The x2c Book

The language, standard library, and compiler

The x2c authors

x2c-lang.dev

Contents

The x2c Book 1
Language guide 2
Install a Native Compiler 3
From C to x2c 6
What x2c adds to C syntax 11
Values and Var 13
Symbols and Atoms 19
Strings, Lists, Arrays, and Maps 23
Pattern Matching 39
Iteration 45
Scopes and Lifetime 50
Contexts and Threads 55
Errors and Cleanup 58
Protocols 63
Classes and System Macros 68
Compile-time Macros 75
Meta Functions 83
Generating Code 96
Packages 107
Wrapping a C Library 115
Commands and Files 122
Experimental REPL 129
Indentation Syntax 136
Programming Idioms 138
Language reference 143
Language Reference 144
Syntax specifications 198
Command-line interface 226
Release Notes 237
Libraries and Packages 239
Standard Library 240
Advanced Topics 567
Compiler API (provisional) 581
Compiler API (provisional) 582
Compiler and contributor internals 748
Architecture 749
Implementation Map 761
Building the Compiler 767
Literate Lisp in x2c 769

The x2c Book

x2c extends C with dynamic values, collections, and compile-time code generation. Use the features you need; keep ordinary C where it already works. C headers, libraries, data layout, expressions, and native tools remain available. x2c compiles to C, and its compiler and runtime are written in x2c.

This book explains how to use x2c and specifies its language and library. Install a native compiler explains setup.

What x2c adds

x2c adds these features to C:

- `Var` holds values and pointers with runtime type information, alongside ordinary static types.
- `String` provides immutable text with interpolation. Immutable `List` values share structure and support mapping, filtering, and folding; mutable `Array` and `Map` containers grow automatically and store `Vars`.
- `Symbol` provides compact named values without declaring an enum; `SymbolSet` gives those names an explicit order and indexing.
- `Iter` composes traversal operations, and `foreach` visits collections and `File` contents.
- `protocol` defines interfaces that concrete types explicitly adopt. Namespaced methods, inherited typedef methods, delegation, and static or dynamic dispatch let types share behavior while retaining C layouts.
- `Func` holds callable functions and capturing lambdas. Expression-bodied functions, dot methods, and call chains are also available.
- `Scope` and `Context` manage allocation lifetimes and isolated runtime state without garbage collection. Pools give immutable values nested lifetimes.
- `match` selects the first matching case for structured `List` data and exposes named captures as local variables.
- `Thread` runs work on native threads and returns results through `join`; `Mutex` coordinates access to shared mutable data.
- `Error` carries structured failures. `try`, `catch`, and `raise` handle them; `finally` and `defer` run cleanup when control leaves a block.
- `macro` expands to parsed source at compile time, and names a template declares stay private to each expansion. `Type` holes specialize code for concrete types, including typed collections. `Decorator` macros transform the expression or source item that follows. The optional autodiff package uses them to generate dual-number types from a `Type` hole and forward- and reverse-mode derivative functions with decorators.
- `import` brings packages into a local namespace, including wrapped C libraries and their x2c and Lisp bindings.
- `Lisp` evaluates code inside a running program; compile-time Lisp uses the same interpreter to compute values and generate syntax during translation.

A program can use one feature and keep the rest in C.

Reading the book

If you know C, most source will look familiar. The guide covers the features that replace hand-written tagged unions, adapters, repeated declarations, collection allocation, traversal, cleanup, and pattern matching.

- The language guide shows how to use x2c and when its features help.
- The language reference states exact syntax and semantics.
- The standard library describes the available modules and links to their API reference, generated from the source.
- Compiler architecture explains the self-hosting implementation for readers who want to study or contribute to it.

The book describes implemented behavior. The optional `make doc-examples` translates its x2c samples and compiles them to objects. `make doc-outputs` also links and runs samples with expected output. The optional `make examples` runs the showcase programs and checks their output.

Language guide

Install a Native Compiler

Install a release with one command, or build and install from a source checkout. The release route needs `curl`, `tar`, `shasum` or `sha256sum`, and a C compiler named `cc`:

```
curl -fsSL https://x2c-lang.dev/install.sh | sh
export PATH="$HOME/.local/x2c/bin:$PATH"
```

`X2C_PREFIX` chooses another prefix and `--version <n>` pins a release; the current version comes from `https://x2c-lang.dev/x2c-version.txt`. Running the script again upgrades the prefix and keeps installed packages. `tools/check-install-script.sh` exercises the same script against a release layout built from the current tree.

From a source checkout:

```
make install PREFIX="$HOME/.local/x2c"
export PATH="$HOME/.local/x2c/bin:$PATH"
x2c build hello.x --output hello
./hello
```

A source build needs a C compiler, `ar`, GNU Make, Python 3, and `/bin/bash`; see building the compiler. The install command builds the compiler and installs its matching runtime, headers, compile-time Lisp SDK, unit interfaces written by the installed compiler, runnable examples, an empty package directory, and license. With no `PREFIX`, `make install` runs `make build-install`: it copies the development compiler to `bin/x2c-<branch>`, links `bin/x2c` to that copy, and installs the shipped commands under `libexec/x2c/`.

Upgrading from 0.14.0

Version 0.15.0 reads package native arguments from `<name>.native.rsp` instead of `<name>.link`. The installer keeps existing packages, so rebuild source packages and replace old native bundles with bundles built for 0.15.0.

The SQLite package calls its prepared-statement type `Prepared` instead of `Statement`; change imports and typed variables, and use `Var.prepared` in place of `Var.statement`. Automatic differentiation moved from the runtime to the optional `autodiff` package. Build or install that package and use `import "autodiff"` for its runtime types and macros.

The removed `List.pool_*` operations now belong to `Pool`. The removed `List.reduce`, `Array.reduce`, and `Iter.reduce` operations have no matching method; use `Iter.foldl` when an explicit seed fits. Use `Var.truth` in place of `Var.truthy`.

The x2c home

An installed prefix and a source checkout share one layout, the x2c home:

```
<home>/bin/x2c           the compiler
<home>/include/x2c/      runtime sources and generated headers
<home>/lib/              the runtime archive and unit interfaces
<home>/etc/              compile-time Lisp and macro sources
<home>/packages/         installed packages, one directory each
<home>/examples/         runnable examples (installed prefix)
<home>/src/              language component sources (installed prefix);
                         all compiler sources (checkout)
```

The compiler finds its home by walking up from its own executable, then from the current directory, to the nearest directory holding `include/` and `etc/compiler-sdk.xlisp`. `X2C_HOME` sets the home explicitly and takes precedence. `x2c env` prints the resolved home, layout, package roots, and host tools; `x2c env home` prints one value. Every `import` searches `<home>/packages` after the explicit `--package-dir` and manifest directories, so a package placed there needs no build flags.

`PREFIX` must be absolute and dedicated to x2c. Move or rename the complete prefix to relocate it. Keep `bin`, `include`, `lib`, `etc`, `packages`, and `licenses` together. Paths may contain spaces. Quote the compiler path when invoking it:

```
"/path with spaces/x2c/bin/x2c" build hello.x --output hello
```

An upgrade stages a complete payload, replaces files owned by the previous installation, and removes obsolete owned files using its inventory. It leaves unrelated files alone and refuses to overwrite an unrelated file with the same name. Publication replaces individual files; do not run an installation while programs are compiling against it. Keep `.x2c-install-manifest` with the prefix so upgrades retain ownership and build identity information.

For a packaging staging directory, use `DESTDIR` with the intended prefix:

```
make install PREFIX=/opt/x2c DESTDIR="$PWD/staging"
```

This writes `staging/opt/x2c`. Move that complete directory to its destination; the staging path is not embedded in the compiler's runtime configuration. `make dist PREFIX=/opt/x2c` stages the same installation under `dist/` and writes `dist/x2c-<version>-<platform>.tar.gz` with a `.sha256` beside it.

`make uninstall PREFIX="$HOME/.local/x2c"` removes every file the inventory owns and any directory that becomes empty. Packages installed under `<prefix>/packages` are not owned by the compiler inventory: an upgrade keeps them, and an uninstall leaves the prefix in place when they remain. Installing these support directories directly into a shared tree such as `/usr` is not supported. Native executables and archives run only on the platform and architecture that built them.

An installed prefix records `cc` and `ar` in `lib/x2c/toolchain`; Environment gives the full tool selection order.

Windows

Use WSL2. Inside an Ubuntu 24.04 or newer distribution the Linux release installs with the same command. The installer needs `curl` and a `cc` (clang 15 or newer, or gcc); `make` is needed only for a source checkout or package builds:

```
sudo apt-get install -y clang make curl
curl -fsSL https://x2c-lang.dev/install.sh | sh
```

Keep the prefix and your projects on the Linux filesystem, under your WSL home, because builds under `/mnt/c` are slower. Use clang 15 or newer, or `gcc`: Ubuntu 22.04 ships clang 14, which rejects two static initializers in the runtime.

A source checkout also builds under MSYS2 with its `gcc`, given three settings: check the repository out with LF line endings (`git config core.autocrlf false`), enable native symlinks (`export MSYS=winsymlinks:native`, which needs Developer Mode or an administrator shell), and select the compiler with `CC=gcc X2C_CC=gcc`. `make build-safe` then produces a working compiler, and the `check` workflow's `windows_spike` input tests both routes. Native Windows without a POSIX layer is not supported, and under MSYS2 only a source build is supported.

Developer workflow

The `dev` branch is the everyday integration branch; `main` is reserved for releases. Start new development checkouts and branches from `dev`. A source checkout is the development install. After `make build-safe`, the checkout is a complete home with its own `packages/` directory, and `bin/x2c` is the bootstrap compiler. Run `make build-install` before putting `bin/` on `PATH`; it links `bin/x2c` to a copy of the current branch's compiler and copies its shipped commands into `libexec/x2c/`. A compiler in the checkout's `bin/` links `builds/0/libx2c.a`, the runtime built with the headers in `include/x2c`. Select the checkout for one shell with `PATH`, or for one command with `X2C_HOME`:

```
export PATH="$HOME/src/x2c/bin:$PATH"
X2C_HOME="$HOME/src/x2c" x2c env
```

Keep the global prefix on a release. Install it from a tagged `main`, never from a working branch, so a bug seen through it reproduces for every user. Validate a staging candidate in a fresh scratch prefix and use its explicit package index, then remove that prefix afterwards. The staging site supplies the candidate installer and commands; bare `x2c install` still uses the production package index. Compiler version alone does not distinguish two candidates of the same version. Compiler development itself runs through `make` inside the checkout and does not depend on which compiler `PATH` selects.

Releases publish the compiler for macOS and Linux on arm64 and x86_64. The installer's default is the newest release; `--version <n>` installs an earlier one, and every release's downloads stay available.

Build, debug, and use packages

Start a project with `x2c new`:

```
x2c new hello
cd hello
x2c run -q
```

Direct source and `x2c.toml` project builds use the same installed compiler. For source debugging, opt into native debug information and original source locations:

```
x2c build -g --source-map hello.x --output hello
lldb ./hello
```

On macOS this also produces `hello.dSYM`; move that directory with the executable. The build removes generated objects; pass `--build-dir` to keep them. Source mapping gives the location rules.

Compile C against the x2c runtime

A C program can call x2c code through the generated header. Point `-I` at the include root and write the runtime header as `<x2c/x2c.h>`, so every system header still resolves to the C library:

```
#include <string.h>        /* the C library */
#include <x2c/x2c.h>       /* the x2c runtime */

cc app.c -I "$HOME/.local/x2c/include" \
  "$HOME/.local/x2c/lib/libx2c.a" -pthread -lm -o app
```

To compile C generated by `x2c translate` by hand, see external builds. To build a package from a source archive, see packages.

From C to x2c

This chapter lists where x2c syntax or meaning differs from C. Each section links to the chapter that explains it.

x2c translates to C and uses the native compiler, linker, object layout, and calling conventions. Scalars, aggregates, pointers, arrays, and ordinary functions remain useful representations. Each `.x` unit also loads the standard runtime prelude. `Var`, collections, protocols, and closures add operations where you use them. Memory ownership remains explicit: there is no garbage collector or reference counting.

Files, visibility, and the native build

Write or run	Meaning
`module.x`	A unit containing declarations and definitions; translation produces a C source and header
`#include "module.x"`	Make that unit's public declarations and compile-time definitions visible
`#include "native.h"`	Read native declarations and preserve the native include in generated C
`static`	Private function/object; also permitted on a type declaration to control its publication
`import "geo" as g with Vec;`	Import a registered package; use `g.Vec` or the selected bare name `Vec`
`x2c translate --out-dir generated main.x`	Let an existing native build compile and link the output
`x2c build --output app main.x helper.x`	Let x2c own compilation and linking; list the local implementation units
`x2c run main.x -- argument`	Build and execute

Declarations are public by default. A type that a public declaration needs becomes public; its body is published only when the interface needs the layout. Including a local `.x` exposes its declarations, and the build must also link its implementation. See units and visibility and type-owned initialization.

A first-line `#!/usr/bin/env -S x2c script` makes a script unit, whose top-level statements form the program body. See scripting.

An `.xp` file, or `#pragma indent` before code, uses indentation in place of braces and semicolons. The parser then reads the same tokens as the brace form. See indentation syntax.

Types, methods, and references

A method is a function with a type-qualified name. Dotted calls select it from the receiver's static type and pass the receiver as the first argument.

```
typedef struct Counter { int value; } Counter;

void Counter.add(Counter &counter, int amount) => counter.value += amount;
int Counter.get(Counter counter) => counter.value;

Counter count = {0};
count.add(3);
printf("%d\n", count.get());
```

Spelling	Rule
`int T.f(T value, int n)`	Defines `T.f`; generated C uses a function such as `T_f`
`object.field`	Also works through one pointer level when x2c knows the aggregate layout; `->` remains valid
`T &value` parameter	Borrows the caller's addressable `T`; call with `value`, not `&value`
`T &?value` parameter	Also accepts `NULL`; test presence before using the borrowed object
`Self` in a method signature	Preserves the original receiver's static typedef at the marked parameter/result positions
`delegate T field;`	Forwards otherwise unresolved dotted method calls through that field

Spelling	Rule
`result_type f(...) => expression;`	A one-expression function body; a `void` function executes it without returning a value

Reference parameters lower to C pointers and are allowed only on parameters. Use `->` where x2c cannot see the aggregate definition. Some pointer mismatches that C only warns about are x2c errors. The C foundation gives the rules for references, delegation, typedef chains, and conversions.

What `class` and `protocol` mean

`class` is a shipped declaration macro. The declaration chooses an ordinary C representation and supplies applicable constructors, conversions, comparison, hashing, and printing defaults:

```
class Point { int x; int y; };
class Node { int value; } *;
class Count int;

Point p = Point.new(3, 4);       // record value
Node n = Node.new(7);           // pointer; default allocation uses Scope
Point copy = p;                 // ordinary shallow record copy
n.free();
```

Methods can replace the defaults. Copying a record with an Array or pointer field shares that backing object. See classes and system macros.

A protocol declares a compile-time contract. On adoption, the compiler resolves the protocol's members and any adapters they need:

```
protocol Reading(T) {
  int T.value(T);
}

int Gauge.value(Gauge gauge) => gauge.value;
protocol Reading(Gauge);
```

A type participates only through an explicit adoption. `Var(T)` also connects a type to boxing and runtime operations. See protocols.

Runtime values and implicit operations

Type	Representation and consequence
Native C type	Keeps native storage and arithmetic unless an x2c conversion or protocol operation applies
`Var`	Eight-byte tagged value; supported values box/unbox at typed boundaries; wide scalars and some custom values require allocated boxes
`String`	Immutable canonical text; equal content shares storage
`List`	Immutable canonical cons structure; elements are `Var` values
`Array`, `Map`	Mutable objects with identity; copying their handles shares the object
`Symbol`	Encoded 64-bit tag, written `<ready>`; source spelling must fit exactly
`Atom`	Exact name of arbitrary length, used by quoted data; compact names use Symbol representation
`Func`	Callable runtime handle, possibly with captures; calls through it return `Var`

Initializers, assignments, typed arguments, and returns can call converters: `Var v = 42;` boxes, and a typed destination unboxes. Some pairs have no converter. `NULL`, an empty `String` or `List`, and `void` are three different values, and a condition on a runtime value uses its truth operation, so an empty `Array` is false. See values and Var.

Literals: evaluated expressions versus quoted data

```
int n = 4;
Array evaluated = [n, n + 1];
List quoted = %(n $n ${n + 1});
Map settings = {mode: "fast", count: n};
String message = %"count=$n next=${n + 1}";
```

The Array contains `4, 5`. The List contains the Atom `n`, then `4, 5`. The Map's bare keys are Atoms; `{(name): value}` uses a variable as its key.

Form	Contents
`[expressions]`, `{key: expression}`	Evaluate expressions; nested bare collections do likewise
`%(items)`	Whitespace-separated List data; bare names are Atoms
`%[items]`, `%{key: value}`	Array/Map with quoted contents; commas separate entries
`%"text $name ${expression}"`	Interpolated String
`%<<ready busy done>>`	Immutable ordered SymbolSet; literal members only
`$name`, `${expression}` inside quoted data	Insert one evaluated value
`@name`, `@{expression}` inside a quoted List	Splice a List's elements; Arrays/Maps do not support this splice

Bare `{}` depends on its destination: it builds a `Map` or `Array`, or zero initializes a native object. Inside `%()`, commas and apostrophes are Lisp reader punctuation. After a cast's closing `)`, `%(` reads as modulo, so write `(List) (%(a b))`. Numbers also accept `0o` octal and `0b` binary. See collections and the literal rules.

Operators, indexing, and small declaration additions

Spelling	Meaning
`a == b`, `a != b`	Uses the type's equality operation when one exists; two `Var` numbers compare as their C values do
`a === b`, `a !== b`	Identity tests; two `Var` numbers are identical with the same type and bits
`value is Type`, `value is not Type`	Require a Var left operand; test its exact tag, not ancestry or convertibility; a tag Symbol may replace `Type`
`item in collection`	Membership; Maps test keys, Strings test substrings; negate with `!(item in collection)`
`items[-1]`, `items[start:stop:step]`	Negative indexing and slicing on Array/List/String; omitted bounds are allowed, zero step is invalid
`int i, float x;`	Restart declaration specifiers after a comma
`Var (a, b) = values;`	Destructure a List into new bindings
`(int a, String b) = values;`	Destructure into separately typed new bindings
`(a, b) = values;`	Destructure into existing simple identifiers
`threaded`	Spelling of C thread storage; `_Thread_local` and `thread_local` also work

Other operators may call protocol members: `String` `+` concatenates, and `String` comparison uses content. The reference gives the rules for indexing, declaration rows, destructuring, and operator precedence.

Iteration, matching, and closures

```
List rows = %((add 2) (add 5));
foreach (List row in rows) {
  match (row) {
    case %(add ?(int amount)) if (amount > 0):
      printf("%d\n", amount);
    default: break;
  }
}
```

`foreach` evaluates its source once. `match` tries its arms in order: `?name` binds one element, `*rest` binds the remaining elements, and `?(T name)` also checks the tag. See iteration and pattern matching.

```
int total = 1;
Func snapshot = %!() => total;
Func add = %!(int n) using &total => total += n;
total = 10;
Var old = snapshot();            // 1
Var next = add(2);               // 12; total becomes 12
```

Lambdas use `%!(parameters) => expression` or `=> { statements }`. Captured locals are read-only snapshots unless `using &name` shares them. Only a noncapturing lambda can become a C function pointer. See lambdas.

Ownership, cleanup, and errors

C allocations still need their ordinary owner. Runtime mutable allocations and wide `Var` boxes belong to a `Scope`, and canonical `String` and `List` values normally last for the process. See scopes and lifetime.

```
$scope() {
  File file = $auto(tmpfile());
  file.puts("hello\n");
  Array scratch = [1, 2, 3];
}
```

`$scope()` retains an allocation region around a statement, and `$auto(initializer)` releases a local at block exit. `$lock` and `$let` give scoped locking and temporary replacement. See classes and system macros. `defer statement;` runs the statement at block exit, in reverse order of registration.

```
int n = -1;
try {
  if (n < 0) raise %(bad-arg (value $n));
}
catch %(bad-arg *details): {
  printf("bad argument: %s\n", details.repr());
}
finally {
  puts("finished protected work");
}
```

An `Error` has a `Symbol` cause and immutable details. Here `$n` inserts the variable; a bare `n` would be an `Atom`. See errors and cleanup. A `Context` bounds allocation and error state for a task, and a `Thread` runs one on a native thread. See contexts and threads.

Compile-time code and source substitution

```
macro Expression $twice(Expr $value) => $value + $value;
int result = $twice(3);
```

A macro takes parsed source. Its hole kinds, such as `Expr` and `Stmt`, state what each argument accepts. An argument used twice is evaluated twice: `$twice(f())` calls `f()` twice.

Spelling	Purpose
`$name(arguments)` in code	Invoke a macro or visible meta function
`$!(expression)`, `$!{ statements }`, `$!Unit{ ... }`	Quote source as syntax for compile-time construction
`$(...)`	Evaluate compile-time Lisp
`meta` function	Expose a function to compile-time calls; `meta native` declares a callable supplied by the compiler or a loaded native module
`keyword`	Give a macro a source keyword alias; shipped `class` and `foreach` use this mechanism
Decorator macro	Transform the following expression, statement, field, function, named type, or unit item

A `meta` function can also be called at runtime; `$f(...)` runs it at compile time. `with expression as name { ... }` substitutes the expression at each use of `name`. See macros, meta functions, and generating code.

Before bringing over a C source file

x2c parses source before host preprocessing, so most conditional branches must parse, including ones the native build disables. A C macro that supplies missing grammar may need an x2c macro or a rewritten call site. See host preprocessing.

- Use `(void)` for an ordinary no-argument function declaration. Do not assume every C dialect, wide/UTF literal spelling, or identifier spelling is accepted; identifier scanning is ASCII. The lexical specification records the exact boundaries.
- `_Static_assert`, designated initializers, `_Generic`, GNU attributes, and statement expressions are supported native forms. `_Generic` has no x2c result type until a cast supplies one, so it cannot directly drive boxing or method lookup.
- `in` and `match` remain ordinary identifiers outside their contextual forms. Other new keywords can collide with C names. Check the keyword inventory.
- Literal printf-family formats can trigger conversions of Var arguments. A computed format with a Var argument is rejected. For native arguments, respect the native variadic ABI.

The x2c syntax map lists every addition. The syntax specifications give the exact productions, and the language reference states the exact rules. Idioms shows how to combine them.

What x2c adds to C syntax

For a C reader, x2c adds eleven useful syntax families. Most declarations, expressions, and control flow keep their familiar shapes. Start with the families needed by your program; macros and compile-time code can wait.

This map groups additions from the source grammar and lexical specification. It is onboarding material, not a second language specification. The language reference owns each construct's meaning.

The short list

Extension family	Grammar additions	Lexical changes
Source organization	import with aliases/member selection, package-qualified alias.Type/alias.member names, module member text splicing before parsing, and script statements outside an explicit main.	import is a keyword; a leading shebang is handled before lexing.
Optional indentation	The parser receives ordinary braces, parentheses, and semicolons.	In .xp or after #pragma indent, a token pass rewrites layout and colons.
Declarations and references	Mixed-type rows, T.method names, & and &? parameters, delegate fields, destructuring, class/NamedType forms, and static type visibility.	delegate and threaded are keywords; Self is contextual. Existing punctuation gains declaration roles.
Protocols	Protocol definitions/adoptions, associated types, member aliases, and as/tag adoption modifiers.	protocol and associated are keywords; as and tag are contextual words.
Short functions and lambdas	Expression bodies use =>; lambdas permit typed or bare parameters and using captures.	%! is a lambda opener; => remains two tokens; using is contextual.
Evaluated collections	Bare Array/Map literals, destination-typed brace expressions, void as a value, and trailing ordinary call-argument commas.	No new delimiter tokens: [], {}, commas, colons, and void acquire expression roles.
Quoted data and text	List/data readers, interpolation, splicing, reader prefixes, typed pattern captures, Symbols, and SymbolSets.	Percent openers select reader modes; braced interpolation/splicing re-enters code; angle Symbols, atom boundaries, byte escapes, signed data numbers, and explicit 0o octal extend scanning.
Operators and slices	and !==, in, is/is not, and [start:stop:step] extend the C expression ladder.	**and !== are operator tokens; in is** contextually retagged; is and not remain contextual words.
Iteration and matching	foreach binders and match cases with patterns and optional guards; case/default otherwise remain C forms.	match is contextually retagged; foreach is a shipped keyword alias; pattern sigils are data-mode atom spellings.
Cleanup and errors	with/as blocks, defer statements, try/catch/finally, and raise with structured details.	defer, try, catch, finally, and raise are keywords; with/as are contextual; error patterns reuse List mode.
Syntax-producing code	Typed macro holes/results, @ sequence parameters and splices, aliases, decorators, source quotations, Lisp escapes, and meta/native functions.	$ inserts one result; @ splices a sequence. $(and @(enter compile-time Lisp in code; $! uses ordinary ! after $; macro/keyword/meta/native are contextual.

The grammar column identifies added alternatives or contexts. It does not classify an entire shared production as non-C. For example, == remains C; === is an addition within the same equality production. An & before a parameter name is new reference syntax; unary `&value` is ordinary C.

Grammar changes need not introduce tokens. Bare `[a, b]` Arrays and `{key: value}` Maps reuse C punctuation in new expression positions. Conversely, percent literals change how their contents are tokenized. => is a grammar spelling composed of two tokens, not a new lexer token.

What was subtracted

The comparison baseline is C11, using its lexical clauses and syntax summary: WG14 N1570, sections 6.4 and Annex A. Ordinary C declarations, pointer declarators, expressions, control flow, designated initializers, compound literals, `_Static_assert`, `_Generic`, and `_Thread_local` are therefore absent from the short list.

This is a reviewed classification of the extracted forms. It is not an automatically proven difference between the languages accepted by two complete parsers. Contextual binding, macro signatures, and lexical modes prevent a simple subtraction of production names from giving that result.

Three boundaries keep the map useful:

- Existing native extensions: GNU statement expressions (`({ ... })`), GNU attributes, vendor `^` declarators, linkage strings/groups, and native annotation macros support interoperability. They are not x2c inventions.
- Later C overlap: C23 includes binary `0b` integers, the `thread_local` keyword, and empty initializers. These are outside the C11 baseline but are not uniquely x2c. See WG14 N3096, sections 6.4.4.1, 6.7.1, and 6.7.10. x2c's `threaded` is another spelling of C thread storage. Its `{}` can construct an Array or Map according to the destination, which adds meaning.
- Implementation differences: ASCII identifier scanning, separate wide/UTF string-prefix tokens, numeric and escape edge cases, required `(void)` in ordinary no-argument declarations, and aggregate-body limits belong in the specifications' compatibility notes. They are not onboarding features or evidence of complete C conformance.

What syntax alone cannot show

Names such as `String`, `Var`, and `List` use ordinary type-name syntax. Boxing, conversion, lifetime, truthiness, pointer-dot access, method-call resolution, and operators backed by protocols add behavior without always adding syntax. `(a, b) = values` also needs its assignment interpretation to explain destructuring. A grammar-only diagram must not imply these behaviors follow from punctuation alone.

`class` and `foreach` are shipped aliases built on x2c's macro system. User macros can add further source forms through the same typed positions. Consequently, the short list describes the built-in and shipped language surface; it cannot enumerate every future alias or macro invocation.

Where the table comes from

The feature records in `etc/syntax/extensions.x` connect each family to grammar productions, lexical rules, example spellings, and a reference section. The table above is generated from those records. The grouping and C comparison are reviewed annotations; grammar and lexer data do not infer pedagogical categories themselves.

Values and Var

Most data in an x2c program has an ordinary C type: `int`, `double`, `char *`, or a struct you declared.

Some code has to hold whatever the caller supplied. The element of a heterogeneous list, the value stored behind a map key, `Error` detail, the argument to a routine that prints anything: none of those have one static type. In plain C you would write a tagged union by hand and a switch for every operation. x2c provides that type: `Var`.

A value that carries its own type

A `Var` is a single 64-bit value holding a payload plus a tag that names what the payload is. Assigning a typed value into a `Var` boxes it, and the tag records where the value came from.

```
Var count = 42;
Var name = "ada";
Var setting = <verbose>;
Var numbers = %(1 2 3);

printf("%s %s %s %s\n",
       count.tag().str(), name.tag().str(),
       setting.tag().str(), numbers.tag().str());
```

```
i32 string symbol list
```

Three queries read the tag. `Var.tag` returns the exact family, as above. `Var.kind` returns the coarse category: `<integer>`, `<floating>`, `<symbol>`, `<object>`, `<pointer>`, `<reference>`, or `<void>`. Use it when a whole group of tags is treated alike. `value is i32` tests for one specific tag.

Every `Var` has the same size and shape whatever it holds, so heterogeneous containers work. `List`, `Array`, and `Map` all store `Var`, so a list can mix a number, a `String`, and a `Symbol`.

```
List mixed = %($count $name verbose);

foreach(Var item, mixed)
  printf("%s = %s\n", item.tag().str(), item);
```

```
i32 = 42
string = ada
symbol = verbose
```

A `Var` is eight bytes, and most families keep their payload inside those bits with no allocation. Five native families cannot: `long`, `unsigned long`, `long long`, `unsigned long long`, and `long double`. Boxing one of those allocates a small immutable box in the active scope, so the boxed value follows the scope lifetime described in scopes and lifetime. Those five tags are spelled `<long>`, `<ulong>`, `<llong>`, `<ullong>`, and `<ldouble>`. Their payload widths follow the corresponding native C types.

The complete tag set and the mapping from native source types to tags are in the language reference.

Getting the value back out

Assigning a `Var` to a typed target unboxes it, and the compiler inserts the conversion for you.

```
Var boxed = 42;

int native = boxed;
double widened = boxed;
printf("%d %f\n", native, widened);
```

```
42 42.000000
```

`Var.convert` performs every numeric conversion, and compiler-inserted unboxing calls it before using the target's extractor. `int native = boxed;` and an explicit conversion behave the same way.

A `Var` argument unboxes the same way at any call whose prototype x2c has parsed. x2c does not read system headers, so the runtime declares the C99 `<math.h>` functions itself; a `Var` holding a number goes straight into one with the ordinary include:

```
#include <math.h>
Var ratio = 2.5;
printf("%f\n", sin(ratio));

0.598472
```

The scalar-named readers, `Var.int`, `Var.short`, `Var.uchar`, and their siblings, work the same way. A matching tag reads its payload directly, and any other numeric tag converts through `Var.convert`, so a reader call and an assignment agree:

```
Var small = (unsigned char) 44;

int converted = small;
int reader = small.int();
printf("tag=%s converted=%d reader=%d\n",
       small.tag().str(), converted, reader);

tag=u8 converted=44 reader=44
```

A nonnumeric source raises the same structured `Error` either way. The raw payload readers, `Var.integer`, `Var.floating`, and the wide `*_value` readers, are silent low-level decoders that return zero for a tag they do not handle.

A crossing to a pointer-shaped runtime type has nothing to convert. The payload either has that tag or it does not. Assigning to `List`, `Map`, `Array`, `Symbol`, or a type with a declared `protocol Var(T)` conversion uses that type's exact reader, and a wrong tag yields NULL. An alias of such a type, declared with `class` or `typedef`, uses the reader of its nearest ancestor that has one. A `String` destination reads a `String` payload as itself, displays any other value through `.str()`, and reads `void` and `Null` as NULL. The explicit readers `.list()`, `.map()`, `.array()`, `.block()`, `.buffer()`, `.bytes()`, `.file()`, `.as_iter()`, and `.string()` extract their payload and raise `<no-convert>` for any other tag. Their `_payload` forms, such as `.list_payload()`, read the payload or NULL without raising, as an assignment does:

```
Var listy = %(1 2 3);
String text = listy;
List same = listy;
printf("text=%s same=%d\n", text, same != NULL);

text=( 1 2 3 ) same=1
```

Call `.list()` when a value must be a `List`, so a wrong tag is reported where it is read. `Var.pointer` is the unchecked read, for a raw native pointer that has no tag to check.

Formatting dynamic values

`String` interpolation displays a `Var` with `Var.str`, whether it holds a number, a `String`, or another dynamic value:

```
Var key = 23, val = 3.14, answer = "hello";
puts(%"output: ($key: $val) = $answer");
```

A numeric segment of an interpolated `String` renders through its nearest declared `T_str` converter when it has one and through `Var.str` otherwise. Converter lookup follows the source's typedef chain, and an exact converter wins. A segment already typed `Var` renders through `Var.str` whatever tag it holds:

```
int count = 2;
Var value = 3.14;

String counted = %"processed $count items";  /* processed 2 items */
String read = %"v = $value";                   /* v = 3.140000 */
```

A printf-family call with a single static format literal can use the format to request native numeric arguments:

```
Var key = 23, val = 3.14, answer = "hello";
printf("output: (%d: %.2lf) = %s\n", key, val, answer);
```

For numeric conversions, the compiler inserts the checked `Var.convert` path used by typed assignment and then passes the format's promoted C vararg type. A `Var` supplied for `%s` uses `Var.str`. This applies to `printf`, `fprintf`, `sprintf`, `snprintf`, and the `printf` methods on `String`, `File`, and `Buffer`.

The format must be a direct static literal whenever a variadic argument is a `Var`. Automatic lowering covers integer, floating, `%c`, `%s`, width, and precision arguments with the `hh`, `h`, `l`, `ll`, and `L` modifiers. Use an explicit native conversion for `j`, `z`, `t`, `%p`, `%n`, wide character/string conversions, positional formats, or a dynamic format. Calls containing only native arguments keep C's printf behavior.

Null and void

`Null` is the all-zero `Var`: the null pointer, or the external `nil`. It is ordinary data. You may store it in a `List`, an `Array`, or a `Map`, and iterating a collection can return it.

`void` is the all-ones `Var`. APIs return it to report that no value is available: an index out of range, an absent key, or an exhausted iterator. No collection or iterator holds it. You cannot push it into an `Array`, store it as a `Map` key or value, or receive it as a successful iterator payload. In an expression, lowercase `void` is the literal for this sentinel. In a declaration, cast, `sizeof`, or parameter list, `void` keeps its C meaning.

You can inspect the sentinel without it becoming data. Two `void` operands are equal with `==` and identical with `===`, and `!=` and `!==` are their inverses. A sentinel and any ordinary value are unequal. The same rules apply to `Var.equal`, `Var.fallback_equal`, `Var.same`, and the four comparison operations accepted by `Var.binary`. Display and readable output both spell the sentinel `void`.

`void` is what you get from an out-of-range `Array` or `List` read and from a missing key in a plain `Map`. A `String` read out of range returns `-1`, a packed typed `Array` bracket is unchecked, and a typed `Map` raises `<bad-arg>` for a missing key:

```
List items = %(10 20 30);

Var present = items[1];
Var missing = items[7];
printf("present=%d missing=%d\n", present is void, missing is void);

present=0 missing=1
```

A missing `Map` key reads the same way, but `Map` also offers a status-bearing form that keeps presence separate from the payload. Prefer it:

```
Map settings = {width: 80};

Var height;
if (settings.try_get(<height>, height)) {
  int value = height;
  printf("height is %d\n", value);
} else {
  printf("height is unset\n");
}

height is unset
```

`Null` is a value that happens to be zero. `NULL` converts to the same value when an x2c crossing requires a `Var`:

```
Var absent = (Var) { .u64 = 0 };
Var also_absent = NULL;
List row = %($absent 1);

printf("null=%d void=%d truthy=%d len=%d\n",
       absent.is_null(), absent is void, absent.truth(), row.len());

null=1 void=0 truthy=0 len=2
```

In a condition, `Null` is false, and a truthiness test on `void` raises `<void-op>`. A matching filtered `catch` receives that error; without one, the default abort terminates the program. Do not write `if (maybe_missing)` on a value that might be `void`. Test it with `Var.is_void`, or use the status-bearing API and never hold a `void`.

The representation of `void` and its exclusion from collections and iterators are consistent. Its meaning in a particular API can vary: missing, exhausted, or invalid. Read that API's description before interpreting the result.

Truthiness

When a condition's static type participates in a protocol with `truth`, the compiler applies that member. Dynamic `Var` uses `Var.truth`; built-in `Strings`, `Lists`, `Arrays`, `Maps`, and `Buffers` use their concrete content rule. This covers `if`, `while`, `do`, classic `for`, `?:`, unary `!`, and each participating operand of `&&` and `||`. Short-circuit evaluation is untouched. Each operand is converted only when C would evaluate it, so the right-hand side of `&&` does not run when the left is false.

```
Var name = "";
Var full = "ada";

if (!name) printf("no name\n");
if (full && full is <string>) printf("name=%s\n", full);

no name
name=ada
```

False: numeric zero, `Symbol` zero, `Null` and null pointers, the canonical empty `String` and empty `List`, and empty `Array`, `Map`, `Block`, `Bytes`, or `Buffer` values. True: everything else nonzero or nonnull, including NaN and the infinities, which are numbers. A nonnull `Iter` is true even when it is exhausted. The condition tests the iterator pointer; it does not try to advance it.

Arithmetic on Var

If either operand of `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `^`, or `|` is a `Var` and the other is a `Var` or statically numeric, both operands are boxed as `Var`, the runtime performs the operation, and the result is a `Var`. You write C syntax; the compiler inserts the boxing.

```
Var count = 10;

Var total = count * 3 + 1;
double scaled = count * 2.5;
printf("total=%s scaled=%f\n", total, scaled);

total=31 scaled=25.000000
```

Expressions with no `Var` operand stay native C and make no runtime calls. Integer operands use integer promotion; arithmetic keeps the promoted type's low bits, so overflow wraps to that type's width. Floating operands work in the widest participating floating family with host behavior. Remainder, the shifts, and the bitwise operators reject a floating operand. There is no dynamic unary numeric operator, and no dynamic `++` or `--`.

A statically nonnumeric peer is a compile-time error:

```
Var count = 10;
String label = "items";
Var bad = count + label;   /* xform: dynamic numeric operators require
                              numeric operands */
```

A `Var` that *happens* to hold a `String` passes the static check and fails at run time instead.

Comparisons have their own rule. A comparison of two numeric `Vars` behaves as the same comparison between their native values in C, after the usual arithmetic conversions. `===` and `!==` on numbers require the same type and the same bits. Other `Var` operands compare through `Var.equal`, `Var.same`, and `Var.compare`.

```
Var three = 3, wide = 3L, nan = 0.0 / 0.0;
printf("3 == 3L: %d  3 === 3L: %d  -1 < 1u: %d\n",
       three == wide, three === wide, (Var) -1 < 1u);
printf("NaN == NaN: %d  NaN === NaN: %d\n", nan == nan, nan === nan);

3 == 3L: 1  3 === 3L: 0  -1 < 1u: 0
NaN == NaN: 0  NaN === NaN: 1
```

`Map` keys and sorting keep the numeric types apart; see Collections. See the language reference for the dispatch.

Handling dynamic-operation failures

`Var.convert`, `Var.binary`, `Var.truth`, `Var.update`, and `Var.postfix` raise cause-specific `Errors`. Their default policies fail fast, and compiler-inserted conversions raise the same causes. Install a filtered `catch` where your code can recover from one of them.

```
Var quotient = void;
try quotient = Var.binary(10, </>, 0);
catch %(div-zero *):
  printf("no quotient: division by zero\n");

no quotient: division by zero

Var narrowed = Var.convert(300, <u8>);
int value = narrowed;
printf("%s holds %d\n", narrowed.tag().str(), value);

u8 holds 44
```

Integer-to-integer conversion keeps the destination width's low bits and does not detour through floating point, so 300 narrowed to `<u8>` is 44. Float-to-integer conversion truncates toward zero, and raises `<conv-range>` for NaN, the infinities, and anything not representable. Other dynamic-operation causes include `<div-zero>`, `<bad-shift>`, `<no-convert>`, `<bad-op>`, `<bad-types>`, and `<void-op>`; the full list and their exact meanings are in the language reference.

Compound assignment

Numeric compound assignment works when either side involves a `Var`. The compiler takes the address of the lvalue once, and the runtime loads, computes, converts to the target's declared family, and stores only after every step has succeeded. A failed operation or conversion leaves the target unchanged.

```
Var total = 1;
total += 5;

int native = 7;
Var delta = 10;
native += delta;

int shown = total;
printf("total=%d native=%d\n", shown, native);

total=6 native=17
```

Both directions work: a `Var` lvalue with any numeric operand, and a native scalar lvalue updated by a `Var`. C indexing and member access are fine, since those are addressable lvalues. Enum and bitfield targets are excluded.

`Array` and `Map` indices support compound assignment too. The compiler uses the collection's update operation and evaluates the receiver, index, and right operand once:

```
Array counts = [1, 2, 3];

counts[0] += 1;

Var shown = counts[0];
printf("%s\n", shown);
```

```
2
```

Numeric `map[key] += value` also initializes a missing key from `value`; other indexed updates require an existing destination. `List` and `String` indices cannot be updated. The language reference describes the supported operations and conversion rules.

When to use Var

Use `Var` when the dynamic type is part of the problem. Use a static type otherwise.

Use `Var` when:

- the value is heterogeneous: a `List` element, a `Map` key or value, or `Error` detail;
- the tag itself is what your code inspects or reports;
- you are crossing a boundary with no target type, such as a variadic call or a generic callback signature.

Prefer a plain C or runtime type when:

- The static type already says what the function takes. `long checksum(String)` tells a reader more than `Var checksum(Var)`, and the compiler checks it. A `Var` moves those type errors from translation time to run time.
- The code is on a hot path. A dynamic operation calls into the runtime, which validates the encoding and decodes both tags. The native operation is one machine instruction. Boxing a `long` or a `long double` also allocates a box in the active scope.
- A call already takes a `Var` parameter. Pass your `String`, `Symbol`, collection, or scalar straight in, and the compiler converts it. A local `Var` declared only to pass the value adds a name and hides the type.

Written out, the two cases look like this:

```
/* Var fits here: the tag is the answer. */
String describe(Var value) {
  return value.tag();
}

/* Var would only erase a contract the caller already knows. */
int clamp(int value, int low, int high) {
  return value < low ? low : value > high ? high : value;
}
```

Where to go next

- Symbols and atoms for the `<name>` values used above as tags and outcome codes.
- Strings, lists, arrays, and maps for the containers that store `Var`.
- Iteration for `Iter`, `try_next`, and the exhaustion rule that `void` participates in.
- Scopes and lifetime for what owns a boxed wide value.
- Idioms for practical usage, and the language reference for the full rules.

The generated Var, conversion, and operation references list every callable by API tier.

Symbols and Atoms

In C, a name in data is an `enum` constant or a `char *`. An `enum` constant compares quickly and has no text. A `char *` has text, and each comparison costs a `strcmp`. x2c adds two name types that have text and compare with one machine comparison:

- A `Symbol` is a short name packed into a 64-bit integer.
- An `Atom` is an exact name of any length. Equal spellings produce the same value.

A Symbol is a name packed into an integer

`Symbol` is a typedef for `unsigned long`. Its literal syntax is angle brackets: `<name>` for a bare spelling, and `<"punctuated name">` when the spelling contains characters that would otherwise end the token. The characters are encoded in the value itself. There is no allocation, table, or pointer.

```
Symbol state = <ready>;
Symbol also_ready = <ready>;
Symbol arrow = <"->">;

printf("equal: %d\n", state == also_ready);
printf("%s has %d characters\n", state.str(), state.len());
printf("repr: %s\n", arrow.repr());
```

The whole name is encoded in the integer, so a symbol literal is a compile-time constant. A `Symbol` works as a `case` label, and a closed set of names dispatches through a C `switch`.

```
static Symbol check_port(int port) {
  if (port < 0) return <negative>;
  if (port > 65535) return <too-big>;
  return <ok>;
}

static const char *explain(Symbol outcome) {
  switch (outcome) {
    case <ok>:       return "usable";
    case <negative>: return "below the valid range";
    case <too-big>:  return "above the valid range";
  }
  return "unrecognized outcome";
}
```

Zero is the empty symbol. Construction returns it for empty or missing input, it is falsy in a condition, and its representation is `<>`.

How a Symbol is encoded

A `Symbol` stores its characters in the integer. Two `<ready>` literals are equal for the same reason two `42`s are equal, and comparing them is one integer comparison. An integer holds a limited number of characters, so the spelling selects one of two encodings:

- A restricted 5-bit alphabet holds up to ten characters. It covers the letters, `*`, `+`, `?`, `!`, and a separator. Uppercase folds to lowercase, and `-` and `_` are the same separator.
- The general 7-bit encoding holds up to seven ASCII characters and preserves each byte.

A source literal must reproduce its spelling when decoded. The compiler rejects case folding, separator folding, and truncation. `Symbol.new` accepts runtime text and applies those normalizations:

```
printf("%d\n", Symbol.new("Alpha") == <alpha>);          /* 1 */
printf("%d\n", Symbol.new("read_only") == <read-only>);  /* 1 */
printf("%s\n", Symbol.new("verylongidentifier").str());  /* verylongid */
printf("%d\n", <"Token@!">.len());                       /* 7 */
```

Two runtime names can normalize to the same value, so use `Atom.intern` when the exact dynamic spelling matters. Keep a closed set of `Symbol`s short and spell its decoded names in source.

The encoded integers are not in alphabetical order. Use `Symbol.compare`, which compares the decoded text:

```
printf("raw integers say b < abc: %d\n", <b> < <abc>);
printf("Symbol.compare says b > abc: %d\n", <b>.compare(<abc>) > 0);
```

`Symbol.str` decodes the value and returns the canonical `String`, so two calls return the same pointer. Call `Symbol.str` or `Symbol.repr` only when you need the text.

An Atom is an exact name

Use an `Atom` when the spelling must survive intact. `Atom` is a `Var`, and `Atom.intern` turns a spelling into one. Interning one spelling twice produces identical `Var` bits, so equality is a bit comparison. Interning keeps the spelling exactly: it does not fold case or separators, and it does not truncate.

```
Atom exact = Atom.intern("VeryLongIdentifierName");
Atom again = Atom.intern(String.new("VeryLongIdentifierName"));
Atom lower = Atom.intern("verylongidentifiername");

printf("%s\n", exact);              /* VeryLongIdentifierName */
printf("same value: %d\n", exact == again);
printf("case matters: %d\n", exact == lower);
```

The source literal `<VeryLongIdentifierName>` is rejected because its encoded value would decode as `verylongid`. `Symbol.new("VeryLongIdentifierName")` is the lossy runtime operation, and it returns `verylongid`.

	`Symbol`	`Atom`
written as	`<name>`, `<"name">`	`Atom.intern(text)`, bare `%()` element
spelling	exact in source	exact
length limit	10 or 7 characters	none
storage	none; encoded in the value	the canonical `String`, when needed
constant?	yes, usable as a `case` label	no; interned at run time

`Atom.intern` requires a non-empty spelling. A null or empty spelling is a programming error and terminates the process.

How an Atom is stored

`Atom.intern` stores an atom in one of two forms:

- If `Symbol` encoding reproduces every byte of the spelling, the atom is that compact immediate symbol and needs no storage.
- Otherwise the atom is a private `<lsym>` value that points to the canonical `String`.

`Atom.intern("alpha")` is an immediate with no storage, while `Atom.intern("Mixed_Case_Identifier")` points at a canonical `String`. Most code does not need to know which form it has. When it does, `value.is_atom()` is true for both, and `value is Symbol` is true only for the compact form. `Atom.str` returns the exact bytes without copying. For a `String`-backed atom, hashing uses the `String`'s cached hash.

Identity and equality

For both types, equal values are identical.

A `Symbol` is an integer, so `==` compares encoded values. Two different runtime spellings can have one encoding, so `Symbol.new("Alpha") == <alpha>` is true.

An `Atom` is a `Var`, so `==` compares values and `===` compares identity. Interning makes the two agree:

```
printf("equal:    %d\n", left == right);
printf("identical: %d\n", left === right);
printf("raw bits:  %d\n", left.u64 == right.u64);
```

All three print `1`. An equality test is one comparison, even for a long name.

Two cautions:

- Equality is exact-spelling equality. `Atoms` differing only in case are different atoms.

- `compare` orders two long atoms by their spellings and two compact atoms by their decoded text. Between a long atom and a compact atom, the order does not follow the spelling. For a text order over a mixed set, compare `Atom.str` values.

Where each one belongs

Choose the type by who wrote the names.

Use a `Symbol` for a closed set of names that you wrote, such as outcome and lifecycle values. `Lisp.read` reports `<value>`, `<eof>`, `<incomplete>`, or `<malformed>`, and `Error` causes include `<bad-sig>`, `<no-symbol>`, `<bad-arity>`, `<bad-types>`, and `<bad-result>`. A debugger shows each one by name, where an `enum` constant prints as `3`. The compiler uses symbols the same way: AST node heads and the match operator and predicate names are compact symbols, because the compiler chose them.

`Symbol`s also make good map keys for fixed configuration:

```
Map config = {host: "localhost", port: 8080};
printf("%s:%d\n", config[<host>].string(), config[<port>].int());
```

Keep numeric enums where the numeric value matters: indexes, packed fields, arithmetic, and external encodings.

Use an `Atom` for names that come from outside your code: identifiers read from source text, keys in serialized data, user-supplied tags, anything where truncating to ten characters or folding case would corrupt the data. Bare elements of a `%()` list literal are atoms for that reason. A bare element is already the compact `Symbol` when its spelling fits. Brackets inside a `%()` `List` literal only quote text that would otherwise parse as another kind of element:

```
List names = %(alpha Mixed_Case_Identifier 1 <1>);

printf("%s\n", names.getindex(1));   /* Mixed_Case_Identifier */
printf("alpha is compact: %d\n", names.getindex(0) is <symbol>);
printf("number then Symbol: %d %d\n",
       names.getindex(2) is <i32>, names.getindex(3) is <symbol>);
```

`alpha` is short and lowercase, so the compact encoding reproduces it and the `Atom` is that `Symbol`. The compact encoding cannot reproduce the mixed-case name, so that `Atom` points to its canonical `String`. The brackets on `<1>` quote a `Symbol` spelling that bare `1` would parse as an integer. `<"">` is the empty `Symbol` and has no bare `Atom` spelling. Pattern matching uses both types. `Match` operators are compact symbols, and named binder keys such as `?name` are canonical atoms, so long and case-sensitive binder names work.

`Atom` representation escapes delimiters, comment openers, numeric-looking prefixes, whitespace, control bytes, and backslashes, so a printed atom reads back as the same value through `%()` and through the Lisp reader.

Cost

A `Symbol` literal is a compile-time constant. A `Symbol` needs no allocation, table, or lifetime management. `Symbol.str` and `Symbol.repr` are its only costly operations.

`Atom.intern` canonicalizes the spelling through the `String` pool: a hash, a pool lookup, and an allocation the first time the spelling is seen. It then tries the compact encoding. Intern a name once and pass the atom around. After interning, equality and identity are one bit comparison. Hashing a `String`-backed atom reads the `String`'s cached hash. Interning uses the `String` pool and has no separate atom table.

Use a `String` for text you slice or concatenate, such as a message or a file's contents. Use a name type when the main task is recognizing and comparing names.

Lifetime of long atoms

A compact atom is an immediate value and has no lifetime. A long atom holds the canonical `String` and follows that `String`'s pool lifetime. If you intern inside a child pool and the atom must outlive it, call `Atom.promote` before the pool closes:

```
Pool.open_named("scratch-names");
Atom row = Atom.intern(String.printf("row-%d", 314159));
Atom.promote(row);
Pool.close();

printf("%s\n", row);
```

`List.promote` promotes the long atoms inside a list, so a promoted list keeps every identity it contains. `Atom.promote` is for a standalone atom. See Scopes and Lifetime for the lifetime rules.

Where to look next

- The language reference has the literal and representation rules.
- The standard library overview has the runtime rules, including identity and equality for the other types.

Strings, Lists, Arrays, and Maps

x2c keeps C's `char *`, `struct`, and `malloc`, and adds four runtime data types: `String`, `List`, `Array`, and `Map`. Much of the compiler and runtime is built with them.

Four rules apply throughout this chapter:

- Brackets and braces evaluate. `[a, b]` builds an `Array` and `{key: value}` builds a `Map` from ordinary x2c expressions. Only a bare identifier key is quoted, as an `Atom`.
- Percent quotes. The delimiter after `%` selects the literal grammar for the contents: `%"..."`, `%(...)`, `%[...]`, `%{...}`, or `%<<...>>`.
- Dollar unquotes. Inside quoted `List`, `String`, `Array`, or `Map` content, `$name` or `${expression}` inserts one ordinary x2c value.
- `String` and `List` are immutable and canonical: equal values share storage. `Array` and `Map` are mutable: each object has its own identity, even when its contents equal another's.

All four are represented by C pointers and can be boxed into `Var`, so they can share a heterogeneous container. See Values and Var for boxing and conversion.

Array and Map literals

A bracket in operand position builds an `Array`, and a brace whose entries are `key: value` builds a `Map`. Elements, values, and keys are ordinary x2c expressions, except that a bare identifier key is an `Atom`:

```
int port = 8080;
String host = "localhost";

Array ports = [80, 443, port];
Map server = {host: host, port: port, tags: ["web", <public>]};
Map empty = {};

printf("%s %s %d\n", ports.repr(), server[<host>].repr(), empty.len());

[ 80, 443, 8080 ] "localhost" 0
```

A symbol value is written `<public>`, and a computed key is parenthesized, as in `{(host): port}`. `[]` is always a fresh `Array`. `{}` is a fresh empty `Map` when its destination is a `Map` or a `Var`, and a fresh empty `Array` when its destination is an `Array`. For a struct or scalar destination, `{}` is the C zero initializer. A plain string literal is a `String` wherever one is expected, and `String` methods apply to it directly: `"x2c".len()` is `3`.

The percent forms below quote their contents. Use them for symbolic data such as ASTs, patterns, and configuration trees, where the names are the content.

Percent literals

In operand position, `%` followed by a literal delimiter is the quoting sigil. The delimiter selects both the result type and the grammar used inside it. "Quoted" means a literal grammar is reading the contents. It does not mean every percent form holds unevaluated symbolic data.

Form	Builds	Element syntax
`%"text"`	`String`	quoted text, with `$name` / `${expr}` unquote
`%(a b c)`	`List`	quoted symbolic content; bare names are `Atom`s
`%<<a b c>>`	`SymbolSet`	quoted literal `Symbol`s only
`%[a, b, c]`	`Array`	quoted comma-separated values
`%{k: v, k: v}`	`Map`	quoted key/value pairs

The empty forms are `%""`, `%()`, `%<<>>`, `%[]`, and `%{}`.

```
List codes = %(200 301 404);
Array queue = %[10, 20, 30];
Map status = %{200: "ok", 404: "missing"};
```

```
String greeting = "hello, world";

printf("%s %s %s %s\n",
       codes.repr(), queue.repr(), status.repr(), greeting.repr());

(200 301 404) [ 10, 20, 30 ] { 200: "ok", 404: "missing" } "hello, world"
```

The four forms print differently. `repr` is the readable representation for each type, and the `List` form round-trips through the same reader the Lisp showcase uses.

A number's `repr` prints its own value and, where C has one, the suffix that gives the width it was boxed at: `5u` for an `unsigned`, `5l` for a `long`, `0.1f` for a `float`. Floating text is the shortest decimal that reads back as the same value. The reader itself has no suffixes, so it is the `int` and `double` forms that read back unchanged; a suffixed form is for you to read, not for the reader.

Quoted data and x2c expressions

Inside `%(...)`, `%[...]`, and `%{...}`, a bare spelling is an `Atom`, an exact, case-sensitive name. Use `$name` to insert one identifier expression or `${expression}` to insert any other x2c expression.

```
int radius = 3;

List named = %(radius);      // one Atom whose text is "radius"
Array named_too = %[radius]; // the same Atom in an Array
Array valued = %[$radius];   // one element holding the number 3
List inserted = %($radius);

printf("%s %s %s %s\n", named.repr(), named_too.repr(),
       valued.repr(), inserted.repr());

(radius) [ <radius> ] [ 3 ] (3)
```

These forms quote symbolic data such as ASTs, patterns, and configuration trees, where the names are the content. A short `Atom` uses the compact `Symbol` representation; a longer one retains its exact spelling in an `<lsym>`. To put an x2c value in that quoted content, unquote it with `$name` or `${expression}`.

Bare `(...)`, `[...]`, and `{...}` nest `List`, `Array`, and `Map` data inside any of these collection forms. Bare `"..."` is an x2c `String`, with the same interpolation and escape rules as `%"..."`. `${"..."}` instead inserts an ordinary C string expression. Arrays and Maps have no splice form.

Inserting and splicing into List literals

`%()` supports four insertion forms:

- `$name` inserts one value named by an identifier.
- `${expression}` inserts one value from an arbitrary expression.
- `@name` splices a `List`'s elements into the surrounding `List`.
- `@{expression}` splices the `List` an expression produces.

```
String name = "quartz";
List tail = %(3 4);

List one = %(mineral $name);
List computed = %(len ${name.len()});
List spliced = %(1 2 @tail);
List reversed = %(1 2 @{tail.reverse()});

printf("%s\n%s\n%s\n%s\n",
       one.repr(), computed.repr(), spliced.repr(), reversed.repr());

(mineral "quartz")
(len 6)
(1 2 3 4)
(1 2 4 3)
```

`$` leaves `List` grammar, parses and evaluates one x2c expression, converts the result to one `List` value, and returns to `List` grammar. `@` makes the same crossing and splices the evaluated `List`'s elements instead of inserting one value. The identifier forms are the short versions of the braced expression forms.

Lisp reader prefixes in List literals

A `List` passed directly to `Lisp.eval` can use the familiar reader prefixes instead of spelling out their names:

```
lisp.set_global("tail", %(3 4));
List result = lisp.eval(%(`(1 ,(+ 1 1) ,@tail)));
```

That literal constructs the same data as `%(quasiquote (1 (unquote (+ 1 1)) (unquote-splicing tail)))` and evaluates to `%(1 2 3 4)`. `'` is `quote` the same way, so `%('foo)` is `%(quote foo)`. A `List` literal reads these prefixes as Lisp source does, and evaluation gives each form its meaning. `$` and `@` are x2c interpolation and splicing, performed while constructing the `List`.

Braces delimit the x2c expression being unquoted. Nested calls, indexing, assignments, conditional expressions, and other parentheses stay part of that expression:

```
String name = "quartz";
String source = "feldspar";
int count = 0;
List called = %(result ${name.upper().len()});
List assigned = %(result ${count = source.len()});
```

Inside `%()`, `$name(...)` reads differently. `$name` inserts one value, and the following `(...)` is a nested symbolic `List`. Write `${name(...)}` when the unquoted expression is a runtime call. Inside `%""`, parentheses after `$name` are text, so the same runtime call is again `${name(...)}`.

Nested `(...)` in `%()` stays quoted `List` data. Bare `"..."`, `[...]`, and `{...}` switch to `String`, `Array`, and `Map` literals. All four retain quoted data syntax, and `String` takes `$` unquote the way `%"..."` does:

```
String source = "quartz";
List shapes = %(
  (nested symbolic list)
  "len=${source.len()}"
  [${source.len()}, 2]
  {name: $source}
  ${source.upper()}
);
```

Do not repeat the `%` sigil on those. A bare `%` inside `%()` is an `Atom`, so `%(k %"txt")` is the three elements `k`, `%`, and `"txt"`. Write `%(k "txt")`.

Atom spellings inside Lists

A bare name in `%()` is an `Atom` and preserves its exact bytes, however long. The `Atom` is a compact `Symbol` when the encoding reproduces those bytes, and the private `<lsym>` representation otherwise. Do not write `<ready>` inside a `List`. `ready` produces the same compact value, and a long bare name keeps its full spelling.

A C preprocessor macro name is a bare name too. The preprocessor does not expand names inside a literal, so after `#define ROWS 64` the List `%(ROWS 4)` is `(ROWS 4)`, and reading its first element as a number yields the `Symbol`'s numeric value rather than 64. Unquote a typed value, `%(${(long) ROWS} 4)`, to insert the macro's value; the bare `$ROWS` has no x2c type either, because the preprocessor's text never reaches the type checker. The compiler warns when a bare name in a literal matches an object-like `#define` it has already passed in the same unit; macros from included C headers are not seen and get no warning.

Angle brackets quote text that `List` syntax would read as another kind of value. `1` is an integer, `<1>` is the `Symbol` `1`, and `<"a b">` quotes a `Symbol` containing a space. The escaped bare `Atom` spellings `\x31` and `a\ b` produce those same `Symbol`s. An angle spelling that does not fit the compact encoding is rejected instead of becoming an `<lsym>`. `<"">` is the empty `Symbol`, which has no bare `Atom` spelling.

```
List names = %(ready VeryLongExactName 1 <1>);
printf("%s %s\n", names.getindex(0).tag().str(),
                  names.getindex(1).tag().str());
```

```
symbol lsym
```

Symbols and Atoms covers the two representations and when to use each.

Ordered Symbol sets

`%<<...>>` declares a closed, immutable set of compact `Symbols`. Its source order is also its dense numeric order, starting at zero:

```
SymbolSet colors = %<<red green blue>>;

printf("%d %d %d\n",
       colors.index(<green>),
       colors.contains(<blue>),
       colors.index(<orange>));

1 1 -1
```

Bare entries are `Symbols`, so write `%<<red green blue>>` rather than `%<<<red> <green> <blue>>>`. Angle brackets still spell a `Symbol` explicitly. A member whose spelling is not a bare word is quoted, and the quotes may hold the `>>` that would otherwise end the literal:

```
SymbolSet assignments = %<<"+=" "<<=" ">>=">>;
```

Interpolation and splicing are not supported. Every member must be known when the file is compiled. Two spellings that encode as the same `Symbol` are a compile-time duplicate error.

`index` maps a `Symbol` to its source-order integer and returns `-1` for a nonmember. `contains` is the corresponding membership test. `getindex` maps an integer back to its `Symbol`, accepts negative indexes, and returns zero when out of range. A `SymbolSet` also supports `foreach(Symbol value, colors)` in source order. `<blue> in colors` is the operator spelling of that test: a `SymbolSet` adopts the one-member `Contains(T)` protocol, so the `in` operator reaches it without the set becoming a boxed `Var` container.

The compiler generates a perfect hash and the ordered `Symbol` table as one read-only byte string. A lookup computes one candidate index and checks the `Symbol` stored there. It does not allocate, initialize a runtime table, or search the set. The byte string has static lifetime, so a literal is valid in file-static declarations and public inline code.

String interpolation

`%"..."` quotes text. `$name` and `${expression}` unquote one value when its type converts to `String`.

```
String who = "world";
String greeting = %"hello, $who";
String shouted = %"${who.upper()}!";
printf("%s %s\n", greeting, shouted);

hello, world WORLD!
```

Supported native numeric expressions interpolate directly:

```
String who = "world";
printf("%s\n", %"$who has ${who.len()} bytes");

world has 5 bytes
```

An already-boxed `Var` interpolates through its `String` conversion, the same text `str()` returns. Extract the native value to interpolate it as that type instead:

```
Var answer = 42;
printf("%s\n", %"answer=${answer.integer()}");
printf("%s\n", %"boxed=${answer}");

answer=42
boxed=42
```

`${...}` unquotes one x2c expression inside a quoted `List`, `String`, `Array`, or `Map`. In ordinary x2c code, `$(...)` enters compile-time Lisp. Nesting them inserts a compile-time Lisp result into a quoted collection:

```
List generated = %(${$(+ 40 2)});
Array generated_array = %[${$(+ 40 2)}];
```

The outer `${` unquotes from `List` grammar into x2c. The inner `$(` is read in x2c and enters compile-time Lisp. `$macro(...)` is a macro invocation in x2c. None of these parser meanings change Lisp `quote` or `quasiquote`. `%!` is still the lambda-literal prefix, and `%` between two operands is still modulo.

Percent strings are byte strings. They may span physical lines, and their `\x`, `\u`, and `\U` escapes consume one or two hexadecimal digits instead of C's fixed escape widths. C `"..."` literals keep C rules. The escape table is in the language reference.

File-static literals

A file-static x2c object can keep its literal beside its declaration:

```
static String endpoint = %"$scheme://example.test";
static String scheme = "https";
static List labels = %(service $scheme $endpoint);
static Array routes = [endpoint];
static Map route_by_name = %{default: $endpoint};
static Var boxed_labels = %(boxed $scheme);
```

The compiler builds these values once in the translation unit's guarded initializer. Cached immutable pieces are ready before the assignments, and referenced file-static objects are assigned before their dependents, so a forward reference such as `endpoint` to `scheme` is valid. A referenced object without `static`, or a cycle among the declarations, is diagnosed at compile time.

Immutable versus mutable

List and String are canonical

`cons` canonicalizes `Lists`: constructing the same head and tail twice returns the *same cell*. `String.intern` canonicalizes non-empty `Strings` the same way. Equal content means an identical pointer, and equality is a pointer compare.

```
List literal = %(1 2 3);
List built = cons(1, cons(2, cons(3, NULL)));
printf("same cell: %d\n", literal == built);
printf("nil is the null pointer: %d\n", %() == NULL);

same cell: 1
nil is the null pointer: 1
```

`List` and `String` operations do not mutate their input. They return new canonical structure, sharing whatever they can:

```
List x = %(1 2 3);
List longer = x.append(%(4));
printf("x=%s longer=%s\n", x.repr(), longer.repr());

x=(1 2 3) longer=(1 2 3 4)
```

You can hand a `List` or `String` to any function without copying it, store it in several places, and use it as a `Map` key. Nothing can change it under you.

Array and Map are objects

`Array.new`, `Map.new`, `[]`, and a `{}` whose destination is a collection each produce a fresh allocated object with its own identity. Two separately written empty literals are two different `Arrays`. Assigning one to another variable aliases it.

```
Array a = [1, 2, 3];
Array alias = a;
Array copy = a.copy();
alias.push(4);
printf("a=%s alias=%s copy=%s\n", a.repr(), alias.repr(), copy.repr());
printf("distinct empty literals: %d\n", [] === []);

a=[ 1, 2, 3, 4 ] alias=[ 1, 2, 3, 4 ] copy=[ 1, 2, 3 ]
distinct empty literals: 0
```

Mutating operations mutate in place and return the same object, so `array.sort()` is not a sorted copy:

```
Array numbers = [5, 3, 9, 1];
Array sorted = numbers.sort();
printf("numbers=%s  same object=%d\n", numbers.repr(), sorted === numbers);

numbers=[ 1, 3, 5, 9 ]  same object=1
```

`Array.reverse` behaves the same way. `Array.copy`, `Array.concat`, `Array.map`, and `Array.getslice` return new `Array`s. The generated Array reference says which operations mutate and what identity each returns.

Empty values are not all alike

```
String empty = "";
List nil = %();
Array none = [];
Map blank = {};
printf("empty String is null: %d  nil is null: %d\n",
       empty == NULL, nil == NULL);
printf("empty Array allocated: %d  empty Map allocated: %d\n",
       none != NULL, blank != NULL);

empty String is null: 1  nil is null: 1
empty Array allocated: 1  empty Map allocated: 1
```

The empty `String` and the empty `List` *are* the null pointer. An empty `Array` or `Map` is an allocated object. `Null` is not a valid empty `Array` or `Map` value, and boxing a null pointer with either tag breaks an invariant.

Emptiness tests follow from that. A bare condition on `String`, `List`, `Array`, or `Map` uses its `truth` protocol member, so every empty collection is false even when an empty `Array` or `Map` owns storage. Compare the pointer when you are asking about the allocation instead of the content:

```
Array none = [];
Var boxed = none;
printf("pointer test: %d   boxed truthiness: %d\n",
       none != NULL, boxed ? 1 : 0);
printf("static truthiness: %d   len test: %d\n",
       none ? 1 : 0, none.len() == 0);

pointer test: 1   boxed truthiness: 0
static truthiness: 0   len test: 1
```

Equality and identity are explicit

```
Array left = [1, 2];
Array right = [1, 2];
printf("array structural: %d  identity: %d\n",
       left == right, left === right);

Map a = {k: 1};
Map b = {k: 1};
printf("map structural: %d  identity: %d\n", a == b, a === b);

array structural: 1  identity: 0
map structural: 1  identity: 0
```

`Array.equal` and `Map.equal` compare contents, and ==/`!=` call them. They compare elements and values with ==, so two numbers compare as their native values do in C. ===/`!==` are the identity operators for typed objects

and boxed `Var` values; two numbers are identical when they have the same type and the same bits. Boxed `Array` and `Map` hashing is identity-based, so they make poor `Map` keys:

```
Map by_list = {};
by_list[%(1 2)] = "found";
printf("List key, rebuilt: %s\n", by_list[cons(1, cons(2, NULL))].repr());

Map by_array = {};
by_array[[1, 2]] = "found";
printf("Array key, equal contents: %s\n", by_array[[1, 2]].repr());

List key, rebuilt: "found"
Array key, equal contents: void
```

Key your `Maps` with `Strings`, `Symbols`, `Atoms`, or numbers. Those compare by content. Keys compare by `Var.equal`, which agrees with `Var.hash`, so a number key also keeps its type: `3` and `3L` are two keys although `3 == 3L`.

```
Map by_number = {};
by_number[3] = "int";
by_number[3L] = "long";
printf("keys: %d  [3] == [3L]: %d\n", by_number.len(), [3] == [3L]);

keys: 2  [3] == [3L]: 1
```

A `List` key compares by content only as far as its elements do. `List.equal` and `List.hash` read the head's exact bits and the tail's identity, which is a constant-time test that works because cells are canonical. An element that lives inside the `Var` word compares by content: a small number, a `Symbol`, an `Atom`, a canonical `String`, or a nested `List`. An element that the `Var` word only points at compares by address: a wide number boxed in a `Scope` (`long`, `unsigned long`, `long long`, `unsigned long long`, or `long double`), an `Array`, or a `Map`. Two `Lists` built from one `long` value hold two boxes, so they are different keys:

```
Var first = Var.box_long(5l);
Var second = Var.box_long(5l);
Map counts = {};
counts[%($first)] = "boxed";

printf("same number, rebuilt key: %s\n", counts[%($second)].repr());
printf("small number key: %s\n", %(5) == %(5) ? "matches" : "differs");

same number, rebuilt key: void
small number key: matches
```

Key on the number itself, or on a `String` or `Symbol` spelling of it, when a value that wide has to be part of the key. `==` on two `Lists` is the same test, boxed in a `Var` or not, so it keeps numeric types apart even though `==` on two numbers follows C: `%(1 2) == %(1L 2L)` is 0, while `(Var) 1 == (Var) 1L` is 1. Pattern literals compare numbers the same way.

Working with Lists

A `List` is a canonical cons chain, as in Lisp. `car` gives the head, `cdr` gives the tail, `nil` is `NULL`, and the composed accessors `cadr`, `caddr`, `cddr`, and the rest are available.

```
List form = %((op add) (args 1 2) (flag on));
printf("head: %s\n", form.car().repr());
printf("second: %s\n", form.cadr().repr());
printf("args value: %s\n", form.assoc(%(args).car()).repr());
printf("length: %d\n", form.len());

head: (op add)
second: (args 1 2)
args value: 1
length: 3
```

`List.assoc` treats the `List` as an association list of `(key value ...)` pairs and returns the `cadr` of the first matching pair. `List.get` dispatches on the key: integers index, anything else does an `assoc` lookup.

The functional `List` operations take `Func`. A fixed nonvariadic function or function pointer converts automatically, and you can pass a capturing lambda directly. Each element is supplied as a value. A callback with a reference parameter is rejected when the first element is invoked, and it never receives an alias into `List` storage. Empty inputs do not invoke or arity-check the callback. The language reference has the conversion and lifetime rules.

```
List numbers = %(1 2 3 4);
List doubled = numbers.map(%!(x) => x * 2);
List big = numbers.filter(%!(x) => x > 2);
Var total = numbers.foldl(0, %!(a, b) => a + b);
printf("%s %s %s\n", doubled.repr(), big.repr(), total.repr());
```

```
(2 4 6 8) (3 4) 10
```

For `map`, `filter`, and `foldl`, a callback is required when an element needs an invocation. A null callback then raises `<bad-arg>` through `Func`. Empty inputs make no call. A fold with a `void` seed also makes no call for a singleton: its only element becomes the result. This rule applies to these operations on `List`, `Array`, `String`, and `Iter` where provided. Lisp nil callbacks follow the same rule. Other operations, including `zip_with`, queries, `map2`, and `scan`, retain their own callback contracts.

Transformations and folds keep the callback's `Var` result. The predicates `List.filter`, `List.find`, `List.any`, and `List.all` apply `Var` truthiness, so they accept more than an `int` result. Direct functions, function-pointer values, and capturing lambdas all use the same methods.

The other operations are `reverse`, `concat`, `last`, `index`, `contains`, `sort`, `unique`, `head`, `tail`, `flatten`, `flatten_all`, `sublis`, `array`, and `str` / `repr`. `List.len` walks the chain, so it is O(n). Do not call it inside a loop over the same `List`.

You cannot write through a `List` index; see Indexing and slicing below. For scratch state you mean to mutate, use an `Array`.

Pattern Matching takes `Lists` apart using the same `%()` notation, with `?binder` and `*binder` holes naming the parts.

Typed cons chains

`typed-list.x` declares seven `Lists` whose car always carries one known tag: `ListChar`, `ListShort`, `ListInt`, `ListFloat`, `ListDbl`, `ListString`, and `ListSymbol`. They are typedefs of `List` sharing its canonical pool, so a typed chain and the plain literal that spells it are the same cells, and `%()` builds one when that is the declared type. An element of the wrong tag raises `<no-convert>`.

They give you typed methods, not more speed. `cons` is still a pool probe. Use `typed-array.x` for indexing or bulk numeric storage, and these for sharing, canonical identity, or a chain.

```
#include "typed-list.x"

int main(void) {
  ListInt values = %(4 5 6);
  printf("car=%d last=%d\n", values.car(), values.last());

  int total = 0;
  for (ListInt cursor = values; cursor; cursor = cursor.cdr())
    total += cursor.car();
  printf("total=%d\n", total);

  ListString words = %("alpha" "beta");
  List widened = values;
  printf("first=%s widened=%s\n", words.car(), widened.str());
  return 0;
}
```

```
car=4 last=6
total=15
first=alpha widened=( 4 5 6 )
```

`car` and `last` return the element type. The structural methods inherited from `List` return `Self`, so `cdr`, `reverse`, `sort`, `append`, `head`, `tail`, and the slices keep the typed spelling without generated wrappers, and a walk never

reconverts. `car` of `nil` is the element's zero, not `void`. There is no `long` family, because `Var.box_long` allocates a `Scope`-owned box: a cell would outlive its car and every `cons` would miss the pool.

These types inherit List bracket reads: `values[0]` returns a `Var`. Use `car` for the family's native element type and `nth_cdr` for a tail. Predicate-only `filter` keeps the typed spelling, since it cannot replace an element. Operations such as `map`, whose callbacks can produce new values, need a widening assignment to `List`, as above.

Working with Strings

A canonical `String` is an interned `char *`. It is NUL-terminated, so it passes straight to `printf("%s", ...)` and to any C function that takes a string. Every operation works on bytes: lengths, indices, slices, padding, case conversion, and iteration. `String` does not decode Unicode characters.

```
String line = "  name = x2c  ";
String trimmed = line.strip(NULL);
List halves = trimmed.partition(" = ");
printf("[%s] -> %s\n", trimmed, halves.repr());
printf("upper=%s find=%d startswith=%d\n",
       trimmed.upper(), trimmed.find("="), trimmed.startswith("name"));

[name = x2c] -> ("name" " = " "x2c")
upper=NAME = X2C find=5 startswith=1
```

`strip`, `lstrip`, and `rstrip` take a C string of characters to remove, or `NULL` for the default whitespace set. `partition` returns a three-element `List` of before/separator/after. `split` and `split_lines` return `Lists` of `Strings`. Use `words()`, `lines()`, or `splits(sep)` when the fields should stream directly through `foreach` instead. `String.join` is the inverse of eager `split`, with the separator as the receiver:

```
List parts = "a:b:c".split(":");
printf("%s -> %s\n", parts.repr(), "-".join(parts));

("a" "b" "c") -> a-b-c
```

Numeric parsing reports success separately from the result:

```
long value;
if ("42".try_long(&value)) printf("parsed %ld\n", value);
if (!"abc".try_long(&value)) printf("rejected abc\n");
if (!"4x".try_long(&value)) printf("rejected 4x\n");

parsed 42
rejected abc
rejected 4x
```

`try_long` requires the whole string to be the number. Leading and trailing whitespace are allowed, and any other unconsumed character fails, so `4x` is rejected instead of read as 4. It accepts a sign and recognizes `0x`, `0o`, and `0b` radix prefixes as well as C's leading-zero octal. `String.try_double` is the floating counterpart on the same terms.

Concatenation with `+` produces a canonical `String`. Comparing a `String` with a C string literal using `==` or `!=` promotes the literal to `String` and compares their contents:

```
String built = "hello" + ", world";
printf("%s  %d\n", built, built == "hello, world");

hello, world  1
```

The literal may appear on either side and may be parenthesized. A `char *` variable or another C pointer expression keeps native pointer comparison; convert it to `String` explicitly when you want content comparison. The identity operators `===` and `!==` also keep pointer comparison and do not promote literals.

Building text

Canonical `Strings` are immutable, so repeated `+` in a loop allocates and interns at every step. There are two better tools.

`String.malloc` creates a transient mutable buffer in the active canonical-value pool. It becomes a canonical `String` only when `String.intern_free` finalizes it. Bracket assignment on a `String` is rejected, because writing through a canonical `String` would corrupt every equal `String` that shares its storage. Bind the buffer to a `char *` to write it with native indexing, then finalize it with `String.intern_free`:

The byte count includes room for the terminating NUL, and nothing writes that NUL for you. `String.intern_free` measures the buffer with `strlen`, so an unterminated buffer is read past its end.

```
String buf = String.malloc(6);
char *bytes = buf;
for (int i = 0; i < 5; i++) bytes[i] = 'a' + i;
bytes[5] = '\0';
String canonical = buf.intern_free();
printf("%s %d\n", canonical, canonical == "abcde");
```

```
abcde 1
```

Write bytes that way only into such a buffer. On an interned `String` it writes through shared canonical storage. To change one byte of a canonical `String`, use the copy-producing `String.withindex`:

```
String word = "hello";
String capital = word.withindex(0, 'H');
printf("%s %s\n", word, capital);
```

```
hello Hello
```

For anything longer, use `Buffer`. It grows, it tracks indentation, and a `String` destination calls `Buffer.str` for the finished canonical `String`.

```
Buffer out = Buffer.new(0);
foreach(Var item, [1, 2, 3]) {
  out.write("item ");
  out.write(item.str());
  out.newline();
}
String report = out;
out.free();
printf("%s", report);
```

```
item 1
item 2
item 3
```

`String.free` releases a transient buffer early and does nothing to a canonical process-lifetime `String`. See Scopes and Lifetime for what owns what.

Working with Arrays

An `Array` is a growable contiguous sequence of `Var`. Use it for indexed mutation, a stack, a queue, or scratch storage. `Array.len` is O(1) and returns a `size_t`.

```
Array stack = [];
stack.push(1);
stack.push(2);
stack.push(3);
Var top = stack.take_last();
stack[0] = 100;
printf("%s top=%s len=%zu\n", stack.repr(), top.repr(), stack.len());
```

```
[ 100, 2 ] top=3 len=2
```

The positional operations are `push`, `take_last`, `shift`, `unshift`, `insert`, `remove`, and `splice`; searching is `find` / `indexof`, `contains`, and `count`; and there are `map`, `map2`, `foldl`, `sort`, `reverse`, `concat`, `copy`, and `join`.

Use `sort_by` to order by a computed key, or `sort_with` for a comparator. Both preserve the input order of ties. Arrays change in place and return the same object; Lists return a sorted copy. The existing `sort()` still uses `Var.compare` and does not promise stable ties.

```
Array rows = [];
rows.push(%(2 "first"));
rows.push(%(1 "low"));
rows.push(%(2 "second"));
rows.sort_by(%!(List row) => row.car());
// (1 "low"), (2 "first"), (2 "second")
List descending = %(3 1 2).sort_with(
  %!(int left, int right) => right - left);
```

`sort_by` evaluates its key once per element, front to back, and compares the keys with `Var.compare`. `sort_with` passes two values to its callback and converts the result to `int`: negative orders the first value before the second, zero ties, and positive orders it after. Comparators must give a consistent ordering. Callbacks are borrowed, run synchronously, and must not mutate the Array being sorted; they may sort another container. If a callback or comparison fails, the Array keeps its original element order. Callback side effects are not undone. Empty inputs call neither callback; a singleton calls its key once but never calls a comparator.

`Array.map`, `Array.map2`, and `Array.foldl` accept the same `Func` values as the `List` operations and pass elements by value. `String.filter` and `String.map` do the same for bytes. Each byte is boxed from `char`, predicates use `Var` truthiness, and `String.map` converts the result to `int` before rejecting NUL and storing the byte.

The shared `Block` protocol also supplies `pop`, `truncate`, and `free`. `pop` discards the final element and returns nothing; use `take_last` when the removed value is needed. `truncate` shortens the `Array` without changing its identity, and `free` releases its storage before the owning scope ends.

```
Array items = ["a", "c"];
items.insert(1, "b");
items.unshift("start");
Var gone = items.remove(0);
printf("%s removed=%s\n", items.repr(), gone.repr());
printf("find b at %d, joined %s\n", items.find("b"), items.join("-"));
```

```
[ "a", "b", "c" ] removed="start"
find b at 1, joined a-b-c
```

`Array` also carries binary min-heap helpers, which turn it into a priority queue without a second data structure:

```
Array heap = [];
heap.heap_push(30);
heap.heap_push(10);
heap.heap_push(20);
printf("smallest first: %s\n", heap.heap_pop().repr());
```

```
smallest first: 10
```

`Array.push` rejects `void`, and so does the counted construction used by non-empty Array literals. Raw `Null` is `Array` data. An out-of-range read returns `void`. `Array` has no status-bearing read.

Packed typed Arrays

`typed-array.x` declares seven `Arrays` whose elements use a concrete field rather than `Var`: `ArrayChar`, `ArrayShort`, `ArrayInt`, `ArrayLong`, `ArrayFloat`, and `ArrayDbl` hold native scalars, and `ArrayString` holds canonical `String` pointers. Each stores its elements packed, at the width of the C type, and carries the same positional, slicing, searching, and `Block` operations that `Array` does. An Array literal builds one when that is the declared type, so `ArrayInt counts = [1, 2, 3];` packs the literal; an element that is not a number raises `<no-convert>`.

The bracket is where they differ. `numbers[i]` is native indexing on a concrete element pointer: no null test, no bounds test, and no negative-index normalization. An index outside the elements is undefined, as it is for the equivalent C pointer. The generated code is the same indexed load or store that C would use.

```
#include "typed-array.x"

int main(void) {
  ArrayDbl samples = ArrayDbl.new();
  for (int i = 0; i < 4; i++) samples.push(i * 0.5);
  samples[3] = samples[3] * 2.0;
```

```
  double total = 0.0;
  for (int i = 0, n = (int) samples.len(); i < n; i++) total += samples[i];
  printf("len=%zu total=%.2f\n", samples.len(), total);
  return 0;
}
```

```
len=4 total=4.50
```

Bound the loop with `len`, as above. When the index may be out of range, `try_get` is the checked read. It returns zero and leaves its output untouched instead of reading past the end, and it normalizes a negative index against the length the way `Array` does.

```
#include "typed-array.x"

int main(void) {
  ArrayInt values = ArrayInt.new();
  values.push(10);
  values.push(20);

  int found = -1;
  int present = values.try_get(1, found);
  printf("%d %d\n", present, found);
  present = values.try_get(5, found);
  printf("%d %d\n", present, found);
  present = values.try_get(-1, found);
  printf("%d %d\n", present, found);
  return 0;
}
```

```
1 20
0 20
1 20
```

The other members keep their checks. `push`, `insert`, `remove`, `take_last`, `shift`, and the slicing operations raise on a bad argument or an empty receiver, and the slice bounds normalize negatives. Only the bracket is raw. A typed `Array` boxes into a `Var` and iterates, so `foreach(Var item, numbers)` and dynamic indexing through a `Var` receiver both work. The dynamic bracket reaches the same raw member.

Packed `Arrays` cross a `Context` or `Thread` boundary like other values; Contexts and Threads gives the export rules.

Compound update is native as well. `values[i] += 1`, `values[i] <<= 2`, and `values[i]++` compute in the element type and compile to the one instruction the operator names. Overflow wraps, a narrow element truncates a result too wide for it, and a signed right shift keeps the sign, as C does. Three failures are kept, since they turn a hardware trap into something you can catch: integer `/= 0` and `%= 0` raise `<div-zero>`, a shift count outside the element's width raises `<bad-shift>`, and `%` or a bitwise operator on a float element raises `<bad-types>`. Each raises before storing, so a failed update leaves the element as it was.

An `ArrayChar` *store* loop has an extra cost: it cannot keep the element base in a register, because a byte store may alias any object, including that base. Wider element types and reads are unaffected. For a loop that is nothing but byte stores, bind `array.bytes` to a `char *` and write through that.

Working with Maps

A `Map` is a mutable Robin Hood hash table from `Var` keys to `Var` values. Keys and values are both `Var`, so a `Map` can be heterogeneous.

```
Map counts = {};
List words = %("to" "be" "or" "not" "to" "be");
foreach(Var w, words)
  counts[w] += 1;
printf("%s\n", counts.repr());
```

```
{ "be": 2, "to": 2, "not": 1, "or": 1 }
```

Numeric `+=` treats an absent destination as zero, inserts the right-hand side, and returns the stored value, so `counts[key] += 1` counts in one call. The first value takes the right-hand side's numeric tag, and later updates keep that stored tag.

Only numeric `+=` initializes. A missing or `void` right-hand side, a nonnumeric right-hand side, and every other compound operator require an existing destination. Inserting the first value is a structural `Map` change and may invalidate an outstanding traversal.

`getdefault` reads with a fallback and inserts nothing. `setdefault` inserts the fallback and returns it. `contains` reports presence whatever the stored value is, and `key in map` is the operator spelling of that same test, as it is for a `List`, an `Array`, and a `String`. `copy` and `merge` combine `Maps`.

Nested literals work, and `Symbols` make readable keys for fixed schemas:

```
Map config = {
  name: "x2c",
  targets: ["c"],
  limits: {depth: 8}
};
printf("%s\n", config.repr());
printf("targets is an Array: %d\n", config[<targets>] is Array);

{ <targets>: [ "c" ], <limits>: { <depth>: 8 }, <name>: "x2c" }
targets is an Array: 1
```

Map lookup, deletion, and traversal

`Map.get` and bracket reads return `void` for a missing key. The `try_` operations report presence separately from the value and leave the output pointer untouched on failure.

```
Map ages = {"ada": 36, "grace": 45};
ages["alan"] = 41;

Var found;
if (ages.try_get("ada", found))
  printf("ada is %s\n", found.repr());
if (!ages.try_get("nobody", found))
  printf("nobody is absent\n");

printf("bracket read of a missing key: %s\n", ages["nobody"].repr());

Var removed;
if (ages.try_del("alan", removed))
  printf("removed alan=%s remaining=%u\n", removed.repr(), ages.len());

ada is 36
nobody is absent
bracket read of a missing key: void
removed alan=41 remaining=2
```

`Map.try_next` performs the traversal. It takes a cursor you own, starting at zero, and reports status separately from the key and value it writes, so it can yield an entry whose key is raw `Null` without that looking like exhaustion.

```
Map ages = {"ada": 36, "grace": 45};
unsigned cursor = 0;
Var key, value;
while (ages.try_next(cursor, key, value))
  printf("%s -> %s\n", key, value.repr());

grace -> 45
ada -> 36
```

That is the table's order, and it moves when the hash changes. Sort what you print when the order matters.

`Map.get`, `Map.del`, and the bracket forms are wrappers over these calls. `Map.set` and bracket writes are fail-fast and reject a `void` key or value.

Inserting or deleting invalidates outstanding cursors and iterators. Finish traversing, or collect what you need first, before changing the table's structure. Missing values and exhaustion are reported differently across the library; check each operation's documented result before assuming `void` means absent.

Iterating a Map

A `foreach` over a `Map` yields each value. Two binders yield the key and the value. With native loop types, the loop does not allocate:

```
Map ages = {"ada": 36, "grace": 45};
foreach(Var (key, value), ages)
  printf("%s -> %s\n", key, value.repr());

grace -> 45
ada -> 36
```

Iteration covers the `Map` iterators and traversal order.

Packed typed Maps

`typed-map.x` declares four `Maps` whose keys and values use concrete fields rather than `Var`. `MapIntInt` and `MapLongDouble` store native numeric scalars; `MapStringString` stores canonical `String` pointers, and `MapStringInt` stores a canonical `String` key against a native `int`. Each carries the same lookup, defaulting, update, deletion, traversal, `copy`, and `merge` operations that `Map` does. A Map literal builds one when that is the declared type, converting every key and value to the field type; an entry that cannot be converted raises `<no-convert>`.

```
#include "typed-map.x"

int main(void) {
  MapIntInt counts = {1: 2, 3: 4};
  counts.updateindex(1, <+>, 10);
  counts.set(5, 25);

  MapStringString label = {"name": "x"};
  label.updateindex("name", <+>, "2c");

  int out = 0;
  printf("one=%d present=%d\n", counts.get(1), counts.try_get(5, out));

  unsigned cursor = 0;
  int key, value, total = 0;
  while (counts.try_next(cursor, key, value)) total += value;
  printf("entries=%u total=%d\n", counts.len(), total);
  printf("label=%s\n", label.get("name"));
  return 0;
}

one=12 present=1
entries=3 total=41
label=x2c
```

These types adopt `protocol Var`, so their `getindex`, `setindex`, `updateindex`, and `postfixindex` members also provide bracket reads, writes, compound updates, and numeric postfix updates. The numeric families update with native arithmetic, and so does `MapStringInt`'s `int` value. For `MapStringString`, `<+>` concatenates, and both postfix operators raise `<bad-op>`. In all four families `<+>` initializes an absent key from the right-hand side, and every other update requires an existing one.

Packed `Maps` cross a `Context` or `Thread` boundary under the same export rules.

The element type has no `void`, so absence is reported another way. `get`, `getindex`, `del`, `postfixindex`, and the updates that require a destination raise `<bad-arg>` on a missing key where `Map` would return `void`. Use `try_get`, `try_del`, and `try_next` for status, or `getdefault` for a fallback.

Indexing and slicing

Native C arrays and pointers keep C indexing. `Array`, `List`, `String`, and `Map` also define `value[...]`.

A typedef alias such as `typedef List Row;` keeps its base type's bracket reads.

```
Array digits = [0, 1, 2, 3, 4, 5];
List letters = %(a b c d);
String text = "sphinx";

printf("%s %s %s\n",
       digits[0].repr(), digits[-1].repr(), letters[1]);
printf("%c %c\n", text[0], text[-1]);
printf("out of range: %s\n", digits[99].repr());
```

```
0 5 b
s x
out of range: void
```

`Array`, `List`, and `String` accept negative indices, counting from the end. An out-of-range `Array` or `List` read returns `void`, and a missing `Map` key returns `void` through brackets. An out-of-range `String` read returns `-1`; a successful `String` read returns one byte as an `int` between 0 and 255, so `-1` means out of range on every platform and a high byte is never negative.

The packed numeric `Arrays` are the exception on both counts. Their bracket is raw native indexing, with no bounds test and no negative-index rule. See Packed typed Arrays.

Writing is where the immutable/mutable split shows up in the syntax:

- `array[i] = value` and `map[key] = value` are supported.
- `list[i] = value` is rejected by the compiler.
- `string[i] = ch` is rejected, because a canonical `String` is immutable. Use the copy-producing `String.withindex`, or bind a transient `String.malloc` buffer to a `char *` and write through that pointer.

```
List letters = %(a b c);
letters[0] = "z";   // xform: type ( List ) does not support bracket
                    // assignment
```

Slicing is `value[start:stop:step]`, available on `Array`, `List`, and `String`. Any of the three parts may be omitted. One shared runtime call normalizes negative bounds and negative steps, so the rules are identical across the three types. A zero step is invalid.

A slice returns the same type as its receiver. Slicing an `Array` gives a new `Array`, a `List` gives a new `List`, and a `String` gives a new `String`. The result is a fresh value, and slices do not alias.

```
Array digits = [0, 1, 2, 3, 4, 5];
Array first_three = digits[:3];
Array every_other = digits[::2];
Array backwards = digits[::-1];

List letters = %(a b c d);
List middle = letters[1:3];

String text = "sphinx";
String tail = text[2:];

printf("%s %s %s\n",
       first_three.repr(), every_other.repr(), backwards.repr());
printf("%s %s\n", middle.repr(), tail.repr());
```

```
[ 0, 1, 2 ] [ 0, 2, 4 ] [ 5, 4, 3, 2, 1, 0 ]
(b c) "hinx"
```

`Array.setslice` and `Array.remslice` read `start` and `end` the same way, so `array.setslice(start, end, values)` replaces exactly the elements `array[start:end]` selects and `array.remslice(start, end)` removes them. `Array.splice` is the other shape: it normalizes one position and counts forward from it. A reversed pair of region bounds is swapped.

```
Array digits = [0, 1, 2, 3, 4];
Array region = digits[1:-1];
digits.setslice(1, -1, [9]);

printf("%s %s\n", region.repr(), digits.repr());
```

```
[ 1, 2, 3, 4 ] [ 0, 9 ]
```

The reference gives the full indexing and slicing rules, including the `int` index domain.

Choosing between them

Choose by what the data is.

- `List`: symbolic and structural data, such as ASTs, patterns, s-expressions, association lists, and anything you match over or want canonical identity for. Also the return type when a function produces a persistent value.
- `Array`: indexed mutation, stacks, queues, priority queues, sorting, and scratch accumulation. Also the fast path when you need O(1) length and random access.
- `Map`: keyed lookup, counting, interning tables, and configuration keyed by `Symbol` or `String`.
- `String`: canonical text you compare, key on, or return.
- `Buffer`: building text before you have a finished `String`.
- `Symbol`: short immediate names, tags, and closed sets of status values.

Some guidelines:

- To build a sequence in a loop, accumulate into an `Array`, then call `Array.list()` if the caller wants a `List`. `List.map` and `List.sort` do the same internally.
- To look something up by name, use a `Map` keyed by `String`, `Symbol`, or `Atom`. Scan a `List` with `assoc` only when the `List` is the data.
- To compare two sequences cheaply, use `List`: equality is a pointer compare. `Array` equality walks the elements.
- To change an element in place, use an `Array`.
- To pass a sequence to code that must not change it, use a `List` or a `String`. No copy is needed.
- For a `Map` key, use a `String`, `Symbol`, `Atom`, number, or `List`. `Array` and `Map` hash by identity.
- When every element is the same C scalar and the loop is hot, use a typed family, preferably a packed `Array`, which has a raw bracket. Measure speed when it matters.

A small program that uses each type for what it is good at: a `String` split into words, a `Map` for counting, an `Array` for ranking, and `List` pairs to carry each entry.

```
String text = "the quick the lazy the end";

Map counts = {};
foreach(Var word, text.words())
  counts[word] += 1;

Array ranked = [];
foreach(Var (word, count), counts)
  ranked.push(%($count $word));
ranked.sort();
ranked.reverse();

foreach(List entry, ranked) {
  Var (count, word) = entry;
  printf("%s %s\n", count, word);
}
```

Where to look next

- The standard library overview links to the API reference for each collection.
- The language reference has the percent literal, indexing, and slicing rules.
- Iteration for `foreach` and lazy pipelines, Pattern Matching for taking `List`s apart, and Idioms for recommended usage.

Pattern Matching

A `match` statement takes a `List` apart by its shape. You write a pattern that looks like the data and name the pieces you need. The names become ordinary local variables. One pattern replaces a chain of `List_car`, `List_cdr`, length checks, and tag tests.

The `match` statement and the runtime matcher share one pattern language. The exhaustive rules are in the language reference.

The match statement

A `match` has a subject in parentheses and a sequence of arms. Each arm is `case pattern:` followed by one statement, and an optional final `default:` arm runs when nothing matched.

```
List reply = %(status ok 200);
match (reply) {
  case %(status ok ?code):
    printf("ok with code %d\n", code);
  case %(status ?state ?code):
    printf("%s with code %d\n", state, code);
  default:
    printf("not a status reply\n");
}
```

The subject is evaluated once. Arms are then tried in source order, and the first one that matches runs. The second arm above would also have matched, but it is never tried. If no arm matches and there is no `default`, the whole statement does nothing.

An arm's body is a single statement, so use braces when you want several. Ordinary structural patterns use `%(...)` `List` literals. Compile-time matching can also use Macro values to recognize the code a macro constructs.

For a single arm you may drop the outer braces, the same way you can with `if`:

```
List value = %(one);
match (value)
  case %(one):
    printf("no braces needed for a single arm\n");
```

Arm order is also cost order, so put cheap, common shapes first.

Typed captures and expression guards

Write `?(typename name)` to capture a value as a native typed local. The pattern tests the value's `Var` tag; a different tag fails the pattern without converting the value. The tag test precedes the native local declaration. The supported types are the same as for `value is typename`. The `?(` opener must be adjacent; `? (String text)` remains a wildcard followed by a sublist pattern.

Inside `?(`, `typename` uses ordinary source type syntax, so write `?(String text)` or `?(List payload)`. This is an exception to the pattern's bare-spelling rule. The compiler represents a named type such as `String` with the type List `%("String")`; `%(String)` instead contains an `Atom`. See type syntax in macro values.

Typedefs use their underlying `Var` tag. Since `Type` is a typedef for `List`, `?(Type type)` tests for a List payload. It does not check whether the List contains valid type syntax.

```
List reply = %(message "ready");
match (reply) {
  case %(message ?(String text)) if (text.len() > 3):
    printf("long message: %s\n", text.str());
  case %(message ?(String text)):
    printf("short message: %s\n", text.str());
}
```

An optional `if (expression)` before the colon runs after the pattern matches, with its captures in scope. A false guard tries the next arm; errors propagate normally. The guard uses ordinary expression truth rules.

A typed name has the same type throughout its arm, including other occurrences written as `?name` and occurrences in alternative patterns. Repeated names still require equal values. `!quote` remains opaque, and the

existing rule that each binder must be assigned in every alternative still applies. Explicit `(!is ?name type string)` keeps `name` as a `Var`.

The shorthand lowers to the existing `!is` pattern predicate and ordinary typed local declarations. An expression guard lowers to an ordinary `if`; both forms match exactly as the explicit patterns do.

The pattern vocabulary

Patterns are written in `List`-literal notation, which is described in strings, lists, arrays, and maps.

Literals

A bare spelling in a pattern is an exact `Atom` that must appear in that position. Strings compare by value. A number matches only a number of the same type and value, so `%(3)` matches `3` but not `3L`, `3.0`, or `3u`, and `%(?x ?x)` does not match `%(3 3L)`. This differs from == on two numbers, which follows C and calls `3` and `3L` equal. `$name` or `${expression}` unquotes a value to compare against by the same rule. A pattern can require a `match` with something computed at run time.

Wildcards

`?` matches exactly one element and discards it. `*` matches any number of elements and discards them.

Binders

`?name` matches one element and binds it; `*name` matches a run of elements and binds them as a `List`.

```
List command = %(move 10 20 fast quiet);
match (command) {
  case %(move ?x ?y *flags):
    printf("to %d,%d with %d flag(s)\n",
           x, y, flags.len());
}
match (command) {
  case %(move ? ? *): printf("some move\n");
}
```

A `?name` binder is a `Var`; a `*name` binder is a `List`. A `*name` takes the leftmost split that lets the rest of the pattern succeed, and matching a `*name` against typed `nil` binds an empty `List` rather than failing.

Repeating a binder name in one pattern means "these must be equal": `%(?x ?x)` matches `%(same same)` and rejects `%(other same)`.

Binder names follow the C identifier grammar, `[A-Za-z_][A-Za-z0-9_]*`. A sigil followed by anything else is a mistake, and the compiler reports it with a position. `%(node ?bad-name)` reports `invalid match binder name`.

Nested shapes

A parenthesized group inside a pattern is a sublist, so nesting states nested structure without any index arithmetic:

```
List tree = %(tree (node 3 4) leaf);
match (tree) {
  case %(tree (node ?a ?b) ?rest):
    printf("%d\n", a * b);
}
Symbol wanted = <leaf>;
match (tree) {
  case %(tree ? $wanted): printf("ends with the wanted tag\n");
}
```

Guards

Six guard operators turn a position in a pattern into a test. Each is written as a sublist headed by the operator. `(!or a b ...)` succeeds when any alternative matches, `(!and a b ...)` when all of them do, and `(!not a b ...)` when none of them do:

```
List reply = %(status created 201);
match (reply) {
  case %(status (!or ok created) ?code):
    printf("success %d\n", code);
  case %(status (!not ok) ?code):
    printf("failure %d\n", code);
}
match (%(color green)) {
  case %(color (!and (!not red) (!not blue))):
    printf("neither red nor blue\n");
}
```

`(!set PATTERN ...)` is membership: it succeeds when any member matches. `(!quote PATTERN)` compares the input to the pattern as literal data, with no binder interpretation inside it. `(!is ...)` is the type and category predicate: `(!is type string)` checks a `Var` tag, and `(!is atom)`, `(!is binder)`, `(!is op)`, `(!is var binder)`, and `(!is list binder)` check the named category.

Every guard except `!quote` accepts an optional leading binder, which captures the slice of input the guard checked. A leading binder needs at least one operand after it, so `(!not ?y)` tests `?y` rather than capturing it, and `(!set ?w a b)` captures the `a`-or-`b` membership:

```
match (%(node 7 8)) {
  case %(!set ?whole (node ?a ?b)):
    printf("%s holds %d and %d\n",
           whole, a, b);
}
match (%(key "abc")) {
  case %(key (!is type string)):
    printf("a String payload\n");
}
match (%(key 42)) {
  case %(key (!is ?found atom)):
    printf("an atom: %s\n", found);
}
match (%(tag (a b))) {
  case %(tag (!quote (a b))):
    printf("literally (a b)\n");
}
```

Runtime-built patterns may put an interned operator `Symbol` in the head position. `List.match(input, %($op ...))` applies the same operator semantics as its literal spelling, and the runtime operator suite checks `!or`, `!not`, `!quote`, `!is`, and nested dynamic forms. Source `match` arms should spell operators literally so the compiler can determine which named binders are available to the arm body.

The `?binder?`, `*binder?`, and `!op?` spellings are reserved in `!is` operand positions. They are not named binders and do not execute a predicate. Use the `(!is ...)` forms above.

Where binders live

Each arm declares its binders as local variables before its body is parsed. A `?name` has type `Var`, a `*name` has type `List`, and either works with method syntax. A binder is visible only inside the arm that introduced it, and it shadows any outer name for the length of that arm:

```
Var value = 99;
List input = %(pair 1 2);
match (input) {
  case %(pair ?value ?other):
    printf("inside the arm value is %d\n", value);
}
printf("outside the match value is still %d\n", value);
```

The same binder name can appear in two different arms, and a binder does not exist after the `match` statement ends. There is no "result of the `match`" to read afterwards; assign to a variable you declared outside if you need to carry something out.

Method syntax follows those types:

```
match (%(node (a b))) {
  case %(node ?child):
    printf("%d\n", child.list().len());
}
```

Named binders must also be definitely assigned whenever their arm matches. Binders inside the operands of `!not` are never available, though its leading binder is, and a binder under `!or` or membership-style `!set` must occur in every alternative. `!quote` is opaque literal data. The compiler diagnoses a maybe-bound name at the pattern.

Lists, Var, and nil

The subject of a `match` is a `List`, so the shapes you can match are `List` shapes. `Arrays`, `Maps`, and `Strings` are not `match` subjects.

A `Var` subject works because the compiler inserts the conversion to `List` for you. A `Var` whose tag is not `<list>` is not a `List`: it matches no `List` pattern, not even `%()` or `%(*rest)`, and reaches the `default` arm when there is one. A `Var` holding an empty `List` still matches `%()`. See values and Var for the tag rules and symbols and atoms for what a bare pattern spelling means.

Nil is typed `List` data, not the absence of a `List`. A null `List` is a legal subject, `%(*rest)` matches it and binds an empty `List`, and an explicitly stored empty `List` inside a larger structure is a visible node that search can find. What is *not* a node is the implicit terminal cdr of a proper `List`.

A `match` statement binds captured values to arm-local variables. Captured values keep their ordinary lifetime rules. A pattern built from literals, nested Lists, unique `?` binders, typed or not, and a final `*` or `*name` is tested in place: the compiler emits direct checks and captures. Other patterns use the runtime matcher with the same semantics.

break and continue inside an arm

`break` in an arm body exits the `match`, not any enclosing loop. `continue` skips to the next iteration of the enclosing loop:

```
List items = %((skip) (keep) (skip));
int seen = 0, kept = 0;
foreach(Var item, items) {
  seen++;
  match (item) {
    case %(skip): continue;
    case %(keep): break;
  }
  kept++;
}
printf("seen %d, kept %d\n", seen, kept);
```

That prints `seen 3, kept 1`: the two `skip` items jump past `kept++`, while the `keep` item leaves the `match` and falls through to it.

The runtime matcher

The runtime matcher exposes the same pattern language as `List` methods. Use it when the pattern is data, when you need the bindings as a value, or when you are rewriting rather than dispatching.

`try_match` reports success separately from the bindings, and leaves its output pointer untouched on failure. `match_replace` matches once and expands a template from the bindings:

```
List input = %(define x 10);
List bindings = NULL;
if (input.try_match(%(define ?name ?value), bindings))
  printf("%s = %d\n",
         bindings.assoc(<?name>),
         bindings.assoc(<?value>));
List rewritten = input.match_replace(%(define ?name ?value),
```

```
                                    %(assign ?name ?value));
printf("%s\n", rewritten.str());
```

Bindings are an association `List` keyed by the binder `Atom`, so you read them with `assoc` and the same spelling you wrote in the pattern: `<?name>` for a `?` binder and `<*rest>` for a `*` binder. These are standalone `Symbol` literals, outside the `List` syntax that accepts bare `Atoms`.

The search family walks a whole structure instead of matching at the root. `try_search` returns the first depth-first hit, `search` collects every hit, and `search_replace` rewrites all of them:

```
List tree = %(root (item 1) (wrapper (item 2)));
Var node = void;
List bindings = NULL;
if (tree.try_search(%(item ?id), node, bindings))
  printf("first id %d\n", bindings.assoc(<?id>));
printf("%d matches in all\n", tree.search(%(item ?id)).len());
printf("%s\n", tree.search_replace(%(item ?id),
                                   %(entry ?id)).str());
```

The `try_` forms report success separately from their results. `match` returns bindings, `%(())` for a binder-free success, or `nil` on a miss. `match_replace` returns the original input on a miss and `nil` for a successful scalar replacement. `search` returns its results in reverse visitation order, so do not read the list as document order. The standard library overview has the full API list.

Each of these operations runs a pattern program rather than the pattern value. When the pattern argument is a literal written at the call, the compiler gives that call its own program, prepared once for the life of the process. A pattern computed at run time -- built with `cons`, interpolated with `$`, or received as a parameter -- goes through a plan cache instead: the first call prepares its program and later calls with the same pattern reuse it, so a loop over one computed pattern pays a single preparation. The cache is local to the enclosing `Context` and holds a bounded number of recent patterns, so a loop that alternates between more distinct patterns than it holds prepares them again; hoist the pattern out of the loop when that matters. The arms of a filtered `catch` get their own per-site program, like a literal.

Match for shape, traversal for search

Use `match` when the data's structure determines what to do. Literal elements specify what must be present; `?name` and `*tail` name the parts you will use. Nested patterns describe sublists, and `default` handles the remaining cases. A pattern made entirely of wildcards often calls for iteration instead.

A loop can find candidate values, then match each candidate:

```
List program = %((set x 1) (call print x) (set y 2));
int assignments = 0;
foreach(Var node, program) {
  match (node) {
    case %(set ?name ?value):
      assignments++;
    case %(call ?fn *args):
      printf("call to %s\n", fn);
  }
}
printf("%d assignment(s)\n", assignments);
```

Use a plain `if` chain instead when there is no shape to state: one tag test, one length check, one comparison. A `match` with a single arm whose pattern is `%(?x)` is no clearer than the test it replaces. Use iteration when order, accumulated state, or early exit matters. A loop and a pattern can each do part of the work.

A plain binder does not check the captured value's type. `?value` captures whatever is in that position. When the arm's body depends on the type, use a typed capture such as `?(String value)`, which fails the arm on a tag mismatch.

Limitations

`default` must be the last arm. The compiler diagnoses a later arm. A final `default` runs only when every earlier case failed. Each branch of a conditional group may end with its own `default`, as long as no arm follows the group.

A `case` pattern must be a `%(...) List` literal. The runtime matcher accepts a standalone `Symbol` pattern such as `<?whole>`, but the statement form does not. The compiler rejects any other case pattern with a positioned x2c diagnostic before generating C.

Source guard operators must be literal. Runtime-built patterns may interpolate an interned operator `Symbol`, but a source arm needs a literal operator so definite binder assignment can be checked. The `?binder?` family is reserved and is not a working predicate.

The generated Match reference lists the runtime operations by API tier.

Iteration

x2c has two ways to walk a collection. `foreach(T name, collection)` is the short form and covers most cases. `Iter` chains combine operations that process one element at a time, then collect, aggregate, or loop over the result.

At runtime, an immediately consumed chain uses caller stack storage for every stage. Name that storage yourself when the iterator has to stay live between expressions. In a meta function, the destination-free form allocates iterator state in the compile-time session `Scope`. The storage convention covers the runtime forms.

The language reference states the `foreach` rules, and the standard library overview links to every iterator operation.

Foreach

The loop header declares the loop variable and gives the collection:

```
List values = %(10 20 30);
int total = 0;
foreach(int value, values)
  total += value;
printf("total = %d\n", total);
```

`break` and `continue` behave as they do in a C loop.

A single-binder loop over a `List`, `Array`, or `String` compiles to that type's `try_next` cursor: a caller-owned index or cell pointer and one call per element, with no `Iter` in between. Over an `Array` or a `String` it costs what hand-written indexing costs, and over a `List` it is faster than a hand-rolled `cdr` walk, which pays a `car` and a `cdr` call per cell. Prefer `foreach` and let the lowering choose the step.

What you can iterate

`List`, `Array`, `Map`, and `String` each provide an iterator adapter in their own module, so all four work directly:

```
Array scores = [3, 4, 5];
foreach(int score, scores)
  printf("score = %d\n", score);

Map ages = {"ada": 36, "alan": 41};
foreach(Var (name, age), ages)
  printf("%s is %d\n", name, age.int());

String word = "hi";
foreach(int ch, word)
  printf("char = %c\n", ch);

foreach(char ch, word)
  printf("char = %c\n", ch);
```

A `Map` yields its values, like every other collection. Two binders yield the key as well; with native loop types the loop does not allocate. For an iterator, `map.iter(&storage)` yields the values, `map.keys(&storage)` yields the keys, and `map.enumerate(&storage)` yields `(key value)` `Lists`, allocating on each pull. For `MapLongDouble`, `keys` also boxes each `long` key in the active scope. The generated families in `lib/typed-map.x` publish the same three iterators, so a `MapIntInt` iterates as a `Map` does. Traversal follows the table's internal slot order, so do not depend on the sequence. Inserting or deleting an entry invalidates live traversal state. Two binders over a `Var` that holds a `Map` yield its keys and values the same way. A `Var` that holds a typed `Map` does not support two binders; iterate the typed `Map` itself.

The explicit destination keeps ordinary local traversal allocation-free. When an iterator itself must be stored or returned, `Iter.new()` allocates zeroed iterator storage in the active `Scope`; that scope, the source, and any callback must remain live until the last pull.

A `String` yields its bytes as numeric `Var` values. Declare the loop variable as `int` when the byte is used numerically, or as `char` when it is passed to a character-oriented C API. Both forms traverse bytes, and neither decodes Unicode characters.

Lazy String fields

`String` provides three ways to iterate over fields without first building a `List`:

```
foreach(String word, text.words())
  printf("word=%s\n", word);

foreach(String line, text.lines())
  printf("line=%s\n", line);

foreach(String field, text.splits(":"))
  printf("field=%s\n", field);
```

`words()` coalesces runs of C whitespace, ignores leading and trailing whitespace, and yields no empty `String`. `splits(sep)` treats every explicit separator independently, so `" a  b ".splits(" ")` yields empty fields around and between the words. `lines()` recognizes LF, CR, and CRLF and strips the endings.

These cursors allocate one scope-owned traversal descriptor. There is no `List` and no object per field. Each yielded field is a canonical `String`. After a field has been interned once, finding an equal field of at most 256 bytes needs no heap allocation. Distinct fields remain in the active canonical-value pool. For a large input, use `Pool.open()` and `Pool.close()` to reclaim temporary fields, and promote the values that must survive.

A `Var` is iterated through its descriptor, so a boxed `List` or `Array` iterates the way the unboxed value does. A boxed value with no iterator adapter produces an already-exhausted iterator, and the loop body runs zero times. A receiver with a statically known type must have a visible `Iter` conformance, from its own adoption or from the nearest typedef ancestor. `typedef List Ast;` lets `foreach` over `Ast` reuse `List`'s conformance without `protocol Iter(Ast);`. A type with neither is rejected as not iterable.

An `Iter` is itself iterable, so `foreach` can consume a hand-built pipeline. That combination appears at the end of this chapter.

What the loop variable's type means

The general `Iter` path yields a `Var`. The declared loop type selects the conversion applied to each element as it arrives. It does not filter the elements. `foreach(int value, values)` means "unbox each element with `Var.int`" on that path. Cursor-backed collections deliver their typed outputs directly.

The conversions available for a loop variable are the `Var` conversions: `char`, `short`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `String`, `Symbol`, `List`, `Array`, and `Map`. Declaring the loop variable as `Var` applies no conversion. Do that when the element type varies or when the body dispatches on the tag:

```
foreach(Var item, values)
  printf("%s\n", item);
```

If you declare a loop type that has no `Var` conversion, such as one of your own structs, x2c still translates the loop, and the C compiler reports an incompatible assignment to the loop variable. When that happens, pick a converting type or take a `Var` and convert in the body.

What foreach guarantees

The collection expression's static type selects the traversal. Cursor-backed collections use their typed `try_next` member. A `Var` dispatches through its runtime descriptor, and an `Iter` is used as it stands. Foreach keeps the traversal state in local variables and converts each output to the declared loop type, so the loop itself allocates nothing and needs no cleanup. The collection expression can still allocate, as the lazy `String` field cursors do. Converting a `long`, `unsigned long`, `long long`, `unsigned long long`, or `long double` element to a `Var` loop variable boxes it in the active scope.

Iterators by hand

An `Iter` points to a small struct holding the object being iterated, one `Var` of state, a `Func` slot, and a `next` callback that reports success separately from its result. Advancing it calls `next`, which fills a `Var *out` and returns 1, or returns 0 to report exhaustion. The callback is cleared at exhaustion.

You read an iterator two ways:

- `iter.try_next(out)` is the preferred form. It returns success separately from the payload, and writes `out` only when it returns nonzero.
- `iter.next()` returns the element directly and `void` at exhaustion. It is unambiguous because no iterator may yield `void` as an element.

An `Iter` is single-pass. There is no rewind, and the aggregates below consume their input, so build a fresh iterator for a second traversal.

`range` is the simplest source:

```
struct Iter storage;
Iter counts = range(1, 4, 1, &storage);
Var value;
while (counts.try_next(value))
  printf("%d\n", value.int());
```

That prints 1 through 4. A range is inclusive when the endpoint lies in the step direction. `range(3, 1, -1, ...)` counts down 3, 2, 1. A step that would cross the endpoint stops before it, so `range(0, 9, 2, ...)` yields 0, 2, 4, 6, 8. A range pointed the wrong way is empty, and a zero step is rejected at construction.

You can write a source yourself by initializing storage with your own callback. `Iter.init` takes the object, the callback, and the initial state, and the receiver is the storage:

```
struct Iter storage;
Iter powers = Iter.init(&storage, 64, _powers_next, 1);
foreach(int value, powers)
  printf("%d\n", value);
```

Inside such a callback, write `iter->next` when you need the field. `iter.next(...)` is receiver syntax and calls the method instead of the stored callback.

The storage convention

An `Iter` is a pointer to caller-owned storage. Each source and lazy operation takes a final `Iter` destination internally, initializes it, and returns the same pointer. The operation's state is stored in that `struct Iter`. There is no second state object and nothing to free.

Fluent code omits those destinations when the whole chain is consumed immediately:

```
static Var double_value(Var value) {
  return value * 2;
}

static int divisible_by_four(Var value) {
  return value % 4 == 0;
}

int main(void) {
  $scope() {
    Array values = range(1, 8, 1)
      .map(double_value)
      .filter(divisible_by_four);
    foreach(int value, values)
      printf("%d\n", value);
  }
  return 0;
}
```

x2c inserts a distinct `struct Iter` compound literal for each missing final destination. Those objects have automatic storage and remain alive through the enclosing block. Completion applies only when the chain ends in `try_next`, `next`, `done`, `list`, `array`, `foldl`, `any`, `all`, `find`, `count`, `sum`, `product`, `min`, `max`, or `foreach`. A `List` or `Array` destination ends the chain the same way: it calls `list` or `array` for you, which is why the chain above needs no explicit collector.

It does not apply when an iterator is assigned, returned, or passed as an argument. Then declare one `struct Iter` for each stage:

```
struct Iter source_storage, map_storage, filter_storage;
Iter source = range(1, 8, 1, &source_storage);
Iter doubled = source.map(double_value, &map_storage);
Iter selected = doubled.filter(divisible_by_four, &filter_storage);

Var first = selected.next();
// selected and all of its sources remain available here.
```

Storage must outlive every iterator that refers to it. Never return an `Iter` built over local storage; take the destination storage as a parameter or return a collected `List` or `Array`. One `struct Iter` holds one live stage, so reusing it within a pipeline overwrites the earlier stage.

A lazy callback stage also stores its `Func`. A direct function uses one file-static binding, and a dynamic function pointer or capturing lambda belongs to the active `Scope`. That `Scope` must stay alive until the iterator is finished. To return a lazy iterator, its source storage, destination storage, and any dynamic or captured callback must outlive the returned value.

The iterators need no cleanup. A collected `List` or `Array` is scope-owned data, so the first program brackets its work with `$scope()`. See scopes and lifetime for the region lifetime.

Pipeline operations

These operations are lazy. The signatures show the fluent form; add a final `&storage` when retaining a result. There is no `take_while`, `flat_map`, or lazy sort. For eager transformations over a `List`, see collections.

The callback operations accept `Func`, so direct functions, function-pointer values, and capturing lambdas use the same method names. Every source element and accumulator is passed as a value. A callback with a reference parameter is rejected when it is first invoked, and it never receives an alias into source storage. An empty source does not invoke or arity-check its callback.

`map`, `filter`, and `foldl` reject a null callback only when they invoke it. Lazy `map` and `filter` construction does not pull the source. Their first pull consumes one source element before a missing callback raises `<bad-arg>`. A seeded fold consumes one element before that failure; a seedless fold consumes its seed and the next element. Consumed inputs are not restored, just as when a callback itself raises. Empty inputs and seedless singleton folds need no invocation.

This replaces the earlier null-callback identity behavior of `map`, immediate exhaustion of `filter`, and drain-without-combining behavior of `foldl`. The corresponding String identity and List accumulator-preservation defaults also end; Lisp nil callbacks use the same native contract.

- `range(start, end, step)`: each `int` in the range.
- `iter.filter(pred)`: elements accepted by `Var` truthiness.
- `iter.map(fn)`: `fn(element)`.
- `iter.head(count)`: at most `count` leading elements.
- `iter.enumerate(start)`: `(index element)` `Lists`.
- `iter.unique()`: the first occurrence of each element.
- `iter.chain(other)`: all of `iter`, then all of `other`.
- `iter.zip(other)`: `(left right)` `Lists` until either side ends.
- `iter.zip_with(other, fn)`: `fn(left, right)` pairwise.
- `iter.map2(other, fn)`: the same pairwise mapping with `fn` required.
- `iter.scan(seed, fn)`: each new accumulator.
- `iter.accumulate(initial)`: a running numeric sum with `Var` promotion.
- `iter.unzip(&shared, &storage)`: two independent column iterators. This operation always keeps both arguments explicit.
- `Iter.repeat(value, count)`: `value`, `count` times.

Some operations need a little explanation:

- `filter` keeps its predicate in the destination `Iter`, like the callbacks used by the other lazy operations.
- `scan` applies its callback to the accumulator and each element and yields every new accumulator, never the seed. Its callback must not return `void`.

- `accumulate` is the numeric case. It adds through `Var.binary`, keeps the promoted result tag, and yields each running total. Use `scan` for anything else.
- `map2` is `zip_with` with the callback required. A `zip_with` given no callback yields pairs.
- `unzip` requires two-element `Lists` and rejects other shapes. It yields two column iterators that can be consumed independently, buffering only the lag between them.
- `head` and `repeat` treat a negative count as zero.

Nothing runs until you ask for an element. Each stage then requests an element from its source. Taking two elements from a `map` over a range calls the mapping function twice. The pipeline above starts when `array()` requests its first element.

Collecting and folding

Constructing an iterator does not consume it. Collectors and aggregates do:

```
List numbers = range(1, 4, 1);
Array boxed = range(1, 4, 1);
printf("%s and %s\n", numbers.str(), boxed.str());

printf("sum = %d\n", range(1, 10, 1).sum().int());
printf("first over two = %s\n",
       range(1, 10, 1).find(over_two));
```

`iter.list()` and `iter.array()` build a fresh `List` or `Array` under the current scope, and a `List` or `Array` destination calls them for you, as above. The aggregates are `count`, `sum`, `product`, `min`, `max`, `foldl`, `any`, `all`, and `find`. `sum` and `product` use `Var` arithmetic promotion. `min` and `max` use total `Var` ordering and keep the first of equal values. `min`, `max`, and `find` return `void` when there is nothing to return, and `foldl` given a `void` seed uses the first element as its seed.

Handing a pipeline to foreach

An `Iter` is iterable, so `foreach` can consume a fluent pipeline directly. The compiler supplies stack storage for every stage, including both sides of a `zip` or `map2`, before the loop starts pulling.

```
foreach(int value, range(1, 10, 1).filter(is_even))
  printf("%d\n", value);
```

Nested `foreach` loops receive separate storage for each chain. Use the explicit form when the same iterator must be paused and resumed outside one loop.

Choosing between the two

Use `foreach` when the collection has an adapter, the element conversion you want exists, and the loop body is the interesting part. It is shorter and it cannot leak an iterator.

Drive an `Iter` by hand when:

- you are composing streaming stages, especially over a `range`;
- you need to interleave or compare two sources with `zip`, `zip_with`, or `chain`;
- exhaustion status matters to the surrounding code, so you want `try_next` instead of a loop that ends silently;
- you are writing a new source with `Iter.init`;
- the traversal has to be paused, handed to another function, or resumed. The loop form cannot express that.

Idioms shows how iteration combines with other x2c features.

Scopes and Lifetime

x2c has no garbage collector and no reference counting. As in C, you decide when each allocation is freed. x2c storage is of three kinds:

- Small values stored inside a `Var` allocate nothing.
- Canonical `String` and `List` values belong to an interning pool. By default they last until the process exits.
- Mutable storage belongs to a *scope*. A scope frees its allocations together.

The system macros provide `$scope()` for a retained region, `$scope(pointer)` for a temporary destination, and `$auto` for an explicitly owned local. Classes can also supply early `free` operations. The standard library overview links to the detailed API reference.

Where each value is stored

A `Var` is eight bytes. Integers that fit, doubles, pointers, `Symbols`, `Null`, and `void` are stored in those bytes and allocate nothing. The wide scalars do not fit alongside a tag: `long`, `unsigned long`, `long long`, `unsigned long long`, and `long double`. Those go in an immutable box allocated in the active scope, and the `Var` holds its address:

```
Var small = 42;
long huge = 1L << 60;
Var boxed = huge;
printf("%ld %ld\n", small.integer(), boxed.integer());
```

Both `Vars` are eight bytes; only the second one allocated. A boxed `Var` is valid while the scope that allocated the box is alive. See Values and Var for the encoding itself.

A non-empty `String` and every `List` cell built by `cons` are interned: equal content becomes one canonical pointer. An interning pool holds these values. By default they stay valid for the rest of the process. You do not free them, and you do not keep a scope alive to hold them. A canonical `List` does not extend the lifetime of a mutable object stored inside it. The `Pool` calls described below give temporary `String` and `List` structure a shorter lifetime.

All other storage is mutable and belongs to a scope: `Block`, `Array`, `Buffer`, wide `Var` boxes, and anything you request with `Scope.malloc`, `Scope.calloc`, or `Scope.memdup`.

A scope is a group of allocations

`Scope.retain` opens a new region on the currently active scope slot; `Scope.release` frees everything allocated inside it. Between them, the allocation calls go into the retained region:

```
Scope.retain();
char *line = Scope.malloc(32);
snprintf(line, 32, "%d bytes", 32);
puts(line);
Scope.release();
```

Retained regions nest. Each `Scope.retain` must be paired with exactly one `Scope.release`. Use a nested region to free temporaries before the surrounding work finishes.

The runtime checks each release against the active slot. An empty retained region releases normally. An extra release, or a release from a different pushed slot, raises `<bad-state>` and leaves the active region intact. A matching `catch` can handle the error. Without one, the default policy aborts the program.

Storage returned by these calls is managed memory. `Scope.realloc` grows or shrinks it in place, and `Scope.free` frees it before the release. Most storage does not need `Scope.free`, because the release frees the whole region. `Scope.malloc_finalized` allocates storage that runs a function of your choice when it is reclaimed; see attaching a finalizer.

For storage that must outlive the current region, hold a scope in a variable of type `Scope` and allocate into it directly. `Scope.malloc_in`, `Scope.calloc_in`, and `Scope.memdup_in` take the slot's address. They do not make the slot active, so other allocations still go to the active scope. `Scope.destroy` ends such a scope:

```
Scope work = Scope.new_named("request");
char *copy = Scope.memdup_in(&work, "payload", 8);
puts(copy);
Scope.destroy(work);
```

`Scope.new_named` copies a diagnostic name. Name every long-lived scope. At exit the allocator reports each scope that is still alive, by name. Without the `Scope.destroy` line above, the program prints this on standard error before it ends:

```
Scope leak detected:
  live_scopes: 1
  live_allocations: 1
  scope "request": 1 allocations
  live_backing_allocations: 2
```

The last line counts the allocator's own bookkeeping, such as the record holding that name. `Scope.stats` reports the same counters on demand, so you can check that a routine leaves no allocations behind.

When several operations should allocate in the same scope without passing a slot to every call, `Scope.push` makes a slot active and `Scope.pop` restores the previous one. `Scope.pop` with no push outstanding raises `<bad-state>`. `Scope.destroy` raises the same cause when the scope is still in use: the active root, a slot still on the pushed stack, or a scope that a later `Scope.retain` layered another region on top of. The scope is left intact. Uncaught, `<bad-state>` aborts the program.

To release a region on every exit from a block, write it as `$scope()` or put the release in a `defer`. Errors and Cleanup gives the exit rules.

Bounding temporary canonical Lists

`Pool.open` and `Pool.close` give canonical values a shorter lifetime. Bracket work with them when it builds many `List` cells and uses none of them afterwards. `String` and `List` share one canonical pool, so the bracket bounds both:

```
static int evaluate(int input) {
  Pool.open();
  defer Pool.close();

  List temporary = %($input ${build_left(input)} ${build_right(input)});
  return temporary_score(temporary);
}
```

Every `cons` inside the bracket remains canonical. `Pool.close` reclaims cells created in the innermost pool and forgets their identities. Calls nest, and every successful `Pool.open` requires one matching `Pool.close`. Use `defer` when an error or early return can cross the boundary. `Pool.open_named` gives the bracket a name that appears in `Scope` diagnostics, and `Pool.current` returns the active pool.

If a `List` must outlive the bracket, call `List.promote` before `Pool.close`. Promotion is pointer-stable and recursively preserves nested `Lists`, interned `String` cars, and long `Atom` payloads:

```
Pool.open();
List result = build_result();
result.promote();
Pool.close();
return result;
```

An unpromoted `List` dangles after `Pool.close`. Small amounts of data, and values needed for the rest of the process, can stay in the default pool.

Pools and Contexts

A pool bracket frees only canonical values. A `Context` frees more: one open `Context` owns a `Scope`, `Error` state, `Match` state, and, when it is opened isolated, a canonical-value pool of its own. `Context.open_isolated` calls `Pool.open_named` to get that pool, and `Context.close` closes it. Use `Pool.open` when only canonical `String` and `List` structure must be freed. Use a `Context` when the `Scope`, the `Error` state, and the `Match` state must also end with it. See Contexts and Threads.

Moving a value out of a scope

When a region computes one result that must outlive it, use `Scope.move`. It relinks a single allocation onto another scope without copying it. The pointer does not change:

```
Scope keep = Scope.new_named("results");
Scope.retain();
char *text = Scope.memdup("survivor", 9);
Scope.move(text, &keep);
Scope.release();
puts(text);
Scope.destroy(keep);
```

A `String` or a `List` needs no move, because canonical values outlive the scope that was active when they were built.

Attaching a finalizer

A record that holds a resource outside the allocator, such as a handle from a C library, can release it when the record is reclaimed. `Scope.malloc_finalized` takes the function to run:

```
typedef struct Feed { feed_parser *native; } *Feed;

static void _feed_drop(void *ptr) {
  Feed feed = ptr;
  if (feed.native) feed_close(feed.native);
  feed.native = NULL;
}

Feed Feed.open(void) {
  Feed feed = Scope.malloc_finalized(sizeof(struct Feed), _feed_drop);
  feed.native = feed_open();
  return feed;
}
```

The finalizer runs exactly once with the record's pointer: on `Scope.free`, on a `Scope.realloc` to size zero, when the owning scope is released or destroyed, or when shutdown releases its owning root region. A detached `Scope` that is still live at shutdown is reported as a leak, and its finalizers do not run. The finalizer follows the record through `Scope.move`, and `Scope.realloc` keeps it and passes the resized pointer. Records are reclaimed most recent first, so a finalizer can still read older records in the same scope.

A finalizer has three rules. The record is already unlinked when its finalizer runs, so the finalizer must not free or move the record itself. It must not raise. It may allocate into other scopes, and into the scope being destroyed only for scratch that the same destruction reclaims.

A wrapper can also release the native handle early. If its own `free` method clears the native field, the finalizer has nothing left to do.

`Block` and `Buffer`

`Block` is the fixed-width storage primitive. A `Block` owns a contiguous run of elements of one width, chosen at construction, and tracks their length and capacity. Its backing store comes from the active scope, so a `Block` created inside a retained region is freed when that region ends, even if you never call `Block.free`.

```
Block ids = Block.new(sizeof(int));
for (int i = 0; i < 4; i++) ids.push(&i);
int *values = ids.bytes;
printf("%zu of %zu, last %d\n", ids.len(), ids.capacity(), values[3]);
ids.free();
```

The example reads `values` after the last push. Growth reallocates and moves the `bytes` pointer, so a copy taken before an append can be stale afterwards. The `Block` handle stays valid across growth, so pass the handle around. If you work through the raw `Bytes` pointer, use the forms that return the pointer, `Bytes.append` and `Bytes.reserve`, or recover the handle with `Bytes.block`. `Block.append` and `Block.reserve` raise `<bad-arg>` on a bad argument and leave the storage unchanged. Capacity and allocation failures raise `<size-limit>` and `<alloc-`

`fail>`. None of these raises returns to the call. `Array` is a `Block` whose element width is `sizeof(Var)`; see Strings, Lists, Arrays, and Maps.

`Buffer` is a text builder built on `Block`. It rejects an embedded NUL, because it builds canonical `String`s, and it tracks the byte position and leading indentation of the current line for `push`, `pop`, `indent`, and `newline_indent`. Use `Block` for checked storage of some element type, and `Buffer` for assembling text.

```
String text;
$scope() {
  Buffer out = $auto(Buffer.new(0));
  out.write("case ").printf("%d", 7).write(":").newline_indent();
  text = out;
}
puts(text);
```

The `String` destination calls `Buffer.str`, which interns the accumulated text, so `text` is canonical. It stays valid after the `Buffer` cleanup and the region release free every byte the builder used. Most x2c code follows this pattern: mutable storage inside a region, a canonical result outside it.

Choosing when to free storage

Free storage when the work that needs it is finished: a request has ended, a file has been parsed, or temporary values have been combined into one result. Do not wrap a scope around canonical values that already outlive it. That scope frees nothing and adds a release you can forget.

A scope does not prevent dangling pointers:

```
static char *leaked_label(void) {
  Scope.retain();
  char *label = Scope.memdup("temporary", 10);
  Scope.release();
  return label;
}
```

`Scope.release` freed that storage. The returned pointer is dangling, and every later use of it is undefined behavior. A use can read stale bytes, corrupt the allocator, or seem to work for a while. `Scope`s make lifetimes explicit and cheap to end. They do not check at runtime that a value has stopped being used. The same applies to a boxed wide `Var`: if you keep the `Var`, you must keep the scope that allocated the box.

The compiler warns about this example because the escape is visible in the source: `label` is allocated between a retain and its release and read after it.

To avoid dangling pointers:

- Return a canonical value (a `String`, a `List`, or a `Symbol`) when a result must cross a scope boundary.
- Use `Scope.move` when a mutable allocation must outlive its region.
- When unsure, let the caller create the scope and pass the slot down. The lifetime is then chosen where the caller can see it.

Lifetime warnings

Every allocation in this chapter belongs to a region: a part of the program that owns storage and ends at a known point. Regions are `$scope()` blocks, `Scope.retain` and `Scope.release` pairs, `$scope(&slot)` pushes, `$auto` locals, `Scope` locals that `Scope.destroy` ends, `Pool.open` brackets, and the function's own locals and parameters.

The rule is: a value allocated inside a region must not be reachable after that region ends. `Scope.move`, `Context.export`, `List.promote`, and returning a canonical value all follow the rule, because each gives the storage an owner that outlives the region.

Translation warns when code breaks the rule. The warning includes the value's name and the way it leaves, and a note gives the line that opened the region. A value leaves when it is:

- returned;
- assigned to a local declared outside the region;
- stored through a parameter, through an unknown pointer, or into a static;

- stored into an object that belongs to another region;
- held by a List cell, as in `cons(a, rest)` or `%($a)`, or captured by a closure that leaves;
- handed to a function in the same unit, or to a runtime operation such as `Array.push`, that stores it in one of those places.

The address of a local or parameter is reported the same way when it outlives the function. Two more warnings come from the same check. `after-free` reports a local read after `Scope.free`, `Scope.realloc`, or `Array.list_free` consumed it. `bad-free` reports `Scope.free` or `Scope.realloc` given a literal, a local's address, or other storage no `Scope` allocator returned. In a `meta` function, each of these findings is an error; see Meta Functions.

Code that follows the patterns in this chapter compiles without warnings:

```
static Entry adopt(Scope *keep) {
  $scope() {
    Entry entry = Scope.calloc(1, sizeof(struct Entry));
    Scope.move(entry, keep);
    return entry;
  }
  return NULL;
}

static String label(void) {
  $scope() {
    Buffer out = Buffer.new(0);
    out.write("ready");
    return out;
  }
  return NULL;
}
```

`Scope.move` moves the storage into a scope the caller owns. The `String` return destination calls `Buffer.str`, which produces a canonical `String` that its pool owns.

A warning does not stop translation. When a warning describes a lifetime you arranged another way, you may keep the code. The warnings cover one lexical pattern. The Region Model lists what they do not cover, the exemptions, and how the check works. To observe a dangling read at run time, link the program against `libx2c.a` and compile with `cc -fsanitize=address,undefined`.

Ordinary C storage still works

x2c is a superset of C, and it does not change automatic or static storage. Locals, arrays, and `struct` values still live on the stack and end at the closing brace; `static` and file-scope objects still last for the program. `Scopes` govern only what you explicitly allocate through them.

Many library types store their state in memory the caller provides, so a loop can iterate without calling the allocator:

```
struct Iter storage;
Iter walk = items.iter(&storage);
Var value;
while (walk.try_next(value)) printf("%ld\n", value.integer());
```

`storage` is an automatic variable. Nothing here needs freeing or a scope; see Iteration for the protocol. An automatic local or parameter you assigned directly keeps its value when a `raise` transfers control to a `catch`. Your source needs no optimization-specific qualifiers for that.

Where to look next

Read the standard library overview and the language reference for the rules. Idioms shows how scopes combine with other x2c features. The Region Model covers the lifetime check: its guarantees, its limits, and how it compares with the checks in other languages.

Contexts and Threads

A `Context` gives a task its own `Scope`, `Error` state, and `Match` state. An isolated `Context` also gives it a child canonical-value pool, which its `Strings` and `Lists` share. Closing the `Context` restores the state that was active before it opened and reclaims everything that was not exported.

If opening a `Context` fails, the parts already opened are closed before the failure reaches an outer `catch`. A failed open does not change `Context.current()` or leave a pool active.

A `Thread` runs a callback in an isolated `Context` on a native worker thread. The worker may read parent values and may access shared mutable objects. The program must keep those objects alive and coordinate any access that could race.

Use a Context for bounded work

`Context.open` and `Context.open_named` use the current canonical-value pool. Use them when the work needs fresh `Scope`, `Error`, and `Match` state and its `Strings` and `Lists` should live as long as the caller's.

`Context.open_isolated` and `Context.open_isolated_named` add a child pool. New temporary `Strings` and `Lists` are then reclaimed when the `Context` closes. Values already in a parent pool stay visible and keep their identity. The name given to a `_named` form appears only in `Scope` leak diagnostics.

The following program builds a cyclic mutable result around a private `String`, exports it, and closes the `Context`. `Context.export` moves the `Array` and `Map` storage into the caller's `Scope`, copies the private immutable values into the caller's pool, and preserves the cycle:

```
#include <stdio.h>

int main(void) {
  Context input = Context.open_isolated_named("input file");
  String temporary = String.new("temporary text");
  Array values = [];
  values.push(temporary);

  Map result = {};
  result[<values>] = values;
  result[<self>] = result;

  result = input.export(result);
  input.close();

  printf("%s\n", result[<values>].array()[0].string());
  return 0;
}
```

Always use the value returned by `Context.export`. Mutable `Arrays`, `Maps`, `Blocks`, `Bytes`, `Buffers`, and wide scalar boxes retain their identity when their storage can be moved. Private `Strings`, `Atoms`, and `Lists` are canonicalized in the destination pool, so their pointers may change. Export walks nested `Arrays`, `Maps`, and `Lists`, including repeated references and `Array`/`Map` cycles. A registered custom value can provide its own `Context` exporter. `void` exports as `void`.

The seven packed `Arrays` from `typed-array.x` and the four packed `Maps` from `typed-map.x` follow the same rule. A container owned by the source `Context` keeps its pointer, and its header and backing storage move together. A borrowed container is returned unchanged. The numeric families keep their native entries and continue growing in the destination `Scope`. `ArrayString` canonicalizes its elements, `MapStringString` its keys and values, and `MapStringInt` its keys, in the destination pool before the container moves, so their `String` pointers may change. If two keys become the same destination `String`, the entry visited later in source bucket order replaces the earlier one. `Map` traversal order is unspecified.

A custom `protocol Var(T)` participant may implement `T.export_context` to move its mutable storage into another `Context`. The member is optional, like the other `Var` methods. Its implementation recursively exports nested values and moves its owned allocations through the `Context` helpers; application code still calls `Context.export` on the complete value.

Export covers values allocated through the active `Context` and values inherited or borrowed from its caller. Code that manually creates an unrelated `Scope` or immutable pool must manage that lifetime itself; returning such a value as though the `Context` owned it is undefined behavior.

The complete program is in `examples/power/contexts.x`.

Pass shared values to a worker

`Thread.start` copies the input structure's bytes. It does not copy anything reached through pointers in that structure. A pointer, `String`, `List`, `Map`, or other handle therefore refers to the same parent object in the worker. The copied bytes have `max_align_t` alignment, matching ordinary allocation; do not pass a structure that requires extended alignment beyond that.

Parent immutable `String`s and `List`s can be read directly. Shared mutable objects can also be read or changed, but normal C data-race rules apply: a read is safe only when no thread can write at the same time, and coordinated writes need a `Mutex` or another suitable mechanism. Every referenced parent object, including the `Mutex`, must remain alive until all workers using it have joined.

```
typedef struct Work {
  int iterations;
  int *counter;
  Mutex mutex;
} Work;

static Var count(const void *input, size_t input_size) {
  const Work *work = input;
  if (input_size != sizeof(Work))
    raise %(bad-arg (operation "count worker"));

  for (int i = 0; i < work.iterations; i++) {
    $lock(work.mutex) (*work.counter)++;
  }
  return work.iterations;
}

int main(void) {
  int counter = 0;
  Mutex mutex = Mutex.new();
  Work work = { 1000, &counter, mutex };
  Thread first = Thread.start(count, &work, sizeof(work));
  Thread second = Thread.start(count, &work, sizeof(work));

  first.join();
  second.join();
  first.free();
  second.free();
  mutex.free();
  return counter == 2000 ? 0 : 1;
}
```

Each worker gets a private canonical-value pool, so its temporary `String`s and `List`s are not kept for the life of the process. The callback returns one `Var`. `Thread.join` waits for the worker and exports that result into the joining `Scope` and active pool. Until the join, the result stays in the sealed worker pool, even after the callback finishes. Once the native join succeeds, the `Thread` consumes that storage, even if exporting the result raises. A failed export cannot be retried. Call `Thread.free` after `Thread.join` returns or after catching an `Error` raised by the join.

A `Thread` cannot be detached or cancelled. Join every started `Thread` before the process exits. If a worker is still unjoined at shutdown, even one whose callback has finished, the runtime prints a message and aborts. The join reclaims the worker's result `Scope` and pool.

A worker uses the starting thread's `Error` policy. A cause whose policy is `<collect>`, `<log>`, or `<ignore>` returns to its `raise` inside the callback, as it does on the starting thread. A cause whose policy is `<abort>` leaves the callback. That includes every shared cause and every code that no policy names. `Thread.join` then raises `<join-fail>` with the worker's captured errors, and that `raise` does not return to the joiner. So `Thread.join` either returns the worker's exported result or transfers. A callback that returns `void` joins as `void`.

Captured errors stay in the sealed worker storage and enter the joining pool only during `Thread.join`. A worker `Error` whose policy is `<log>` also passes through the global `Logger` error handler. `Logger` serializes its sinks. A memory sink copies worker `Strings` and `Lists` into the pool captured when its `Logger` was created, so logged entries stay valid after the worker exits.

The more complete `examples/power/threads.x` uses two workers, parent `Strings`, private temporary `Lists` and `Strings`, a shared counter protected by a `Mutex`, joined `Map` results, and a global memory `Logger` sink.

All type and protocol registration must finish before the first successful `Thread.start`. Starting native execution freezes descriptor declaration. Boxing is not frozen: a worker may box the first value of a declared class, and every thread then reads that class's complete row.

`Thread.start` also turns on locking for the canonical-value pools. A process that has never started a worker does not lock them. Create native threads through `Thread`; one made by calling `pthread_create` directly can share interned values with no lock protecting them.

Errors and Cleanup

x2c adds structured errors to C. `raise` records an `Error`, a filtered `catch` selects it by cause, and `finally` and `defer` run cleanup when a block exits early. The system macros attach the same `defer` cleanup to managed locals and to `$scope`, `$let`, and `$lock` bodies. The language reference gives the full statement rules.

One error protocol

The code that first detects a failure raises one structured `Error`:

```
raise %(size-limit (operation "Block.reserve") (bytes $requested));
```

The first item is a compact `Symbol` naming the cause. The remaining items are `(key value)` detail pairs. `Error` also records the source location. A code identifies a cause that a handler can branch on. Put the reporting mechanism and the module that found the cause in the details.

Use the most specific shared cause. For example, `<bad-arity>` identifies an argument count, `<bad-types>` identifies unsupported dynamic operand types, `<not-found>` identifies a missing external resource, and `<invariant>` identifies an impossible internal state. A raiser may add any detail keys without changing the code. Detail values must be immutable data: `Null`, numbers, enum values, `Symbols`, `Atoms`, `Strings`, and `Lists` of those. The reference gives the full rule.

When an Array failure reports both a type and a method action, `operation` is the Array type String, and `action` is the method Symbol. For example, a push to a null `ArrayInt` raises `(bad-arg (operation "ArrayInt") (action push))`. The action `<take-last>` means the public method `take_last`.

When several codes seem to fit, choose in this order: callable shape, dynamic operation, resource or external operation, API contract, then `<invariant>` for an impossible internal state. The reference lists the 31 shared codes.

Every shared cause aborts by default and never returns to the raising call. It can transfer to a matching filtered `catch`, after the cleanup between them runs. So a constructor whose preconditions hold returns an initialized object, and a failed open, read, write, or format returns no null, `void`, or fallback value.

Return values still carry ordinary results. Clean EOF, a missing `Map` key, iterator exhaustion, and numeric text that does not parse are not `Errors`. APIs such as `File.readline_into`, `Map.try_get`, `Iter.try_next`, and `String.try_long` keep those expected outcomes separate from their value output.

Catching by cause

`try` requires a following filtered `catch`, `finally`, or both. Adjacent `catch` arms select the newest `Error` by matching `%(CODE @DETAIL)`:

```
try {
  block.reserve(requested);
}
catch %(size-limit * (bytes ?count) *): {
  printf("could not reserve %ld bytes\n", count);
}
catch %(bad-arg *detail):
  report_bad_argument(detail);
catch:
  report_unknown_error();
```

The pattern language and binder rules are the same as `match`. A `?name` binder declares a `Var`, a `*name` binder declares a `List`, and arms run in source order. `catch:` is an optional default arm and must be last. Filters are evaluated once when the `try` is entered. Like a literal pattern written at a `match`, each arm's program is prepared once for the life of the process; an arm whose pattern interpolates a run-time value with `$` is prepared on every entry instead.

When no arm matches, the `Error` continues outward. Selecting an arm consumes the errors accumulated since that `catch` was registered. Older errors remain available to the enclosing handlers.

The selected `catch` is removed before its body runs. A handler can therefore translate a lower-level cause without catching its replacement:

```
try parse_config(source);
catch %(malformed *cause):
  raise %(bad-state (operation "load-config") (cause $cause));
```

The matched details are borrowed from the selected `Error` and remain valid through that `catch` arm. If they must survive the arm, copy them first with `Error.snapshot`. If translation is unnecessary, omit an arm and let the original cause continue outward.

A wildcard code binder handles any cause while retaining access to its details. Snapshot the details inside the arm when they must survive it:

```
Symbol code = 0;
List detail = nil;
try raise %(bad-arg (operation "load"));
catch %(?caught *fields): {
  code = caught;
  detail = Error.snapshot(fields);
}
printf("%s %s\n", code.str(), detail.repr());

bad-arg ((operation "load"))
```

A pattern in the code position selects several codes in one arm, and `raise` accepts the bound code as `$code`. This arm rewraps either cause under its own code and lets every other cause continue outward:

```
try raise %(conv-range (value 300));
catch %((!or ?code bad-enc conv-range) *cause): {
  List lower = cons(code, cause);
  try raise %($code (index 0) (cause $lower));
  catch %(?outer *detail): printf("%s %s\n", outer, detail.repr());
}

conv-range ((index 0) (cause (conv-range (value 300))))
```

Put a wildcard arm after any specific filters that should handle a cause first. Like any matching arm, it consumes the selected errors and does not catch an error raised by its own body.

Handlers and policy

A filtered `catch` transfers control out of the raising call. Embedders may also install observing handlers with `Error.push`. An observing handler runs inside the raising call, receives the errors accumulated since its registration, and returns:

- `<handled>` to consume those errors and stop;
- `<declined>` to leave the errors accumulated and continue outward;
- `<fatal>` to terminate without allocating.

`<unwind>` is reserved for compiler-generated filtered catches.

An observing handler cannot consume a non-returning cause with `<handled>`. If it tries, `Error` terminates the process. Use a filtered `catch` to recover outside the failed call.

The handler borrows its `Errors` for the duration of the callback. `Error.pop` closes the handler and reclaims everything accumulated since its registration, including declined or collected `Errors`. Use `Error.snapshot(value)` to retain a value beyond a callback; it copies into the caller's ordinary `Scope` and canonical `List` and `String` pools. `Error.since(mark)` also returns a snapshot in the caller's scope and pools.

If every handler declines, the code's policy selects the result:

Policy	Result
`<abort>`	render if possible, then terminate without further allocation
`<log>`	render once, consume the newest `Error`, and continue
`<collect>`	retain the `Error` without rendering and continue

Policy	Result
`<ignore>`	consume the newest `Error` without rendering and continue

Every shared cause defaults to `<abort>` and cannot be reconfigured. A process may set policy for unknown and user-defined codes only. The code that raises chooses the cause; the embedding caller chooses the policy.

`Error.policy_set` accepts only the four dispositions in the table. Passing another disposition raises `<bad-arg>` and leaves the previous policy unchanged. It also rejects `<log>`, `<collect>`, or `<ignore>` for every non-returning cause, leaving those policies at `<abort>`. Unknown `Error` codes remain legal and default to `<abort>`.

The default `Logger` registers an observing `Error` handler. An uncaught `Error`, one whose policy is `<abort>`, is reported as one line on standard error, and the process then aborts:

```
x2c: uncaught <bad-state> in main (config.x:12): operation load-config, path x.toml
```

The line names the cause, the raising function and source line, and the detail pairs. An `Error` raised inside the runtime library shows its detail without the runtime's own source location; the `operation` detail names the call. A `<log>` entry is rendered as a structured `<err-report>` log event. Setting `X2C_DEBUG` in the environment renders uncaught errors as that event too, with the complete detail and location. A program that installs its own global `Logger`, or adds a sink to the default one, receives the structured event for both policies, so a configured log never loses information. After rendering, the handler declines, and `Error` transfers control or consumes the errors as required.

`finally` and `defer`

Both forms run cleanup when control leaves their region: on normal exit, on `return`, `break`, `continue`, or an outward `goto`, and when an `Error` transfers through the region to a selected `catch`. They do not run when an uncaught `Error` aborts the program, or when native code calls `exit` or `longjmp`. The two forms differ in what they attach to.

`finally` attaches to one `try` and runs after its body or selected handler:

```
static String first_line(String path) {
  File input = File.open(path, "r");
  String line = NULL;
  try {
    line = input.readline();
  }
  finally {
    input.close();
  }
  return line;
}
```

The `defer` statement attaches to the enclosing block. Multiple `defer` statements in a block run last-in, first-out:

```
static String first_line(String path) {
  File input = File.open(path, "r");
  defer input.close();
  String line = input.readline();
  return line;
}
```

Prefer `defer` when a block owns a resource but has no handler. The release then sits next to the acquisition, so a reader can compare them. Keep the deferred statement small. Use `finally` when cleanup belongs to an existing `try` statement.

Use either form to release resources. Recovery belongs in a `catch`. Keep expected absence, exhaustion, and parse outcomes in return values, and let `Error`s propagate.

A `finally` body cannot define a label. Its statements run on every path that leaves the region, so the generated C repeats them, and a label among them would be defined more than once. x2c reports an error at the label. A `goto` that leaves the region is allowed, as is one inside the `try` body.

A `return`, `break`, or `continue` that leaves a `finally` or `defer` body leaves its frame first. When that body is running for a propagating `Error`, the transfer discards the `Error`, as a handler that returns does.

Raising during cleanup

A `raise` inside a `finally` or `defer` body transfers to the enclosing handler, never back into the cleanup that raised it. Cleanup runs at most once per exit path. If that cleanup was already running for an `Error`, the new `Error` replaces it: the pending one is discarded with its detail, and only the replacement reaches a `catch`. With no matching handler anywhere outward, the replacement terminates the program through the error floor like any other unhandled `Error`. This is why cleanup is for release, not recovery; handle a failure a finalizer can produce inside the finalizer.

Return expressions run before cleanup

A `return` expression is evaluated and saved before cleanup runs. Cleanup then executes, and the saved value is returned:

```
static String first_line(String path) {
  File input = File.open(path, "r");
  defer input.close();
  return input.readline();
}
```

Here `readline` runs while the `File` is open, then the `File` closes, then the saved `String` is returned. Cleanup cannot change which value the return expression already produced.

`break` and `continue` run cleanup only up to their target loop or `switch`. Cleanup registered outside that boundary remains active until a later normal exit, `return`, or `Error` transfer reaches it.

`goto` follows the same cleanup ancestry rule. A jump within the same cleanup region runs nothing. An outward jump runs every cleanup region it exits. A jump into a protected region, or sideways into a sibling protected region, is a compile-time error because that region's cleanup was never registered on the source path.

Cleanup and scopes

`Error` transfer does not release a `Scope` automatically. Write the region as `$scope()` whenever a raising call can cross the release; its deferred release runs on the error path too:

```
static void render(String text, int width) {
  $scope() {
    char *line = Scope.malloc(width + 1);
    if (!text) {
      raise %(bad-arg (operation "render"));
    }
    line[0] = 0;
  }
}
```

The same rule applies to pushed scope slots, `File`s, and every other resource with paired teardown. See Scopes and Lifetime for ownership details.

Calling fallible APIs

Distinguish an API's return value from the `Error`s it raises:

- If an operation raises, either provide a matching `catch` or let the `Error` continue outward. A handled resumable failure returns the documented sentinel; a non-returning cause transfers and never returns to that call.
- If an operation reports expected absence or exhaustion, inspect that result; a `try` does not convert it into an `Error`.
- Keep these outcomes distinct when adding a fallback.

For example, raw `File` readers separate data from clean EOF in their result, while a host read failure raises `<io-fail>` and transfers:

```
Block line = Block.new(1);
File input = File.open("/etc/hosts", "r");
defer input.close();
try {
```

```
  if (input.readline_into(line) == <eof>)
    printf("empty file\n");
}
catch %(io-fail *):
  printf("read failed\n");
```

The generated API reference states each callable's result, `Error`, and ownership rules.

Process shutdown

Shutdown hooks run in reverse registration order. `Error` initializes before `Logger` installs its hook, so later subsystems may still raise during shutdown. `Logger` then reports any collected root errors once and closes its handler. Finally, `Error` destroys its empty handler chain, stack, policy, pools, and `Scope`. A `raise` after that point terminates without allocating. The raw `Scope` leak report runs after all shutdown hooks.

Protocols

`Cleanup(T)` declares `void T.cleanup(T)` for explicitly owned locals. Classes select their defaults and participate through the same protocol and method rules.

A protocol declares methods that other concrete types can adopt. Its *base* is the type that declares those methods; a *participant* is a type that explicitly adopts them. The compiler selects each participant method or adapts a base method by converting its arguments and result.

The selected methods support dot calls, operators, and, for `Var`, calls on boxed values. Adoption works at compile time; it creates no runtime interface object. Functions with matching names do not imply adoption.

Protocols adapt values. Registration, ordering, and generated initialization code belong to decorators. Startup and shutdown operations with no receiver have no value to convert and are outside the protocol system.

Declaration and adoption

A declaration names the base type and uses a fresh name, here `T`, for the participant. It then declares optional associated types and the methods:

```
protocol Var(T) {
  associated Key = Var;
  associated Value = Var;

  String T.str(T);
  int T.truth(T);
  Value T.getindex(T, Key);
  Value T.setindex(T, Key, Value);
}
```

Associated types must be declared before methods. Their names and the participant name are local to the protocol body. The compiler warns if the participant name shadows a visible type, with code `shadow`: that may be an adoption written as a declaration by mistake.

Participation is a separate, bodyless declaration:

```
protocol Var(Vec);
```

`Vec` must name an existing type. This declaration makes it a participant in `Var(T)`; defining converters or matching typedefs is not enough. The base itself is not a participant. Expressions with the base's exact static type continue to use its own methods.

A typedef can inherit the nearest visible ancestor's resolved protocol methods without its own adoption. It also inherits the associated types, conversions, and representation unchanged. The compiler creates no new adoption, adapter, alias, descriptor, tag, or registration for that use.

An explicit adoption on the typedef takes precedence and resolves those choices again. If it is invalid, the compiler reports the error; it does not fall back to an ancestor.

A type that is only a typed view of an existing `Var` representation can name that representation explicitly:

```
typedef List Row;

Var Row.var(Row value);
Row Var.row(Var value);

protocol Var(Row) as List;
```

`Row` has its own methods, function signatures, protocol participation, and conversions to and from `Var`. Compatible `List` methods satisfy `Var(Row)` directly, without forwarding methods on `Row`.

Boxed values still carry `<list>` and use `List`'s descriptor for dynamic calls; no `Row` descriptor is registered. Consequently, `value is Row` and `value is List` mean the same thing. This is a compile-time distinction, not a new runtime class. The type after `as` must already have a fixed `Var` tag, as `List`, `int`, and `void *` do.

A distinct runtime class whose type name does not encode the desired tag can name a shorter custom tag:

```
protocol Var(ArrayString) tag <arraystr>;
```

The participant and its converters still use `ArrayString`, but boxed values carry `<arraystr>`. The tag must be a `Symbol` literal, must not be built in, and must be unique across the process. `tag` registers the participant's own descriptor; `as` reuses another type's descriptor. The two modifiers cannot be combined.

Any adoption may be local to its translation unit:

```
static protocol Prepared(LocalPlan);
```

A protocol body may also be `static`, which keeps it private to its source file. Three visibility patterns cover most code:

Protocol	Participant	Typical spelling
public	public	`protocol Base(Participant);`
public `Var`	private	`protocol Var(PrivateType);`
private	private	`static protocol PrivateBase(PrivateType);`

A private type can therefore participate in the public `Var(T)` protocol without becoming public itself:

```
static typedef struct LocalPlan *LocalPlan;
static Var LocalPlan.var(LocalPlan plan) => Var.new(<localplan>, plan);
static LocalPlan Var.localplan(Var value) => (LocalPlan) value.pointer();
protocol Var(LocalPlan);
```

An adoption marked `meta`, such as `meta protocol Iter(List);`, also makes that conformance's witnesses available to compile-time code, as described in native C functions.

Placement

Put an adoption beside its converters. The compiler resolves it in the declaring unit, which must see the protocol, the participant, the converters, and the participant methods that fix its associated types. A unit that includes the declaring unit reuses that result. The reference gives the linkage and visibility rules.

Conversions

An adoption needs only the conversions that its generated adapters use. Name the forward conversion `Base P.base(P)` and the reverse `P Base.p(Base)`. The forward conversion must accept every participant value. The reverse may fail for a base value that holds no `P`. `Var(T)` descriptor thunks both unbox arguments and box results, so most object participants need both:

```
Var Vec.var(Vec value);
Vec Var.vec(Var value);
protocol Var(Vec);
```

A missing conversion is an error at the adoption, naming the method that needs it. The reference lists which adapter needs which conversion.

How members are chosen

Each adopted member resolves on its own. The participant's own method wins, then the nearest inherited method, then the protocol's default. An explicit adoption can use a method inherited through the participant's typedef chain:

```
typedef List Domain;
protocol Iter(Domain);
```

`Domain` already iterates through its inherited `List` adoption. The explicit adoption selects `List.iter`, so iteration calls `List_iter` directly, with no `Domain.iter` forwarding method.

`associated Name = Default;` declares a type that each participant binds from its own signatures. `Array` binds an index `Key` to `int`, while `Map` binds it to `Var`. Member resolution gives the full rules, including signature conflicts and `Var(T)`.

Operators

Operators call the same members as dot syntax: `+` calls `add`, `==` calls `equal`, `<` compares through `compare`, conditions call `truth`, and indexing calls `getindex`. A participant may also declare `void T.discard(T)` to release each unnamed temporary in an operator chain. The reference lists every operator and its member, with the rules for mixed operands and updates.

When protocols share a method

A participant method satisfies every protocol that declares the same compatible member. Two protocol defaults for one member are an error; implement the member on the participant to choose. The reference gives the full rule.

Before adding a protocol member, inspect the conformance dump for every participant homonym and its signature:

```
./builds/0/x2c translate --dump-conformance path/to/unit.x
```

An existing method with the same name must have both the intended meaning and a compatible signature. Rename it before extending the protocol if it means something else. Incompatible signatures are compilation errors.

Worked examples

Vec: declared boxed behavior

Vec supplies the conversions that `Var(T)`'s descriptor thunks use, implements only `str`, and explicitly adopts the protocol:

```
typedef struct Vec {
  int x;
  int y;
} *Vec;

Var Vec.var(Vec value) {
  return Var.new(<vec>, value);
}

Vec Var.vec(Var value) {
  return (Vec) value.pointer();
}

String Vec.str(Vec value) {
  return %"Vec(${value.x},${value.y})";
}

protocol Var(Vec);

int main(void) {
  Vec value = Scope.malloc(sizeof(struct Vec));
  value.x = 1;
  value.y = 2;
  Var boxed = value;
  printf("%s\n", boxed);
  return 0;
}
```

`value.str()` calls `Vec.str`; `boxed.str()` calls a typed descriptor thunk that unboxes once and then calls `Vec.str`. Vec does not gain `add`, indexing, or any other omitted member. Removing `protocol Var(Vec);` removes protocol participation even though both conversion functions still exist.

Polar: a total forward view

Polar implements `magnitude` and inherits the `Cartesian.angle` member. The generated `Polar.angle` needs only the total forward view:

```
#include <math.h>

typedef struct Cartesian {
  double x;
  double y;
} *Cartesian;

typedef struct Polar {
  double radius;
  double theta;
} *Polar;

protocol Cartesian(T) {
  double T.magnitude(T);
  double T.angle(T);
}

Cartesian Polar.cartesian(Polar value) {
  Cartesian result = Scope.malloc(sizeof(struct Cartesian));
  result.x = value.radius * cos(value.theta);
  result.y = value.radius * sin(value.theta);
  return result;
}

double Cartesian.magnitude(Cartesian value) {
  return sqrt(value.x * value.x + value.y * value.y);
}

double Cartesian.angle(Cartesian value) {
  return atan2(value.y, value.x);
}

double Polar.magnitude(Polar value) {
  return value.radius;
}

protocol Cartesian(Polar);

int main(void) {
  Polar value = Scope.malloc(sizeof(struct Polar));
  value.radius = 2.0;
  value.theta = 0.5;
  printf("%.1f %.1f\n", value.magnitude(), value.angle());
  return 0;
}
```

No `Cartesian.polar` reverse conversion is required because this conformance generates only forward calls. A same-name incompatible `Polar.angle` would be a hard error at `protocol Cartesian(Polar);`.

Bytes: ordinary defaults and punctuation

The built-in manifest declares `protocol Block(Bytes);` centrally. The `Bytes.block` forward view is declared in `common.x` and implemented in `block.x`; no reverse `Block.bytes` adapter is needed. Members such as `len`, `truth`, `pop`, `free`, and `truncate` forward through that view.

```
int main(void) {
  Bytes bytes = Bytes.new(sizeof(int));
  int value = 7;
  bytes.push(&value);
  printf("%zu %d\n", bytes.len(), bytes ? 1 : 0);
  bytes.pop();
  bytes.free();
  return 0;
}
```

`bytes.truth()` and truth punctuation select the same ordinary `Block.truth` semantics. A scalar that adopts `Var(T)` without implementing truth or equality gains no static method, so its native punctuation is not replaced by `Var.fallback_truth` or `Var.fallback_equal`.

Native protocols

A native protocol maps members directly to C functions:

```
typedef void *Handle;

protocol void *(T) {
  void T.release(T) = free;
}

protocol void *(Handle);
```

Native participation is declared by the same bodyless adoption form; a matching typedef alone is insufficient. Each occurrence of `T` is checked in the C conversion direction that the alias uses. No named bidirectional pair is required.

The participant's translation unit emits a `_Generic`/`_Static_assert` signature check and an alias such as `#define Handle_release free`. A fully public adoption places them in the participant's generated header. If the protocol, participant, native target, or adoption is private, both remain in the participant's C file. Variadic native targets and unsafe conversion directions are rejected.

Classes and System Macros

The compiler ships a small set of macros for declaring ordinary types and managing block lifetimes. They use the same methods, protocols, allocation, and `defer` rules as handwritten x2c. Their definitions are available without an import; runtime operations still require their ordinary declarations, usually supplied by `#include "x2c.x"`.

Facility	Availability	Purpose
`class`	Shipped keyword alias	Declare a type and its applicable defaults
`$scope`	Shipped decorator	Retain a region or select a Scope destination
`$let`	Shipped decorator	Temporarily replace one captured storage location
`$lock`	Shipped decorator	Hold a Mutex while a statement runs
`$auto`	Shipped expression macro	Attach cleanup to an initialized local
`foreach`	Shipped keyword alias	Traverse through the ordinary Iter protocol
`with`	Contextual grammar	Substitute a source expression within a block
`$switch`, `$dedent`, `$assert`, `$todo`, `$unreachable`, `$time`	Shipped, explicit import	Remove boilerplate C leaves to every caller
Project macros	Explicit import	Supply project-specific syntax and policy

Only `class` and `foreach` have bare keyword aliases. Use the `$` spelling for `$scope`, `$let`, `$lock`, and `$auto`. For authoring your own macros, continue with Compile-time Macros.

Classes are ordinary named types

Use `class` to give a named type the supporting operations you would otherwise write yourself: construction, conversion to and from `Var`, comparison and hashing, and readable output. These operations let your values work with generic containers and protocol-based code. You choose the representation; the class supplies applicable defaults, which explicit methods can replace.

For example, a small value record can be constructed, used as a Map key, and printed without writing its constructor, converters, hash, or printer:

```
class Point { int x; int y; };
Point point = Point.new(3, 4);
Map labels = $auto({});
labels[point] = "origin";
printf("%s: %s\n", point.repr(), labels[Point.new(3, 4)]);
```

Which methods does a class supply?

For a class named `T`, the possible defaults are listed below. A new class means one declaring a representation, rather than an alias of an existing named type. Defaults depend on that representation; not every class generates every method.

Method	Default behavior and applicability
`T.alloc`	Returns zeroed Scope storage for a new pointer class with a record representation. Replace it when the object lives elsewhere.
`T.new`	Constructs a scalar, value record, or pointer object. When the class defines `init`, `new` takes the remaining parameters of `init`, calls it on storage from `alloc` (or a zeroed value), and returns the object. An alias forwards the applicable constructor of its underlying named type. Some layouts require an explicit constructor or `init`, as described below.
`T.free`	Releases a new pointer class's Scope allocation early. When the class defines `void T.drop(T)`, `free` first calls it on a non-NULL object. `free` accepts NULL.
`T.cleanup`	Calls the selected `free` for a new pointer class and supplies `Cleanup` participation for `$auto`.
`T.var`	Converts a new class to `Var`: a scalar uses its underlying representation, a pointer boxes identity, and a value record boxes a Scope-owned copy.
`Var.t`	Converts a `Var` back to the new class; for example, `Var.point` for `Point`.

Method	Default behavior and applicability
`T.equal`, `T.hash`	Use identity for new pointer classes and fields for supported flat value records. Other value records require compatible explicit methods.
`T.str`, `T.repr`	Provide string output for new pointer and record classes. Pointer `str` shows identity; value-record `str` delegates to `repr`. Record `repr` shows fields.
`T.write_str`, `T.write_repr`	Write the corresponding output into a Buffer for new pointer and record classes.

Scalar classes retain their underlying type's comparison, hashing, and output operations. Aliases of existing named types inherit supporting operations, including cleanup, rather than generating a second implementation. Explicit methods replace applicable defaults; the output methods can be customized independently, as described under Boxing and readable output.

`init` is not generated. Some constructors require you to supply it. Generated cleanup releases the object's allocation; ownership of resources stored in its fields requires your own cleanup code.

Choose a representation

The name comes first. The remaining declaration specifies its representation:

```
class Count int;
class IntPointer int *;
class ValuePointer Var *;
class Values Array;
class Point { int x; int y; };
class HeapPoint { int x; int y; } *;
```

`Point` is a value aggregate; the short braces abbreviate a struct definition. `HeapPoint` explicitly names a pointer representation; short braces accept the same trailing `*` as an explicit struct declaration. Prefer a value for local state without retained identity, and use `Point &` for a method that updates the caller's Point. A value may live in a local, a field, an array, or allocated storage. Its declaration does not select a storage location. `Values` follows the existing Array type chain and constructor; it does not acquire a second allocation layout. A forward declaration such as `class Point;` reserves the identity until its visible definition supplies the representation.

An enum cannot be a class's value representation, because a `Var` has no fixed tag for a type whose integer width C leaves to the implementation. Give the enum a typedef and name that typedef, as `class Tone Shade;` does for `typedef enum Shade { LIGHT, DARK } Shade;`. A pointer to an enum is an ordinary pointer representation and needs no typedef.

Construction and early release

A scalar class constructor accepts its scalar value. A class pointing to an eligible value allocates and copies one pointee. A flat record of supported value fields accepts positional arguments in field order:

```
class Point { int x; int y; };
class HeapPoint { int x; int y; } *;
Point value = Point.new(3, 4);
HeapPoint object = HeapPoint.new(3, 4);
printf("%d %d\n", value.x, object.y);
object.free();
```

The heap constructor allocates in the current Scope. Its generated `free` allows early release; otherwise the Scope reclaims that allocation. Generated cleanup does not recursively free fields and is not a Scope finalizer. Positional fields include native numeric values and enums, Symbol, Var/Atom, String, List, and their aliases. Var slots are shallow copies.

A class that defines `init` gets a constructor built from it. The constructor obtains zeroed storage, passes it to `init` with the constructor's arguments, and returns it. A heap class obtains that storage from `alloc`, whose default is a zeroed Scope allocation. A heap class's `init` receives its bare handle. Prefer a reference receiver for a value class's `init`; the existing pointer receiver is also accepted. `new` takes the parameters after that receiver in the same order:

```
class Counter { int value; };
void Counter.init(Counter &c, int initial) {
  c.value = initial;
}
static void Counter.step(Counter &c, int amount) => c.value += amount;
Counter counter = Counter.new(2);
counter.step(3);
printf("%d\n", counter.value);
```

```
5
```

The Counter record is a local value and its methods update that value through references. A record containing a mutable handle such as Array, Map, Iter, or another heap class requires `init`, as do nested records and arrays. A flat record that defines `init` uses it in place of its positional constructor.

A heap class that owns more than its own allocation defines `drop`. The generated `free` calls `drop` and then releases the object, so `drop` only releases what the fields own:

```
class History { int limit; Array items; } *;
void History.init(History self, int limit) {
  self.limit = limit;
  self.items = [];
}
void History.drop(History self) {
  self.items.free();
}
History history = $auto(History.new(8));
history.items.push(42);
```

A heap class's `init` may return `int` instead of `void`. When it returns zero, `new` releases the storage with `Scope.free` and returns NULL; `drop` is not called, so an `init` that returns zero must first release whatever it acquired. A class that replaces `alloc` so the object lives outside its Scope allocation also replaces `free`.

An explicit `new` replaces the default and may take different arguments. It also replaces the default constructor's obligation to call `init`. An explicit `free` replaces the default, and with it the call to `drop`. Explicit methods can appear later in the owning source; only the selected default or explicit method is emitted. Two ordinary definitions still conflict, and an importing consumer cannot replace a provider's defaults.

Derived classes forward an existing parent's constructor with its arguments and retain parent cleanup. For example, Array's constructor still chooses storage for Var elements. A variadic parent requires an explicit constructor when its arguments cannot be forwarded by an existing facility. A custom constructor that changes a heap class's allocation contract must also supply compatible cleanup. A failed `init` returns no object; it does not roll back arbitrary side effects inside the initializer.

Boxing and readable output

Heap classes box the existing object identity. A copied value aggregate boxes a Scope-owned copy and unboxes by value. The box never points at the original local's stack storage:

```
class Point { int x; int y; };
Point point = Point.new(3, 4);
Var first = point, second = Point.new(3, 4);
Map labels = $auto({});
labels[first] = "origin";
printf("%s\n", labels[second]);
```

Supported value fields receive compatible field-based equality and hashing, so independently boxed equal values work as the same Map key. Opaque layouts require explicit compatible methods. Equal values must have equal hashes. Heap classes retain identity equality and hashing.

`str` and `repr` are independently replaceable. A heap object's default `str` prints pointer identity. A new value aggregate's `str` delegates to `repr`; scalar and derived value classes retain their parent's printing. Record `repr` visits fields in declaration order, using readable value output and an address fallback for opaque fields. It does not dereference unknown pointers.

Nested Array, Map, List, and class rendering shares an active rendering path. A repeated identity on that path prints its pointer form; a repeated reference outside the active path prints fully again. Custom printers that recurse outside these operations must manage their own recursion. Readable output containing addresses is not a serialization format.

Class boxing uses the existing Var descriptor registry. Every record and heap class declares its descriptor at startup, and declaration freezes when worker startup freezes registration. A class takes a `Var` row only when a value of it is first boxed, which may happen on any thread. The first 30 custom tags boxed in a process get direct rows; these include classes and `protocol Var(T)` participants. Later heap classes box through a small process-lifetime cell, one per class and address, and later records carry their descriptor in front of the boxed copy, so a program can box any number of classes. Scalar and alias classes declare nothing. Class tags derive from the canonical source file and full name, keeping private classes in different files distinct. Tags use deterministic compact spellings, while diagnostics retain the full name. Tag collisions are errors.

Retain or select a Scope

Without an argument, `$scope()` retains one region around the following statement. Its expansion keeps the body inside the deferred release:

```
$scope() { Array values = []; values.push(1); }
```

The block placement is equivalent to:

```
{
  Scope.retain();
  {
    defer Scope.release();
    { Array values = []; values.push(1); }
  }
}
```

Place the decorator around a loop for one region, or around its body for one region per iteration. No hidden loop changes the meaning of `break` or `continue`:

```
$scope() for (int i = 0; i < 3; i++) { Scope.malloc(8); }
for (int i = 0; i < 3; i++) $scope() { Scope.malloc(8); }
```

Place it before a file-scope function definition to retain one region around that whole function, without indenting the body a further level:

```
$scope() static int distinct(List words) {
  Map seen = {};
  foreach (String word, words) seen[word] = 1;
  return seen.len();
}
```

The `Map` belongs to the retained region, and the region closes on every exit, so the `return` releases it. `$lock` and `$let` decorate a function the same way.

With one Scope-pointer argument, `$scope` evaluates the argument once, pushes that destination, and restores the previous destination on exit:

```
Scope destination = $auto(Scope.new());
$scope(&destination) { Scope.malloc(8); }
```

The corresponding body placement is:

```
{
  Scope.push(&destination);
  {
    defer Scope.pop();
    { Scope.malloc(8); }
  }
}
```

`pop` restores the destination; it does not release the selected Scope. `return` and error transfer use the ordinary defer boundaries in both forms.

Temporarily change one location

`$let(place, value)` captures the address of an assignable place, saves its previous value, installs the new value, and restores that same storage on exit. Both locating the place and computing the new value happen once:

```
int levels[2] = { 1, 2 }, index = 0;
$let(levels[index++], 7) {
  printf("%d\n", levels[0]);
  index = 1;
}
```

Changing `index` does not redirect restoration. The place's storage must outlive the body. Ordinary typing rejects const assignment or taking a bitfield's address. By comparison, `with` substitutes its source expression on each use and creates no temporary; it is not a saved-value binding.

Hold a Mutex

`$lock(mutex)` evaluates one Mutex expression, acquires it, and defers unlock until the decorated statement exits. Unlock is registered only after lock succeeds:

```
Mutex mutex = $auto(Mutex.new());
$lock(mutex) { printf("held\n"); }
```

This preserves Mutex's error behavior. It does not replace descriptor locks, file-stream locks, or a project's conditional-acquisition policy. Logger's private synchronized decorator keeps its own policy.

Clean up an initialized local

`$auto` is the complete initializer of an ordinary local declaration:

```
Array items = $auto([]);
Buffer text = $auto(Buffer.new(0));
items.push(42);
text.write(items.repr());
printf("%s\n", text.str());
```

Each declaration initializes the local and registers cleanup for the acquired value in the same enclosing block. The initializer runs once, and cleanup is registered only after it succeeds. Compound declarations preserve acquisition order, with each successful acquisition protected before the next initializer runs. Cleanup runs in reverse registration order.

`Cleanup(T)` declares `void T.cleanup(T)`. Participation is explicit, follows the existing typedef/protocol ancestry, and keeps ordinary method precedence:

Types	Cleanup action
Block, Bytes, Array, typed Arrays	Release the backing Block allocation
Map and typed Maps	Release the record and both backing Blocks
Buffer, Mutex	Their existing `free` operation
File, Context	Their existing `close` operation
Scope, Lisp	Their existing `destroy` operation
Generated heap classes	Their selected `free` operation
Other types	An explicitly adopted compatible cleanup method

Cleanup releases the acquired value even if the local is reassigned. Each `$auto` below protects its own Array:

```
Array current = $auto([]);
Array replacement = $auto([]);
current = replacement;
```

Both Arrays are cleaned up at block exit. Reassignment neither releases the first Array early nor transfers its cleanup to the replacement. Returning or storing an alias does not cancel cleanup; the caller must choose an appropriate lifetime. A managed local cannot be managed again: `$auto` of a local that an earlier `$auto` manages, or of another `$auto`, is a compile-time error, because the value would be released twice. There is no move or recursive ownership system. Var itself has no owning cleanup contract, and Pool's `free` operation requires a separate allocation argument.

Only an initialized automatic local that is a compound-statement item accepts `$auto`. Parentheses and macro construction preserve that position. Static or threaded storage, fields, assignment, returns, call arguments, for-header declarations, and wrappers nested inside arithmetic or conditionals do not supply this enclosing-block lifetime.

Include the rest of the set

`class`, `foreach`, `$scope`, `$let`, `$lock`, and `$auto` need no include. A second set ships in `lib/system-macros.x` and becomes available with one line:

```
#include "system-macros.x"
```

Give every case its own block and its own break

C makes you write the block and the `break` yourself, and forgetting either is a familiar defect. `$switch` takes the condition and the body, keeps each run of `case` and `default` labels where you wrote it, and turns the statements after a run into one block carrying that run's `break`:

```
int main(void) {
  int code = 2;
  String reached = "none";
  $switch(code)
  {
    case 1:
    case 2:
      String shared = "low";
      reached = shared;
    case 3:
      String shared = "three";
      reached = shared;
    default:
      reached = "other";
  }
  return reached == "low" ? 0 : 1;
}
```

Both cases declare `shared`, which plain C rejects without hand-written braces. A run that already transfers gets no added `break`, so a case ending in `return` stays free of unreachable code. Items that are not labels pass through untouched, including a declaration before the first case, a `goto` label, and a preprocessor directive. To fall through, nest an ordinary `switch`.

Write indented text without indenting the text

`$dedent` removes the indentation a block of text was written with. The prefix is the run of spaces and tabs opening the first content line, after one leading LF or CRLF newline is dropped, so indentation past that prefix survives and the block renormalizes as a unit:

```
int main(void) {
  String usage = $dedent(%"
    usage: report [options]
      --verbose   explain each step
      --quiet     report only failures
  ");
  return usage.startswith("usage:") ? 0 : 1;
}
```

A literal whose spelling carries no escape and no interpolation hole is dedented during translation and reaches C as a finished constant. Every other form calls `String.dedent` at run time, so an interpolated block means the same thing and costs one call. Use `String.dedent` directly on text that arrives from a file or an argument.

Report the check you actually wrote

`$assert` raises `<invariant>` carrying the failing expression as written and the source line it was written on. It transfers to a catch like any other x2c error, and `NDEBUG` does not remove it:

```
int main(void) {
  int limit = 50;
  String written = NULL;
  try $assert(limit > 0 && limit < 10);
  catch %(invariant (check ?check) (at ?where)): written = check;
  return written == "limit > 0 && limit < 10" ? 0 : 1;
}
```

`$todo` and `$unreachable` mark a path that is unwritten or excluded by the surrounding logic. Both raise `<invariant>` with a note and the same location detail, so an unfinished path reports where it is rather than returning something plausible.

Measure a statement in place

`$time` wraps its target in a monotonic clock pair and reports the duration on stderr. The caller includes `<time.h>`:

```
int main(void) {
  long total = 0;
  $time("sum")
  {
    for (int i = 0; i < 1000; i++) total += i;
  }
  return total == 499500 ? 0 : 1;
}
```

The report runs from a `defer`, so decorating a whole function body still reports when that body returns or a cause transfers out of it.

A complete resource-using program

This program collects line lengths from a file. The retained region owns all allocations; the managed locals close the file and release the mutable history before that region ends. The History method makes ownership of its Array explicit.

```
class History { Array lengths; } *;

void History.init(History self) {
  self.lengths = [];
}

void History.drop(History self) {
  self.lengths.free();
}

int main(int argc, char **argv) {
  if (argc != 2) return 0;
  $scope() {
    History history = $auto(History.new());
    File input = $auto(File.open(argv[1], "r"));
    for (String line; (line = input.readline()) != NULL; )
      history.lengths.push(line.len());
    printf("%s\n", history.lengths.repr());
  }
  return 0;
}
```

Compile-time Macros

For using the shipped `class`, `$scope`, `$let`, `$lock`, and `$auto` facilities, see Classes and System Macros. This chapter explains how to author macros, including the ordinary facilities those definitions compose.

x2c macros generate source at compile time using C-like x2c syntax. They bind parsed, typed source rather than preprocessor text. The compiler constructs, binds, and types their templates and arguments as it does ordinary source. Hygiene keeps generated names from colliding with names in the caller. A reusable global macro is invoked explicitly with a `$` name; a local macro uses a bare name inside its defining block. Both declare the syntax kinds they accept and produce one declared kind of source.

Choose according to what the code does:

- a function computes with runtime values;
- a meta function also computes during compilation;
- a macro constructs source at translation time;
- a decorator transforms the expression or source item immediately following it.

Start with templates. Use meta functions when the result requires computation.

A first expression macro

The result kind follows `macro`; typed holes appear in the invocation pattern:

```
macro Expression $project.minutes(Expr $value) =>
  $value * 60;
```

Read the definition from left to right:

Source	Meaning
`Expression`	the expansion must be one expression
`$project.minutes`	the explicit, qualified invocation name
`Expr $value`	bind one caller expression to `$value`
`$value * 60`	insert that expression into the replacement

The `$` marks macro syntax. It does not make a runtime variable dynamic. Macro names are in their own namespace, so qualification is useful for project and library names. The `x2c.*` namespace is reserved for compiler-shipped facilities.

The replacement preserves ordinary evaluation rules. If a hole appears twice, the caller's expression appears twice and may be evaluated twice. Use a generated temporary when the macro promises single evaluation, or use a function when that fits better.

Local expression shorthand

Use `with` when the repeated source is local to one braced block and does not need a reusable macro:

```
with point {
  _.x = 1;
  _.y = 2;
}
```

The optional `as name` form replaces _ with a chosen lexical name. Like a macro hole, every use substitutes the original expression and may evaluate it again; `with` does not create a runtime temporary. It declares no reusable generator and has no name outside its block.

Keep a generator inside one block

Use a local macro when one function or nested block needs a typed generator:

```
macro Expression scaled(Expr $input) =>
  $input * scale;
```

```
return scaled(value);
```

The definition is visible from that point to the end of the block and through nested blocks. An inner definition of `scaled` temporarily shadows it; a later definition in the same block replaces it for following calls. A name the body reads without declaring, such as `scale`, resolves where the macro is invoked, as in a global macro, unless the macro lists it with `using`. Names the caller writes in arguments resolve there too.

The bare call shape belongs to the local macro before a file `keyword` alias or ordinary function. A use without parentheses is an ordinary identifier, and `(scaled)(value)` explicitly calls a function. `$scaled(value)` uses only a global or imported macro, so local and global definitions may share a name.

Local expression, statement, field, `Map`-entry, and enumerator generators keep their usual result positions. Local decorators are available when their target also occurs inside the block. `Unit` generators and decorators over functions or translation-unit items remain global because they have no invocation position inside the local name's lifetime. A macro may generate a local definition; it has the same source-order visibility and capture behavior as a direct one.

`with` remains the smaller choice when the only need is repeated substitution of one expression. A local macro is worth defining when it needs typed holes, declarations, control flow, computed source, or more than one input.

Holes say what binds

Every hole has a syntax kind:

Hole kind	Accepts
`Expr`	an expression
`Name`	an identifier binding
`Type`	a type
`Param`	a function parameter
`Function`	a complete function definition
`Decl`	one declaration without its trailing semicolon
`Stmt`	one block item
`Field`	one struct or union field
`Entry`	one `Map` `key: value` row
`Enumerator`	one enum member
`Unit`	one translation-unit item
`Literal`	one literal, such as `42` or `<char>`

The reference lists every hole kind, including the less common `NamedType`, `DeclaratorRow`, `Catch`, `Captures`, and `MatchRow`.

An `@` prefix makes the final hole a sequence. The replacement uses the same prefix to splice the captured sequence:

```
macro Expression $project.call(
  Expr $callee,
  Expr @arguments
) =>
  $callee(@arguments);
```

A `raise` template can also insert expression holes for the error code and field keys. A sequence hole between field pairs supplies alternating key and value expressions, in source order:

```
macro Stmt $fail(Expr $cause, Expr $op, Expr @fields) {
  raise %($cause (operation ${$op}) @fields);
}
```

Here `${$op}` inserts the captured x2c expression into the literal payload. The holes are substituted before the ordinary raise expression binding and lowering; they do not evaluate Lisp code.

Typed holes tell the reader whether an argument is an expression, binding, type, declaration, or sequence, without requiring knowledge of the AST.

Type holes are compile-time generics

A `Type` hole lets one `Unit` macro generate the same typed interface for several concrete C types. Each invocation is checked independently and emits ordinary declarations and functions specialized to the types it receives:

```
macro Unit $value.family(Type $box, Type $item) {
  static inline $box $box.new($item value) {
    $box box = { value };
    return box;
  }

  static inline $item $box.get($box box) {
    return box.value;
  }
}

$value.family(IntValue, int);
$value.family(DoubleValue, double);
```

Each expansion is a distinct C type. `IntValue.get` returns `int`, `DoubleValue.get` returns `double`, and the generated C has no runtime type parameter or hidden cast. The caller supplies named concrete types, and the macro builds their shared implementation.

Result kinds match source positions

The declared result kind determines where an invocation may appear:

Result kind	Invocation position
`Expression`	expression
`Stmt`	statement
`Field`	struct or union body
`Entry`	`Map` literal row
`Enumerator`	enum body
`Unit`	translation unit
`Decorator`	as a prefix immediately before its target

An `Entry` macro can produce zero, one, or several comma-separated `Map` rows. `Entry` holes capture complete rows, and `@rows` captures or inserts a row sequence. A `Map` literal reads quoted data, so `${...}` crosses into x2c before invoking either a `$handler(...)` or a keyword alias:

```
macro Entry $project.handler(Literal $key, Expr $value) {
  $key: $value
}

int main(void) {
  int open = 1, close = 2;
  Map handlers = %{
    ${$project.handler("open", open)},
    ${$project.handler("close", close)}
  };
  return handlers.len() == 2 ? 0 : 1;
}
```

`Stmt` macros are useful for a small, repeated control-flow shape:

```
macro Stmt $project.guard(Expr $condition) {
  if (!$condition) return 0;
}
```

A `Stmt` macro whose body is one expression statement, or one invocation of another `Stmt` macro, can write it after `=>`, like an expression macro:

```
macro Stmt $project.note(Expr $value) =>
  printf("%s\n", %"value ${$value}");
```

That expression may also be a call to a `meta` function that returns a statement, as shown under Compute with meta functions.

An expression macro is not a statement macro, and a unit macro cannot appear inside a function. The compiler diagnoses the mismatch at the invocation.

Private names

Declare scratch bindings normally. A declaration written literally in a macro body and every literal reference to it receive a private binding for each expansion:

```
#include "meta.x"
meta static List project_type(TypeInfo type) => type.assoc(<type>);

macro Stmt $project.swap(
  Expr $left,
  Expr $right
) {
  $project_type($left) temporary = $left;
  $left = $right;
  $right = temporary;
}
```

The helper returns a type into the declaration's type slot. The same explicit call form also supplies expressions and generated names in their own slots.

Each invocation receives a compiler-private binding for `temporary`, so a name the body declares never collides with caller source.

Free names resolve where the expansion lands

A name the body reads without declaring means what it would mean if the expansion were written by hand at the call site. A caller's declaration of that spelling supplies it, including a local:

```
static int scale = 3;

macro Expression $scaled(Expr $value) => $value * scale;

int main(void) {
  printf("%d\n", $scaled(2));
  {
    int scale = 10;
    printf("%d\n", $scaled(2));
  }
  return 0;
}

6
20
```

The second call reads the block's `scale`, as `2 * scale` written there would. Names the caller writes in arguments resolve at the call site too. A macro that should reach a caller's variable by design takes the name as a `Name` argument, as `$repeat` does in the reference.

Keep a file-scope name with `using`

A macro that must reach a particular file-scope declaration, whatever the caller declares, lists that name with `using`. An `Expression` macro writes the list after its signature. A braced body writes it as its first item:

```
static int scale = 3;
static int bump(int value) => value + 1;
```

```
macro Expression $scaled(Expr $value) using scale => $value * scale;

macro Stmt $bumped(Name $target) {
  using bump;
  $target = bump($target);
}

int main(void) {
  int scale = 10, bump = 0;
  printf("%d\n", $scaled(2));
  $bumped(bump);
  printf("%d %d\n", scale, bump);
  return 0;
}

6
10 1
```

Each listed name keeps the declaration that file scope supplies where the macro is written: a function, object, or enumerator of that file or of a header it includes, or a native name such as a C library function. The expansion reads that declaration with its type. A caller's local of the same spelling receives a separate generated C spelling, so the expansion still reaches the file-scope name. Only names, not types, can be listed; a type name in a body always resolves where the expansion lands.

List every helper that a macro calls when the macro is meant for other files. A helper reached from another translation unit must be nonstatic, or `module` within a module unit, because a `static` function is private to its own unit.

Generated names with `using $name`

Use a leading `using $name;` directive only when compile-time Lisp or a nested macro needs a private `Name` hole before an ordinary declaration can introduce it:

```
macro Unit $project.generated() {
  using $private;
  @project_generate($private)
}
```

The directive creates syntax identity, not runtime storage. It can therefore name a generated helper function or label as well as an automatic variable. All `using` directives must precede the body's ordinary items.

The three spellings of `using` do different things:

Spelling	Where	Effect
`using name`	macro signature or first body item	keep the file-scope declaration of `name` where the macro is written
`using $name`	first body item	create a fresh private name for each expansion
`using &name`	lambda	share a captured local by reference

Put reusable macros in ordinary source modules

An ordinary `.x` file may hold runtime code, macro definitions, keyword aliases, meta functions, and top-level compile-time Lisp. Include the module wherever its public definitions are needed:

```
#include "project-macros.x"
```

Nonstatic definitions become available at the include line, including through transitive includes. Use `static macro` or `static keyword` for helpers that belong only to the declaring file. Ordinary includes are tracked translation dependencies and each canonical file contributes once per unit; include cycles terminate without loading a second copy.

`$(import "helpers.xlisp")` remains the loading form for Lisp files. A Lisp import executes in the translation-unit Lisp session and rejects import cycles.

A package publishes the nonstatic macros and aliases its entry source and ordinary includes define. `import "name";` makes that syntax available at the import position.

Prefer a qualified name such as `$test.run` or `$project.logging.trace` in a shared module. It identifies the project or library at each call and avoids ambiguous short global names.

Decorators transform one target

A decorator receives the following expression or source item as its required first parameter. Invocation arguments follow normally:

```
macro Decorator $project.trace(
  Function $function,
  Expr $label
) {
  printf(
    "[%s] enter %s\n",
    $label,
    $Code.name($function)
  );
  defer printf("[%s] leave\n", $label);
  @Code.body($function)
}

$project.trace("request")
int answer(int value) {
  return value * 2;
}
```

`$Code.name` and `@Code.body` receive the complete captured function. `Code.name` returns its name, and `Code.body` returns its body as the sequence that `@` inserts.

There is no semicolon after the decorator invocation. Decorators stack closest-first, and each expansion must leave exactly one target for the next. A function decorator preserves the function's original name, storage, signature, and method ownership.

A statement macro or statement decorator may be used directly as an `if`, `else`, or loop body when its production contains exactly one statement. An explicit compound statement or `do { ... } while (0)` counts as one; an unwrapped sequence of block items does not. The compiler never inserts braces, so the production still decides scope and control flow.

An `Expr` target makes the decorator a prefix expression:

```
macro Decorator $project.checked(Expr $target) =>
  checked($target);
```

Expression decorators bind like a prefix unary operator. They capture the following primary/postfix, unary, or cast expression before surrounding binary, conditional, assignment, or comma operators. Parenthesize a larger target: `$project.checked() value + 1` checks `value`, while `$project.checked() (value + 1)` checks the complete addition. Chained expression decorators remain closest-first.

Give macros local keyword spellings

A `keyword` declaration removes the qualified `$` spelling while preserving the macro's existing invocation grammar. This makes a parameterized decorator look like a new control construct without adding a new parser for its header:

```
macro Decorator $control.range(
  Stmt $body,
  Name $index,
  Expr $start,
  Expr $stop
) {
  {
    int begin = $start, end = $stop;
    for (int $index = begin; $index < end; $index++) $body
  }
}

keyword range $control.range;
```

```
range(index, 0, 3) {
  consume(index);
}
```

`range(index, 0, 3) TARGET` is exactly `$control.range(index, 0, 3) TARGET`. The macro still handles the typed `Name` and `Expr` arguments, hygiene for its private bounds, and parsing of the following block.

Ordinary macros can use the same declaration. Their parentheses remain mandatory, even when they take no arguments:

```
#include "meta.x"
meta static List project_type(TypeInfo type) => type.assoc(<type>);

macro Stmt $control.swap(
  Expr $left,
  Expr $right
) {
  $project_type($left) temporary = $left;
  $left = $right;
  $right = temporary;
}

keyword swap $control.swap;

  swap(left, right);
```

`Decorator` arguments use the same comma-separated typed and sequence parameters as direct macro calls. A decorator with no explicit arguments uses `ALIAS TARGET`; one with arguments uses `ALIAS(arguments) TARGET`. An all-sequence decorator uses `ALIAS() TARGET` when the sequence is empty, so a parenthesized target of a zero-argument decorator remains unambiguous.

A `Decl` argument ends at the invocation's comma or closing parenthesis, so its trailing semicolon is omitted. It accepts one declarator with an optional initializer, or flat destructuring with two or more simple identifier targets. For example, `foreach(int value, values)` supplies `int value` as one `Decl` argument.

The declaration and uses are source ordered. A public alias in an included module becomes available at that include; a `static keyword` declaration stays in its source file. A module may keep an alias beside its macro:

```
#include "compiler-keywords.x"

swap(left, right);
```

A later alias does not reinterpret an earlier macro template. Use `static keyword swap $swap;` when the short spelling is private, while the qualified macro remains public.

Compute with meta functions

Write compile-time calculations as ordinary x2c functions marked `meta`. A macro calls one with `$helper(...)`; its captured holes supply code rather than the future runtime values of those expressions:

```
meta static int project_offset(void) => 2;

macro Expression $project.answer(Expr $base) =>
  $base + $project_offset();

int main(void) {
  printf("%d\n", $project.answer(40));
  return 0;
}
```

```
42
```

Include `meta.x` when the calculation needs compiler operations for diagnostics, identifiers, types, bindings, source text or function bodies. The meta-function guide introduces these operations; the language reference specifies them.

A helper can also receive a captured statement, a description of a type (`TypeInfo`), or the exact source text of an argument (`Source`). Generating Code shows each.

Prefer source templates for generated code and meta functions for the calculations that supply their arguments. Use the optional `$(...)` Lisp entry when integrating existing Lisp helpers or changing the compiler's Lisp session. For example, `$(project_offset)` calls the helper above from Lisp. Local macro names are lexical; Lisp definitions instead remain available for the rest of the translation unit after the defining block ends.

Generate several views from one table

A meta function can keep one table of facts from which small macros generate enums, lookup tables, switch cases, declarations, and repetitive functions. Changing the table then changes every generated form.

Use a table when the outputs repeat the same facts. Write names, types, and status values explicitly so a reader can understand the generated code without working through the generator. Keep unrelated outputs as direct source.

Declare methods and converters before adoption

Shallow collection loads included macro definitions and expands file-scope `Unit` macros. It evaluates preceding top-level Lisp contributions when the producer needs them. Generated private helpers stay available in their source module. Public declarations and protocol rows reach includers. The full parse binds the retained result without running its producer again.

A generated public function definition can complete a preceding prototype in the same expansion when their canonical signature and ownership agree.

Put converter and public method prototypes before an adoption so shallow collection can classify the conformance and downstream code can discover the functions. Included macro definitions survive intervening preprocessor includes, so a definition can be included before the headers its invocations need.

Choosing the smallest tool

Use:

- a function for runtime computation;
- a meta function for compile-time calculation and compiler queries;
- a protocol for explicit participation in shared typed behavior;
- a macro for repeated source whose invocation has one obvious expansion;
- a decorator for uniform policy on one following expression or source item;
- direct source when generation would hide more than it removes.

Count the macro's definition as well as its calls: a useful macro reduces the total, preserves useful diagnostics, and makes each binding clear.

For a complete generator built from these pieces, see Automatic Differentiation: a `Type`-hole family for dual numbers, and `Unit` decorators whose meta functions transform a function's typed code into its derivative.

For the complete hole grammar, result validation, hygiene rules, limits, and compiler Lisp SDK, see Compile-time macros in the language reference.

Meta Functions

x2c extends C. A `meta` function uses the x2c backend and the host compiler selected by `--meta-cc`. It runs as native code during translation, and its body is ordinary x2c: anything the unit can compile, it can run during translation. Two questions remain, both at the boundary between the program and compile-time code:

1. Which arguments can cross: a `$helper(...)` call passes constants, captured syntax, or the results of other `$` calls.
2. Which results can become program code: returning a value to another meta function and inserting it with `$helper(...)` have different limits.

`meta` is not a purity annotation. A body can mutate locals, arrays, maps and structs, and pass local addresses to other meta functions. Its compile-time objects belong to the compiler; they are not the objects the eventual program allocates.

Start with an ordinary function. This one takes an integer and returns an integer, using the same braces, `return`, and arithmetic as C:

```
int poly(int n) {
  return n * n + 3 * n + 1;
}
```

Calling `poly(7)` gives 71. Now put `meta` before its definition:

```
meta int poly(int n) {
  return n * n + 3 * n + 1;
}
```

The body has not changed. `meta` makes it available to the compiler during translation as well as to the finished program. You can use it for an ordinary calculation; it does not have to inspect types or generate code.

A `meta` function with a body can sit beside the code that calls it, as in the samples of this chapter, or in an ordinary `.x` module that several units include; see Sharing a `meta` function between units.

Call it in the program

An ordinary call uses parentheses and commas, just as before. Here the argument is a local variable:

```
meta int poly(int n) {
  return n * n + 3 * n + 1;
}

int main(void) {
  int n = 7;
  printf("%d\n", poly(n));
  return 0;
}
```

```
71
```

This uses the function's run-time form. The compiler emits C for the body, and the finished program calls it.

Ask the compiler to calculate it

Use `$poly(7)` to call the same function during translation:

```
meta int poly(int n) {
  return n * n + 3 * n + 1;
}

int main(void) {
  printf("%d\n", $poly(7));
  return 0;
}
```

```
71
```

The `$` prefix requires the compiler to run the function and insert its result. The call uses the same parentheses and commas as an ordinary x2c call. Here it inserts the integer 71 before the program is built.

The optional Lisp equivalent is `$(poly 7)`. It calls the same implementation; use it when working with Lisp code already in the compiler session.

These are the two forms of a `meta` function: one runs in your program, and one runs in the compiler. Both come from the same body. A rewrite rule in your program changes only the program's copy; the copy the compiler runs is compiled as written. Functions that call a compiler query or build a source template, directly or through another meta function, are exceptions: they only run during translation. We will reach those after ordinary calculations.

Compute the arguments too

An explicit call can calculate its arguments before invoking the function:

```
meta int twice(int n) => n * 2;

int main(void) {
  printf("%d\n", $twice(3 + 4));
  return 0;
}
```

```
14
```

The arguments must be resolvable during compilation. They may be computed expressions and calls to other meta functions, rather than only literals. An unresolved runtime input is an error; the explicit call cannot fall back to runtime execution. An existing macro with the same name takes precedence. Inside an executing meta body, arguments use that evaluation's current values.

Dynamic runtime calls

A meta function can be stored in a `Func` and called with runtime arguments:

```
meta int poly(int n) => n * n + 3 * n + 1;

int main(void) {
  Func calculate = poly;
  printf("%d\n", calculate(7));
  return 0;
}
```

```
71
```

This dispatches the program's compiled implementation, not the copy the compiler runs during translation.

Typed Func calls during compilation

Inside a meta body, a `Func` retains its callable signature through parameters, returns and collection storage. Lambdas and available named functions use the same argument checks as native calls. For example, an `unsigned char` parameter receives 1 from 257, and a `float` parameter rounds 16777217 to 16777216. A typed result converts before it is boxed as `Var`.

The callee is evaluated once. Arity is checked before argument evaluation; arguments are prepared once from left to right, then the adapter checks and converts them in that order. A reference parameter takes the live address of the caller's object without reading it, including an output-only local:

```
meta int write_answer(int n) {
  int output;
  Func write = %!(int &value) => value = 41;
  (void) write(output);
  return output + n;
}
```

`$write_answer(1)`, `write_answer(1)` and a runtime call with argument 1 all produce 42. References to locals, fields, dereferenced pointers and C-array elements share the original storage. Passing one object twice or forwarding a reference preserves immediate aliasing. Leading qualifiers may be strengthened; the remaining source type must match exactly. An invalid lvalue, null address, different underlying type or discarded qualifier raises `bad-types`. A concrete typed value parameter rejects `void`; a `Var` parameter carries it.

Ordinary calls run in the program

A call without `$` always calls the compiled function, even when every argument is a literal. `poly(7)` stays a call in the generated C; write `$poly(7)` to have the compiler calculate it.

Use x2c conveniences

The body can also use familiar x2c conveniences. An expression-bodied function uses `=>` in place of braces and `return`:

```
meta int poly(int n) => n * n + 3 * n + 1;
```

A `String` parameter gives access to string operations through dot syntax:

```
meta static int width(String text) => text.len() * 2;

int main(void) {
  printf("%d\n", $width("abcd"));
  return 0;
}
```

```
8
```

`static` follows `meta` and keeps its usual meaning for the emitted function. A body can use local variables, ordinary control flow, `foreach`, and `String`, `List`, `Array`, and `Map` operations with compile-time bindings. The restrictions below describe where this stops.

When `meta` is a keyword

`meta` is contextual. Besides functions, it can mark the file-static values and protocol adoptions described below. Outside those declaration forms it remains an ordinary identifier:

```
List meta = %(a b);

struct MetaHolder { int meta; };

int main(void) {
  meta = %(a b c);
  struct MetaHolder holder = { .meta = 3 };
  printf("%d %d\n", meta.len(), holder.meta);
  return 0;
}
```

```
3 3
```

A bodyless `meta` prototype declares a native C function for compile-time code, and `meta native` before a definition does the same for the function it defines. Native C functions below describes which functions the compiler provides.

At file scope, `meta` can also advertise one initialized static value to compile-time code:

```
meta static const double pi = 3.1415;
meta static int compile_counter = 0;
```

The emitted program keeps the ordinary C declarations and initializers. Compile-time code gets separate per-translation-unit values built from the same source initializers, so compile-time mutation never changes the eventual program's object. An explicit dollar call reads and writes the compile-time values; an ordinary call reads the program's. Each value lives in bytes the compile-time session owns, so taking its address and reading

or writing through a correctly typed pointer has the same aliasing effect as in C. `const` prevents compile-time writes.

`meta` marks functions, values and protocol adoptions, not types. Compile-time code can use any type the compiler sees: scalars, typedefs, classes with a `Var` representation, and any complete struct, including an anonymous inline `struct { ... } value`. A struct keeps its C layout; see C objects during compilation. Elsewhere `meta` remains an ordinary identifier.

A type's methods are not callable at compile time on their own. Each function needs its own `meta`:

```
struct Cell { int value; };
typedef struct Cell Cell;

meta int Cell.twice(Cell cell) => cell.value * 2;

meta static int doubled(void) {
  Cell cell = { 21 };
  return cell.twice();
}

int main(void) {
  printf("%d\n", $doubled());
  return 0;
}
```

```
42
```

Without `meta` on `Cell.twice`, the definition of `doubled` reports `no binding for Cell_twice`.

Native C functions

A bodyless `meta` prototype binds to the function of the same name that the compiler itself links:

```
meta double sin(double);
```

Compile-time code that calls `sin` runs the compiler's native copy. The declared signature must match that function exactly. A mismatch is reported at the declaration:

```
sample.x:1:1: type: native meta function declaration does not match its target
  meta float sin(double);
  ^^^^
  note: name: sin signature: ((func ((double))) float)
```

Two kinds of function follow a rule. An iterator operation takes its destination last: its last parameter and its result are `Iter`. Compile-time code may omit that destination; the call then allocates one with `Iter.new` before calling the native operation. A declared `Func` parameter matches any `Var` parameter of the compiler's target, which receives the compile-time callable and adapts it.

A `meta` protocol adoption, such as `meta protocol Iter(List);`, declares each witness of that conformance the way a bodyless prototype would.

Native definitions

A function that has an x2c body can be declared native where it is defined. Write `native` after `meta`:

```
/** Returns the number of set bits in `value`. */
meta native int bits(unsigned value) {
  int count = 0;
  for (; value; value &= value - 1) count++;
  return count;
}
```

This means the same as a bodyless `meta` prototype followed by the ordinary definition. The body is compiled for the program and never staged with the unit's `meta` group. Compile-time code that calls the function runs the compiler's native copy, and the signature must match that copy just as a prototype's must. `native` applies only to functions; before any other declaration it is an error:

```
sample.x:1:1: parse: a native meta declaration must be a function
  meta native static int value = 3;
  ^^^^
```

`native` is a marker only directly after `meta` and before a declaration. Elsewhere, including as a type name after `meta`, it is an ordinary identifier.

Keep the bodyless prototype for a function that has no x2c body, such as `sin` from the C library.

The compiler checks a `meta` body's storage against what each native callee allocates and keeps; see Regions. The runtime's own functions state this in a table. For any other native function, the compiler infers it from the signature, where a handle is any type other than a scalar, `Var`, `Symbol`, `String`, `List`, `Array`, `Map` or `Func`. A parameter that points at one of those, such as a `const char *` or an `int *` result, is not a handle.

The compiler links every function declared in `lib/cmath.x` and `lib/clibc.x`. Both are part of the implicit prelude, so their functions need no declaration of your own:

- `lib/cmath.x` declares all of C99 `<math.h>`, in both the `double` and `float` forms. This includes functions that return a second result through a pointer, such as `frexp`, `modf` and `remquo`.
- `lib/clibc.x` declares `abs`, `labs`, `llabs`, `atoi`, `atol`, `atoll`, `atof`, `strcmp` and `strncmp`.

The `<ctype.h>` functions are not included, because a C library may define them as macros. A compile-time `String` passes to a `const char *` parameter, so `$atoi("42")` answers 42.

The compiler also links the pure text operations of three optional modules: `Json.parse`, `Diff.lines`, `Diff.unified`, and `Path.join`, `dirname`, `basename`, `extension` and `stem`. Include `json.x`, `diff.x` or `path.x` to call them from a `meta` body. They give the same results as at run time. The Path operations that read the filesystem are not available.

A pointer argument can be the address of a compile-time local. The native function writes through it, and the caller reads the result afterward:

```
meta static double parts(double x) {
  int exponent;
  double mantissa = frexp(x, &exponent);
  return mantissa + exponent;
}

int main(void) {
  printf("%g %g\n", $parts(8.0), $sin(0.0));
  return 0;
}
```

```
4.5 0
```

A prototype for a function the compiler does not link is accepted. A meta function that calls it is diagnosed where it is defined:

```
sample.x:2:1: macro: this function cannot run at compile time
  meta static int roll(void) => rand();
  ^^^^
  note: reason: no binding for rand
```

Your own native functions come from a native module.

A native function binds into the compile-time session the first time compile-time code calls it. `sin(1.0)` stays a call in the generated C; write `$sin(1.0)` to compute the value during translation.

Native modules

The project's bodied `meta` functions need no module of your own: the compiler builds the project meta module from them and caches it, as How a meta function runs describes. A native module is built ahead of time instead, by you or by a package. It lets compile-time code call functions that are not part of the project meta module, and it can be shared between projects. Define each function with `meta native`:

```
meta native int triple(int x) { return 3 * x; }
```

Save that as `helpers.x` and build it as a module:

```
x2c build --kind meta-module --output helpers.so helpers.x
```

The module contains every function that a `meta native` definition or a bodyless `meta` prototype in its own sources declares. Code that includes the declaration can call the function during translation when the compiler loads the module:

```
#include <stdio.h>
#include "helpers.x"

meta static int nine(void) => triple(3);

int main(void) {
  printf("%d %d\n", $nine(), triple(5));
  return 0;
}

x2c run --native-module helpers.so helpers.x main.x

9 15
```

`$nine()` runs the module's `triple` during translation. The ordinary call `triple(5)` uses the copy linked into the program, so `helpers.x` is also one of the program's sources. `--native-module` works with `translate`, `build`, `run`, and `repl`, and the REPL accepts a bodyless prototype for the same function.

A module function may take or return a handle:

- A returned handle is a fresh allocation in the active `Scope`. Allocate it with `Scope.malloc`, so that compile-time code can release it with `Scope.free` or hold it in an `$auto` local.
- A handle argument is borrowed for the call. Keeping it after the call returns is not supported.
- A handle whose type is a runtime class, or adopts `Var` or `Cleanup`, is owned by its cleanup wherever it appears.
- A function that takes a handle and returns one, when neither is owned by its cleanup, might return or keep its argument. Its ownership cannot be inferred, so its prototype is rejected:

```
helpers.x:3:1: type: unproved native meta lifetime
  meta Widget widget_wrap(Widget inner);
  ^^^^
  note: name: widget_wrap signature: ((func (("Widget"))) "Widget") it might return or keep its
argument; ownership cannot be inferred
```

The compiler loads a module only when an option, a manifest or an import names it. Without `--native-module`, `nine` reports `no binding for triple`. A package with a compile-time part builds its module with the package, and importing the package loads it. In a manifest, a target lists the module targets its translation loads:

```
[target.helpers]
kind = "meta-module"
sources = ["helpers.x"]

[target.app]
sources = ["helpers.x", "main.x"]
native-modules = ["helpers"]
```

A module target builds before the targets that load it, and a changed module retranslates them. A target binds only the modules it names, even when an earlier target in the same build loaded others.

A module's functions bind like the functions the compiler links. The prototype's signature must match the one the module was built from, and a mismatch is reported at the declaration. A function the compiler links takes precedence, and the prototype reports a warning that the compiler's own function hides the module's. When two named modules define the same name, the one named first supplies it, and the prototype reports a warning. Modules named by an option or a manifest come before the modules of imported packages.

A module runs inside the compiler and uses the compiler's own runtime, not a copy of it. Only the compiler that built a module can load it, so a module must be rebuilt after the compiler changes. The compiler checks this before it loads any of the module's code:

```
x2c: error: native module 'helpers.so' was built by another compiler; rebuild it
```

Every function the module exports needs a `meta native` definition or a bodyless `meta` prototype in the module's sources, and a module whose sources declare none fails to build.

A module can call any runtime function, because the compiler links the whole runtime. This holds for a compiler built from a checkout and for one `make install` installs. A loaded module stays loaded until the compiler exits. Native modules work on macOS, Linux and WSL. On other platforms, loading one reports that native modules are not supported.

A package's compile-time part can instead be linked into the compiler itself, which is the only route where modules do not load. `x2c build --extension <dir>`, given the compiler's sources, translates the package's sources into the compiler and registers its `meta` functions under the package's name. Any number of packages can be linked this way. Importing such a package loads no module and needs no `builds/<name>.module`; its functions come after the modules an option or a manifest names.

C objects during compilation

Compile-time code keeps C objects the way the program does, because it is the same C. Structs, unions, bitfields, arrays, enums of any width, local statics, pointers, and `goto` all behave as C defines them. Struct assignment copies bytes into the destination's existing storage, so an address taken earlier stays valid. Passing a struct by value gives the callee its own copy:

```
struct Pair { int x; int y; };

meta static struct Pair pair(int x) {
  struct Pair p = { x, x + 1 };
  return p;
}

meta static void bump(struct Pair *p) { p->x += 10; }

meta static int spare(struct Pair p) {
  p.x = 0;
  return p.y;
}

meta static int pairs(void) {
  struct Pair a = pair(1);
  int *ax = &a.x;
  a = pair(5);
  bump(&a);
  return *ax * 100 + spare(a) + a.x;
}

int main(void) {
  printf("%d %d\n", $pairs(), pairs());
  return 0;
}
```

```
1521 1521
```

`ax` still points into `a` after the assignment, so `*ax` reads 15. `spare` changes its own copy and returns 6, and `a.x` is still 15.

A compile-time call frees its locals and parameters when it returns. A `meta` function whose body lets the address of one outlive the call is rejected where it is defined, with an error at the statement where the address leaves: a `return`, a store into a `meta static`, or a store through a parameter or through a pointer whose target the compiler cannot identify. A local array counts as the address of its first element. The check follows the address through pointer locals, either arm of `?:`, and the `meta` functions defined before the caller:

```
struct Box { int value; };

meta static int *field_of(struct Box *box) { return &box->value; }

meta static int *leak(int seed) {
  struct Box box = { .value = seed };
```

```
  return field_of(&box);
}
```

`field_of` is accepted, because the object it borrows from belongs to its caller. `leak` is rejected at its `return`, because `field_of` hands back an address inside `box`.

The check is the region check of ordinary code, and every finding it makes in a `meta` body is an error, including a value from a `Scope` region that outlives the region. In ordinary code the same findings are warnings; see The Region Model. As in C, an address the check does not follow, such as one computed with pointer arithmetic, is undefined behavior once its storage ends.

A struct, a C `bool`, or a pointer stays inside compile-time code: another meta function can take it, but a `$` call cannot pass one in or insert one into the program. Such a call reports `this function cannot run at compile time` with a reason that states the kind of value: a struct or union result, an address result, or a parameter or result with no Var form.

Heap objects during compilation

`Scope.malloc`, `Scope.calloc`, `Scope.memdup`, `Scope.realloc`, and `Scope.free` work as they do in the program, and `sizeof` is C's. A meta function can therefore build structs on the heap: a linked list, an array of structs, or a buffer that grows. `p->f`, `p.f`, `*p`, and `p[i]` read and write the heap bytes, and a pointer plus or minus an integer moves by whole elements, as in C.

```
struct Node { int value; struct Node *next; };

meta static struct Node *push(struct Node *head, int value) {
  struct Node *node = Scope.malloc(sizeof *node);
  node->value = value;
  node.next = head;
  return node;
}

meta static long heap_sum(int count) {
  struct Node *head = NULL;
  for (int i = 1; i <= count; i++) head = push(head, i);
  long *squares = Scope.calloc(1, sizeof(long));
  int size = 0;
  for (struct Node *at = head; at; at = at->next) {
    squares = Scope.realloc(squares, (size + 1) * sizeof *squares);
    squares[size] = at->value * at->value;
    size += 1;
  }
  long total = 0;
  for (long *at = squares + size - 1; at >= squares; at = at - 1)
    total += *at;
  Scope.free(squares);
  while (head) {
    struct Node *next = head->next;
    Scope.free(head);
    head = next;
  }
  return total;
}

int main(void) {
  printf("%ld %ld\n", $heap_sum(4), heap_sum(4));
  return 0;
}
```

```
30 30
```

The heap belongs to the compiler while it translates. A heap pointer can pass between meta functions, but like any pointer it cannot cross `$`. Return a value computed from the heap objects instead.

The region check treats `Scope.realloc` as the end of the pointer it is given, as `Scope.free` is. In a `meta` body, a read through a pointer after `Scope.free` or `Scope.realloc` ended it is an error, and so is freeing or reallocating a literal, a local's address, or other storage no Scope allocator returned.

How a meta function runs

A translation runs in two phases. Before any unit is translated, the compiler gathers the bodied `meta` functions the inputs reach: those of each ordinary source module an input reaches, directly, through includes, or through a package. It emits them, with the declarations they use, as C through the ordinary backend, compiles them with the host C compiler (`--meta-cc`, default `cc`, never the target `--cc`), and links them with the host runtime into one helper program, the project meta module. A group sees the headers its unit includes, found from the unit's directory and the `-I` directories. The translation then sends each `$` call, and each compile-time Lisp call, of one of those functions to the helper and inserts the reply. A project whose inputs reach no `meta` function builds nothing extra.

A call runs with the functions of the file that wrote it. An included `.x` file computes its file-scope constants, such as an enumerator or an array size, with its own `meta` functions, as its own translation does. The including unit also receives nonstatic meta functions and can call them at compile time. Static meta functions remain private to their provider.

The helper is kept under the cache directory (`$X2C_CACHE_DIR`, `$XDG_CACHE_HOME/x2c`, or `~/.cache/x2c`), named by a hash of the meta source paths, the compiler, the C compiler, and the flags, including `-D` and `-U`. It is built again when any file its build read changes, x2c source or C header; a change to program code alone recompiles the groups but keeps the helper when no group object changes. The shipped meta code of the compiler's own source modules and of `lib/meta.x` is linked into the compiler and runs without a helper. The REPL evaluates its supported submissions through Lisp. It accepts bodyless `meta` prototypes for linked or explicitly loaded native functions; user `meta` function definitions entered at the prompt are outside its supported subset. Its `:load` command compiles a source file's public bodied meta functions into a separate project helper and makes them callable from interpreted submissions.

Inside a `meta` body the whole body runs at compile time, so `$f(x)` there is an ordinary call of `f`. `$` keeps its meaning only where program code meets compile-time code, and says evaluate now and insert the result.

A `match` or `catch` pattern in a `meta` body is the exception. Its captures come from its value, so a `$f(...)` call in it runs while the body is translated. Only meta code linked into the compiler can run then, such as the source-form builders. A call of a project function waits until the body runs, so a pattern built by one cannot capture.

Each unit gets its own `meta static` values: the helper runs their initializers again before that unit's first call reaches their provider. What those initializers allocate, a Job one starts included, lasts until the next unit's reset or the end of the helper.

A project `meta` function receives arguments and returns a value. Quotations, `x2c_ident`, `Code.name`, `Code.binding_spelling`, `Type.parts`, `x2c_diagnostic_fail`, and `x2c_diagnostic_warn` work in the helper. The typing and invocation queries, such as `Type.resolve`, `Type.fields`, `Code.type`, and `Type.resolve_member`, ask the compiler while the call waits. The compiler answers in the state of the `$` call's site, so a project function gets the answer that compiler-linked meta code gets at the same place. `Code.source_text` reads only the text a `Source` parameter carries.

Running compile-time code needs what building the program needs: the C compiler and the runtime headers. A project meta module that does not compile is reported at the call with the C compiler's first error and the directory that keeps its C.

A raise inside a `meta` body becomes a diagnostic at the `$` call. A body that crashes, exits, or overflows the stack ends the helper; the call is reported with the function and the reason, and the next call starts a new helper. A `$` call that runs longer than 60 seconds is stopped the same way; `X2C_META_TIMEOUT` sets another limit in seconds, and `0` turns the limit off.

A process that compile-time code starts ends no later than its helper. The helper leads a process group of its own, and everything in that group ends when the helper ends, whether it quits, crashes, or passes the deadline, and when the translation that started it ends. Since that group is not the terminal's, compile-time code that reads the terminal is stopped until the call's deadline.

Arguments and results

A `$` call's arguments are evaluated before the call. Each is a numeric, character, or String literal, a `%(...)` syntax literal, arithmetic over constants, a Symbol, captured macro syntax, or another `$` call. A captured literal passed to a parameter that is not syntax, such as a `String`, arrives as the literal's value. Anything else, such as a program variable, reports `an argument must be a constant, captured syntax, or a meta call`.

Compile-time Lisp calls a `meta` function by its name with Lisp values. The native function checks each argument against its declared type, so passing `0` where a `List` is declared is an error rather than an empty list.

Lifetime and state

A block that holds a cleanup keeps its C meaning in a meta body. `defer`, `$scope`, and `$let` run their cleanup when the block ends and on every exit through it:

- A `return` computes its value first, then runs every pending cleanup, innermost first, then returns.
- A `break` or `continue` runs the cleanups between it and its loop.
- An error raised inside the block, such as a failed compiler query, runs the cleanup before it becomes the diagnostic for the explicit call.

```
meta static int closed = 0;

meta int exits(int n) {
  int total = 0;
  for (int i = 0; i < n; i++) {
    defer closed += 1;
    if (i == 1) continue;
    if (i == 3) break;
    total += 10;
  }
  {
    defer total += 1;
    if (n > 5) return total * 100;
  }
  $let(total, 0) total = 7;
  $scope() {
    Array scratch = [total];
    total += (int) scratch.len();
  }
  return total;
}

meta int closed_count(void) => closed;

int main(void) {
  printf("%d %d %d\n", $exits(2), $exits(9), $closed_count());
  return 0;
}
```

```
12 2000 6
```

`$exits(2)` runs both turns, adds 1 as its block ends, restores `total` after `$let`, and adds the one element counted in `$scope`. `$exits(9)` leaves the loop at the `break` after four cleanups. Its `return` computes 2,000 before the block's cleanup adds 1 to `total`. A loop with a cleanup on its iteration path still runs each turn in constant space.

A scratch collection that only builds the result takes `$auto`. The return converts the Array to a new List first, then the cleanup frees the Array:

```
meta static List evens(int n) {
  Array out = $auto([]);
  for (int i = 0; i < n; i++) out.push(2 * i);
  return out;
}

int main(void) {
  List values = $evens(4);
  printf("%s\n", values.repr().str());
  return 0;
}
```

```
(0 2 4 6)
```

Meta code manages lifetimes with the same operations as native code, and they mean the same thing: a compile-time call runs inside a Scope the compiler opens for it, and a returned collection belongs to the compiler, not to the eventual program's heap. As in C, a value built inside a region must not be kept, in a `meta static` or a Lisp definition, after that region is released.

A file-scope variable a `meta` body reads is the compiled group's own copy, initialized as the declaration says; it does not expose future runtime state. Mark it `meta static` to say that compile-time code owns it.

Results: compute or insert

A meta function can pass and return any value to another meta function. Inserting a result into the program adds a separate requirement: the compiler must construct code representing that value.

Result	Explicit `$helper(...)` insertion
Native integers and floating values	Preserves the numeric Var family, including width, signedness and floating precision.
Computed string	Inserts a quoted C string literal.
`Symbol`	Inserts a Symbol literal.
Identifier or nonempty code `List`	Binds the returned code through normal compiler binding and typing, in ordinary code and in macro bodies alike. A List is code when it is an expression node, a macro application, a quotation, or an identifier; inside a macro expansion every List is code. A call written as a whole statement may also return a statement, while a call inside a larger expression must return an expression. A data List is not automatically an expression, and a result used as a `case` pattern stays data.
Boxed `Var`	Insertion follows the contained value.
`{}` stored in a `Var`	A fresh empty Map, as in compiled code; inserted like any other Map.
`Array` or `Map`, nested at any depth	Constructs fresh collections through the ordinary literal constructors.
Struct value, C `bool` or pointer	Diagnosed; the value has no Var form.
`Func`	No direct materialization of the compiler's object.

An inserted Array or Map is a snapshot of the compile-time result. It becomes an ordinary `[...]` or `{...}` literal, so each runtime execution of the expression allocates fresh collections in the current Scope, at every depth; assigning the result to another variable still aliases it. Array order and element types are preserved. Map keys use their ordinary runtime key semantics, but reconstruction does not promise the same traversal order. A boxed Var may contain the result. Collection items, including Map keys, can contain raw `Null`, distinct from the empty List `%()` and `void`.

Scalars, Strings, Symbols, and Lists that hold only those values are immutable and are emitted once as constants, as the same values written in source would be. A List inside a result is data. A List that holds raw `Null`, an Array, or a Map is built at runtime like one written in source.

Each Array and Map in a result must appear once. A result that contains itself, or holds the same collection in two places, is diagnosed rather than copied, because the inserted code would build separate collections.

The same functions can be called during compilation or with ordinary C-style calls at runtime:

```
meta static double meta_half(double n) => n / 2.0;
meta static int meta_whole(void) => (int) meta_half(9.0);
meta static String meta_label(String stem) => stem.upper() + "!";

int main(void) {
  printf("%.1f %d %s\n", $meta_half(9.0), $meta_whole(),
    $meta_label("ready"));
  printf("%d %s\n", meta_whole(), meta_label("ready").str());
  return 0;
}
```

```
4.5 4 READY!
4 READY!
```

Floating constants retain their binary value using hexadecimal literals; nonfinite values use the native compiler's infinity/NaN expressions. A pointer into the compiler is never a valid address to embed in the future program.

Compile-time arithmetic and evidence

Compile-time code uses the x2c backend and the host C compiler selected by `--meta-cc`, which defaults to `cc`. The target `--cc` does not select that compiler. Arithmetic, conversions and casts follow the host build's native semantics. The `meta-differential` compiler fixture checks narrow and unsigned arithmetic, wide intermediate values, floating operations and conversions against runtime calls. The `comptime-lowering`, `meta-native-constructs`, `meta-import` and `meta-cursors` fixtures cover collections, callable values, local addresses, C constructs, method chaining and iteration; the `meta-records` and `meta-heap-objects` fixtures cover structs and the heap.

Generating code

A `meta` function can also take code as a `List` and return new code. Generating Code covers macro helpers, source templates, quotations, and the questions the compiler answers.

Sharing a `meta` function between units

Define a shared `meta` function in an ordinary `.x` module, beside any macros and runtime code that use it. The project meta build compiles the provider's meta code into the project meta module. A meta function must be written in source: a definition inside a macro template, or one produced by a `$` call, is reported where it is written.

The provider includes its own dependencies. A consumer includes the provider:

```
#include "shape.x"
```

Nonstatic meta functions become callable during translation at the include position, including through transitive includes. A `meta static` helper and its static state remain in the provider. Public meta functions can use those helpers inside the provider's compiled group. The caller does not receive a second definition of the helper.

Repeated includes contribute one public declaration per canonical path. For a meta function with a runtime form, ordinary linkage applies: the provider owns its runtime definition and including sources receive its declaration. Compile and link that provider as you would any other source module. The compile-time helper build and the runtime build are separate.

Lisp interoperability

Use the x2c calls above for ordinary meta work. The following notes apply when importing or calling Lisp helpers, or working with the compiler session.

The compiler loads Lisp support into an embedded evaluator. The built-in macros' algorithms are compiler functions (`src/builtins.x`), bound under the names `etc/builtin-macros.x` and `etc/lisp-bindings.x` call.

One table for compiled code and Lisp

Compile-time Lisp calls a `meta` function by its name. A table kept as a `meta` function therefore serves both an inserted value and a Lisp helper that reads it during expansion, without a second copy of its rows:

```
meta static Map widths(void) => { %(char): 1, %(short): 2, %(int): 4 };

macro Expression $sample.width(Type $type) =>
  $(x2c.literal.int (Map.getindex (widths) $type));

static Map table = $widths();

int main(void) {
  printf("%d %d\n", $sample.width(short), table[%(int)].int());
  return 0;
}
```

```
2 4
```

`lib/native-scalar-types.x` keeps the compiler's exact C scalar table this way: `src/type.x` inserts it, and the `lib/lisp/lisp.x` access records read their tags from it.

Names the compile-time library already defines

A unit's compile-time session inherits the compiler's own Lisp library and cannot replace one of its definitions. A `meta` function is installed under its C name, so `meta int add(int x)` reports that it could not be installed. A macro file or a `$(...)` form that defines an inherited name reports:

```
sample.x:2:1: macro: compile-time Lisp evaluation failed
  $(defun filter (a b) 42)
  ^^
  note: form: (defun filter (a b) 42) error: (bad-state (operation "def")
  (reason "inherited") (name filter))
```

The library holds many ordinary words, so the name to avoid is often one you would reach for first: `filter`, `last`, `map`, `search`, `len` and `apply` are all defined. Give your own definition a different name, or a prefix of your own.

`eval` reads globals only

A lambda's body reads its captures and then the session's globals. `eval` is an ordinary procedure, so the form it is given is evaluated in the globals alone and does not see the bindings around the call:

```
(let ((z 7)) (eval (quote (add z 1))))   error: (unbound (name z))
```

Pass the value instead of the name: `(eval (list (quote add) z 1))`.

A `let` binding is not visible inside its own initializer, so a helper that calls itself by name must be a `defun`:

```
(let ((h (lambda (n) (h n)))) (h 3))   error: (unbound (name h))
(defun h (n) (h n))                    reads its own name
```

Generating Code

A `meta` function can take the code being compiled as a `List`, inspect it, and return new code for the compiler to insert. Read Compile-time Macros first. The macro declares what code to capture; the `meta` function implements the transformation in x2c. Include `meta.x` to use the compiler's operations: the methods of `Code` and `Type`, and the `x2c_` helpers.

Calling a meta function from a macro body

A macro body can call a meta helper with `$helper(args)`. Passing a captured hole directly supplies its code to the helper; it does not evaluate the future runtime expression. This form works for code transformations such as the field query below. Complete captures also retain their source information for text and location queries; constructed subtrees do not acquire it.

```
#include "x2c.x"
#include "meta.x"

typedef struct Point { int x, y, z; } Point;

meta static List field_count(TypeInfo type) =>
  $!int{ ${((List) type.assoc(<fields>)).len()} };

macro Expression $probe.count(Expr $value) => $field_count($value);

int main(void) {
  Point p = { 1, 2, 3 };
  printf("count %d\n", $probe.count(p));
  return 0;
}
```

```
count 3
```

The macro body is one call and nothing else. That is the usual shape: the macro declares the holes and the result kind, and the `meta` function does the work.

`field_count` declares its parameter `TypeInfo`, so it receives a description of the argument's type rather than its code. The compiler computes that description at the `$` call:

```
((name "Point") (kind struct) (type ("Point"))
 (fields (("x" (int)) ("y" (int)) ("z" (int)))) (methods ()))
```

`name` is the type's name, or `""` when it has none. `kind` is `struct`, `union`, `enum`, `pointer`, `scalar` or `other`. `type` is the canonical type, and `fields` lists the `(name type)` rows of a struct or union's named fields in declaration order. `methods` lists the names of the type's direct dotted methods. Read a part with `List.assoc`. Pass the same hole twice when a function needs both the code and its type, as `shape_reads` does below.

The function's result determines the expansion. A `List` that a quotation built represents code. A typed quotation of one value, such as `$!int{ $count }` or `$!String{ $name }`, is an expression holding that value.

A `Stmt` macro can use the same arrow form. Its body is then the statement `$helper(args);`, and the helper may return any statement that fits where the macro is invoked, not only an expression:

```
#include "x2c.x"
#include "meta.x"

meta static List repeated(List code) => %(block $code $code);

macro Stmt $sample.twice(Stmt $code) => $repeated($code);

int main(void) {
  int n = 0;
  $sample.twice(n += 2;);
  $sample.twice(if (n == 4) puts("four"););
  printf("%d\n", n);
  return 0;
}
```

```
four
four
4
```

`repeated` receives each captured statement as code and returns a block that holds it twice. The second invocation passes an `if` statement, which an expression position could not accept. A `$helper(args);` statement in an ordinary function may also return a statement. There, only an expression node, a macro application, a quotation, or an identifier is code. A `%(block ...)` such as the one `repeated` builds is data there and becomes a runtime `List` value. A helper for ordinary code therefore returns statements as a `$!{ ... }` quotation.

Methods, source text, and embedded files

A `TypeInfo` parameter's `methods` part lists the type's direct dotted methods. A generator can test for one and build the dotted call, which the compiler resolves as it would the same call in source:

```
#include "meta.x"
meta static List project_write(TypeInfo type, List receiver, List value) {
  List methods = type.assoc(<methods>);
  if (!methods.contains("write")) return %();
  return $!( $receiver.write($value) );
}
```

The list does not search delegate fields. A delegated result would also need the field-projected receiver, which this operation does not return. Generated dotted calls still use delegation during ordinary expression resolution.

To read the exact source text of a complete captured argument, declare the parameter `Source` and use `Code.source_text`:

```
#include "meta.x"
meta static String project_text(Source value) => Code.source_text(value);
macro Expression $project.source(Expr $value) => $project_text($value);

int main(void) {
  return strcmp($project.source(1 /* kept */ + 2),
                "1 /* kept */ + 2");
}
```

Passing a complete capture to a meta helper preserves the source information for this query. Forwarding it through another source macro does too. Constructing or selecting an AST subtree does not invent source text; use `List.repr()` to render code data.

`x2c_embed_text` reads a regular text file into a compile-time String and records it as a build dependency. A captured String literal retains the caller's file location for resolving a relative path:

```
#include "meta.x"
meta static String project_notice(Source path) => x2c_embed_text(path);
macro Expression $project.notice(Literal $path) => $project_notice($path);

String notice = $project.notice("notice.txt");
```

A captured path stays relative to its caller even when the macro is defined in an imported file. A project meta function also accepts an absolute String path; a relative one needs the `Source` that locates it.

Empty files are valid. Directories, embedded NUL bytes, unreadable files and files too large for a String are rejected.

Source templates from meta functions

A Unit or Stmt macro used as an expression inside a meta function constructs a deferred template invocation. Its arguments are values computed by the meta function. The returned code expands when inserted into the program, using ordinary macro substitution, scope and binding rules. Expression macros retain their usual expansion behavior.

```
macro Unit $sample.make_function(Name $name, Expr $result) {
  int $name(void) { return $result; }
}
```

```
meta static List build_function(String name, int n) {
  List node = $sample.make_function(name, n * 2);
  return %($node);
}

macro Unit $sample.make_answer() { @build_function("answer", 21) }

$sample.make_answer();
int main(void) {
  printf("%d\n", answer());
  return 0;
}
```

```
42
```

The constructor returns one code node. The helper wraps it in a List because `@` inserts a sequence of nodes. It does not return an already expanded function declaration for the helper to inspect. Name, Type, parameter and statement captures still follow the source macro's declared hole kinds.

A decorator can make that choice from the function it captures. Here the `meta` helper selects one of two named Stmt templates, then the decorator inserts the selected invocation into the function body:

```
#include "x2c.x"
#include "meta.x"

static int calls = 0;

macro Stmt $sample.counted(Stmt @body) {
  calls++;
  @body
}

macro Stmt $sample.plain(Stmt @body) {
  @body
}

meta static List choose_body(List function) {
  List body = Code.body(function);
  String name = Code.name(function);
  List node = name == "tracked" ? $sample.counted(body) : $sample.plain(body);
  return %($node);
}

macro Decorator $sample.count_if_tracked(Function $function) {
  @choose_body($function)
}

$sample.count_if_tracked()
static int tracked(void) { return 7; }

$sample.count_if_tracked()
static int ordinary(void) { return 9; }

int main(void) {
  printf("%d %d %d\n", tracked(), ordinary(), calls);
  return 0;
}
```

```
7 9 1
```

`choose_body` runs while the compiler expands the decorator. It reads the captured function, computes the choice, and returns a deferred invocation. The chosen template expands when the returned code is inserted. Its body capture becomes the function's statements; `calls++` runs only when `tracked` is called at runtime. The template names are known in source, while `meta` chooses which one to use for each function.

The reverse gradient generator in `packages/autodiff/src/autodiff-macros.x` uses this pattern. Its `ad_reverse_with` returns an invocation of the `ad.gradient` source macro. The macro contains the generated function declaration, tape storage, reverse-pass label and cleanup. Meta helpers compute the parameter and

statement sequences. The macro owns the fresh scratch names and passes those same names to the fragment helpers, keeping all generated references in the intended function scope.

Quoting code with `$!`

A template that a `meta` function uses once can be written where it is used. `$!( expression )` builds an expression, `$!{ items }` builds statements, and `$!Unit{ ... }`, `$!Field{ ... }`, or another category name before the braces builds that kind of code. A `$name` inside refers to the function's local `name` and inserts its value: syntax where the position takes syntax, and a number, String, or Symbol as a literal. `@name` splices a `List` local. `${expression}` inserts any value; see below. Each local's first position determines what it fills.

```
#include "x2c.x"
#include "meta.x"

meta static List numbered(List subject, List arms) {
  Array cases = [];
  int index = 0;
  foreach (List arm, arms) {
    cases.push($!{ if ($subject == $index) $arm });
    index++;
  }
  return cases.list_free();
}

macro Stmt $sample.choose(Expr $subject, Stmt @arms) {
  @numbered($subject, $arms)
}

int main(void) {
  int pick = 1;
  $sample.choose(pick, printf("zero\n");, printf("one\n");, printf("two\n"););
  return 0;
}
```

```
one
```

A quotation is an anonymous macro applied at once to the locals its body names, so it follows the same rules as any template: arguments arrive as the syntax or values the locals hold, names the body declares are private to the expansion, and other names resolve where the code lands. It keeps nothing from the function once it has been applied. Inside a `%(...)` List, `$` inserts a value, so bind a quotation to a local first and insert the local. `$` inserts one value and `@` splices a sequence in templates and quotations. Runtime List literals use the same prefixes. Native C varargs retain their `...` spelling. An `@name` sequence splices into call arguments and braced initializers such as `$!( { @items } )` or `(T){ $first, @rest }`, and `%[@items]` Array literals; Map entries splice as `%{${@rows}}`. A quotation expands where code is bound, spliced, or resolved: return it, pass it to a slot, or use it as another quotation's hole. A quotation inside hand-built expression syntax, such as a call's argument list or a declaration initializer, expands when the compiler resolves that expression.

A local declared `Type` fills a type position, which lets a quotation declare a variable whose type it computes:

```
meta static List counter(String name) {
  Type type = %(unsigned);
  return $!{ $type $name = 0; };
}
```

A `Type` spells C type keywords such as `unsigned` and `char`, and the `*` of a pointer, as Symbols. It spells every other type name as a String: `%("String")`, `%(* "Point")`, `%(struct "Tag")`. `%(String)` holds an `Atom`, so it is an error where it fills a type. The compiler reports `type name 'String' must be a String: write %("String")`.

Holes that name an expression

A `$name` hole needs a local. When the value is a field, an element, or the result of a call, `${expression}` inserts it without one:

```
#include "x2c.x"
#include "meta.x"

typedef struct Call { List callee; List arguments; Type result; } Call;

meta static List scaled_call(List callee, List a, List b) {
  Call d = {.callee = callee, .arguments = %($a $b), .result = %(int)};
  return $!( ({
    ${d.result} value = ${d.callee}(@{d.arguments});
    value * ${d.arguments.len()};
  }) );
}

macro Expression $sample.scaled(Expr $f, Expr $a, Expr $b) =>
  $scaled_call($f, $a, $b);

static int sum(int a, int b) => a + b;

int main(void) {
  printf("%d\n", $sample.scaled(sum, 2, 5));
  return 0;
}
```

```
14
```

The braces hold ordinary code of the function that contains the quotation. The function evaluates each `${...}` once, when it evaluates the quotation, before the quotation builds its code. Holes are evaluated in the order they are written. Names in the braces resolve where the quotation is written. A name that the quotation's body declares is not visible in the braces, because the body is code for another place. A `${...}` hole therefore means the same as a local declared just before the quotation and named with `$`:

```
return $!( ${d.callee}(@{d.arguments}) );
// means
List callee = d.callee;
List arguments = d.arguments;
return $!( $callee(@arguments) );
```

The expression's declared type gives the hole its kind, as a local's type does. A `Type` expression fills a type position, as `${d.result}` does above. A `String`, `int`, or `Symbol` expression inserts a literal, so `${d.arguments.len()}` inserts `2`. A `List` expression inserts the syntax it holds. `@{expression}` splices a `List`, like `@name`. Each `${...}` or `@{...}` is a separate hole, even when two are spelled alike.

A `${...}` belongs to the innermost quotation that contains it. To put one quotation's code inside another, write the inner quotation as a hole: `$!( $inner * ${$!( $a - $one )} )` builds the inner code first and inserts it.

Inside a `%(...)`, `%[...]`, `%{...}`, or `%"..."` literal in a quotation's body, `${...}` keeps its literal meaning: it inserts a value when the built code runs. Write `${${expression}}` there to insert a value that the function computes. A template Array splices a computed sequence through the same code boundary: `%[${@{items}}]`. This constructs template slots; ordinary Arrays and Maps do not support runtime splicing. A List's `@{expression}` still runs when the generated program constructs that List.

Literal characters need no new staging operator. Write an ordinary string such as `"$ @"`, or `%"\$ @"` for a percent string. Quoted atom escapes such as `%(\$ \@)` preserve sigils as data.

`${...}` works only in quotations. A named template has no function to evaluate it in; it computes with a `$meta(...)` call or a `$(...)` slot, which run where it expands.

Code that knows its type

A quotation is bound where it lands, so until then its code has no type. Sometimes the code a function builds must state its type at once: another operation reads the type before the code lands, or the code goes where nothing binds it again. Write the type between `$!` and the braces that hold the expression:

```
meta static List doubled(List value) {
  List twice = $!double{ $value * 2 };
  // Code.type(twice) is (double) already.
```

```
  return twice;
}
```

`$!T{ expression }` builds the expression where it is written and gives it the type `T`. `T` is one identifier that names a type, such as `int` or `String`. Put any other type in parentheses: a type of several words, as in `$!(unsigned long){ ... }` or `$!(char *){ ... }`, or a `Type` local or `${...}` hole whose value is the type, as in `$!($result){ $callee(@arguments) }`. A kind name such as `Unit` or `Type` after `$!` always names a kind, so a type spelled like one also takes parentheses: `$!(Type){ ... }`.

A typed quotation states the type the code has. It converts nothing, so write a cast inside the braces when the code needs one.

The rest of the code is built the way a quotation builds it, with two differences:

- Nothing is bound. Holes keep the syntax their locals hold, a binding becomes a reference to it, and a number, String, or Symbol becomes a literal. A number keeps its exact value and type, and a String is a String literal when `T` is `String`. A `@items` sequence makes each number, String, or Symbol it holds a literal in the same way. A name the expression reads without declaring stays a name and is bound wherever the code is bound, if it is bound at all.
- The code declares nothing. A typed quotation has no expansion to keep names private to, so a name it needs comes from a hole, such as a local that holds `x2c_ident("total")`. For the same reason it applies no template; build that code first and insert it. Inside another quotation, write a typed quotation as a hole: `${$!int{ $b }}`.

Only the outermost expression receives `T`. Inner expressions carry no type until something binds them. Write `T` exactly: the compiler trusts it as it trusts the type a typed call already carries.

A typed reference to a binding the function holds is the shortest form: `$!($type){ $binding }` is the expression that reads `binding` as a `type`.

Types and parameters

A quotation can also build a type or one parameter. `$!Type{ ... }` holds a type written as a cast writes it. It builds the `Type` List that a `Type` local holds:

```
typedef struct Point { int x; int y; } Point;

meta static String point_types(void) {
  Type point = $!Type{ Point };
  Type pointer = $!Type{ const $point * };
  Type reader = $!Type{ int (*)($point *) };
  return point.repr() + " " + pointer.repr() + " " + reader.repr();
}

("Point") (* const "Point") (* (func ((* "Point"))) int)
```

A `Type` local or a `${...}` hole of type `Type` stands for a type in the braces. The declarator around it builds a new type from it, so `$!Type{ const $point * }` is a pointer to a constant `Point`. A type name keeps its spelling: `$!Type{ String }` is `%("String")`.

`$!Param{ ... }` holds one parameter declaration. Put the parameters in a function with a sequence hole:

```
meta static List sum_function(String name, int count) {
  Array params = [];
  List total = $!( 0.0 );
  for (int i = 0; i < count; i++) {
    List x = x2c_ident(%"x$i");
    params.push($!Param{ double $x });
    total = $!( $total + $x );
  }
  List declared = params.list_free();
  List function = $!Unit{ double $name(@declared) { return $total; } };
  return %($function);
}
macro Unit $sample.define_sum(Literal $name, Literal $count) {
  @sum_function($name, $count)
}
$sample.define_sum("sum3", 3);   // sum3(1.0, 2.0, 3.5) is 6.5
```

```
6.5
```

Both build their List where they are written, as a typed quotation does. Nothing binds them until the function that holds them is bound. The braces may not declare a name of their own. A parameter takes its name from a hole, so the body reads it through the same `x2c_ident(...)` local. There is no `$!Decl{ ... }`, because `$!{ ... }` and `$!Unit{ ... }` already build declarations.

What the compiler answers

`lib/meta.x` declares the compiler operations. An operation on a value is a method of its type: `Code` for captured code and `Type` for a semantic type. The rest are `x2c_` helpers with no value to act on, such as `x2c_ident`, `x2c_invocation_line` and `x2c_diagnostic_fail`. Code is built with quotations, so the module declares no constructors.

`x2c_ident`, `Code.name`, `Code.binding_spelling`, `Type.parts`, `x2c_diagnostic_fail` and `x2c_diagnostic_warn` work in every `meta` function. The queries that read the compiler's symbol table, such as `Code.type`, `Type.fields` and `Type.resolve`, also work in every `meta` function: the compiler answers a project function's query while the call waits, in the state of the call's site. `TypeInfo` and `Source` parameters still deliver a description in one argument. The operations are grouped here by the task, not by signature; the module reference lists every declaration, and the language reference gives their semantics.

Building code

A quotation builds the code, and a typed quotation of one value, such as `$!int{ $count }`, is the literal expression that holds it. `x2c_ident` checks a name and returns identifier code, which a quotation uses where the code reads the name:

```
#include "x2c.x"
#include "meta.x"

meta static List call_of(String callee, List argument) {
  List name = x2c_ident(callee);
  return $!( $name($argument) );
}

macro Expression $probe.twice(Expr $value) => $call_of("twice", $value);

static int twice(int n) => n * 2;

int main(void) {
  int n = 21;
  printf("twice %d\n", $probe.twice(n));
  return 0;
}

twice 42
```

Asking about types and fields

A `meta` function receives a type as a `TypeInfo` parameter, described above, or asks the compiler with the queries below. `Code.type` answers the canonical `Type` of an expression or binding. `Type.fields` answers the named fields of a struct or union `Type`, each as a metadata row whose first element is the field name. `Type.layout`, `Type.resolve` and `Type.is_value` answer the remaining generated-code questions. `Type.parts` reads only the `Type` it receives. `Type.resolve_member` answers which field or method a member access selects. These answers live in the compiler's symbol table, so a macro body cannot derive them from the code it captured.

Source text and location

A parameter declared `Source` receives a captured hole together with the text the developer wrote for it: `((text T) (file F) (syntax S))`. `Code.source_text` returns that text, and `x2c_embed_text` reads the file a captured `String` literal names beside the source that wrote it and records it as a translation dependency. Both read what the argument carries. `x2c_embed_text` also accepts an absolute `String` path. `Code.binding_spelling` returns the

name a binding was declared with. `x2c_invocation_file`, `x2c_invocation_line` and `x2c_invocation_column` give the site of the macro invocation.

Failing with a diagnostic

`x2c_diagnostic_fail` reports a message at the invocation and stops the expansion. It does not return. Use it when the argument is wrong in a way the macro can see:

```
#include "x2c.x"
#include "meta.x"

meta static List one_word(Source node) {
  String text = Code.source_text(node);
  if (text.contains(" "))
    x2c_diagnostic_fail("this argument must be one word", %());
  return $!String{ $text };
}

macro Expression $probe.word(Expr $value) => $one_word($value);

int main(void) {
  int seconds = 90;
  printf("word %s\n", $probe.word(seconds));
  return 0;
}
```

```
word seconds
```

The helper receives the complete capture with its source text, which `Code.source_text` reads. A computed subtree is code data, not a new source capture, and cannot be passed as a `Source`.

Writing `$probe.word(seconds * 2)` instead reports the message at that invocation:

```
sample.x:15:23: macro: this argument must be one word
  printf("word %s\n", $probe.word(seconds * 2));
                      ^
```

`x2c_diagnostic_fail_at` reports under a category, such as `<macro>` or `<type>`, at a node the macro received, such as one captured statement, instead of at the invocation.

Placing code beyond the result

`x2c_enclosing` answers the syntax around the invocation: the initialized `<declarator>`, the block item (`<statement>`), the `<function>`, or the `<unit>`. `x2c_place` adds code the result cannot hold: `%(after-statement)` after that declarator or block item, `%(block-exit)` on every exit from the block after it, as a `defer` there runs, `%(function-entry)` at the entry of the enclosing function before its first statement, `%(unit-support KEY)` among the unit's file-scope declarations once for each key, and `%(unit-init)` in the unit's initialization. Code placed at function entry follows the parameters and keeps the order of its placements. A macro in a lambda body places it in the function that holds the lambda; a `$rewrite` rule, which runs while the body lowers, places it in the lambda's own function. A macro in a `defer` finalizer places it in the function that holds the `defer`; a `$rewrite` rule that places it while the finalizer lowers is rejected, because the finalizer runs as its own callback. A failed expansion leaves none of it behind. Placed code that declares a name takes it from `x2c_binding_fresh(role)`, which returns a binding the compiler names, so the result can refer to the same binding. Code placed once under `%(unit-support KEY)` takes its name from `x2c_binding_for(KEY, role)`, which returns the same binding for the same key in a unit, so every result that refers to it agrees. Here the counter runs after each statement that traces a value:

```
#include "x2c.x"
#include "meta.x"

static int traced = 0;

meta static List trace_after(List value) {
  x2c_place(%(after-statement), $!{ traced++; });
  return value;
```

```
}

macro Expression $trace(Expr $value) => $trace_after($value);

int main(void) {
  printf("%d %d\n", $trace(1), traced);
  printf("%d\n", traced);
  return 0;
}

1 0
1
```

`$auto` is written this way: it places a release after the declarator it initializes, and the declaration checks that the expansion is its complete initializer before it binds that release.

Rewriting a function definition

`$rewrite` from `rewrite.x` registers a `meta` function as a rule for the code its macro builds. When that macro is a `Unit` macro whose body is one function definition, or only its one `Function` hole, the rule receives each function definition the program writes once, after binding and typing and before the body lowers. A definition that the program's own `meta` code places or returns is the program's. A `meta` function whose body uses only what the runtime library provides also has a runtime form, an ordinary C function in its unit, and that definition is the program's, so the rule receives it. A definition that the compiler writes while lowering is not offered: a lambda's function, a `defer`'s cleanup, a `Func` adapter, a protocol's forwarding member, or the compiler's own replacement of a definition. The statements such a definition holds from the program still reach the program's statement rules. The rule returns the definition unchanged to decline, or a replacement definition. The replacement is lowered as written and offered to the other rules, but not again to the rule that returned it. A pattern that spells its return type, such as `int $name(...)`, is tried only on definitions with that return type and storage, before the rules whose pattern holds a hole there:

```
#include "x2c.x"
#include "rewrite.x"

static int entries = 0;

macro Unit $any_function(Function $function) { $function }

$rewrite($any_function)
meta Code count_entries(Code function) {
  if (!Code.name(function).startswith("counted_")) return function;
  match (function) case %(function ?type ?declarator ?):
    return %(function $type $declarator
      (block ${$!{ entries++; }} @{Code.body(function)}));
  return function;
}

int counted_next(int x) { return x + 1; }

int main(void) {
  counted_next(1);
  counted_next(2);
  printf("%d\n", entries);
  return 0;
}

2
```

The compiler binds each of these under its x2c name and derives the Lisp name from it: a method keeps its spelling, so `Code.binding_spelling` is `Code.binding_spelling` in Lisp, and in a helper each `_` becomes `.`, so `x2c_invocation_line` is `x2c.invocation.line`. A Lisp form has no quotation, so the Lisp library also defines `x2c.literal.string`, `x2c.literal.int` and `x2c.literal.symbol`, which build the literal a `$(...)` form returns for a value.

The compiler's shipped components lower language features with the same `$rewrite` decorator. A program definition spelled like a shipped component's `meta` function or helper is a separate definition: shipped rules and templates resolve their own names to the copies the compiler links.

Functions that need the compiler

Compiler queries exist only inside a compiler. A `meta` function that calls one, directly or through another `meta` function, therefore has no valid runtime form, and the compiler emits no definition for it. A body containing a source-template constructor is also compile-time-only, as are its meta callers. A call to one from a runtime body is diagnosed where it is written. Calling `one_word` from an earlier section at run time gives:

```
sample.x:19:22: macro: 'one_word' can only be called at compile time
    List n = one_word(node);
                     ^
  note: reason: it reaches a compiler operation, so no unit emits a
  definition for it; call it from a macro or another meta function
```

In the complete shape example below, `shape_names` and `shape_reads` construct source code and have no runtime forms. The consumer uses their returned code.

A `meta` function that needs no compiler query, like `poly` above, keeps both forms and is emitted normally.

A complete example

Two ordinary source files. `shape.x` holds the `meta` functions and the macros that call them. `shape_fields` reads the fields from the `TypeInfo` the compiler sends. `shape_names` and `shape_reads` turn them into code.

```
#include "meta.x"

/* The named fields of a struct-typed expression, in declaration order. */
meta static List shape_fields(TypeInfo type) => type.assoc(<fields>);

/* One `String` literal holding those field names, comma separated. */
meta List shape_names(TypeInfo type) {
  Array names = [];
  foreach (List field, shape_fields(type)) names.push(field.car());
  return $!String{ ${String.join(", ", names)} };
}

/* `{ p.x, p.y, p.z }`, built from the fields rather than written out. */
meta List shape_reads(List receiver, TypeInfo type) {
  Array reads = [];
  foreach (List field, shape_fields(type)) {
    String member = field.car();
    reads.push($!( $receiver.$member ));
  }
  List items = reads.list_free();
  return $!( { @items } );
}

macro Expression $shape.names(Expr $value) => $shape_names($value);

macro Expression $shape.reads(Expr $value) => $shape_reads($value, $value);
```

The second file includes `shape.x` and uses its public macros. The include also makes its nonstatic meta functions available.

```
#include "x2c.x"
#include "meta.x"

#include "shape.x"

typedef struct Point { int x, y, z; } Point;

int main(void) {
  Point p = { 2, 3, 4 };
  int reads[3] = $shape.reads(p);
  printf("names %s\n", $shape.names(p));
```

```
  printf("reads %d %d %d\n", reads[0], reads[1], reads[2]);
  return 0;
}

names x, y, z
reads 2 3 4
```

The field names never appear in the program. The compiler supplied them, and the `meta` functions built `"x, y, z"` and `{ p.x, p.y, p.z }` from them.

Packages

x2c has one flat namespace, so an included `.x` library merges its names into yours. A package's public names carry its prefix. It is compiled once into an archive, and an importing program reaches its names through an alias.

What a package looks like

A package is a directory named for the package. Only its `src/` directory holds package sources; everything else in the directory consumes it like any other program.

```
packages/greet/
  Makefile          PACKAGE := greet, then include ../package.mk
  src/greet.x       the entry unit; nonstatic definitions are public
  tests/            consumers that import the package
  examples/
```

That is the whole layout for a package with no native dependency. One that wraps a C library adds the pin, license texts, and profile that Wrapping a C Library covers.

The entry unit needs no manifest and no export list:

```
typedef struct GreetingData {
  String subject;
  int count;
} *Greeting;

Greeting Greeting.new(String subject);
String Greeting.line(Greeting greeting);

Greeting Greeting.new(String subject) {
  Greeting greeting = Scope.malloc(sizeof(struct GreetingData));
  greeting.subject = subject;
  greeting.count = 0;
  return greeting;
}

String Greeting.line(Greeting greeting) {
  greeting.count++;
  return %"hello, ${greeting.subject} (${greeting.count})";
}
```

`greet.x` spells its own names bare. Compiled as a package they become `greet__Greeting`, `greet__Greeting_new`, and `greet__Greeting_line`, so a second package may publish its own `Greeting` in the same program.

Building one

`packages/package.mk` holds the recipe. A package Makefile is two lines:

```
PACKAGE := greet
include ../package.mk
```

An explicit `--package-dir`, or a target's `package-dirs` in `x2c.toml`, selects package ownership when compiling that package's source files. For `packages/greet/src/greet.x`, pass `--package-dir packages`; its public names then use the `greet__` prefix. `package.mk` supplies this directory for you. The automatic `<home>/packages` search locates imports; source compilation uses explicit package directories.

`make build` translates every `src/*.x` in package mode and produces:

```
builds/greet.h           the generated public header
builds/libgreet.a        the archive a consumer links
builds/greet.native.rsp  the native arguments a consumer needs, one per
                         line, empty when none
builds/greet.module      the compile-time part, only for a package that
                         has one
```

`make test` builds and runs `tests/test-*.x`, and `make clean` removes `builds/`. A package with a pinned native dependency adds a `dependency.json` and one more Makefile line, `DEPENDENCY_PREFIX_VAR := GREET_PREFIX`, naming the variable that overrides the cache prefix; `make prepare` then fetches and builds the dependency into a shared cache outside the worktree and leaves a `deps` symlink in the package directory.

Using one

Register the directory that holds packages and import by name:

```
import "greet" as g;

int main(void) {
  g.Greeting greeting = g.Greeting.new("x2c");
  printf("%s", %"${greeting.line()}\n");
  return 0;
}
```

```
x2c build --package-dir packages --output greeter greeter.x
```

The driver reads the unit's recorded dependencies, adds the package's `builds` and `src` directories to the C include path, and links `libgreet.a` together with the arguments in `greet.native.rsp`. It never rebuilds the package, so consumers can use a read-only package tree. A target in `x2c.toml` can set `package-dirs` instead of passing the flag.

`as` is optional; `import "greet";` binds the alias `greet`. The alias is only a way to spell names. The generated C always uses the package's own prefix.

Both clauses are optional and combine:

```
import "<package>" [as <alias>] [with <Name> [as <Local>] {, ...}] ;
```

A `with` clause drops the alias from names you use often:

```
import "greet" with Greeting;             // bare Greeting
import "greet" with Greeting as Hello;    // bare Hello
import "greet" as g with Greeting;        // g.Greeting and Greeting

int main(void) {
  Greeting greeting = Greeting.new("x2c");
  printf("%s", %"${greeting.line()}\n");
  return 0;
}
```

A `with` name is still only a spelling: `Greeting.new` compiles to `greet__Greeting_new`. Any public name works, functions included. The alias stays registered, so `with` never takes it away, and a local variable of the same name shadows the binding. See the language reference for what does and does not cross an import.

An imported package may add a method to an external type. PCRE2, for example, adds `Var.regexpmatch`, so `value.regexpmatch()` resolves after `import "pcre2"`. A method declared in the consumer takes precedence. If two imported packages provide the same receiver method, the call is ambiguous; call the selected package function explicitly, such as `pcre2.Var_regexpmatch(value)`.

That external type is often one from the package's own vendored header, which crosses the import under its own name. raylib's `Image` and `Vector2` are raylib's, so a consumer writes `Image.new(...)`, reads `image.width`, and uses `a + b` with no `with` entry for either type. The types come from the header and the methods from `raylib__`.

Packages with compile-time parts

A package has a compile-time part when its sources declare a native `meta` function, with a `meta native` definition or a bodyless `meta` prototype. Building the package then also builds a native module from the package's sources, `builds/<name>.module`, and importing the package loads it:

```
meta native int tally_sum(int n) => n * (n + 1) / 2;
```

```
import "tally";

meta static int ten(void) => tally.tally_sum(4);
```

`$ten()` runs the package's `tally_sum` during translation. The import alone loads the module; no `--native-module` option or `native-modules` entry is needed. A module named by either one is still consulted first. A package whose sources declare no native `meta` function builds no module and is only linked. `x2c install` builds the module of a source package, and `make bundle` carries it in the bundle.

The consumer's translation records the module as a prerequisite, so rebuilding the package retranslates the units that import it. The import never builds the package. Only the compiler that built a module loads it, so a package with a compile-time part must be rebuilt after the compiler changes; until then the import reports:

```
main.x:1:1: driver: package 'tally' was built by another compiler; rebuild it
```

Modules load on macOS, Linux and WSL. On other platforms, importing a package that has a module reports that native modules are not supported. There, link the package's compile-time part into a compiler built from a checkout's `src/*.x` instead:

```
./x2c build --extension packages/tally --output /tmp/x2c src/*.x
```

That compiler then calls `tally`'s `meta` functions through `import "tally"` alone, never loads a module, and needs no `builds/tally.module`. The build translates the package's `src/*.x` and compiles its `src/*.c` with the compiler, so only a package without a native dependency links in this way.

A package can publish macros beside its compile-time functions in ordinary source:

```
// src/tally.x
meta int tally_sum(int n);
macro Expression $tally.sum4() => tally__tally_sum(4);
```

The consumer needs one import:

```
import "tally";
meta static int ten(void) => $tally.sum4();
```

The import loads the package's public declarations, compile-time module or linked extension, and its nonstatic macros and keyword aliases. Ordinary includes propagate those definitions through the entry source. Static helpers stay in their declaring files.

Macros keep their declared names; `as` changes the package alias, not macro spellings. For a direct `#include`, compile the provider as ordinary source so its public names stay flat. Consume a provider compiled in package mode through `import`, which reaches its prefixed names through a package alias.

Expose macros through one import

The `autodiff` entry source includes its runtime dependencies and macro module:

```
#include "typed-array.x"
#include "autodiff-macros.x"
```

A consumer writes one line, `import "autodiff" with AdTape, AdNode;`. The same include rules apply to macros and ordinary runtime declarations.

Packages that wrap a C library

`packages/yyjson/` is the worked example. It publishes the vendored `src/yyjson-0.12.h` because four of its options methods take yyjson's own flag types, so a consumer needs the upstream header on its system include path:

```
make -C packages/yyjson prepare
make -C packages/yyjson build
make -C packages/yyjson test run run-lisp
x2c build --package-dir packages \
```

```
  --c-include-dir packages/yyjson/deps/include \
  --output catalog catalog.x
```

`prepare` verifies and builds the pinned native dependency. `build` creates the package archive and link file. The three checks exercise the x2c API, example applications, raw C header, and Lisp bindings. A program imports the ordinary API; code that needs an unwrapped option calls yyjson's real declarations through `yyjson-0.12.h`, and a runtime Lisp session installs the package's group with `JsonLisp.install`. The package README states which values own native storage and when borrowed views expire.

A source distribution carries `packages/<name>/`, including `src/`, `dependency.json`, `LICENSES/`, its README, examples, and tests, together with `packages/package.mk`, `packages/dependency.mk`, `packages/tools/deps.py`, and `packages/tools/bundle`, which `make bundle` runs through `x2c script`. It does not carry the ignored `deps` symlink or `builds/` output. From the repository root, produce a source archive with ordinary `tar`:

```
package=yyjson
tar --exclude="packages/$package/deps" \
  --exclude="packages/$package/builds" \
  -czf "$package-source.tar.gz" \
  "packages/$package" packages/package.mk packages/dependency.mk \
  packages/tools/deps.py packages/tools/bundle
```

The package directory includes its `Makefile`; keep any additional source files or licenses that its build needs. Include other x2c packages it imports in the same archive, or distribute them separately under a registered package root. Native dependency sources are fetched and verified from `dependency.json` during preparation.

Shared support resolves beside those files, so the bundle needs no Git checkout. Unpack it into a directory where its sources and build outputs can remain together. Select an installed compiler and an explicit dependency cache. Use the native prefix installer for a compiler that can move independently of its producer checkout. A development build can instead use the absolute path to `builds/0/x2c`; keep that compiler's support tree in place. A package source archive does not install or bundle the compiler.

```
mkdir package-sources
tar -xzf yyjson-source.tar.gz -C package-sources
cd package-sources
export X2C_DEPS_DIR="$HOME/.cache/x2c-dependencies"
make -C packages/yyjson prepare build X2C=/path/to/x2c
```

Register the directory containing the package with `--package-dir`. Native compilation and archiving use the same driver actions and retained state as ordinary builds, keeping the generated public headers, archive, and link flags together. Package tests that include `unittest/test-support.x` still need that repository test support; it is not part of the source-package build contract.

A package over a third-party C library also has to decide what to expose, who owns each returned value, how to preserve the library's error codes, and whether to provide Lisp bindings. That is Wrapping a C Library.

Package examples

The packages include example applications that use their x2c interfaces without requiring public services:

- `packages/sqlite/examples/observations.x` stores readings and queries slow or failed responses. `observation-history.x` imports a file batch in one transaction, then reopens the database and reports endpoint history.
- `packages/libcurl/examples/page-titles.x` fetches pages concurrently with `CurlEasy.get_all`; `endpoint-report.x` handles each batch response or Error independently, then uses the ordinary single-request operations. Batch results keep input order and borrow their lifetime from `CurlBatch`. The call stays on the caller's thread; the pinned resolver can still block during DNS lookup. `examples/packages/http-json-releases/` composes libcurl's response bytes with yyjson parsing.
- `packages/termbox2/examples/incident-filter.x` handles terminal input and rendering. Its standard run is driven through a pseudo-terminal.
- `packages/blis/examples/page-rank.x` expresses PageRank with BLIS operators; `risk-report.x` shows the broader matrix workflow.

- `packages/libuv/examples/process-report.x` supervises child processes; `thread-notify.x`, `network-report.x`, `ipc-report.x`, and `datagram-report.x` exercise wakeups, TCP, named pipes, and UDP; and `release-checks.x` reads produced artifacts through asynchronous files.
- raylib's standard examples render PNG files in memory. Use `run-interactive` for the windowed showcase; package checks do not run it.
- `packages/torch/examples/fit-line.x` fits a line by gradient descent through libtorch autograd: `x.matmul(w) + b`, `backward`, and an in-place update under `Torch.no_grad`, with every operator temporary reclaimed by the step's `Scope`. `mlp.x` trains a composed model with Adam and reloads it from a checkpoint; `mnist.x` trains a convolutional network for one epoch on the MNIST files named by `TORCH_MNIST`; and `jit-infer.x` runs a TorchScript model exported from Python. Training and Inference with torch is the chapter for the package.

Run the package-local examples from the repository root:

```
make -C packages/libcurl run
make -C examples/packages/http-json-releases test
make -C packages/termbox2 run
make -C packages/blis run
make -C packages/libuv run
make -C packages/raylib verify
make -C packages/torch run
```

SQLite rows and transactions

The SQLite package takes SQL text and binds ordinary x2c values. `Database.open` creates a file-backed or `":memory:"` connection. Prepare a statement once, bind a positional List or a Map of exact parameter names, and iterate copied row Lists. Rows preserve column order and duplicate names; `Prepared.columns` returns the names separately. Free statements before closing their connection, with `defer` beside each acquisition.

```
import "sqlite" with Database, Prepared;

void show_readings(String filename) {
  Database db = Database.open(filename);
  defer db.close();
  Prepared query = db.prepare(
    "SELECT url, status FROM observation WHERE status >= ? ORDER BY url"
  );
  defer query.free();
  query.bind(%(400));
  foreach (List row, query)
    printf("%s", %"${row[0]}: HTTP ${row[1]}\n");
}
```

SQL NULL is `Var.null()`, distinct from exhausted iteration. Integers retain SQLite's signed 64-bit domain; oversized unsigned inputs raise `conv-range`. Text becomes `String`, while blobs and text containing NUL become copied `Bytes`. These values survive later steps and statement release within their ordinary x2c lifetimes. `db.transaction(%!() => { ... })` commits on success and rolls back on Error; nested managed transactions are rejected. SQLite's own code, message, and operation remain available in Error details.

`SqliteLisp.install` adds query, execute, NULL, and byte-list operations to an embedded Lisp session. Each query or execute call opens its own connection, so use a database filename to retain changes between calls. Query results are ordinary nested Lists. The complete pinned raw SQLite API and the same native handles remain available for advanced operations. See `packages/sqlite/README.md` for ownership, the admitted native profile, and the executable examples. SQLite is verified on macOS and checked separately:

```
make -C packages/sqlite prepare
make -C packages/sqlite test run run-lisp
```

Install packages

Package management runs in the shipped `x2c-package` command. The `x2c install`, `x2c remove`, and `x2c list` forms select its operations. Project builds use the same command to prepare declared dependencies and maintain `x2c.lock` before compilation starts.

`x2c install` places a package under `<home>/packages`, where every build finds it without `--package-dir`:

```
x2c install pcre2                      # a name from the package index
x2c install ./mylib                    # a local package directory
x2c install ./mylib-1.2.tar.gz         # a local archive
x2c install https://example.com/mylib-1.2.tar.gz --sha256 <hex>
x2c list
x2c remove pcre2
```

A name resolves through the package index, `index.txt` at `https://x2c-lang.dev/packages/` unless `--index <url-or-path>` names another. Each index row is `name version kind platform url sha256`. The compiler takes the bundle row for its own platform, otherwise the source row, downloads the archive with `curl`, and rejects it when its digest differs. A URL given directly needs `--sha256`.

A bundle, the output of `make bundle` below, installs as built. Its `BUNDLE.json` records the x2c version that built it. Another compiler version rejects the bundle unless `--force` is given, because a bundle has no stable ABI across releases. A source package is a `<name>/src/<name>.x` tree with no `dependency*.json`. The compiler translates it in package mode and archives it, the same two commands `packages/package.mk` runs, and records the origin and digest in `SOURCE.json`. A directory or archive carries no version of its own, so installing one over an installed package keeps the version that package recorded and a first install records none, which `x2c list` shows as `-`. A source package with native dependencies is refused; publish its bundle instead.

Installed packages are not owned by the compiler's install inventory, so a compiler upgrade keeps them and `x2c remove` is the way to delete one. A directory under `<home>/packages` without `BUNDLE.json` or `SOURCE.json` is never replaced or removed. Installs and removals in one home run one at a time. A second one waits until the first finishes and, unless `-q` is given, prints a message while it waits.

Upgrade x2c before installing packages by name. Each release publishes the index that `x2c install <name>` reads, which lists the bundles built by that release's compiler. A bundle built by a different compiler version installs only with `--force`.

Pin packages in a project

A project manifest can list the packages it needs, and a build installs them. Each entry pins one exact version:

```
[dependencies]
pcre2 = "10.48"
```

`x2c build` and `x2c run` install anything missing through the index before planning the build, then write `x2c.lock` beside the manifest recording what they resolved. Commit that file. A later build whose lockfile is already satisfied by the installed packages reads no index, so it works offline, and one that has to reinstall a package takes it from the archive the lockfile recorded, so every build uses the same archives.

To change a version, edit the manifest and build again; the lockfile is rewritten. `x2c remove` still deletes an installed package, and the next build reinstalls it. See project manifests for the rules.

Movable native bundles

`make bundle` builds a package and assembles its public source interfaces, generated headers, wrapper archive, licenses, and declared native headers and static libraries:

```
make -C packages/yyjson bundle
mkdir native-packages
tar -xzf packages/yyjson/builds/bundle/yyjson-native.tar.gz -C native-packages
x2c build --package-dir native-packages app.x --output app
```

The directory `packages/yyjson/builds/bundle/yyjson` is also directly usable; register its parent with `--package-dir`. `BUNDLE_DIR=/path/to/output` changes the output parent. The directory and archive contain the same files. Move the whole package directory, keeping `src`, `builds`, `native`, and licenses together. Extracted bundle, compiler, header/archive, and consumer paths may contain spaces. Paths used while producing a package must not contain spaces.

A bundle is specific to its host platform, architecture, native profile, and matching x2c compiler/runtime. `BUNDLE.json` records those build identities. Native dependency toolchain information comes from its existing cache receipt when available; an explicit external prefix may have no receipt. A bundle provides no stable ABI across compiler releases, no cross-platform execution, and no automatic dependency resolution. Native system libraries and frameworks remain supplied by the host.

Most bundles link statically. libtorch is a set of shared libraries, so the torch bundle contains those libraries under `native/lib`, and a program built against it records a run-time search path to that directory. The program loads libtorch from the installed package, so moving or removing that package breaks the program.

Every package carries `builds/<name>.native.rsp`. The compiler reads its quoted native arguments, expands the literal `{package}` to the resolved package directory, and applies C include/define options during native compilation and ordered archives/system options during final linking. Arguments remain individual argv values even when an expanded path contains spaces. They are not shell commands and `@` does not expand another response file. Definitions do not change the preceding x2c source preprocessing.

Native bundle metadata covers yyjson, PCRE2, BLIS, libuv, termbox2, libcurl, raylib, and torch. Releases publish torch for macOS arm64 and Linux x86_64, the platforms PyTorch ships prebuilt CPU libtorch for. SQLite remains separately distributed as source in this release. Pure and mixed C/x2c packages with no external native inputs need no dependency manifest; their bundle carries an empty native response file. Distribute any other x2c package imports alongside them under the registered package root.

Libcurl bundles include its pinned OpenSSL static inputs in link order. Host libcurl uses the host's system certificate bundle.

Raylib also requires its native type declarations during x2c source discovery. Use the existing source include option for its bundled headers:

```
x2c build --package-dir native-packages \
  --x-include-dir native-packages/raylib/native/include chart.x --output chart
```

The default raylib bundle uses the headless profile. To produce its separately admitted macOS desktop profile, select its dependency manifest explicitly:

```
make -C packages/raylib clean-builds
make -C packages/raylib bundle DEPENDENCY_MANIFEST=dependency-desktop.json
```

Prepare and validate the selected native profile before distributing it. The bundle target preserves existing dependency checksum, header, and license checks through the ordinary package build; consumer builds do not download missing inputs.

Declare a package's native distribution

The optional `distribution` object in `dependency.json` lists selected files and directories with the existing `copies` shape and ordered native arguments:

```
"distribution": {
  "copies": [
    {"from": "{prefix}/include/yyjson.h",
     "to": "native/include/yyjson.h"},
    {"from": "{prefix}/lib/libyyjson.a",
     "to": "native/lib/libyyjson.a"}
  ],
  "native_args": [
    "--c-include-dir", "{package}/native/include",
    "{package}/native/lib/libyyjson.a"
  ]
}
```

Copy sources expand `{prefix}` to the prepared native prefix and `{package}` to the producer package. Destinations are relative to the bundle. An optional `archive_objects` list holds objects the package compiles outside `x2c build`, such as torch's C++ shim; they are added to the bundled `lib<name>.a`, so a consumer links them with no further input. Native arguments retain `{package}` for consumer-time expansion. An optional `platform_args` object appends arguments keyed by the producer's platform (`darwin` or `linux`) when a shared dependency manifest has platform-specific system requirements. The manifest that the package Makefile selects sets the

native profile. Unless `DEPENDENCY_MANIFEST` names one, the Makefile uses the first manifest present among `dependency-<os>-<arch>.json`, `dependency-<os>.json`, and `dependency.json`, where `<os>` is `darwin` or `linux` and `<arch>` is the machine name, such as `arm64` or `x86_64`.

Accepted options are native include directories (`-I`, `--c-include-dir`, `--c-system-dir`), `-D`, `-U`, `-L`, `-l`, `--rpath`, archive inputs, `-pthread`, and `-framework <name>`. A bundle of shared libraries passes its library directory to both `-L` and `--rpath`, under `{package}`. Keep native archives in dependency order after the wrapper archive. Options that replace the consumer's output, command, or tools and unrestricted compiler/linker escape options are not package metadata.

Wrapping a C Library

The previous chapter explains how to build and import a package. This chapter explains how to design one around a C library.

Provide an x2c API for ordinary use, with strings, collections, indexing, iteration, `defer`, and `Error` where they simplify the caller's work. Keep the pinned C API available under its own names for features the wrapper does not expose. The raw header exposes every C function; the wrapper adds x2c types and keeps the C names recognizable.

PCRE2 and yyjson illustrate the choices. Their packages are in `packages/pcre2/` and `packages/yyjson/`.

The library we are wrapping

The examples wrap a small fictional C library:

```
typedef struct feed_parser feed_parser;
typedef int (*feed_visitor)(const char *title, void *user);

feed_parser *feed_open(const char *bytes, unsigned long len);
void         feed_close(feed_parser *parser);
int          feed_next(feed_parser *parser, const char **title,
                       unsigned long *len);
int          feed_scan(feed_parser *parser, feed_visitor visit, void *user);
const char  *feed_strerror(int code);
```

A real package includes a pinned header here. The examples declare these six names in hidden lines, visible through the eye icon, so each example compiles on its own.

These six functions leave five things for the caller to manage:

Concern	Here
allocation and release	`feed_open` / `feed_close`, and nothing frees it for you
pointer plus length	`feed_next` yields `const char *` and a separate length
borrowed storage	that pointer is invalidated by the next `feed_next`
status versus absence	`feed_next` returns 1, 0 for end, negative for failure
callback and its state	`feed_scan` calls back into a C frame with a `void *`

Identify these obligations before designing types. For each value, decide whether to copy it, borrow it with a stated lifetime, convert it, or leave it as a native C value.

Also check for conditions this library does not have: process-global or thread-local state, callbacks on threads the caller did not start, and bytes that `String` cannot hold, such as embedded NUL.

Start with the tasks

Start by listing the tasks a developer brings to this kind of library. Then ask whether the imported x2c API can perform each task without requiring the raw C header or a hand-written loop over lower-level operations. Measure the wrapper by the tasks the x2c API performs.

For a regular-expression library such as PCRE2, the list might be:

- Compile a pattern, match, and read named and numbered captures.
- Select case-insensitive, multiline, dotall, or extended matching.
- Find every match in a subject.
- Read capture offsets and distinguish unmatched from empty captures.
- Replace the first match or all matches, with backreferences.
- Match at an anchor or from an offset.
- Match UTF-8 text and Unicode properties.
- Bound backtracking and JIT-compile a frequently used pattern.
- Report PCRE2's code, message, and offset for a bad pattern.
- Split a subject on a pattern.

- Replace each match with a computed value.
- Quote a literal for use inside a pattern.
- List the named groups a pattern declares.

Demonstrate each task with a small program. Reading the wrapper's source can miss an operation that several existing methods already express together. Features outside the x2c API must remain reachable through the raw header; name them in the README. For PCRE2 these include DFA matching, serialization, callouts, partial matching, and binary subjects.

Pin the library before you wrap it

A package with a native dependency carries a `dependency.json` naming the version, the source URL, the archive hash, and the build profile. `make prepare` fetches and builds it into a cache shared by every worktree of the clone and leaves a `deps` symlink in the package directory, so ordinary `make test` needs no network.

```
make -C packages/feedparse prepare
make -C packages/feedparse build
```

Document the build you provide: version and URL, archive and public-header hashes, compile options and disabled features, static or dynamic linkage, dependencies and platform frameworks, licenses and required notices, and any x2c limits on foreign declarations. The available raw API depends on these choices. `packages/FOREIGN-C-QUALIFIERS.md` records where x2c's observed types drop a `const` that the upstream header declares.

Keep the raw API reachable

Expose the raw API through a small header in the package's `src/`. It selects the build profile, rejects the wrong version at compile time, and includes the upstream header. Do not copy declarations, rename aliases, or add forwarding functions.

```
#pragma once

#include <feedparse.h>

#if FEEDPARSE_MINOR != 4
#error "this package pins feedparse 1.4"
#endif
```

Pin it under a versioned name. The generated header for a package unit is named after that unit, so a package called `feedparse` already produces `feedparse.h`; a vendored header of the same name beside it would be unreachable. `pcre2-8.h` and `yyjson-0.12.h` are the two spellings here.

Publish the shim when a public method takes one of the library's own types. The generated header needs that foreign declaration before it can publish a wrapper type that uses it.

The entry unit

The entry unit holds the public types and method definitions together. Nonstatic declarations are public; static helpers stay in the source.

```
typedef struct Feed *Feed;

Feed Feed.open(String text);
Feed Feed.close(Feed feed);
List Feed.titles(Feed feed);

static struct Feed {
  feed_parser *native;
  String source;
};
```

`Feed` is opaque to consumers: they hold the pointer and call methods on it. The public forward type does not require its static body, so only this source file needs the layout. A public value type or inline field access would require that definition in the header.

For functions with no receiver, a package can declare a type used only as a namespace. The runtime does this for `Json.parse`:

```
typedef Symbol Json;
```

Ownership and defer

Write the acquisition and release functions together. Release should be idempotent and return `NULL`, which makes a double close harmless and lets the caller clear their handle in the same expression.

```
Feed Feed.open(String text) {
  Feed feed = Scope.calloc(1, sizeof(struct Feed));
  feed.source = text.intern();
  feed.native = feed_open(text, text.len());
  return feed;
}

Feed Feed.close(Feed feed) {
  if (!feed) return NULL;
  if (feed.native) {
    feed_close(feed.native);
    feed.native = NULL;
  }
  return NULL;
}
```

`Regexp.free` and `JsonDocument.free` are both shaped this way.

`Scope` owns the small x2c record. With `Scope.calloc` it releases the `struct Feed` when its scope ends and does not call `feed_close`, so the caller must release the native resource. With `Scope.malloc_finalized` the record carries a finalizer that calls `feed_close` when the scope reclaims it, so unnamed temporaries, such as the results of operator expressions, are released with the region. Keep the explicit `free` method either way: it clears the native field, so after an early release the finalizer does nothing. See attaching a finalizer.

Adopt `Cleanup(T)` beside the type so a caller can write `$auto`. Without it the managed initializer is refused and every caller repeats the release. When one method releases the handle, `$cleanup.by` from `cleanup.x` defines `Feed.cleanup` to call it and adopts the protocol:

```
#include "cleanup.x"
$cleanup.by(Feed, close);
```

Keep the public adoption beside its converters. The release method may be defined later; its nonstatic declaration remains public. A cleanup that does more than one call is written as `void Feed.cleanup(Feed)` with `protocol Cleanup(Feed);`. A borrowed view over storage another handle owns adopts nothing.

The caller then names the lifetime at the acquisition:

```
static int count_titles(String document) {
  Feed feed = $auto(Feed.open(document));
  return feed.titles().len();
}
```

Where a type has no `Cleanup` adoption, or the acquisition is a native call the wrapper does not own, the caller puts the release on the line after the acquisition:

```
static int count_titles(String document) {
  Feed feed = Feed.open(document);
  defer feed.close();
  return feed.titles().len();
}
```

A `defer` in a function that returns the handle closes it before the caller ever sees it. The release belongs to whoever will finish with it. A handle whose operations reuse internal state is not reentrant: state that in the README. One `Regexp` cannot be shared across threads for this reason.

Copy, borrow, convert, or stay raw

Copy results a later call invalidates

Copy a result that another native call will invalidate, or retain a separate owner for it. `feed_next` hands back a pointer into storage the next call reuses, so the wrapper copies each title into an immutable `String` as it goes:

```
List Feed.titles(Feed feed) {
  Array titles = [];
  const char *title = NULL;
  unsigned long length = 0;
  int status;

  while ((status = feed_next(feed.native, &title, &length)) > 0)
    titles.push(String.new_len(title, (int) length));
  if (status < 0) _feed_error("next", status);
  return titles.list_free();
}
```

PCRE2 reuses match data on the next call. The wrapper copies captures into immutable `Lists` so later matches do not change earlier results. The copied results have the lifetime of their canonical pools.

Borrow views into an owned document

When the native graph is large, ordered, duplicated, or lazily walked, own the document and hand out borrowed views, with an explicit invalidation rule. A view keeps a reference to its owner and checks it:

```
typedef struct FeedEntry {
  Feed owner;
  const char *title;
  int length;
} *FeedEntry;

static void _feed_live(FeedEntry entry) {
  if (!entry || !entry.owner || !entry.owner.native) {
    raise %(bad-state (library "feedparse") (operation "entry")
             (reason "the feed that owned this entry is closed"));
  }
}

String FeedEntry.title(FeedEntry entry) {
  _feed_live(entry);
  return String.new_len(entry.title, entry.length);
}
```

yyjson preserves object order, duplicate names, and the distinction between signed, unsigned, and real numbers. Its wrapper owns the native document and returns borrowed views that retain those distinctions. Call `to_x2c` explicitly when that information is unnecessary.

Make lossy conversion opt-in

When conversion does not cover the native value's whole domain, name the loss and make conversion opt-in.

Keep bytes x2c cannot hold on the raw API

When x2c cannot represent the bytes, keep that path raw. `String` excludes embedded NUL, so PCRE2's explicit-length binary matching stays on the raw header.

Add indexing and iteration when they simplify the caller's code. Inside a protocol member named `getindex`, a bare `item.getindex(...)` would resolve to the member itself, hence the cast:

```
typedef List FeedItem;
typedef struct FeedItemIndex *FeedItemIndex;

protocol FeedItemIndex(T) {
  associated Key = Var;
  associated Value = String;

  Value T.getindex(T, Key);
```

```
}

protocol FeedItemIndex(FeedItem);

String FeedItem.getindex(FeedItem item, Var key) {
  if (!key.is_integer()) return NULL;
  return ((List) item).getindex(key.int());
}
```

Do not add a protocol, operator, macro, or wrapper type to display a language feature. An operator that creates native resources must also give its intermediate results an automatic lifetime; concise syntax must not hide manual cleanup.

Errors carry the library's codes

A failure raises one structured `Error` carrying the library name, the operation, the library's own numeric code, its own message, and a location when it has one. Do not invent a second set of status codes that callers have to translate back.

```
static void _feed_error(String operation, int code) {
  String message = String.new((char *) feed_strerror(code));
  raise %(malformed (library "feedparse") (operation $operation)
          (code $code) (message $message));
}
```

Pick the cause from the shared codes described in Exceptions and Cleanup: `<malformed>` for bad external input, `<bad-state>` for a stale or released handle, `<alloc-fail>` for a failed native allocation. When the library reports only a message and no code, raise one documented cause with the text under `(message ...)` and no `code`.

An error helper shared by several callers must take the operation name as an argument. Otherwise a failed file write, for example, could be reported as a failed string conversion.

Expected absence is not a failure. No match, exhausted iteration, a missing key, and a timeout's "no event yet" all stay on the return value. `feed_next` returning 0 above ends the loop instead of raising, and `Regexp.match` returns an empty result instead of an error.

An `Error` must never unwind through a C frame. When the library calls you back, `catch` at the boundary, record what has to survive, return a status the library understands, and `raise` once x2c is back in control:

```
typedef struct FeedScan {
  Array titles;
  int failed;
} *FeedScan;

static int _feed_visit(const char *title, void *user) {
  FeedScan scan = (FeedScan) user;
  try {
    scan.titles.push(String.new((char *) title));
  }
  catch: {
    scan.failed = 1;
    return 0;
  }
  return 1;
}

Array Feed.scan(Feed feed) {
  struct FeedScan scan = { [], 0 };
  int status = feed_scan(feed.native, _feed_visit, &scan);

  if (scan.failed) {
    raise %(bad-state (library "feedparse") (operation "scan")
            (reason "a title could not be copied"));
  }
  if (status < 0) {
    String message = String.new((char *) feed_strerror(status));
    raise %(malformed (library "feedparse") (operation "scan")
            (code $status) (message $message));
```

```
  }
  return scan.titles;
}
```

Check which thread invokes each callback. It must be safe to use the x2c runtime on that thread before the callback calls into it.

Write two example applications first

Sketch two applications before implementing the wrapper: one short and one more complete. Read the short one first.

One short application should fit on a screen, do one useful thing, and make the result more prominent than the setup. One broader application should exercise the library's defining operations, ownership, results, and failure paths while still performing a task rather than enumerating outcomes.

```
import "feedparse" with Feed;

int main(void) {
  Feed feed = Feed.open(File.open("headlines.xml", "r").string_close());
  defer feed.close();

  foreach(String title, feed.titles())
    printf("%s", %"- $title\n");
  return 0;
}
```

If the applications still require the same setup, conversion, and cleanup as the C API, revise the wrapper around those tasks. Keep exhaustive tests of malformed input, limits, and cleanup separate from the examples.

Read the four in this repository in this order: `packages/pcre2/examples/parse-log.x` (35 lines), `packages/yyjson/examples/service-health.x` (24 lines), then `packages/pcre2/examples/request-report.x` and `packages/yyjson/examples/release-catalog.x`.

Include Lisp bindings in the package

If the library is useful from Lisp, include its bindings in the package behind one public installer. Bindings defined only in an example are unavailable to other importing programs.

```
typedef enum FeedLisp {
  FEEDLISP_NAMESPACE
} FeedLisp;

$lisp.binding(feed_lisp, "feed-titles")
static List _lisp_feed_titles(String document) {
  Feed feed = Feed.open(document);
  defer feed.close();
  return feed.titles();
}

void FeedLisp.install(Lisp lisp) {
  $lisp.install(lisp, feed_lisp);
}
```

The group compiles into the archive, so a consumer that only links it gets every binding from one call:

```
import "feedparse" with FeedLisp;

int main(void) {
  Lisp lisp = Lisp.new();
  defer lisp.destroy();
  FeedLisp.install(lisp);

  printf("%d\n", lisp.eval(%(length (feed-titles "<rss/>"))).int());
  return 0;
}
```

Bindings should take and return values that Lisp can use. Allocate and free native handles within each call instead of returning opaque objects to the session. Use x2c when a handle must persist between calls.

Test the results with ordinary Lisp operations. A parsed JSON array is an x2c `Array`, and an object is a `Map`; Lisp's `car`, `cdr`, and `length` accept neither. The package supplies `json-len`, `json-keys`, and `json-list` for those values. A parse-and-serialize test alone would not show whether Lisp could use them.

Test both APIs

The x2c tests cover the paths the two applications use, plus ownership and idempotent release, use after release, absence separately from failure and from a present `Null`, boundary sizes and embedded NUL where the library allows them, iteration and indexing including empty and duplicate cases, the native error detail, and every deliberate conversion loss.

One further test calls representative upstream declarations directly through the shim. This checks that the raw API is accessible without repeating upstream's own test suite.

```
make -C packages/feedparse test
```

A test must check the effect rather than a round trip. Writing a document with a pretty-print flag and then re-reading and re-serializing it compactly passes identically with the flag absent. Assert on the bytes in between.

Completion checklist

- The imported x2c API supports every task on your list.
- Everything left off it is reachable through the shim and documented as such.
- Both example applications work, and the short one fits on a screen.
- Ownership, invalidation, non-reentrancy, and every conversion loss are written down and tested.
- `Error`s carry the library's own code and message; absence is not an error.
- Lisp bindings, if provided, are in the package.
- The source version, build options, hashes, and licenses are recorded.

Reread the short application as a new reader would, and revise it until a new reader can follow it.

See `packages/pcre2/README.md` and `packages/yyjson/README.md` for complete examples of package documentation.

Commands and Files

A script works with three kinds of value. A command is a `List`. A running or finished command is a `Job`. A filesystem location is a `Path`. Every failure is an `Error` that a `catch` can select.

The modules are optional in ordinary programs. Include the ones a file uses:

```
#include "process.x"
#include "path.x"
#include "args.x"
#include "digest.x"
#include "regex.x"
#include "diff.x"
```

Write a script

A source file whose first line is a shebang is a script. It can use the modules above without including them, and its statements at file scope run in order as the program:

```
#!/usr/bin/env -S x2c script
String branch = %(git rev-parse --abbrev-ref HEAD).job().output();
foreach (String file, %(git diff --name-only).job().lines())
  printf("%s: %s\n", branch, file);
```

`args` holds the arguments after the script's name as `Strings`. Script units in the language reference describes which forms stay at file scope, how a script that defines `main` runs, and the exit status of an uncaught failure.

`x2c script` builds the file once and then runs the cached executable. The first run builds it; later runs start the cached executable in a few milliseconds until the script or something it uses changes:

```
x2c script tools/release-notes.x v0.12.0
```

A script can include its own local `.x` modules, which `x2c script` builds and links with it. A script with an executable mode runs by name through its shebang line. Every argument after the file reaches the program unchanged. The command reference describes the cache and the changes that make a script build again.

Commands

Each element of a command `List` becomes one argument. The words reach the program without a shell, so a value inserted with `$` stays one argument whatever it contains:

```
String message = %"it's two words";
%(git commit --allow-empty -m $message).job().run();
```

Bare words in a command are x2c atoms, so most flags and names need no quotes. Write a word as a string when it contains punctuation with another meaning in collection syntax: a comma, `$`, `@`, or a redirection such as `2>&1`. Inside a string, `$` still interpolates; write `\$` for a literal dollar sign.

`job` turns a command into a `Job`, and a command converts to a `Job` where one is expected. The first request for a result starts the program, waits for it, and records what it did. Every later request reads that same record, so a job runs exactly once however many results you take from it.

Reading results

Method	Result
`status()`	the exit status, or 128 plus a signal
`output()`	captured standard output
`lines()`	captured standard output split into lines, without endings
`errors()`	captured standard error, or NULL when it was not captured
`check()`	the job itself; raises when the status is not zero
`run()`	shows output live, waits, and raises when the status is not zero

By default a job captures standard output and passes standard error through to the terminal, the way the shell's `$(...)` does. The script receives the output, and error messages appear on the terminal.

```
Job diff = %(git diff --stat);
if (diff.status() == 0)
  printf("%ld lines of summary\n", (long) diff.lines().len());
```

Showing output live

Call `live` before a job starts to show its output on the terminal, as for a build or a test run. A live job does not capture standard output, so `output` and `lines` return NULL. `run` is the short form of `live().check()`:

```
%(make all).job().run();
if (%(make test).job().live().status())
  printf("tests failed\n");
```

Failures

`check`, `run`, `output`, and `lines` raise `<cmd-fail>` when the status is not zero. The detail has `command` and `status` entries, plus `output` and `errors` entries when they were captured. `status` and `errors` do not raise for a nonzero status, so a script can read standard error after one. When a program cannot start, the job raises `<not-found>` for a missing program or directory, and `<io-fail>` for anything else:

```
try %(git pull --ff-only).job().run();
catch %(cmd-fail *detail):
  printf("pull failed with status %ld\n", detail.assoc(<status>).integer());
catch %(not-found *):
  printf("git is not installed\n");
```

In a script, a `<cmd-fail>` that nothing catches ends the script with the command's status.

Captured output is text. When the captured bytes contain a NUL, `output`, `lines`, and `errors` raise `<bad-arg>`, and `status` still reports the run. Send binary output to a file with the `stdout` option instead.

Pipelines

A `List` of commands is a pipeline, and `pipe` adds a stage to a job. These two jobs are the same:

```
List units = %((git ls-files) (grep "\\.x\$")).job().lines();
List same = %(git ls-files).job().pipe(%(grep "\\.x\$")).lines();
```

The job's status is the status of its last failing stage, so a failure in an early stage sets the status even when a later stage succeeds.

Options

`options` attaches settings before the job starts:

```
Path root = "build";
%(make all).job().options({dir: root, env: {CC: <clang>}}).run();
String upper = %(tr a-z A-Z).job().options({input: "quiet"}).output();
```

Key	Value
`dir`	working directory
`env`	a `Map` of variables added to the inherited environment
`input`	a `String` given to the first stage as standard input
`stdout`	`capture` (the default), `inherit`, or a file path
`stderr`	`inherit` (the default), `capture`, `stdout` to merge, or a file path

Without `input`, the first stage reads the script's standard input, and an empty `input` gives it an empty one. `stdout` applies to the last stage of a pipeline, and `stderr` to every stage. `live()` is the same as `options({stdout: <inherit>})`. An unknown key, such as a misspelled `dir`, raises `<bad-arg>`. Changing options or adding a stage after the job has started also raises `<bad-arg>`.

Background jobs

`start` begins a job without waiting and returns the job. `ready` reports whether a started job has finished. `Job.wait_any` removes and returns the first finished job from an `Array`. This loop uses it to bound how many jobs run at once:

```
Array running = [];
foreach (Var host, %(alpha beta gamma delta)) {
  if (running.len() == 2) Job.wait_any(running).check();
  running.push(%(ping -c 1 $host).job().live().start());
}
while (running.len()) Job.wait_any(running).check();
```

`kill` sends a signal to every stage still running. A job held with `$auto` is terminated and reaped if it is still running when its block exits, including when an error leaves the block. Each stage receives `SIGTERM`, and a stage still running a second later receives `SIGKILL`. A job abandoned without `$auto` is terminated and reaped the same way when the Scope that owns it ends:

```
{
  Job server = $auto(%(python3 -m http.server 8000).job().live().start());
  %(curl -fsS "http://localhost:8000/").job().run();
}
```

Paths

A `Path` is a `String` that holds a filesystem location. A literal or a `String` converts to a `Path` wherever one is expected, and a `Path` passes wherever a `String` parameter is expected. Only a `Path` has the file methods.

```
Path source = "src/parse/parse.x";
Path object = Path.join("build", %"${source.stem()}.o");
```

A method can be called on a `Path` value, or through the type with a literal:

```
if (Path.exists("build")) Path.remove_tree("build");
foreach (Path unit, Path.glob("src/**/*.x"))
  printf("%s %ld\n", unit, unit.size());
```

Examining a path

`join`, `dirname`, `basename`, `stem`, `extension`, and `glob_match` operate on the text alone. `stem` and `extension` return a `String`; the others that return text return a `Path`. `absolute` resolves a path against the working directory.

Asking about a path

`exists`, `is_dir`, `is_file`, `is_executable`, `size`, and `modified_time` query the filesystem. `modified_time` keeps the fraction of a second that the filesystem records.

Listing

`list_dir` returns the sorted names in one directory. `walk` lazily yields every path below a directory. `glob` returns the paths that match a pattern, where `**` matches any number of directories. As in a shell, a wildcard does not match a leading dot, so `*` skips `.git` while `.*` finds it. A pattern that ends with a slash, such as `src/*/`, finds only directories.

Changing the filesystem

`make_dirs`, `remove_file`, `remove_tree`, `copy_file`, `copy_tree`, `move_to`, and `symlink_to` change the filesystem. Removing something that is already gone succeeds. Copying a file onto itself leaves it unchanged, and copying a directory into itself raises `<io-fail>`. `read_text` and `write_text` read and replace a whole file. `Path.temp_dir` creates a private directory:

```
Path work = Path.temp_dir();
work.join("out").make_dirs();
work.join("out/report.txt").write_text("done\n");
```

```
printf("%s", work.join("out/report.txt").read_text());
work.remove_tree();
```

A missing path raises `<not-found>`, and any other failure raises `<io-fail>`. Both details have `operation` and `path` entries.

`+` joins a `Path` with a `String`, another `Path`, or a C string literal and produces a `String`, as in `path + ".o"`. Slicing is a `String` operation. Convert a `Path` to a `String` to slice it.

`examples/scripts/line-counts.x` combines commands and paths: it builds a small tree, counts lines with parallel jobs, and writes a report.

Environment

`Env.get` returns the value of one of the script's own environment variables, or NULL when it is unset. The `env` option sets variables for a child instead:

```
String site = Env.get("X2C_SITE");
if (!site) site = "https://x2c-lang.dev";
```

Arguments

`Args.parse` reads an argument `List` against a spec, which is a `List` with one row per option or operand, and returns a `Map` from each row's name to its value. In a program with `main`, `Args.from_argv` makes that `List` from `argc` and `argv`; a script already has it as `args`.

```
List spec = %(
  (-v --verbose (help "Report each input"))
  (-o --output (value file) (default "report.txt") (help "Write <file>"))
  (-I --include (value dir) repeated (help "Also search <dir>"))
  (inputs repeated required (help "Files to read")));
Map options = Args.parse(Args.from_argv(argc, argv), spec);
String output = options["output"].str();
foreach (String input, options["inputs"])
  if (options["verbose"]) printf("%s: %s\n", output, input);
```

A row that begins with dashed words is an option with those spellings. Its name is the first long spelling without the dashes, or the short spelling when it has no long one, so the rows above are named `verbose`, `output`, and `include`. A row that begins with any other word is an operand of that name. The rest of a row describes it:

Row part	Meaning
`(value file)`	the option takes a value, shown as `<file>` in usage
`(default "report.txt")`	the value when the row is not given
`(help "text")`	the row's description in usage text
`required`	a missing option or operand raises `<bad-arg>`
`repeated`	every value is kept in a `List`; an operand takes the rest

A long option takes its value as `--output out.txt` or `--output=out.txt`, and a short one as `-o out.txt` or `-oout.txt`. Short flags may share a word, as in `-vv`. Operands may come before, between, or after options, and every word after `--` is an operand. Operand rows take operands in spec order.

Every name in the spec is present in the result. A flag holds the number of times it appeared, a `repeated` row holds a `List`, and any other row holds its last value as a `String`. A row that was not given holds its default, or else zero, an empty `List`, or a NULL `String`, all of which test false. Values stay `Strings`; convert one to a number where a number is needed.

An unknown option, a missing or unexpected value, an extra operand, or a missing `required` row raises `<bad-arg>` with `reason` and the offending `option` or `operand`. The script chooses the message and the exit status. `Args.usage` returns help text generated from the same spec:

```
List spec = %(
  (--prefix (value path) required (help "Install under <path>"))
```

```
  (-q --quiet (help "Print nothing on success")));
try {
  Map options = Args.parse(Args.from_argv(argc, argv), spec);
  String prefix = options["prefix"].str();
  if (!options["quiet"]) printf("installing to %s\n", prefix);
}
catch %(bad-arg *detail): {
  Stderr.printf("%s", Args.usage("install.x", spec));
  return 2;
}

Usage:
  install.x [options]

Options:
      --prefix <path>         Install under <path>
  -q, --quiet                 Print nothing on success
```

A script with subcommands reads the first word itself and parses the rest with that command's spec. `etc/x2c-payload.x`, which `make install` runs, works this way.

Checksums

`digest.x` computes SHA-256 digests, spelled as the lowercase hexadecimal `String` that `shasum -a 256` prints. `String.sha256` hashes the text of a `String`. `File.sha256` hashes the raw bytes from a stream's position to its end, so it also works on binary files that `read_text` rejects:

```
File shell = $auto(File.open("/bin/sh", "rb"));
printf("%s  /bin/sh\n", shell.sha256());
String abc = "abc".sha256();
```

Patterns

`regex.x` matches regular expressions over the bytes of a `String`. A `Regex` is a compiled pattern:

```
Regex row = Regex.compile(%"^ *(?<count>\\d+) (?<file>.+)$$");
foreach (String line, %(wc -l lib/regex.x lib/path.x).job().lines()) {
  RegexMatch found = row.match(line);
  if (found) printf("%s: %s\n", found[<file>], found[<count>]);
}
```

Since `%"..."` interpolates `$`, write `$$` for an end anchor, or use a plain C string literal for the pattern.

Syntax

A pattern is text with these forms. Anything else matches itself.

Form	Matches
`.`	any byte except newline
`[abc]`, `[a-z]`, `[^0-9]`	one byte in, or not in, the set
`\d` `\w` `\s`, `\D` `\W` `\S`	a digit, word byte, or space, or their opposites; also inside `[...]`
`\b` `\B`	a word boundary, or its absence
`^` `$`	the start and end of the text
`x*` `x+` `x?` `x{m}` `x{m,}` `x{m,n}`	repetition, as many times as possible
`x*?` `x+?` `x??` `x{m,n}?`	repetition, as few times as possible
`a|b`	either
`(x)`, `(?<name>x)`	a numbered capture, also named
`(?:x)`	a group that does not capture
`\.` `\\` `\t` `\n` `\r`	a literal byte
`(?i)` `(?m)` `(?s)`	at the start: ignore ASCII case, let `^` and `$` match at line breaks, let `.` match newline

Matching is byte by byte, so `.` and `\w` treat a multi-byte character as several bytes, and `(?i)` folds only ASCII letters. There is no lookahead, lookbehind, or backreference. The pcre2 package has all of those and Unicode. Its types are `Regexp`, `RegexpMatch`, and `RegexpCapture`, with the same methods. To switch to it, import the package with those names, rename the types, and add the options argument to `compile`.

A pattern that does not parse raises `<malformed>` with `reason`, the `pattern`, and the zero-based byte `offset` of the problem. So does a count above `INT_MAX`, 2147483647.

Matching

`match` returns the first match in the text, or NULL. `match_from` starts at a byte offset. `find_all` returns every non-overlapping match as a `List`. A `RegexMatch` is indexed by capture number, or by a name as a `Symbol` or `String`. The whole match is capture 0. A capture that did not take part is NULL, and so is one that matched no bytes, since an empty `String` is NULL; `matched` distinguishes the two:

```
Regex pair = Regex.compile("(\\w+)=(\\w*)");
foreach (RegexMatch found, pair.find_all("a=1 b= c=3"))
  printf("%s -> %s\n", found[1], found[2] ? found[2] : "(empty)");
```

`capture(key)` returns the `RegexCapture` behind an index, with `text`, `start`, `end`, and `matched`. `capture_names` lists a pattern's names in capture order, and `pattern` returns its text.

Splitting and replacing

`split` returns the text between matches, keeping an empty field where two matches touch or a match sits at either end. `replace` replaces the first match and `replace_all` every match. In the replacement, `$0` through `$9` and `${name}` insert a capture, and `$$` is a dollar sign. `replace_fn` calls a function with each `RegexMatch` and inserts what it returns as is. `Regex.escape` quotes a literal for use inside a pattern:

```
Regex spaces = Regex.compile(" +");
List words = spaces.split("  two   words ");        // "", "two", "words", ""
String dated = Regex.compile("(\\d+)-(\\d+)")
  .replace_all("2026-09 and 2027-01", "$2/$1");      // 09/2026 and 01/2027
String shouted = Regex.compile("\\w+")
  .replace_fn("go now", %!(RegexMatch m) => m[0].upper());
```

A `Regex` is freed by the scope that made it, like any class value. Matching backtracks, so a pattern with nested repetition such as `(a*)*b` can take time exponential in the text; write the repetition once. A repeated byte or set, such as `\w*`, matches any length, but a repeated group such as `(?:ab)*` raises `<size-limit>` past 2,000 repetitions in one match.

Comparing text

`diff.x` compares two texts line by line. `Diff.unified` returns the difference the way `diff -u` prints it, with the two names in the header and three lines of context around each change, or NULL when the texts are the same line for line. This check compares a command's output with expected output stored in a file:

```
String expected = Path.read_text("unittest/expected/help.txt");
String actual = %(./x2c --help).job().output();
String report = Diff.unified(expected, actual, "expected", "actual");
if (report) Stderr.printf("%s", report);
```

`Diff.lines` returns the edits themselves: a `List` of `(same line)`, `(delete line)`, and `(insert line)` forms in order, with line endings removed:

```
foreach (List edit, Diff.lines("a\nb\n", "a\nc\n")) {
  (Symbol kind, String line) = edit;
  if (kind != <same>) printf("%s %s\n", kind.str(), line);
}

delete b
insert c
```

The edits are a shortest script for texts that are mostly alike. Past two thousand edits the differing middle is reported as one run of deletions and one run of insertions.

JSON

A script that reads or writes JSON includes `json.x` itself:

```
#include "json.x"
```

The names in `json.x` match the converting surface of the `yyjson` package. To switch a script to the package, replace that line with `import "yyjson" with Json;`.

`Json.parse` reads JSON text and `Json.read_file` reads a JSON file. The result is made of ordinary values: an object is a `Map` with `String` keys, an array is an `Array`, a string is a `String`, and a number is an integer or `double` `Var`. JSON null is the all-zero `Var`. `true` and `false` are `JsonBool` values, which test true and false in a condition and stay distinct from the numbers 1 and 0:

```
Map release = Json.read_file("release.json");
if (!release["draft"]) printf("release %s\n", release["version"]);
foreach (Map asset, release["assets"]) printf("  %s\n", asset["name"]);
```

An integer that fits in an `int` reads as one, then as a `long` or an `unsigned long`; every other number is a `double`. A repeated object name keeps its last value.

`Var.json` returns compact text, `Var.pretty_json` indents two spaces per level, and `Json.write_file` writes the compact form to a file. Object names are written in byte order, so equal values always produce the same text, and the indented layout is the one Python's `json.dumps` produces with `indent=2` and `sort_keys=True`. A bare key in a `{}` literal is a `Symbol`, which is written as a string, and `Json.bool` makes a boolean. A `Map` or `Array` variable passes to the writers as a `Var`:

```
Map report = {name: "x2c", passed: Json.bool(1), counts: [3, 0]};
printf("%s\n", Var.pretty_json(report));
Json.write_file(report, "/tmp/report.json");
```

Text that is not JSON raises `<malformed>` with `reason`, a zero-based byte `offset`, and one-based `line` and `column` details; `Json.read_file` adds the `path`. Parsing raises the same cause for nesting deeper than 512 arrays and objects, a number too large for a `double`, an unpaired surrogate escape, and `\u0000`, which a `String` cannot hold:

```
try Json.read_file("settings.json");
catch %(malformed *detail):
  printf("settings.json:%ld:%ld: %s\n", detail.assoc(<line>),
         detail.assoc(<column>), detail.assoc(<reason>));
```

Writing raises `<bad-types>` for a value JSON cannot hold, such as a `File` or a `Map` key that is not a `String` or `Symbol`, and `<conv-range>` for NaN or an infinity. A `String` that is not valid UTF-8 is written with U+FFFD in place of each ill-formed byte sequence, the same text that Python and JavaScript produce when they decode those bytes, so the output is always UTF-8 that `Json.parse` accepts.

The yyjson package preserves what a `Map` does not: object order, duplicate names, and whether a number was signed, unsigned, or real. Its converted integers are `long long` or `unsigned long long` values, which do not compare equal to an `int` literal such as the 5 in `{n: 5}`. The package raises `<malformed>` for malformed text.

Experimental REPL

`x2c repl` is a shipped external command that evaluates a supported subset of x2c in a persistent session. The command compiles prompt function definitions to native code. Top-level initializers, statements, and expressions lower to ordinary Lisp and run in the session's evaluator. Compiler macros use native meta compilation and helper execution.

```
x2c repl
x2c repl --help
x2c repl --dump --stats
```

No source file or checkout is needed when using an installed development build. The command reads standard input and accepts no file operands. It also runs directly as `x2c-repl` from the installation's `libexec/x2c` directory.

Try a session

Enter these submissions in order:

```
x2c> int total = 10;
ok
x2c> int plus(int x) { return total + x; }
defined plus
x2c> plus(3);
=> 13
x2c> total = 20;
ok
x2c> plus(3);
=> 23
```

The function calls display `=> 13` and `=> 23`: later calls read the current value of `total`. Definitions display `defined NAME`; initializations and statements display `ok`. A final expression displays its value using `Var.repr`, so character and unsigned values retain their printed tags.

A function is compiled once, when its complete definition is submitted. Compilation needs the matching x2c compiler and host C toolchain. Later calls use the compiled code. Prompt variables have shared native storage: both compiled functions and interpreted input read and write the same objects. Assignment preserves each variable's address, including pointers to record fields. `&total` therefore refers to the same object from either path.

Functions can call earlier functions and recurse directly. A saved `Func` keeps the same bindings, so changes to shared variables remain visible:

```
Func saved = plus;
total = 30;
saved(3);  // 33
```

Submit one declaration or function at a time. Executable input may contain several statements. Semicolons are required. Incomplete input continues at the `...` prompt until it forms a complete submission. Put a multiline `if`/`else` in a block when necessary: a complete `if` can execute before the next line supplies an `else`.

Interactive terminals support the arrow keys, Home, End, Backspace, Delete, and up/down history recall. History holds the last 100 nonempty commands and completed source submissions for the current process; adjacent duplicates are stored once, and history is not written to disk. A rejected or failed submission remains available for correction.

Tab completes names from the compiler's live session namespace, offering only names that fit the grammar position: type positions offer types and type-producing macros, expression positions offer values and callable forms, and statement, block, and submission starts offer their legal union plus contextual keywords. Narrow continuation contexts offer `else`, `catch`, or `finally` where each is legal. Fields and evaluator-callable methods remain available on a typed receiver. At a blank primary prompt completion also offers colon commands, grouped with keywords, session names, types, and other callables. Unsupported REPL declarations and imports are not suggested.

A colon-prefixed line completes commands even at a continuation prompt; `:ast` and `:lowered` then complete published function names, and `:stats` completes `verbose`. A sole match replaces the current name; otherwise

Tab inserts the common prefix, and another Tab lists compact, grouped choices. Completion parses only through the cursor and rolls back its semantic work, so it can use locals from pending multiline input without publishing them. APIs become candidates when their ordinary declarations and compile-time bindings become visible.

Long input wraps across terminal rows. Bracketed paste keeps embedded newlines inside one editable entry, so a pasted multiline submission can be accepted with one Enter. Recalled multiline submissions are also one entry. The editor folds these entries for display and moves over the folded range as a unit; it does not provide arbitrary cursor movement among every line of a whole submission. Typing Enter still submits the current edited entry, and an incomplete submission continues at a fresh `...` prompt.

Inspect and control the session

Command	Meaning
`:help`	Show session commands.
`:stats [verbose]`	Show concise or detailed runtime statistics.
`:symbols [PREFIX]`	List successfully defined names and kinds, optionally matching a name prefix.
`:load "file.x"`	Rebuild all loaded files together, expose this file's public names, and reset loaded native state.
`:info NAME`	Show a session function's argument and return types, source, and execution mode.
`:ast plus`	Show `plus`'s typed x2c AST.
`:lowered plus`	Report that `plus` is compiled and has no Lisp body.
`:cancel`	Discard incomplete input.
`:quit`	Exit, discarding incomplete input.

For the example above, `:symbols` prints the list template `%((function "plus") (value "total"))`. Typed AST output begins with `typed: %(`; `:lowered plus` prints `compiled: native function; no Lisp body`. Inspection does not execute the function, and earlier functions remain inspectable after later submissions. Values have symbol entries but no function AST.

`:symbols pl` lists only names beginning with `pl`. `:info plus` prints a declaration, `source: prompt`, and `execution: native`. For a loaded function, the source is the loaded file's resolved absolute path. Parameter names are omitted from the printed declaration; argument and return types remain. Tab completion for `:info` includes both prompt functions and loaded functions.

Help and inspection commands preserve incomplete input, including when a command reports an error. Colon commands are recognized on separate lines even inside an incomplete string or comment. Missing names, extra arguments, and unknown commands report errors.

`--dump` prints each available typed AST and lowered Lisp form to standard error. `--stats` prints the concise runtime report at exit, to standard error. `--verbose-stats` prints the detailed report and implies exit reporting; when both statistics options are present, one verbose report is printed. The options can be combined with `--dump`.

The report separates absolute live or retained quantities from activity since the REPL opened. Scope allocation objects and live requested bytes are process-wide runtime values; requested bytes count public managed payload. Pool active and depot bytes partition retained backing capacity. Scope and Pool figures overlap and must not be summed. Verbose reports add Scope and Pool traffic and capacity details.

Load functions from a file

Save this source as `numbers.x`:

```
static int twice(int n) { return 2 * n; }
static int calls = 0;
int answer(int n) { calls++; return twice(n + 1); }
int call_count(void) { return calls; }
```

Load it and call its functions with ordinary prompt expressions:

```
x2c> :load "numbers.x"
loaded ("answer" "call_count")
reset all loaded native state
x2c> answer(20);
=> 42
x2c> call_count();
=> 1
x2c> int plus_answer(int n) { return answer(n) + 1; }
defined plus_answer
x2c> plus_answer(20);
=> 43
```

`:load` builds all loaded files into one native helper program. It uses the ordinary include, module, package, and native dependency handling. Public functions with bodies in the loaded file become prompt functions, including `main`. No `meta` annotation is needed. Static helpers such as `twice` remain private. Functions from included files and declaration-only functions remain available to the compiled code but are not imported into the prompt. Public type declarations accompany the loaded definitions. Enumerator constants are not imported. Source macros remain with the compiled source; loading does not install them in the prompt.

Public scalar globals are also available without source changes. Reads and assignments use the same native storage as the file's functions:

```
int counter = 10;
const int limit = 100;
static int private_counter;
int read_counter(void) { return counter; }
```

After loading this file, `counter;` returns `10`. Entering `counter = 20;` makes `read_counter();` return `20`. Compound assignments and numeric prefix or postfix increments also work. Prompt functions can read and assign these globals, including functions saved before a reload. `limit` is read-only, and `private_counter` remains private. `:symbols` lists the public globals.

Supported scalar values are numbers, booleans, enums, Strings, and Symbols, including typedef aliases. Enum values use the compiler's `int` representation. Fixed-size C arrays and structs also expose their scalar fields and elements, including nested arrays and structs:

```
int samples[3] = {10, 20, 30};
struct Point { int x; int y; };
struct Point point = {1, 2};
```

After loading this file, `samples[1];` returns `20`. Assignments such as `samples[1] = 99;` and `point.x = 7;` change the file's native storage. Compound assignments and increments work too. Array indices are checked against the native array's extent and evaluated once per access. These operations also work inside compiled prompt functions.

Enter `samples;` or `point;` to inspect the whole object. Arrays display as Arrays and structs as Maps keyed by field name. Inspection copies the data and visits at most 100 fields or elements, including nested containers; `"..."` marks truncated output. Unsupported fields display `"<unavailable>"`. Inspection is available for the final expression at the prompt. Native arrays and structs are not copied across the helper boundary as ordinary values. They can be passed to loaded functions inside the helper, as shown below. Assign fields and elements individually.

Static globals stay private. Const objects and const fields are read-only. Raw pointers, unions, flexible array members, collections, `Var`, and callable globals do not gain accessors. Included globals become visible at the prompt when their defining file is explicitly loaded. A loaded object can supply a pointer or reference argument to a loaded function; its address stays inside the helper. Taking that address as a prompt value is unsupported.

Compilation happens on `:load` and on each prompt function definition, including `plus_answer` above. Top-level input stays interpreted. Loading needs a matching x2c compiler, the host C compiler, and its runtime. A dependency that the project meta builder cannot compile or link makes the load fail. An explicit `meta` function remains callable under the same data limits.

The session keeps one helper process and its compiled executable. Native globals and static values persist between calls. Each successful `:load` replaces the helper and resets all of those values, including when loading

an additional file or an unchanged file. Prompt variables and functions survive. Calls exchange data values: numbers, Strings, Symbols, Lists, and independent Arrays and Maps. Arrays and Maps are copied across the boundary. Mutating a collection argument in the helper does not mutate the prompt's collection; return the changed collection to use it. Native pointers and records cannot be returned to the prompt. Prompt-owned native objects, callbacks, cycles, and shared mutable collection identity cannot cross this boundary. Selecting a loaded object as a function argument keeps that object inside the helper.

A failed load publishes no names and preserves earlier functions. Fix a rejected file and load it again. Loaded names appear as `native` in `:symbols`; `:info` shows their declarations and loaded file. They have no prompt AST or lowered Lisp for inspection.

Use objects and functions from different files

Save this as `objects.x`:

```
struct Point { int x; int y; };
int samples[3] = {10, 20, 30};
struct Point point = {1, 2};
int first(void) { return samples[0]; }
```

Save these declarations and functions as `operations.x`. The `extern` declaration and function prototype refer to definitions in `objects.x`:

```
struct Point { int x; int y; };
extern int samples[3];
int first(void);

int sum(const int *values, int count) {
  int total = 0;
  for (int i = 0; i < count; i++) total += values[i];
  return total;
}
void shift(struct Point *p, int dx) { p->x += dx; }
int score(struct Point p) { p.x += 100; return p.x + p.y; }
int advance(void) { samples[0]++; return first(); }
```

Load both files in the same session, then use their objects and functions:

```
:load "objects.x"
:load "operations.x"
sum(samples, 3);
shift(&point, 5);
point.x;
score(point);
point.x;
advance();
samples[0];
```

`sum` returns `60`. `shift` changes `point.x` to `6`. `score` returns `108` after changing its own by-value copy; `point.x` remains `6`. `advance` returns `11`, and `samples[0]` also reads `11`.

The helper passes its own array and struct to these functions. No native address or struct bytes travel to the REPL process. Field and element selections also work, such as `sum(&samples[1], 2)`. Repeated selections of the same object alias that object. Const restrictions and array bounds remain checked. Scalar arguments and indices are evaluated once in source order. Compiled prompt functions can make these same calls. Call the loaded function by name for object arguments. Saved `Func` values retain their existing data-value calling limits.

Use normal includes, prototypes, and `extern` declarations for references between source files. Larger programs usually keep common declarations in a header. A source dependency has one provider and one instance of its public storage. Loading that dependency explicitly exposes its public names without adding another instance. Distinct public definitions with the same name conflict; file-static declarations retain ordinary C privacy.

Load a program with main

`:load` also accepts a program with `main`, including a `.c` file. The source needs no `meta` annotation or renamed function. Save this as `hello.c`:

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  for (int i = 0; i < argc; i++) printf("%d: %s\n", i, argv[i]);
  return 0;
}
```

Pass an Array of Strings, including the program name as its first element:

```
x2c> :load "hello.c"
loaded ("main")
reset all loaded native state
x2c> main(["hello", "world"]);
0: hello
1: world
=> 0
```

The adapter supplies `argc`, writable copies of the argument strings, and the final null pointer at `argv[argc]`. `main([])` passes zero arguments. Changes to `argv` do not change the prompt's Array or Strings. `:info main` shows the prompt signature `int main(Array);`.

A program with `int main(void)` or `int main()` is called as `main()`. Both forms return zero if execution reaches the closing brace. Returning from `main` produces a prompt result and leaves the REPL running. Standard output and standard error appear on their usual streams, including output without a final newline. Loading alone does not execute `main`.

The original function keeps its native calling convention inside the compiled program. Its ordinary helpers and references to `main` continue to work. Loading does not modify the source file. Other C dialect and dependency limitations still apply; loading does not provide missing libraries. Calling `exit` ends the helper, as described below under interrupted calls.

`main` follows the same replacement rules as other loaded functions. Saved function values and compiled prompt callers use its replacement. Changing between the no-argument and argc/argv forms requires a new session. A second loaded file cannot publish another `main` in that session.

Edit a loaded file

After editing a file, use the same `:load "file.x"` command. A successful replacement prints `reloaded`. Calls use the new code, including calls from earlier prompt functions. Saved `Func` values use the replacement for calls the REPL already supports. Prompt variables and functions remain. All registered files are rebuilt from their current contents and dependencies, including files other than the one named by the command. Successful loads print `reset all loaded native state`: variables in every loaded file and dependency start over. Previously returned values remain ordinary prompt data.

Existing functions must keep their argument and return types. Parameter names may change. Existing public types must keep their definitions. New functions, globals, and types are allowed when their names do not conflict. Existing globals must keep their declared types and qualifiers, including `const`. Removing an existing function, global, or type, or changing its type, requires a new session. A refused replacement leaves the whole previous native program and its variables intact. This includes a compile, link, or compatibility failure in any loaded file, even when the command names a different file. Prompt definitions still cannot be redefined.

Relative paths, absolute paths, and symbolic links to the same file identify one load. If two loaded files include the same dependency, either load rebuilds both files with the dependency's current contents.

For example, copy the autodiff module from a repository checkout:

```
cp packages/autodiff/src/autodiff-macros.x /tmp/autodiff-repl.x
x2c repl
```

At the prompt, load and inspect the copy:

```
:load "/tmp/autodiff-repl.x"
:symbols ad_do
:info ad_dot
```

The filtered list contains `ad_dot`. Its declaration is `String ad_dot(String);`, followed by the loaded file's absolute path. Keep some test data and a prompt function:

```
x2c> String label = "loss";
ok
x2c> String derive(void) { return ad_dot(label); }
defined derive
x2c> derive();
=> "loss_dot"
```

Edit the copy's `ad_dot` definition to append `"_derivative"` instead of `"_dot"`. Run `:load "/tmp/autodiff-repl.x"` again, then call `derive()`. The result is `"loss_derivative"`; the earlier `label` and `derive` still work. This module also exposes syntax helpers such as `ad_unbind` and `ad_mutates`, which can be called with List values from the prompt.

A native call that exits, crashes, or exceeds `X2C_META_TIMEOUT` fails that submission. The next call restarts the shared helper from its retained executable, resetting all loaded native state. Ordinary errors raised by a function leave the helper and its state running.

Supported subset

The current subset includes initialized simple variables, function definitions, integer arithmetic and narrowing, assignment, conditionals, `for` loops, self recursion, and calls to earlier functions. String values and length, List literals and indexing, and Array construction, mutation, and indexing are exercised by the focused checks. Captured `Func` values can be called directly or passed to supported native callbacks. A record captured by value retains a copy through later submissions. Reference captures share the original local or parameter, whose storage remains alive until the session ends. Capturing an existing reference parameter keeps its caller's alias and does not extend the caller object's lifetime.

An expression lambda can return another captured lambda:

```
Func factory = %!(int n) => %!() => n;
Func nested = factory(3);
nested();  // 3
```

A returned noncapturing lambda or named function needs a `(Func)` cast, as in native code.

The REPL provides the fixed-signature `String.format(String fmt, List values)` operation. Call it as `fmt.format(values)`:

```
String text = "%s=%04d".format(values);
```

It supports `%%`, flags `-+ #0`, numeric and `*` width and precision, integer conversions `d i o u x X` with `hh h l ll`, floating conversions `f F e E g G a A` with default, `l`, or `L`, plus `%c` and `%s`. It converts numeric `Var`s and uses `Var.str` for `%s`. Pointer and write-count conversions, wide strings and characters, positional formats, and `j z t` lengths are rejected. The result follows the process locale, and a failure publishes no partial String. Native `String.printf` retains its ordinary C varargs contract for compiled code.

REPL sessions also provide `print(String)` and `println(String)`. `print` writes the String's bytes; `println` writes those bytes followed by one newline. Use String interpolation for formatting, such as `println(%"count=$count")`. Both functions are REPL-only, reserve their names in the session, and report `ok`. They are not varargs aliases.

A session also declares `Json`, `Diff` and `Path`, so `Json.parse`, `Diff.unified` and `Path.basename` evaluate without an include.

Redefinition is disabled; assign to an existing variable to change its value. Function replacement and mutually recursive forward declarations are not supported. Names beginning `__repl_` are reserved.

Arbitrary C libraries, unions, enums, imports, protocols, user macro/meta definitions entered at the prompt, preprocessor directives, and direct Lisp input are outside this subset. A bodyless `meta` prototype is accepted: it makes a function the compiler links, or one from a native module loaded with `--native-module`, callable in the session. Variables need initializers; declarator modifiers, `const`, and `volatile` are rejected. Function-local `static`, `extern`, and `threaded` storage remain rejected. Top-level statement expressions can contain local declarations followed by one value expression, such as `({ int n=3; n+4; })`. Other statement expression bodies are declined. Reference parameters use the ordinary compiler conversion: pass an addressable value, or write `*p` to

pass a pointer's object. A pointer itself cannot substitute for the referenced object. Records, typedefs, addressed locals and pointer reads/writes use native storage with layouts derived from compiler types. Record assignment copies into existing storage, preserving pointers to its fields. A session global scalar or record keeps its address until the session closes. Other top-level constructs depend on the existing lowering and may be declined. Native function bodies use ordinary compiler semantics; for example, `goto` is available there. Loaded public types can be used in prompt functions within the same native layout limits as interpreted objects. Interpreted code does not distinguish `Var void` from an empty List.

Failure and lifetime

Rejected submissions leave earlier definitions usable. A declaration publishes its names only after all initializers succeed. If a later initializer fails, none of that declaration's names become available, including through inspection. Effects already made on existing values remain, both for failed initialization and failed statements.

Session storage lives until exit. Submission scratch is reclaimed, but canonical syntax, compiler caches, and evaluator allocations can accumulate. Long sessions do not have a bounded-memory guarantee. Compiled modules stay loaded for the process lifetime; captured bindings belong to their session. Each submission may make 40 million interpreted Lisp calls, the evaluator's default budget. SIGINT (Ctrl-C) during evaluation stops the submission with an `(interrupt ...)` evaluation error. Once the budget is exhausted or the submission is interrupted, no further interpreted code runs, including pending `defer` bodies. Ordinary evaluation failures still run those cleanups. The next submission gets a fresh budget. An interrupt cannot stop native callbacks or compiled prompt functions; it takes effect at the next interpreted call. Compiled function loops do not consume the Lisp budget. Their native crashes or nonterminating loops can end or block the REPL process. Loaded files share a helper process separate from the REPL, with the timeout behavior described above.

EOF exits successfully unless input is incomplete, which exits with status

1. With piped input, any submission or command error makes the final status 1, while later input still runs. In an interactive session, recovered errors do not change a normal exit's status from 0.

During interactive editing, Ctrl-C clears both the current edit buffer and any pending incomplete submission, then returns to the `x2c>` prompt without a diagnostic. Ctrl-D on an empty edit buffer exits; pending incomplete source still makes that exit status 1. Terminal canonical mode is restored before a submission runs. Ctrl-C during interpreted evaluation stops only that submission. Terminals without the required escape-sequence support use the basic line reader instead. Piped input retains the same output and exit behavior.

Indentation Syntax

x2c can be written without braces or semicolons. Indentation groups the statements of a block, a line that opens a block ends with `:`, and a line break ends a statement. Name the file `.xp`, or put `#pragma indent` at the top of a `.x` file. The two spellings mean the same program, so an indented file can include, import, and call brace-form code and the reverse.

Here is a small program in brace form:

```
typedef enum Shape { CIRCLE, SQUARE } Shape;

double area(Shape shape, double size) {
  switch (shape) {
    case CIRCLE: return 3.14159 * size * size;
    default: return size * size;
  }
}

int main(void) {
  double total = 0;
  for (int i = 1; i <= 3; i++) {
    if (i == 2) continue;
    total += area(i == 1 ? CIRCLE : SQUARE, i);
  }
  printf("%.2f\n", total);
  return 0;
}
```

```
12.14
```

The same program in the indentation syntax:

```
#pragma indent

typedef enum Shape:
  CIRCLE,
  SQUARE
Shape

double area(Shape shape, double size):
  switch shape:
    case CIRCLE: return 3.14159 * size * size
    default: return size * size

int main(void):
  double total = 0
  for (int i = 1; i <= 3; i++):
    if i == 2: continue
    total += area(i == 1 ? CIRCLE : SQUARE, i)
  printf("%.2f\n", total)
  return 0
```

```
12.14
```

Conditions need no parentheses, except that a `for` header keeps its three clauses in them, and a short body can follow its colon on the same line. `foreach` separates its declaration from the collection with `in` or a comma, as in brace form. The enum keeps its commas, and its name after the dedent completes the `typedef`.

Long lines

A line break inside parentheses, brackets, or braces never ends a statement. Outside them, a line indented deeper than the one before it continues that line. A line that starts with `.` continues a method chain:

```
#pragma indent

int main(void):
  List words = %("indentation" "groups" "blocks")
  int letters = 0
```

```
  foreach String word in words:
    letters += word.len()
  String report = %"${words.len()} words, " +
    %"$letters letters"
  List lengths = words
    .map(%!(Var word) => word.str().len())
  printf("%s: %s\n", report, lengths.str())
  return 0

3 words, 23 letters: ( 11 6 6 )
```

Scopes and decorators

`do:` opens a bare scope when its block is not followed by a `while` line, which is how an indented macro keeps its temporaries to itself. In indentation syntax a decorator line starts with `@`, so the layout wraps the next statement instead of ending the line with `;`. Brace form writes the decorator without `@`:

```
#pragma indent
#include "system-macros.x"
#include <time.h>

int main(void):
  int total = 0
  do:
    int step = 2
    total += step
  @$time("sum")
  for (int i = 0; i < 1000; i++):
    total += i
  printf("%d\n", total)
  return 0
```

Converting a file

`tools/indent-convert FILE...` rewrites brace-form files in place, adding `#pragma indent` so each keeps its name; an executable script stays runnable by name. Every block header, including one whose body had no braces, ends with a colon. The converter proves each conversion by comparing the compiler's AST for the result with the original's, and it leaves a file unchanged when they differ, as they do when indentation misrepresents which statement an `else` or a body belongs to. `--check` reports without writing.

A `.xp` file writes macros the same way. The language reference lists every layout rule. `examples/tours/indentation.xp` is a complete program.

Programming Idioms

Use the shipped classes and system macros where they express a complete constructor or block lifetime: `$auto` for an owned local, `$scope` for a region, `$let` for saved storage, and `$lock` for a Mutex.

These idioms combine the features introduced in the guide.

Keep native values native

Use C scalar, pointer, struct, union, and enum types when the value's type is known at compile time. Use `Var` where the value is dynamic: heterogeneous collection elements, dynamic arithmetic, generic callbacks, or interchange data.

Use values for local state and references for updates

Declare a record as a value when it belongs to one operation and needs no retained identity. A method that updates that record takes a reference:

```
class Counter { int value; };

static void Counter.step(Counter &c, int amount) => c.value += amount;

int main(void) {
  Counter counter = Counter.new(2);
  counter.step(3);
  Counter copy = counter;
  copy.step(1);
  printf("%d %d\n", counter.value, copy.value);
  return 0;
}

5 6
```

`counter` holds the record directly; constructing it needs no allocation for the record. `step` changes the caller's object, and its call needs no `&` or explicit dereference. Copying the record creates a separate Counter. Pass a small record by value when reading a copy is the intended contract.

`class` adds constructors, boxing, and other methods to the declared representation. Use an ordinary value typedef for a context that does not need that bundle. In either case, pass its mutable state through `T &` helpers instead of introducing a pointer typedef and a second local whose only purpose is taking the context's address.

A trailing `*` selects a pointer representation: `class Node { int value; } *;`. Its bare `Node` name is already a pointer, and its methods take that handle. Choose it when shared or retained identity is useful. The spelling `struct` before the braces is optional; short braces work for both representations. See Choose a representation.

Representation and storage location are separate choices. A value can be a local, a field, an array element, or part of allocated storage. A reference borrows one live caller object for a call; it does not own or extend its lifetime. Record copies are shallow: a copied Array, Buffer, or pointer field still reaches the same backing object. Keep cleanup with the owner of those resources, including when the record itself is a value.

An active transaction or builder has one owner even when its record is a value. Copying it shares the saved maps or backing arrays; the copy is not a second transaction or an independently owned builder. Borrow the original through reference methods. `SymTxn` and `MachineBuilder` use this pattern; use their begin or `new` operation to prepare active storage. A `Job` keeps a shared handle because its recorded run and Scope finalizer must reach the same job, while its launch options live directly inside that job.

Borrow required and optional outputs

Use a reference when a result must update a caller's existing object:

```
static int take(int &remaining, int &item) {
  if (!remaining) return 0;
  remaining--;
```

```
  item = remaining;
  return 1;
}

int main(void) {
  int remaining = 2, item;
  while (take(remaining, item)) printf("%d\n", item);
  return 0;
}

1
0
```

Use `T &?` when supplying the object is optional, and test its presence before using it. A required reference takes an addressable value, not a pointer or NULL. An existing pointer `p` supplies its object as `*p`. Keep pointers for buffers and arrays, retained addresses, nullable links, native signatures, and callback state. A nested context that borrows its mutable parent needs a pointer field so both contexts reach the same parent. See Reference parameters.

Adopt a protocol for one real contract

Adopt a protocol when a concrete type should support a shared interface. Define the required conversions beside the type. Similar method names are not enough: the operations must have the meanings the protocol requires.

Implement a member directly when the type already provides that operation. Use a base default when every participant value converts to a valid base value and the forwarded operation has the required meaning. Before extending a protocol, use `--dump-conformance` to check its existing participants.

Box private records through `Var(T)`

A type does not need to become public because internal code stores it in a `Var`, `List`, `Array`, or `Map`. Put both conversions beside the private type and adopt the public `Var(T)` protocol there. Typed arguments, results, assignments, and initializers then box and unbox it without repeated casts.

Keep an explicit `value is Type` check when a `Var` came from mixed or untrusted data. Protocol adoption supplies the conversions; it does not check the dynamic type of an arbitrary value.

Make macro calls easier to read than their expansions

A macro should replace repeated source with a call whose meaning is obvious:

```
macro Stmt $sample.guard(Expr $condition) {
  if (!$condition) return 0;
}
```

Prefer a function when the job is runtime computation. Prefer direct source when a macro needs substantial compile-time Lisp or AST manipulation to save a few tokens. The macro body, imports, and generated declarations are part of the maintenance cost.

Write generated code as the code it builds

A `meta` function that returns code should show that code. A quotation shows the C that will land; a `%(...)` List of node tags hides it. Choose the first row that fits. Meta Functions explains each form with examples.

To build	Write
Code with one shape at several sites	a named macro, applied from the function as in source templates
Code that one function builds	a quotation: `$!( expression )`, `$!{ statements }`, or `$! Unit{ ... }`
Code that uses a field, an element, or a call result	a `${expression}` hole, evaluated once where the quotation is written

To build	Write
Code in which separate quotations share one name	`List name = x2c_ident("total");`, then `$name` in each quotation
An expression whose type is read before it lands	a typed quotation, such as `$!double{ $value * 2 }`
Code the function inspects before it returns	a typed quotation, which builds the code where it is written
A pattern, a data row, or a form with no source spelling	a `%(...)` List

A name that a quotation declares is private to it, so another quotation cannot refer to it. An `x2c_ident` hole declares and refers to one exact name.

When a `meta` function builds one statement, the macro can hand it the work in one line. Write the `Stmt` macro with an arrow body and return the statement itself:

```
#include "meta.x"
meta static List traced(List body) =>
  $!{ { puts("enter"); $body puts("leave"); } };

macro Stmt $sample.trace(Stmt $body) => $traced($body);
```

Keep the braced `{ @helper(...) }` form for a function that returns several block items as a `List`.

Choose collections by mutation and identity

Use `List` for immutable sequences that share structure, `Array` for mutable indexed sequences, and `Map` for mutable key/value pairs. Build text with `Buffer` when repeated concatenation would create unnecessary intermediate `String`s.

```
List syntax = %(call print "hello");
Array work = [1, 2, 3];
Map index = {name: "x2c"};

work.push(4);
index[<count>] = work.len();
```

For a closed vocabulary of `Symbol`s that needs membership tests, a dense index, or ordered iteration, declare a `SymbolSet` literal once and let its perfect-hash `index` replace a hand-written switch; see Ordered Symbol sets for the literal and its operations.

Destructure small fixed results

Flat `List` destructuring is clearest when a producer returns a few values in fixed positions:

```
List bounds = %(0 10 2);
int (start, stop, step) = bounds;

List record = %(7 2.5 "ready");
(int id, float score, String state) = record;
```

Assignment destructuring returns its source unchanged, so one result can feed several flat views without another call or copy:

```
Var left_a, left_b, right_a, right_b;
List values = (left_a, left_b) = %(1 2);
(right_a, right_b) = values;
```

Use `match` for conditional or nested shapes. Use named accessors when only one position matters.

Let collection methods express collection work

Use `foreach` when the operation is naturally sequential or can stop early. Use `map`, `filter`, and folds for direct transformations:

```
List values = %(1 2 3 4);
List squares = values.map(%!(item) => item * item);
Var sum = values.foldl(0, %!(total, item) => total + item);
```

Choose an explicit `Iter` pipeline when values must stream without first materializing a collection.

Keep lambdas small

An expression lambda suits a short operation with a few captured values. Use a block lambda when the operation needs local declarations, branches, or `defer`:

```
List magnitudes = values.map(%!(Var item) => {
  int value = item.int();
  if (value < 0) return -value;
  return value;
});
```

Captured automatic values are read-only snapshots unless a lambda shares the binding with `using &name`. When the operation grows beyond one local task, use a named function and an explicit state object. The collection methods accept capturing lambdas; a C function-pointer API needs a noncapturing one. The reference gives the capture rules.

Separate absence from data

Prefer a `try_*` operation when the result alone cannot represent every outcome, as with `Map.try_get` for presence, `Iter.try_next` for exhaustion, or `String.try_long` for parse success:

```
Map map = {"name": "x2c"};
Var key = "name";
Var value;
if (!map.try_get(key, value))
  printf("missing\n");
```

Sentinel-returning adapters are concise only when the sentinel is impossible in the result domain and the caller does not need a failure reason.

Put cleanup beside acquisition

`Scope` groups allocations by lifetime and `$scope()` retains one region around a statement. `$auto` attaches an owned local's cleanup to its declaration, so the acquisition and its release are one line:

```
$scope() {
  String path = "/tmp/x2c-idioms.txt";
  File output = $auto(File.open(path, "w"));
  output.puts("ready\n");
}
```

Both lower to `defer`. Use `defer` directly where no macro fits: a native release, a conditional rollback, a consuming parameter, or cleanup that writes a result.

```
int *buffer = malloc(64);
defer free(buffer);
```

`$auto` needs a type that adopts `Cleanup(T)`, directly or through its typedef ancestry, as the runtime's owning types do. It releases the value it acquired, even after the local is reassigned. Use `defer` when cleanup must read a binding at exit, such as a `Scope` slot filled later:

```
Scope work = NULL;
defer work.cleanup();
$scope(&work) { Array items = []; items.push(1); }
```

Raise failures; return ordinary outcomes

Report EOF, a missing `Map` key, or a numeric parse failure through the result, as the library operations do. Use `raise` for a failure cause that may cross several frames, and put a filtered `catch` where recovery or reporting belongs. Every intermediate resource owner must have `defer` or `finally` cleanup in place.

Prefer the primary API tier

The generated module reference separates:

- primary operations for ordinary programs;
- advanced storage, ABI, embedding, and diagnostics APIs;
- convenience adapters with a preferred operation;
- runtime-internal callables documented only for source readers.

Start with primary APIs; use the others when you need their specific behavior.

Language reference

Language Reference

This chapter specifies x2c's syntax, behavior, and current limitations.

The runtime API is described in the standard library overview, and advice about which construct to use is in idioms. Flags and the dump options that expose each phase are in compiler options. If you have not built the compiler yet, the website's install page is the quickest route, and building the compiler covers the self-host stages behind it.

The syntax specifications catalog the source rules and provide an extracted EBNF grammar and lexical transition specification.

Source files and visibility

An `.x` file combines declarations and definitions. Translation produces a header and a C source file.

A source file whose first line begins with `#!` is a script unit; every other file is an ordinary unit.

Declarations are public unless marked `static`. A public function contributes its prototype to the generated header, or its whole body when inline. A public object has one definition in the generated C and an `extern` declaration in the header. An object of an anonymous `struct`, `union`, or `enum` type has no repeatable declaration to publish.

The compiler reserves `_x2c_gensym_*`, `_init_guard_`, `_file_init_`, and `_` followed by digits for generated support. Source declarations may use `_x2c_*` outside that subspace. The repository uses those names for shared implementation functions, but the prefix does not change linkage or enforce private access. `static` controls visibility.

Standalone types are public by default. `static typedef`, `static struct`, `static union`, `static enum`, and `static class` declare private types. A type needed by a public declaration becomes public automatically. Its complete definition and field types are published when the interface requires that layout, such as for a value type or field access in a public inline body. A public forward pointer type can keep a static aggregate body private. A static type used only by static functions remains private. The `static` marker on a type controls publication and emits no C storage. `module` in place of `static` declares the same private types; see module blocks.

An ordinary `#include "module.x"` makes that module's public declarations, macro definitions, keyword aliases, and meta functions available. The same rule applies through transitive includes. Compile-time definitions become available at the include's source position. Static definitions belong to the declaring source file. A C header is not an x2c unit, so its `static inline` functions still belong to every C translation unit that includes it.

Every translated header starts with `#pragma once` and also carries a conventional include guard, so ordinary `.x` programs need neither directive. The generated source preserves native includes at their source positions. A late native include enters the header when a later public declaration needs its types.

The advanced `--cpp-symbols` and `--live-symbols` modes run the host preprocessor over raw `.x` include graphs. A module used with those modes needs a source-level `#pragma once` only when its own `.x` includes form a cycle. Ordinary translation resolves includes itself and terminates cycles without it. These modes and `--dump-cpp` reject module units and indented units, because the host preprocessor reads them as C. `--cpp-symbols` and `--live-symbols` also reject script units, because the `#!` line is not C.

Every ordinary `.x` translation unit implicitly loads the `x2c.x` standard runtime prelude. The compiler also emits `#include "x2c.h"` in its generated header. An explicit `#include "x2c.x"` is also accepted; it does not change the semantic environment or generated runtime dependency.

Optional modules shipped with x2c are outside that prelude. They require an explicit source include, such as `#include "typed-array.x"`. Third-party code uses the package `import` described below.

Module blocks

A module block splits one large source file across a directory without splitting the translation unit. The entry file lists its member files, and the compiler reads each member's text in place of the block. The result translates as one unit with two private storage classes: `static` is visible in the file that declares it, and `module` is visible in every file of the unit.

The entry `calculations/calculations.x` lists members beside it:

```
/* calculations/calculations.x */
#include <stdio.h>

module {
  "implementation.x",
}

int main(void) {
  printf("%d\n", helper(40) + counter);
  return 0;
}

/* calculations/implementation.x */
module int counter = 2;
module int helper(int value) { return value + counter; }
```

The block is `module {`, comma-separated string paths with an optional trailing comma, and `}`, with no final semicolon. It appears at file scope, once per file. Before a file-scope declaration `module` is a storage class, described below. Elsewhere it is an ordinary identifier, and `struct module {` declares a tag. Each path is relative to the file that holds the block. Members may be siblings, descendants, or files elsewhere; no same-named companion directory is required. A member that is already being spliced, an unreadable member, a malformed block, or a second block is a driver error at the block.

The block is spliced before C preprocessing, so a preprocessor condition does not choose members. x2c skips a block only inside a conditional arm that C never takes by x2c's own reading: `#if 0`, `#ifdef` or `#if defined` of a name that x2c output never defines, such as `__cplusplus`, and the `#else` arm of `#ifndef` of such a name. Every other arm counts. After `#define OFF 0`, a block under `#if OFF` still reads its members, and blocks in both arms of `#if 1` and `#else` are a second block. The spliced text, including each member's own directives, then reaches C as one unit in splice order.

The entry's syntax mode applies to all spliced text. In a `.xp` entry every member is read with indentation layout, and in a `.x` entry with braces, so a member must use its entry's syntax.

A member's text stands where the block stood, in the listed order. All members share the entry's translation unit. Function declarations are collected across that complete unit, so a function can call a later-defined function without an explicit prototype. Other declarations retain their ordinary placement rules. A member may hold its own block, whose paths are relative to that member; their text is spliced in turn, and the namespace stays flat.

The `module` storage class declares a file-scope name for every file of the module unit. It is written where `static` would be, before an object, function, `typedef`, or `class` declaration:

```
/* calculations/state.x */
module typedef struct Tally { int total; } Tally;
module Tally tally;
module void add(int value) { tally.total += value; }
```

A later member and the entry after the block use `Tally`, `tally`, and `add` by those names. The declaration parses as if `static` were written. The object and function are `static` in the generated C, the type stays private, and the unit's header and interface publish none of them. Outside a module unit `module` means `static`.

Inside a module unit, `static` keeps its C meaning: the name is visible in the file that declares it and in no other file of the unit. That holds for objects, functions, `static typedef`, `static class`, and the `struct`, `union`, or `enum` tag such a typedef declares. Two members may each declare `static int counter` or `static typedef struct Node { ... } Node`, and each file sees its own, and so do the enumerators of a `static typedef enum`. The enumerators of an enum declared without a typedef stay unit-wide, even under `static`. A later declaration without `static` refers to the file's static, as in C. Examples are the definition after a `static` prototype and a block-scope `extern` declaration or prototype. A file's statics take a prefix in the generated C: the file's path relative to the entry's directory without its extension, with each character that cannot continue a C identifier made _, then ___. The prefix starts with `f`, the normalized path's length, and _; a path that starts with a digit gets a leading `x` before its length is measured. Under the entry `parser/parser.x`, a static `advance` in `parser/statements/loops.x` emits as `f16_statements_loops___advance`, and one in the entry itself as `f6_parser___advance`. Trailing _ of the path's

spelling are dropped first, so `a.x` and `a-.x` give one prefix. A package prefix ends in `__`, so a package entry's statics stay apart from the package's public names. Generated names are compiler details; source never spells them, and `--dump-definitions` displays the source name. A static type adopted as a class keeps its short name as its Var tag and descriptor name, so two files of one unit cannot each adopt a class under the same short name. Compile-time declarations, `static macro`, `static $(`, `static protocol`, and `meta` functions, keep their unit-wide behavior. Macro free names follow hygiene.

Three cases are errors. Two files whose paths give one prefix, such as `a_b.x` and `a/b.x`, or a member named like its entry, are a driver error at the block. A `static` object, function, typedef, or tag whose name is a `module` name of the same C namespace, in either order of declaration, is a parse error at the later declaration, because a `module` name means one thing in every file. Two files adopting a class under one short name is a parse error at the second. A file-scope typedef named `module` keeps that word an identifier.

A quoted include of a file beside the member includes that file, in collection and in the generated C. Any other include resolves as it would at the block's position in the entry.

Members are not translation units. A build never lists them; the entry's generated C and header contain everything. The entry's depfile and interface depend on each member's content, so editing a member translates the entry again and changes what an including unit collects. Translated on its own, a member is an ordinary file.

Diagnostics, `--source-map` line directives, dump locations, and editor positions report the member's file and line. A unit that includes the entry collects the public declarations of every member. A unit that includes a member directly collects only that member, as it would any file. A file without a block reads exactly as before.

Script units

A script unit is a source file whose first line begins with `#!`, usually `#!/usr/bin/env -S x2c script` so that `x2c script` runs it. The compiler reads that first line as `#include "scripting.x"`, which brings in `args.x`, `diff.x`, `digest.x`, `path.x`, `process.x`, and `regex.x`. Every other line keeps its line number.

A script unit takes one of two forms, chosen by whether it defines a function named `main` at file scope:

- **With `main`**, it is an ordinary program. Its declarations, including initialized ones such as `int count = 0;`, stay at file scope and `main` runs. Adding a shebang line turns a program into a script.
- **Without `main`**, its top-level statements run in order as the program.

A script unit that defines `main` and also has a top-level statement, such as an expression, control flow, a `with` block, or a statement macro, is an error.

In the second form these top-level forms stay at file scope:

- preprocessor lines, `import`, and protocol declarations and adoptions;
- `macro` and `keyword` definitions, top-level `$(...)` Lisp, and invocations of macros and keyword aliases whose result is file-scope syntax, including decorators of functions and types;
- `typedef`, `static`, `module`, `extern`, and `meta` declarations and static assertions;
- type definitions such as `struct Point { int x; };` and classes;
- function definitions, including expression-bodied ones, and prototypes, meaning declarations that end with a parameter list.

Every other top-level form is a statement, including declarations with an initializer or none, such as `int total = 0;` or `Point p;`. The statements, in source order, become the body of a function that the generated `main` calls with `argc`, `argv`, and `args`, a `List` of the argument `String`s after `argv[0]`. Statements can call the script's functions wherever they are defined, and those functions can call one another. The declarations among the statements are locals of that body, so functions cannot refer to them. Lambdas capture them as usual. Write `static` for a variable that functions share.

Falling off the end of the statements returns zero, and `return` among them sets the exit status. A `<cmd-fail>` that no statement catches prints the command and its status on standard error, and that status becomes the exit status. Any other uncaught `Error` prints its cause and details on standard error and exits with status 1.

```
#!/usr/bin/env -S x2c script
static String greeting(String name) => %"hello, $name";

String name = args ? args.car().string() : "world";
```

```
printf("%s\n", greeting(name));
if (!args) return 1;
```

Indentation syntax

A file can group statements by indentation and end them at line breaks instead of writing braces and semicolons. The file is in the indentation syntax when its name ends in `.xp`, or `#pragma indent` appears before its first line of code. Only comments, blank lines, and other directives may come before the pragma, and it does not reach the generated C. A `.xp` unit is otherwise an ordinary unit: `#include "shapes.xp"` includes its generated `shapes.h`, and builds, packages, and `x2c script` accept `.xp` wherever they accept `.x`.

The indented form is a spelling of the brace form. The tokenizer inserts the braces, semicolons, and condition parentheses, and every later stage reads the same tokens as it would for the brace form:

```
#pragma indent

int sum_positive(int *xs, int n):
  int total = 0
  for (int i = 0; i < n; i++):
    if xs[i] < 0: continue
    total += xs[i]
  return total

int main(void):
  int xs[] = {3, -1, 4}
  printf("%d\n", sum_positive(xs, 3))
  return 0
```

```
7
```

- **Lines.** A logical line ends at a line break outside brackets. A deeper line continues the previous one unless that line ends in a `:` that opens a block, and a line that starts with `.` continues a method chain. The `:` of a `?:` expression never opens a block.
- **Blocks.** A line that ends in `:` before a deeper line opens a block, and each return to an outer indentation closes one. `if`, `else if`, `while`, `foreach`, `switch`, and `match` take their condition with or without parentheses, and `for` keeps the parentheses around its three clauses. The condition belongs to the last of these keywords on the line, so an unbraced `foreach (...)` can continue into an `if x:` block. A `case`, `default`, or `catch` label keeps its colon and opens a block of its own. A `struct`, `union`, or `enum` body closes with `};`, unless its header starts with `typedef`, and enumerators keep their commas.
- **One-line bodies.** `if cond: statement` puts the body after the first colon outside brackets that does not belong to a `?:`.
- **Statements.** Every other line ends with an implied `;`. Directives, enum bodies, a line that is one whole `$(...)` or `@(...)` form, and a statement that is only a macro hole, alone or after a control header, take none.
- **Bare blocks.** `do:` whose block is not followed by a `while` line at the same indentation is a bare `{ ... }` scope. With that trailer it is a `do-while` loop.
- **Decorators.** A decorator invocation is written with a leading `@`, as in `@$time("load")`, on the line before the statement it wraps. Without the `@`, the line would end with `;` as a statement macro call does.
- **Explicit forms.** Braces and semicolons written explicitly keep their meaning, and braces that make values, such as initializers and Map literals, never open blocks.

The tokenizer rewrites each indented form into these brace-form tokens. A block is the deeper lines that follow its header, and it closes where the indentation returns:

Indented form	Brace form the parser reads
`int f(int x):` and a block	`int f(int x) { … }`
`if x:` and a block	`if (x) { … }`
`if (x):` and a block	`if (x) { … }`
`else if x:` and a block	`else if (x) { … }`
`else:` and a block	`else { … }`

Indented form	Brace form the parser reads
`while x:`, `switch x:`, `match x:`	`while (x) {`, `switch (x) {`, `match (x) {`
`foreach T v in c:` and a block	`foreach (T v in c) { … }`
`for (a; b; c):` and a block	`for (a; b; c) { … }`
`if x: s`	`if (x) s;`
`else: s`	`else s;`
`case X:`, `default:`, `catch P:` and a block	`case X: { … }`
`struct S:` and a block	`struct S { … };`
`typedef struct S:`, a block, then `S`	`typedef struct S { … } S;`
`enum E:` and enumerators	`enum E {` enumerators … `};`, commas as written
`do:` and a block	`{ … }`
`do:`, a block, then `while (c)`	`do { … } while (c);`
`@$m(args)`	`$m(args)`, with no `;`
any other statement line	the line followed by `;`
`#pragma indent`	nothing

Inserted tokens have no width and take the position of the token beside them, so diagnostics and source spans report lines of the indented file. At the start of a statement line, `%` and `<` begin a literal, as in `%(ls).job().run()` or `<done>`, even when the line before ends in an operand; on a deeper continuation line they keep their operator meaning.

Indentation uses spaces. A tab in indentation, or a return to an indentation that matches no open block, is an `inconsistent indentation` error at that line.

Packages and `import`

A package is a directory whose name is a valid C identifier. Its entry unit is `<dir>/src/<name>.x`, or `<dir>/<name>.x` for a single-file package. The package is everything that entry unit includes. Nonstatic declarations and compile-time definitions are public. There is no manifest or export list. `--package-dir <root>` registers a directory of packages; a target in `x2c.toml` may set `package-dirs` instead.

An `import` declaration gives a package, binds a local alias, and may list individual members:

```
import "<package>" [as <alias>] [with <Name> [as <Local>] {, ...}] ;

import "geo";                 // alias: geo
import "geo" as g;            // alias: g
import "geo" with Vec, span;  // plus bare Vec and span
import "geo" with Vec as V;   // plus bare V
import "geo" as g with Vec;   // both spellings
```

The alias changes how names are written in x2c. `g.Vec`, `g.Vec.new(2.0, 3.0)`, and `g.span(p)` all compile to the package's C names, which carry the package prefix: `geo__Vec`, `geo__Vec_new`, `geo__span`. Two packages may therefore publish the same type or function name in one program. An existing local or file-scope binding takes precedence over an alias of the same name.

An `import ... with` name changes only the source spelling. The C symbol keeps its package prefix. `Vec v = Vec.new(2.0, 3.0)` emits `geo__Vec` and `geo__Vec_new`. Any public name the package exports may be listed, including free functions. The alias is registered either way, and the clause only adds shortcuts. A name the package does not export is an error at that name, and a local spelling that another binding or a declaration already uses is reported as `package name 'V' is already bound` or `package name 'V' collides with a declared name`. A declaration shadows a `with` name as it shadows an alias.

Names derived from a package type carry the prefix as well. A `Var(T)` participant's tag is `<geo__vec>`, and its converters are `geo__Vec_var` and `geo__Var_vec`. A package's own sources write all of these bare and let the compiler add the prefix. Only a `Symbol` literal, a global interned identity, is written out in full.

A package may publish a method on a type it does not define, such as `Var.json`, or on a type its own vendored header defines. Imported methods take part in dot-call lookup. The receiverless form `Type.member(...)` and protocol punctuation resolve through the same imported names, so a package that adopts one of its own protocols for a foreign type gives the consumer that operator. One imported match is callable. More than one is an error at the call, with the packages listed by name; imports that are never called do not conflict. An alias-qualified function call such as `json.Var_json(value)` selects one package explicitly.

A package's own sources are `<dir>/src/**` and the single-file `<dir>/<name>.x`. Other files in the package directory, including its tests and examples, are ordinary consumers that reach it through `import`.

An import exposes:

- nonstatic functions, objects, standalone types, and types needed by public declarations;
- public macro definitions, keyword aliases, and meta functions;
- public protocol declarations and adoptions; and
- the public declarations of the files the package entry includes.

Compile-time definitions arrive in the order an include of the entry source would deliver them. Static declarations remain in their source files. Macro names retain their declared spellings. One program uses one version of a package.

A public bodyless `meta` prototype crosses the import like any function declaration. When the package has built its native module, `<dir>/builds/<name>.module`, the import loads it, so compile-time code in the consumer can call the function. Modules named by `--native-module` or `native-modules` are consulted first. A module built by another compiler is an error at the import; see packages with compile-time parts.

Names from a C header remain unprefixed, as if the consumer included that header directly. A package prefixes only the names it declares.

An unprefixed public declaration from an x2c file outside the package directory would enter the consumer's namespace unchanged. The compiler rejects it as `package 'geo' exposes unprefixed top-level declaration '...'`. Including the foreign file directly in the consumer avoids that package-prefix conflict. A runtime module, like a C header, crosses unprefixed. A consumer that declares a name in an imported package's `geo__` space is reported as `'geo__x' is reserved for imported package 'geo'`.

C foundation

x2c keeps C declarations, expressions, operators, functions, structs, unions, enums, pointers, arrays, and control flow. It adds language forms around that foundation and keeps C's object model.

Use `#include` to include source and headers. Avoid other C preprocessor constructs in x2c source; use x2c macros and language features instead. Intermixing `#define` with x2c transformations is unsupported. The compiler preserves some native macro behavior on a best-effort basis, but does not guarantee that native macro text follows renamed or file-private bindings.

Member access uses `.` wherever x2c parsed the struct, whether the receiver is a value or a pointer, and whether the declaration came from x2c source or from a C header x2c reads. `->` remains accepted. It is required in two places. One is a pointer to a struct whose layout x2c does not read, such as one from a system header or an unresolved third-party header. The other is a `#define` body. A member of an anonymous union or struct belongs to its enclosing aggregate, and `.` reaches it through a pointer like any other member. `.` reaches through one pointer level, so a pointer to a pointer typedef keeps `(*pointer).field`. x2c also recognizes method-style calls such as `list.len()`. The namespaced function is selected from the static receiver type. When that function declares its first parameter `T *` and the receiver is an addressable `T`, the receiver's address is passed, so `rec.bump(4)` reaches `void Rec.bump(Rec *rec, int by)` without an explicit `&`. A receiver that is a pointer to the declared parameter is one level too far and is a type error; write `(*pointer).method()`. Normal pointer conversion rules still apply, including preservation of `const` and other qualifiers. A qualifier on the receiver itself does not change which methods and operators the type has, so a `const String` has the same `len()` and `+` as a `String`. Method syntax does not make every value dynamically dispatchable; only registered callbacks define custom behavior.

A receiver spelled `struct T` or `union T` uses the methods of the typedef that names the same aggregate. A same-named typedef takes precedence; otherwise the aggregate must have one direct typedef. An unrelated variable with the tag's spelling does not supply a method owner. Generic functions named `struct_method` or `union_method` do not participate in method lookup. Another module may define methods for the type; this rule selects the receiver's type and does not restrict who can define its methods. Type inspection and member completion follow the same lookup.

A method may use `Self` in its result and parameter types to keep the receiver's static typedef:

```
Self List.cdr(Self values);
Self List.append(Self left, Self right);
List List.map(List values, Func fn);
```

For a `ListInt` receiver, dotted lookup treats the first two signatures as `ListInt -> ListInt` and requires a `ListInt` second argument to `append`. Ordinary conversion still applies, so a plain `List` reaches the existing validating `ListInt` converter. `map` remains `List -> List`; x2c never infers covariance for an unmarked method.

`Self` is contextual syntax, valid only in a dotted method declaration or definition with a concrete owner and a compatible first parameter. It may be qualified or nested under pointers. Every occurrence is bound to the original static receiver typedef, even when lookup crosses several typedefs. A direct call to `List_cdr` retains the concrete `List -> List` signature, and generated C keeps that ABI. Prototypes and definitions must agree on both their concrete types and the positions marked `Self`.

Expression-bodied functions

A function that returns one expression may use `=>`:

```
static int twice(int value) => value * 2;

String Pair.describe(Pair pair) =>
  %"${pair.left}, ${pair.right}";
```

This is shorthand for a compound body containing one `return` statement. The expression uses the function's parameter and body scopes and follows the same conversion, cleanup, and lifetime rules as `return expression;`. The semicolon terminates the body; nested compound literals, collection literals, and lambda bodies do not terminate it.

A `void` function has no value to return, so its expression body is the shorthand for a compound body containing one expression statement:

```
static void Counter.step(Counter &self) => self.value++;
```

An expression is required; `=>;` is invalid. The rules for writing `return` in a `void` function are unchanged. Use a compound body when a function needs declarations, several statements, or a comment inside the body.

Statement expressions

x2c accepts the GNU statement expression, which generated C already requires a GNU-compatible compiler to compile. `(` followed by `{` begins one when a `;` stands at the top level of the braces, outside any nested parentheses, brackets, or braces; otherwise the brace keeps its meaning as a braced initializer or a Map literal, so `({})`, `({1})`, and `({a: 1})` are unchanged:

```
int main(void) {
  int area = ({ int width = 6; width * 7; });
  String name = ({ String first = "Ada"; first + " Lovelace"; });
  printf("%d %s\n", area, name);
  return 0;
}
```

```
42 Ada Lovelace
```

Its value and type are those of its final expression statement; a statement expression that ends any other way is `void`. Declarations inside it belong to its block. A template may contain one, and its locals are renamed like any other template local:

```
macro Expression $plus_one(Expr $value) => ({ int t = $value; t + 1; });
```

In a macro body or quotation, a statement hole stands without a `;` of its own, so a hole at the top level of the braces also begins a statement expression unless a comma there makes the braces data. `({ $first $second })` and `({ $only })` are code; data with one element states its type, as in `(Point){ $x }`, or omits the parentheses where its destination gives the type.

A `defer`, or a managed declaration, directly inside a statement expression is an error: its cleanup region would enclose the final statement, and C would lose the value. Either may stand in a block nested inside the statement expression. Leaving a statement expression runs cleanup as described in Errors and cleanup. A statement expression is an ordinary `sizeof` operand, so `sizeof({ ...; })` measures its value's type.

Reference parameters

A function parameter declared `T &name` aliases an addressable `T` supplied by the caller. The parameter name is an ordinary `T` lvalue inside the function, so reading it reads the caller's object and assigning it changes that object:

```
static void swap(int &left, int &right) {
  int temporary = left;
  left = right;
  right = temporary;
}

int main(void) {
  int first = 4, second = 9;
  swap(first, second);
  printf("%d %d\n", first, second);
  return 0;
}
```

The call does not write `&`; x2c takes each argument's address. Passing a reference parameter to another reference parameter forwards the same object. The argument must be an addressable lvalue of the referenced type whose storage remains live for the call. A pointer, `NULL` or `0` is a compile error for a required reference; to pass the object a pointer `p` addresses, write `*p`.

An optional reference parameter is written `T &?name`. It accepts either an addressable `T` or `NULL` (or `0`). A null optional reference means that no caller object was supplied; it is distinct from a caller's pointer-typed object whose value is null. Test the optional reference before reading or writing its object:

```
static int bump_if_present(int &?value) {
  if (!value) return 0;
  value++;
  return value;
}
```

In the nonnull arm of a direct test, or after a null arm that returns or raises, the parameter is an ordinary `T` lvalue: reads, writes, and reference forwarding use the caller's storage transparently. Outside such a proven nonnull path it remains a nullable address and can be tested or forwarded to another optional reference. Required references still reject null at compile time.

Generated C uses a pointer parameter and explicit address-taking and dereferencing. Reference parameters add no runtime representation or ownership behavior. The transparent `&` form is supported only on parameters; reference locals, globals, and return types are not language features. Declarations in those positions are compile errors.

Delegate fields

`delegate` marks a named struct or union field as a fallback for dotted method calls:

```
typedef struct Reader {
  int value;
} Reader;

typedef struct Document {
```

```
  delegate Reader reader;
} *Document;

int Reader.read(Reader reader);
```

If normal lookup finds no applicable `Document.read`, `document.read()` uses the field's static type and emits the same direct call as `document.reader.read()`: `Reader_read(document->reader)`. The original receiver expression occurs once. Pointer and value fields follow the ordinary rules for `.` or `->`, addressability, qualifiers, conversions, and null values. Delegation emits the field access and direct call; it adds no wrapper, temporary, runtime check, symbol, or dispatch table.

Ordinary dot-call lookup runs first. Delegate fields are searched only if those lookups fail, and a member rewrite a program registers for the receiver's own type is tried before them. The shipped component `src/components/delegate-members.x` does this search. `$rewrite` registers it with the receiver pattern `%(expr <delegate> *)`. It runs only when the receiver's aggregate declares a delegate field. The search visits marked fields in source order to produce consistent diagnostics. One resolving path is used; two or more are an ambiguity error, regardless of field order. An explicit outer method therefore resolves an otherwise ambiguous pair.

Delegate fields may chain without a numeric depth limit. Lookup tracks the aggregate types on the current field path. A cycle is reported only when lookup needs that path and finds no valid candidate. One valid path is used even if another contains a cycle; multiple valid paths remain ambiguous.

The terminal field type determines method typing. `Self` parameters and results mean that field's static type, never the outer aggregate, and a method returning `Reader` still returns `Reader`. Delegation does not make the outer type a subtype, adopt a protocol, acquire a conversion or `Var` identity, expose the field's fields, or forward punctuation, indexing, operators, or receiverless calls. `Type.resolve_member` is also direct-only. Its method result cannot represent a projected receiver path.

Thread-local storage

`threaded` is x2c's spelling of C's thread-local storage class, and gives each thread its own copy of an object:

```
threaded int depth;
static threaded Scope current;
```

It pairs with `static` or `extern`, in either order, which is the only pairing C allows. `_Thread_local` and `thread_local` mean the same thing and are accepted, so C that already writes it either way passes through unchanged; all three emit `_Thread_local`.

Pointer type mismatches

Where C reports a pointer type mismatch as a warning, x2c reports an error: passing a `struct Rec *` to a parameter declared `List`, `String`, `Map`, or `Array` is `cannot convert (* "Rec") to ("List")`. Typedef spelling is not a mismatch, so `Ast` and `List` are the same type here. The rule applies only when both sides point at a named struct, union, or enum or at a builtin scalar; `void *`, function pointers, arrays, and types from system headers convert as C defines. A cast still allows the conversion.

Protocols

Protocols declare a compile-time relationship between a concrete base and explicit participant types. The grammar is:

```
protocol-declaration := [ static ] protocol type-name ( identifier ) {
                          associated-declaration*
                          protocol-member*
                        }
protocol-adoption    := [ meta ] [ static ] protocol type-name ( identifier )
                          [ as type-name | tag symbol-literal ] ;
associated-declaration
                     := associated identifier = type-name ;
protocol-member      := type-name identifier . identifier
                          ( parameter-list ) [ = c-identifier ] ;
```

The identifier in a declaration body is a fresh type-variable binder. The identifier in a bodyless adoption must name an existing type:

```
protocol Var(T) {
  associated Key = Var;
  Value T.getindex(T, Key);
}

protocol Var(Array);
```

The optional `as` form applies only to `Var` and says that the participant uses an existing fixed runtime tag:

```
typedef List Row;
protocol Var(Row) as List;
```

`Row` retains its own typed conversions, methods, signatures, and protocol participation. Compatible `List` methods inherited through its typedef chain may satisfy `Var(Row)` without `Row` forwarding methods. A boxed `Row` has the tag `<list>` and uses `List` for dynamic dispatch. Runtime tests therefore cannot distinguish it from another list: `value is Row` and `value is List` test the same tag. The type after `as` must be a scalar, pointer, or fixed runtime class that already has a compiler-known `Var` tag.

The optional `tag` form also applies only to `Var`, but keeps a distinct descriptor under the supplied custom tag:

```
protocol Var(ArrayString) tag <arraystr>;
```

The tag must be a Symbol literal, cannot be built in, and must be unique across the process. The compiler records the full lowercase participant name and rejects a second type with the same tag. `tag` and `as` are mutually exclusive: use `tag` for a distinct runtime class and `as` for a typed view of an existing representation.

Associated declarations precede members. A declaration binder that shadows a visible type receives a warning; use the bodyless form to adopt that type.

`static` may mark a protocol body or a concrete adoption:

```
static protocol Prepared(LocalPlan);
```

That adoption applies only within the translation unit that declares it. It forces local generation even when the complete adoption is public. It is also valid when a private dependency already implies locality; it then records the same local relationship explicitly and does not change linkage.

An adoption is local when its protocol body, participant typedef, required converter, required native target, or adoption row is private. A protocol body declared as `static protocol BASE(T) { ... }` is private to its source file. Local ordinary adapters are `static inline`, appear only in generated C, and are omitted from generated headers. Local native aliases and their signature checks are likewise source-only.

Descriptor-producing adoptions such as `Var(T)` may be local. Their converters and generated thunks remain internal, but descriptor registration is process-wide so boxed dispatch still works. A `Var(T) as R` adoption emits no descriptor for `T`; boxed dispatch uses `R`'s existing descriptor.

One generated C method cannot have incompatible local and external linkage. The compiler reports that conflict at its source location. A protocol whose dependencies are all public retains its public behavior.

Protocol declarations and adoptions appear at the top level. A declaration creates no runtime interface object; a `Var(T)` adoption still emits thunks and registers a descriptor. Each translation unit collects them with its other declarations, and reading source, cached declarations, or a unit interface gives the same result. A byte-identical repeat has no effect, and a conflicting declaration is an error at its location. A fully public native adoption emits its aliases in the header; a local one keeps them in the participant's C file.

Participation must be declared; conversion names and matching typedefs do not imply it. The compiler then determines the adoption's visibility. An adoption resolves in the unit that contains it, so the protocol, participant, required converters, native targets, and constraining participant members must be visible there. A unit that includes it reuses that conformance from the declaring unit's interface; members visible only to the including unit do not change it. When no current interface exists, collection resolves the adoption from the declaring unit's own declarations, so the result does not depend on build order. A declaring unit that imports a package records no conformance. A unit that includes it resolves those adoptions from its own visible declarations.

A typedef descendant with no exact adoption may use the nearest visible ancestor's resolved conformance. It reuses the ancestor's methods, associated types, conversions, and representation without creating an adoption, adapter, native alias, descriptor, runtime tag, registration, or conformance output for the descendant. An exact descendant adoption resolves a new conformance and takes precedence. If that exact adoption is invalid, it is diagnosed and lookup does not fall back to an older ancestor. A local ancestor adoption is inherited only in translation units where that adoption is visible.

Common source forms are a plain adoption for public protocols and public participants, a plain `protocol Var(PrivateType);` for an internal type that crosses `Var`, and an explicit `static protocol PrivateBase(PrivateType);` for a fully private relationship.

Member resolution

Each adopted member resolves independently. Omitting one does not prevent the others from working.

Resolution	Dot call	Mapped punctuation	Descriptor-backed box
`implemented`	participant member	participant member	typed thunk
`native`	checked native alias	checked native alias	not implied
ordinary `base-default`	generated forwarding member	forwarding member	routes through the same winner
`no-member`	no	no	another protocol's owner, else the base fallback
`sig-conflict`	compile error	compile error	compile error

An ordinary base default calls a base implementation after converting the participant through its total forward view. Dot syntax and punctuation use that same semantic member, and `Var(T)` publishes it on `T`'s descriptor even though `Var(T)` itself resolves the member as `no-member`. A `Var` fallback such as `Var.fallback_truth` runs only for a descriptor slot that no protocol filled. It adds no static participant method and does not change native C punctuation.

An explicit adoption may resolve an ordinary implementation through the participant's typedef chain. For example, `typedef List Domain; protocol Iter(Domain);` selects `List.iter` directly; it does not require a forwarding `Domain.iter` method.

An adoption selects each member in this order: the participant's own or imported-package method, then the nearest ancestor's own or imported-package method, then an ordinary base default. Reaching the protocol base uses the same default rule. When checking a method, the compiler treats parameters with the method's exact receiver type as participant parameters. Thus `List.equal(List, List)` can satisfy `int T.equal(T, T)` for a `List` typedef. A concrete result keeps its declared type; a `Self` result preserves the participant's static typedef.

`Var(T)` normally remains exact: an ordinary adoption does not search the participant's typedef chain. `protocol Var(P) as R` is the exception. When `P` inherits from the explicitly named representation `R`, compatible methods of `R` satisfy the adoption without `P` forwarding methods. Boxed values already use `R`'s descriptor, so compile-time conformance uses the same implementation without changing runtime dispatch.

A participant method with the right name but an incompatible signature is a `sig-conflict`. The compiler reports an error at the adoption and identifies the protocol member and conflicting definition.

Associated types

`associated Name = Default;` declares a type variable local to the protocol. A generated member has one owner. The compiler infers a participant's binding by unifying the member signatures the participant declares. Every occurrence must resolve to one type, and the default applies only when no declared member constrains the name.

Calls through a boxed value pass associated arguments and results as `Var`; the generated thunk converts each exactly once. Direct calls keep the participant's static signature.

Dot-call lookup

A dot call visits the receiver's static type, then each typedef parent. At each type it tries the type's ordinary method, then a method an imported package adds to that type, then the member that the type's protocol conformance selects. When the nearest adopted conformance, exact or inherited, resolves the member as anything other than `implemented`, the conformance's member is used, and the type's own method with that

name is not. When that conformance is inherited from an ancestor, a call that the type's own method would otherwise answer is an error: adopt the protocol for that type so its own method is selected.

Conversions required by adapters

The selected methods determine which conversions an adoption needs. The compiler generates adapters for those methods and requires only the conversions each adapter uses:

Adapter	Required conversion
Ordinary default parameter exactly `T`	participant to base
Ordinary default result exactly `T`	base to participant
Dispatcher thunk parameter of `T`	base to participant
Dispatcher thunk result of `T`	participant to base
Native alias occurrence of `T`	the implicit C direction at that position

For a participant `P` and base `Base`, name the forward conversion:

```
Base P.base(P)
```

The forward converter is a total view: every value of the declared participant must produce a valid base value. A reverse converter has the conventional spelling `P Base.p(Base)`, with `Base.as_p` reserved for a real name collision. It may be checked and partial because not every base value must contain a `P`. Signatures cannot express this totality distinction, so it is part of the protocol's rules.

`Var(T)` descriptor thunks both unbox arguments and box results, so most object participants need both directions.

A forwarding protocol whose defaults use `T` only as a parameter may need only the forward conversion. A default returning `T` also needs the reverse conversion to turn its base result into the participant type. The adapter generator does not convert `T` inside compound parameter or result types, so those types are rejected. A missing or unsupported conversion produces an error at the adoption, naming the method that needs it. For an ordinary protocol, a member the participant implements needs no adapter, so its signature may use `T` inside a compound type, as in `int T_area(T *self)`.

Operators

Operators call the same resolved methods as dot syntax. The mappings include:

Syntax	Member
`+ - * / %`	`add sub mul div mod`
direct `+= -= *= /= %=`	`add sub mul div mod`
direct prefix/postfix `++ --`	`add sub` with a converted `1`
unary `-`	`neg`
`== !=`	`equal`
`< <= > >=`	`compare` against zero
conditions, `!`, &&, `	
`needle in value`	`contains`, with the right operand as receiver
indexing and indexed mutation	`getindex`, `setindex`, `updateindex`, `postfixindex`

Matrix multiplication uses the ordinary `left.matmul(right)` method. `Var.matmul` dispatches a registered `matmul` witness. The `@` prefix splices macro syntax; `@` and `@=` are not arithmetic operators.

An operand with no x2c type, such as a preprocessor macro's name, cannot convert for a participant that implements the operator; `images - MEAN` with `#define MEAN 0.1307` is a compile-time type error. Cast the macro, `(double) MEAN`, or bind its value to a local.

A participant may also declare `void T.discard(T)`. The compiler calls it on an unnamed temporary it made for one operator, such as the product in `a * b + c` or the converted `2.0` in `2.0 * x`, immediately after the operator that consumes the temporary returns. A value bound to a name is never discarded. A handle type whose values

own native storage uses this to release each intermediate of a long operator chain before the enclosing scope ends; a value type needs no such member.

A method or function returning a scalar or `void` also discards such arguments after the call. An ordinary call returning a pointer or aggregate may borrow its input, return an interior pointer, or contain a borrowed handle. Unless the callee is a known fresh operator or numeric converter, its temporary arguments remain alive in their enclosing scope. A temporary argument does not make the call's result fresh.

An implemented or native member is eligible. An ordinary base default is also eligible because it is an inherited member reached through a total view. An empty `Var` descriptor slot is not a resolved static member. If no eligible member exists, valid native C behavior still applies; otherwise the compiler reports the ordinary operator error. When exactly one operand of a binary operator is a struct or union participant with the member, or a handle typedef pointing at one, the other operand converts to that type through its declared converter, so `meters + 2.0` and `10.0 - meters` resolve when `double.meters` exists. Scalar pointers and numeric participants keep native C behavior for a mixed operand, so `text + 1` remains pointer arithmetic. The runtime uses the same rows itself: `String.add` implements `add` for `protocol Var(String)`, so `String` `+` and `+=` are ordinary resolved punctuation.

Logical operators preserve C short-circuit evaluation: each operand's `truth` method runs only when C would evaluate that operand. Indexed compound and postfix operations evaluate the receiver, key, and value once, then call the update method once.

Direct compound assignment, `++`, and `--` follow protocol-backed direct updates.

Plain `=`, `===`, and `!==` are never overloaded. `===` and `!==` retain native identity for static participants. Boxed identity is in dynamic truthiness and binary operators.

Members shared by several protocols

A participant implementation satisfies every adopted protocol that declares the same compatible member, and no protocol generates that member.

Two ordinary defaults are an error. The diagnostic names both protocols. Implement the member on the participant to choose its semantics. Declaration order does not decide.

A `Var(T)` adoption puts that same generated member on `T`'s descriptor when `Var(T)` declares the member and `T` does not implement it, so static and boxed behavior agree on one generated C name.

C initializers and static assertions

Array and aggregate initializers accept chained index and field designators. Positional values continue from the designated subobject using C's ordinary brace-elision rules:

```
String labels[4] = {[1] = "first", "second"};
int grid[2][3] = {[0][1] = 7, 8, 9};
typedef struct Record { String names[2]; Var last; } Record;
Record record = {.names[1] = "second", "last"};
```

Each supplied value converts to its destination element or field type. After `.names[1] = "second"` above, the next value initializes `last`, including its ordinary conversion to `Var`. With `.names = "first", "second", "last"`, brace elision fills both array elements before continuing to `last`. Explicit braces delimit a nested initializer. Typedefs and compound literals retain the ordinary destination conversions, including C literals to `String`. The native compiler checks index constant expressions, array dimensions, and bounds.

`_Static_assert(condition, "message");` is accepted at file scope, block scope, and within a struct or union. It emits an ordinary C static assertion and adds no field, binding, or runtime operation:

```
_Static_assert(sizeof(int) >= 2, "int is at least 16 bits");
```

A file-scope assertion stays in its implementation. An aggregate assertion is emitted with the aggregate definition. The native compiler rejects a false or nonconstant assertion. Translation and source analysis alone do not perform that check.

Generic selection

`_Generic(controlling, type: value, ..., default: value)` is a C11 generic selection. x2c resolves the controlling expression and each value, then emits the selection unchanged. The native compiler chooses the

association and reports constraint errors. The selection has no x2c type, so it works where C accepts the selected value directly:

```
int size = _Generic(sizeof(int), size_t: 1, default: 2);
```

A selection does not convert to `Var` or take part in method calls. A cast gives it a type: `Var kind = (int)_Generic(text, char *: 1, default: 2);`.

An association type is a type name of specifiers, qualifiers, and pointers. Name a function-pointer or array type through a typedef.

Mixed declaration rows

A semicolon-terminated declaration at file scope, block scope, or inside a struct or union may restart its declaration specifiers after a comma:

```
int i, float x, char c;
int first, second, const char *name, *alias;
```

Each fresh type starts a declaration with the same scope and source order as if it were on a separate row. Declarators that share a type keep the C spelling: `first` and `second` are `int`, while `name` and `alias` are `const char *` in the second example.

Valid C declarators retain their meaning. In particular, if `T` is a typedef, `int i, T;` still declares an `int` named `T` and hides the typedef. A spelling such as `int i, T value;` restarts at `T`, because the second identifier cannot continue the `int` declarator. Function parameter lists already give each parameter its own type; mixed rows do not extend `for` initializers, `foreach` binders, or macro `Decl` arguments.

Flat List destructuring

Declarations and assignment expressions may destructure the first elements of a `List` into flat identifier targets:

```
Var (a, b, c) = values;
int (start, stop, step) = numeric_values;
(int index, float weight, char code) = %(1 2.5 ${'x'});

Var first, second;
(first, second) = values;
(first) = values;
List copy = (first, second) = values;
```

The leading declaration specifier in `int (start, stop, step)` applies to every name. The parenthesized typed form gives each target a type, written as it would be in a parameter declaration. Both forms declare their names in the enclosing block. Storage classes and per-target initializers are not accepted inside the parentheses. Inside a `%() List` literal, use `${}` to insert a C character expression because the `List` reader reads a bare single quote as `quote`.

The singleton assignment destructures. It assigns element zero; the left side is not a parenthesized scalar assignment.

The source expression is evaluated once. Elements are then assigned from left to right through the same conversions used by ordinary initialization or assignment. Extra source elements are ignored. If the `List` is short, `List.getindex` supplies `void`; a `Var` target receives it, while a typed target follows its existing `Var` conversion and failure behavior.

Assignment destructuring has the same static type, value, and identity as its right side, so it may appear in an initializer, argument, conditional arm, comma expression, return, or another destructuring right side. The right side is still evaluated only once; returning it does not copy the `List`.

The form is flat. Targets must be simple identifiers. Nested targets, members, indexed targets, dereferenced pointer targets, and rest captures are not supported.

Compile-time macros

Macros are explicitly invoked, hygienic compile-time code generators. They differ from C preprocessor macros. Holes bind parsed syntax, definitions use C-like x2c source, and every result is validated for its typed source position before C generation. They make no promise about the runtime semantics of the code they emit; for example, substituting an argument twice may evaluate it twice. A definition's left side looks like its invocation, and its body is x2c with `$hole` markers:

```
macro Expression $twice($value) => $value + $value;

int main(void) {
  printf("%d\n", $twice(21));
  return 0;
}
```

Global definitions are top-level items. A compound statement may instead contain a local definition whose name has no `$`. Both forms emit no runtime declaration. `macro` is a contextual introducer only when the surrounding grammar accepts a definition and a result kind and macro name follow it. It remains legal as an ordinary typedef, variable, parameter, field, or function name everywhere else. A braced definition places `{ ... }` directly after the signature. An expression definition places `=> expression;` after it, including the trailing semicolon. A `Stmt` macro may also use the expression form; its body is then one statement. That statement is an invocation of a `Stmt` macro or a statement decorator, such as `=> $other(args);`, or else the expression statement `expression;`. When that expression is a `meta` call, the call may return a statement, as described under Macro-visible syntax.

Here `guarded` returns an `if` statement, which an expression position would reject:

```
#include "meta.x"
meta static List guarded(List condition, List body) =>
  $!{ if ($condition) $body };

macro Stmt $when(Expr $condition, Stmt $body) =>
  $guarded($condition, $body);

int main(void) {
  int n = 3;
  $when(n > 2, printf("%d\n", n););
  return 0;
}
```

```
3
```

For source compatibility, the compiler also accepts the former `=> { ... }` braced form and a complete parenthesized expression body written as `=> (expression)` without a semicolon. New code should use the forms above.

```
macro Stmt $legacy.guard(Expr $condition) => {
  if (!$condition) return 0;
}
macro Expression $legacy.twice(Expr $value) => ($value * 2)
```

Ordinary global invocation is `$qualified.name(arguments)`. A local macro is invoked as `name(arguments)`. A source file may also give a visible global macro an identifier spelling through a `keyword` declaration, described under Decorators. Macro names are separate from C identifiers and global names may be qualified, as in `$project.logging.trace`. A part after a dot may be a keyword, as in `$error.arg.void`. The `x2c.*` and `lisp.*` macro and Lisp namespaces are reserved for compiler-shipped facilities; `$lisp.bind`, `$lisp.binding`, and `$lisp.install` are the shipped native-binding macros.

Definitions and includes become visible in source order. A definition must precede its first use. A later definition of the same name shadows the earlier definition for later invocations. Earlier expansions keep their original definition, including through transitive includes.

Local definitions

A local definition uses one bare identifier and is a compound-statement item:

```
static int scaled_sum(int scale, int value) {
  macro Expression scaled(Expr $input) => $input * scale;

  return scaled(value);
}
```

Its name becomes visible at the definition and remains visible to the end of that block and in nested blocks. An inner definition of the same name shadows it until the inner block ends. A later definition in the same block replaces it for following source. Fixed C and x2c keywords are not identifiers and cannot be local macro names; `with` is also reserved because it introduces the lexical `with` statement.

A bare `name(...)` selects the innermost local macro before a file-local `keyword` alias or an ordinary call or typedef-style cast. A bare `name` without parentheses remains an ordinary identifier. `(name)(arguments)` is an explicit ordinary call, while `$name(arguments)` selects only a global macro or a `meta` function. A local and global definition may therefore share a name without ambiguity.

A local definition follows the hygiene rules of every macro. `scale` above reads the parameter of `scaled_sum`, where the expansion lands.

Local `Expression`, `Stmt`, `Field`, `Entry`, and `Enumerator` results use their usual positions. A local decorator may target any syntax whose invocation position is reachable before the defining block ends. A local `Unit` result and a local decorator targeting `Function` or `Unit` syntax are rejected. Their invocation positions are outside that lifetime. A macro-generated local definition is published at block position with the same visibility and shadowing behavior as a direct one.

Compile-time Lisp remains one translation-unit session. A local definition may read or change that session, but Lisp definitions, imports, globals, and other effects do not disappear when the local macro name leaves scope.

Holes and sequences

A hole binds parsed syntax, not source tokens. Its kind is normally inferred from every position where it appears in the body: an operand needs an expression, a cast needs a type, and a declarator needs a name. If those uses do not imply exactly one kind, including when a hole appears only inside compile-time Lisp, the definition must annotate it in the invocation pattern.

Hole kinds, result kinds, and decorator targets are one vocabulary of syntactic categories. Each name below selects the same category wherever it appears, spellings match without regard to case, and `Expression` and `Expr` are synonyms. These are the complete hole-kind annotations:

Annotation	Bound syntax	Example argument
`Expr` or `Expression`	expression	`count + 1`
`Type`	type	`FILE *`
`NamedType`	name followed by a complete type definition	`Point { int x; int y; };`
`Decl`	non-function declaration	`static int value`
`DeclaratorRow`	declarator with optional initializer	`second = first + 2`
`Function`	function definition	`int f(int x) => x;`
`Name`	identifier	`checksum`
`Literal`	one literal	`42` or `<char>`
`Param`	parameter declaration	`const char *name`
`Stmt`	block item	`return value;`
`Field`	field declaration	`unsigned ready : 1;`
`Entry`	`Map` row	`key: value`
`Enumerator`	enum member	`ready = 1`
`Unit`	top-level C declaration or definition	`int value;`
`Catch`	one `try` catch arm: pattern and body	supplied by compile-time code
`Captures`	a lambda's complete capture clause	supplied by compile-time code

Annotation	Bound syntax	Example argument
`MatchRow`	one `match` arm or `default`	`case 1: return one;`

A `Catch @arms` hole fills the catch arms of a `try` template, as in `try $body catch @arms finally $cleanup`. `Captures` and `MatchRow` are described below with the templates that use them. Compile-time code supplies `Catch` and `Captures` rows by applying the macro as a value; a source invocation has no argument syntax for them.

Because the definition is already known, an invocation parses each argument according to its hole kind. A type such as `FILE *` therefore works as an argument even though it is not an expression.

An `Expr` argument stays one operand. With the argument `value > 0`, the body `!$condition` means `!(value > 0)`. The same holds for a macro's own result: `macro Expression $twice($value) => $value + $value;` makes `$twice(21) * 2` mean `(21 + 21) * 2`. Generated C has these parentheses only where C precedence would otherwise regroup the expression.

A `Decl` argument captures one declaration without a trailing semicolon. Its comma or closing parenthesis belongs to the macro invocation. The declaration may have an initializer or use flat destructuring with two or more simple identifier targets, but it may not contain multiple comma-separated declarators.

A `DeclaratorRow` argument has no leading type or trailing semicolon. A declaration template supplies the shared type, and a trailing `DeclaratorRow @rows` hole forwards comma-separated declarators in order:

```
macro Stmt $group(Type $type, DeclaratorRow @rows) {
  $type @rows;
}

int main(void) {
  $group(int, first = 1, second = first + 2);
  return second - 3;
}
```

An expression hole may be the receiver of ordinary postfix syntax in the body, including `$value.field`, `$value.method()`, and `$value[index]`. The hole ends at its registered identifier; this does not shorten qualified macro names such as `$project.logging.trace()`. A singular `Name` hole may also appear after `.` or `->`, as in `$value.$member` or `$pointer->$member`. It supplies the captured member spelling unchanged.

A registered `Type` hole may likewise supply the type name before `.` in a method declaration in a `Unit` template, as in `inline Var $type.$method($type value)`. A literal registered type may use the same member form, as in `Logger.$method`. Both expansions use the `Type_method` declaration identity.

Repeated `Unit`-macro applications over `Type` holes are how x2c writes compile-time generic code. Every supplied type must already be a valid named C type, and every expansion produces separately typed declarations and definitions. There is no runtime type argument, erased element representation, or parameterized type spelling such as `Box<T>`. A macro generates named concrete families such as `IntValue` and `DoubleValue`.

During shallow symbol collection, every file-scope `Unit` macro must expand. Collection evaluates preceding top-level Lisp contributions before expanding that producer. Its result is retained for the full parse, which binds the retained function bodies without running the producer again. Static generated declarations stay in their owner; public declarations and protocol rows reach includers. A public function definition can follow its prototype inside the same expansion.

A local `Unit` macro must expand during collection if it contains protocol or adoption rows. Other local `Unit` macros are tried. A failed attempt contributes nothing, and the full parse reports its errors at the invocation.

Protocol declarations and adoption rows are collection-time compiler declarations rather than C declarations. A unit macro may emit them, and importing units receive the retained rows. A `Unit` macro may generate converter and method function definitions before the adoption in the same expansion. Those functions need no separate prototypes when their definitions are already visible. The compiler uses them to resolve the adoption and make the functions visible to later code. Collection discards private declarations with generated bodies, so a private converter needed only within the defining unit can be generated before the adoption.

`@name` is a sequence hole. It captures zero or more arguments of one element kind, must be the final argument, and `@name` in the body is the splice point:

```
static int sum(int a, int b) {
  return a + b;
}

macro Expression $call(Expr $callee, Expr @arguments) =>
  $callee(@arguments);

static int answer(void) {
  return $call(sum, 19, 23);
}
```

The element kind is inferred in the same way as a singular hole and may be overridden, for example `Field @members`. Argument-hole names must be unique, and a name cannot be both singular and sequence-valued. In a quoted `Array` template, an `Expr` sequence hole occupies an element position and inserts its captured expressions in order. It may be empty:

```
macro Expression $values(Expr @items) =>
  %[${10}, @items, ${40}];

int main(void) {
  Array values = $values(20, 30);
  Array empty = $values();
  return values.len() == 4 && empty.len() == 2 ? 0 : 1;
}
```

The `@items` position is part of the macro template; `${expression}` inside an ordinary quoted `Array` still inserts one value.

`Entry` holes capture one `key: value` row; `Entry @rows` captures and forwards zero or more complete rows.

A `Captures` hole holds a complete lambda capture clause as one value. In a template, `%!(@params) using $captures => $body` preserves its value and reference capture rows, with `Param @params` as the trailing sequence. Apply this template as a macro value with the captured rows; recognition binds those rows with `*captures`. Like `Catch` rows, capture rows retain their canonical structure when supplied by compile-time code.

`MatchRow @rows` holds complete `match` arms in a `Stmt` template: `match ($subject) { @rows }`. Each captured row retains its pattern, guarded body, typed captures, and position; preprocessor directive rows stay interleaved with the arms. A `default` arm is one row. Source macro calls may pass comma-separated `case` and `default` arms; compile-time code may pass a List of canonical rows to a macro value. The ordinary match binder handles the rows when the template is expanded.

Result kinds and invocation positions

A macro has one result kind as well as argument kinds:

Annotation	Body	Legal invocation position
`Expression`	`=>` expression `;`	expression
`Stmt`	braced block items, or `=>` one statement	statement
`Field`	braced field declarations	struct or union body
`Entry`	braced, comma-separated `key: value` rows	`Map` literal
`Enumerator`	braced, comma-separated enumerators	enum body
`Unit`	braced declarations and definitions	file scope
`Declaration`	braced declarations with retained public signatures	file scope

The result kind is required between `macro` and the `$` name. An arrow body requires `Expression` or `Stmt`. An anonymous macro's arrow body omits that statement's final `;`, as in `macro Stmt(Expr $v) => $again() $hit($v)`. Braced bodies require `Stmt`, `Field`, `Entry`, `Enumerator`, `Unit`, or `Declaration`. Inside a compound statement, a `Stmt` macro may produce zero or more block items. Where the grammar requires one statement, such as an `if`, `else`, or loop body, the expansion must contain exactly one statement. An explicit `{ ... }` or `do { ... } while (0)` in the production satisfies that requirement; the compiler does not add braces. The invocation must occur in the matching position:

```
#include "meta.x"
meta static List swap_type(TypeInfo type) => type.assoc(<type>);

macro Stmt $swap(Expr $left, Expr $right) {
  $swap_type($left) temporary = $left;
  $left = $right;
  $right = temporary;
}

macro Field $timestamps() {
  long created_at;
  long updated_at;
}

static void reorder(void) {
  int first = 1, second = 2;
  $swap(first, second);
}

typedef struct Record {
  $timestamps();
} Record;
```

An `Enumerator` body contains zero or more enum members. Members may have explicit initializers, and commas separate literal members and sequence splices:

```
#include "meta.x"
meta static List public_name(String name) => x2c_ident(name);

macro Enumerator $status_values() {
  private_start = 3,
  $public_name("STATUS_READY") = private_start + 1,
  $public_name("STATUS_DONE")
}

typedef enum Status {
  STATUS_UNKNOWN,
  $status_values(),
  STATUS_COUNT
} Status;
```

Literal names such as `private_start` are hygienic and may be referenced by later generated initializers in the same expansion. `x2c_ident` publishes the requested name through the generated-name slot. Generated members participate in enum ordering and automatic value assignment, and successful exact members are available to following source. Duplicate generated names and collisions with source members are rejected. A malformed or failed sequence publishes none of its provisional members.

An `Entry` body contains zero or more comma-separated `Map` rows:

```
macro Entry $handler(Literal $key, Expr $value) {
  $key: $value
}

int main(void) {
  int open = 1, close = 2;
  Map handlers = {$handler("open", open), $handler("close", close)};
  return handlers.len() == 2 ? 0 : 1;
}
```

An entry invocation occupies one comma-delimited row position but may expand to any number of rows. In a bare `{...}` literal a direct or keyword-alias invocation is written as a row. In the quoted `%{...}` literal it must sit inside `${...}`, which leaves the quoted Map syntax to parse x2c. Elsewhere in an entry macro body, ordinary expression grammar applies, so an expression macro may generate a key. `@(form)` inserts a `List` of explicit `(map-entry KEY VALUE)` nodes; singular `$(form): value` constructs a generated key.

A `Stmt`, `Field`, or `Unit` invocation ends with its own `;`. `Enumerator` and entry invocations occupy comma-delimited positions and have no semicolon. A result used in the wrong position is rejected at the invocation.

Decorators

A decorator is a macro whose required first parameter is supplied implicitly from the expression or source item following its application:

```
macro Decorator $trace(
  Function $function,
  Expr $channel
) {
  printf(
    "[%s] %s\n",
    $channel,
    $Code.name($function)
  );
  @Code.body($function)
}

$trace("request")
static int answer(void) => 42;
```

The call supplies `"request"` to `$channel`; the compiler supplies the following function to `$function`. The first parameter must be named, singular, explicitly annotated, and one of:

Target kind	Following source	Replacement
`Expr`	one cast expression	one expression
`Function`	file-scope function definition	block items replacing its body
`Stmt`	one statement or local declaration	zero or more block items
`Field`	one struct or union field	zero or more fields
`Unit`	one top-level declaration or definition	zero or more top-level items
`NamedType`	name and type definition, ending in `;`	retained top-level declarations

Parameters after the target are explicit invocation arguments. A decorator application has no semicolon; that omission pairs it with the following target. Only whitespace and comments may separate them. When a statement decorator appears where one statement is required, its production must likewise yield exactly one statement. A compound statement may contain the captured target plus any additional block items without requiring braces at the invocation.

A `Stmt` decorator standing before a file-scope function definition decorates that function's body. The body is parsed as usual, and the decorator's production replaces it. The same function is then rebuilt around the result. Therefore parameters resolve inside the produced items, and a `return` from within the body runs the decorator's deferred cleanup before leaving:

```
$scope() static int total(List values, int bias) {
  int sum = bias;
  foreach (Var value, values) sum += value.int();
  return sum;
}
```

An expression body works the same way. A decorator written this way before a declaration that is not a function definition is rejected.

Block items the production places *after* the captured target are ordinary statements, so a body that returns or that a cause transfers out of never reaches them. A decorator with work to do once the body is finished puts that work in a `defer` written before the target; `$scope`, `$lock`, and `$time` are all written this way.

A `.x` file may give a visible macro or decorator an identifier spelling:

```
keyword ALIAS $QUALIFIED_MACRO;
```

`keyword` is contextual at file scope. It starts this declaration only when followed by an identifier and a `$` macro name, and remains an ordinary identifier elsewhere. `ALIAS` must be a C identifier. A fixed C or x2c keyword does not tokenize as an identifier and cannot be used.

The named definition must already be visible. A macro alias keeps the direct invocation's parentheses, comma-separated typed arguments, sequence arguments, and terminator:

```
keyword swap $project.swap;

swap(left, right);
```

This is `$project.swap(left, right);`. Parentheses remain mandatory for a zero-argument macro: `generate()`.

A decorator with explicit arguments places them before its following target:

```
keyword range $project.range;

range(index, 0, count) {
  consume(index);
}
```

This is `$project.range(index, 0, count) TARGET`. A decorator with no explicit arguments retains the shorter `ALIAS TARGET` spelling. If every explicit parameter is a sequence, `ALIAS() TARGET` supplies an empty sequence; the parentheses distinguish the argument list from a parenthesized target.

Aliases do not add a second argument grammar. Arguments remain comma-separated and are parsed by the macro's own hole kinds. An alias cannot introduce a semicolon-separated control header or capture arbitrary tokens.

The declaration captures the visible macro definition itself, so redefining the same macro name later does not retarget the alias. A later `keyword` declaration for the same alias replaces it only for following source.

Registration and use are source ordered. Expression, statement, field, entry, enumerator, and unit macro aliases retain their result positions. `Decorator` aliases retain their captured `Expr`, `Function`, `Stmt`, `Field`, `Unit`, or `NamedType` target. An invocation in the wrong position receives the same result or target diagnostic as its direct spelling.

Aliases do not globally reserve their identifiers. A macro or a decorator with explicit arguments is recognized only when the next significant token is `(`. The same spelling remains available as a type, declaration, field, label, or function name, but a direct call or typedef-style cast has the invocation shape and is claimed by the alias. A zero-explicit-argument decorator has no parenthesized delimiter. Therefore its spelling is claimed at each parser position that fits its target kind:

- A bare expression decorator can capture a read of a same-named expression.
- A bare block or function decorator can overlap a declaration that begins with a same-named typedef.

A nonstatic `keyword` declaration becomes available to including files at the include position, together with the macro it names. `static keyword ALIAS $macro;` keeps the alias in its declaring file. A reusable ordinary `.x` module may hold both macro definitions and aliases; include that module wherever its syntax is needed.

Macro templates are parsed when defined, so a later alias does not reinterpret an earlier template. Generated `Lists` and strings are not reparsed as alias-bearing source.

An alias adds only a spelling. A decorator cannot turn a block that reads local variables into a C callback. It may wrap an existing callback or a block that captures nothing.

An `Expr` decorator has an arrow-expression body and is parsed as a prefix unary form. Its target is the following C cast expression, including primary and postfix expressions, unary expressions, and casts. It therefore binds before binary, conditional, assignment, and comma operators. Parentheses widen the target explicitly:

```
macro Decorator $nonzero(Expr $target) => $target != 0;

int left = $nonzero() value + 1;
int whole = $nonzero() (value + 1);
```

The first initializer applies `$nonzero` to `value`; the second applies it to the complete addition. The replacement is bound as one typed expression and remains parenthesized at its call site. Surrounding C precedence cannot change its meaning.

Decorators stack closest-first. The inner expansion must leave exactly one target of the required kind for the outer decorator:

```
$logged()
$validated()
int answer(int value) {
```

```
  return value * 2;
}
```

A `Function` decorator preserves the original name, storage, qualifiers, return type, parameters, binding, and method identity. It may splice the old body, replace it, or produce an empty body. A general `Unit` decorator can transform a complete function or declaration, but a public target must retain the same public binding and contract and cannot gain new public siblings. Private targets and hygienically private siblings may be rewritten freely.

`Decorator` adjacency still collects the unchanged source target. A public target's captured source must participate in the decorator's `Match` replacement, and dropping it makes the match fail. Imported decorators work without an extra prototype.

Macro values

A macro is also a compile-time value of type `Macro`, available to `meta` functions. The same value builds code and recognizes it:

```
#include "meta.x"
macro Expression $sum(Expr $left, Expr $right) => $left + $right;

meta static List build_left_twice(List left, List right) {
  Macro sum = $sum;
  List result = sum(left, right);
  Macro twice = macro Expression(Expr $value) => $value + $value;
  match (result) {
    case sum(?a, ?b): return twice(a);
  }
  return result;
}

macro Expression $left_twice(Expr $left, Expr $right) =>
  $build_left_twice($left, $right);

int main(void) {
  printf("%d\n", $left_twice(21, 99));
  return 0;
}
```

```
42
```

`$name` written without an argument list selects the macro as a value. Applying a `Macro` returns a pending invocation. The compiler expands it where the result is bound, so a producer can return it as ordinary syntax. A `case` on a `Macro` value recognizes both the pending invocation and its expanded body, and it binds the same holes in either case. A named case such as `case $sum(?left, ?right):` selects the macro directly. An anonymous `macro Kind(parameters) => body` or `macro Kind(parameters) { body }` is a value without a global name.

A quotation is an anonymous macro applied where it is written: `$!( expression )` for an expression, `$!{ items }` for statements, and `$!Kind{ ... }` for another category, such as `$!Unit{ ... }`. Each `$name` in its body refers to the visible local `name`. Its first use declares a hole, and the hole's position sets its kind. The quotation applies to the locals' values in that order. `@name` is a sequence. A local declared `Type`, which `lib/meta.x` declares as a `List` of type syntax such as `%(double)`, fills a type hole, so `$!{ $type value = $initial; }` declares `value` with the type the local holds. Type syntax spells C type keywords and `*` as Symbols and every other type name as a String, as in `%(unsigned long)`, `%("String")`, and `%(* "Point")`. A type whose name is a Symbol, such as `%(String)`, is an error where it fills a type. A `String` local fills an expression hole as a string literal and an `int` local as an integer literal. A `$name` that matches no local is a macro invocation or a `meta` call, as in a template, so `@rows($items)` splices a `meta` function's result where the quotation expands. A `${expression}`, with `{` directly after `$`, is a hole whose value is an expression of the function that contains the quotation. It means the same as a local declared just before the quotation and named with `$`. The function evaluates each such expression once, where the quotation is written, in the order the holes are written, before the quotation builds its code. Names in the braces resolve at the quotation; names the body declares are not visible there. The expression's declared type gives the hole its kind, as a local's type does: a `Type` fills a type, a `String`, `int`, or `Symbol` inserts a literal, and a `List` inserts

its syntax. `@{expression}` splices a `List`. Each `${...}` or `@{...}` is a separate hole and belongs to the innermost quotation, so `${$!( ... )}` inserts a quotation built first. Inside a `%(...)`, `%[...]`, `%{...}`, or `%"..."` literal in the body, `${...}` keeps its literal meaning and inserts when the built code runs; `${${expression}}` there inserts a value computed at the quotation. For a template Array, `%[${@{items}}]` splices a sequence computed while the quotation is built. It does not add runtime Array splicing. In a List, `@{expression}` retains runtime splicing and does not evaluate during template construction. Literal strings such as `"$ @"`, percent strings such as `%"\$ @"` or `%"$$ @"`, and quoted atom escapes such as `%(\$ \@)` preserve sigils as data. After `case`, `${$name(...)}` is still a macro pattern. These holes exist only in quotations; in a named template's code, `${` still reports a missing macro name. See Quoting code for an example.

A typed quotation is written `$!T{ expression }`, where `T` is one identifier that names a type, or `$!(T) { expression }`, where `T` is any type, a `Type` local, or a `${...}` hole of type `Type`. An identifier after `$!` that spells a macro kind in any case, such as `Unit` or `Type`, always names that kind, so a type with that spelling uses the parenthesized form. A `$!( ... )` group followed directly by `{` is a typed quotation; otherwise it is an expression quotation. A typed quotation builds `(expr T CONTENT)` where it is written. It binds nothing, counts no expansion, and opens no transaction. Its holes take the values a rebuild gives them:

- Syntax is used unchanged.
- A binding identity or `x2c.ident` spelling becomes an identifier expression.
- An `x2c.ident` in a member position becomes its spelling. In a declarator, it becomes the name the declarator declares.
- A number becomes a literal of its own type with its exact value.
- A String or Symbol becomes a literal.

Each number, String, or Symbol item of an expression sequence also becomes its literal. A String is a String literal when the whole expression is the hole and `T` is `String`; elsewhere it is a C string literal. Inner nodes keep the placeholder type of unbound syntax. Free names bind where the code is bound. A typed quotation may not declare a name that its own code spells or apply a template, and inside another quotation it is written as a hole, `${$!T{ ... }}`. The compiler accepts `T` as the expression's type without checking it. A typed quotation states the type the code has; it converts nothing, so write a cast inside the braces when the code needs one. See Code that knows its type.

`$!Type{ T }` builds the type `T`, written as a cast writes a type name. `$!Param{ P }` builds one parameter declaration. A `Type` local or a `${...}` hole of type `Type` may stand for a type or for the base a declarator modifies. Both build where they are written: they bind nothing, declare no name the braces spell, and apply no template. `Type` and `Param` after `$!` always name these kinds. `Decl` names a hole kind only. See Types and parameters.

When one `Name` hole appears in a declaration, a reference, and a member selection, its case checks that the declaration and reference use the same binding and that the member has the binding's source spelling. This also recognizes a local that another macro renamed for hygiene. A different same-spelled binding or a different member does not match. The case captures the original binding syntax.

A template can call a `meta` function in a slot and pass it a sequence hole written with `$name`; the function receives the captured items as one `List`. Applying a macro value with one `List` of syntax as the last argument passes that List as the whole trailing sequence, so `outer(head, items)` and `outer(head, a, b)` build the same code. A `List` of numbers or Strings, such as `%(1 2 3)`, passes the whole sequence too, and each item becomes its literal. A `case` on a macro value captures what a spliced slot built. The capture is bound to the slot's sequence hole, with one element per statement. Each element can then be recognized with a `case` on the macro that built it. The pattern a literal macro value derives is prepared once per process.

A macro's pattern is also a pattern wherever patterns go. In a `case`, `${$name(P, ...)}` is the pattern of code `$name` builds with each argument pattern in its hole, derived while the compiler reads the `case`. Each argument is a binder or wildcard, another `$name(...)` pattern, or a `%(...)` List pattern; a sequence or `Type` hole splices its List pattern's elements. The pattern matches the typed expression or its bare content. Inside a `%(...)` pattern the same form is an element, and in the content position of an `(expr TYPE ...)` shell it is the content alone, so the shell captures the expression's type:

```
#include "meta.x"
macro Expression $neg(Expr $value) => -$value;
```

```
macro Expression $add(Expr $left, Expr $right) => $left + $right;

meta static List classify(List code) {
  match (code) {
    case $add(?left, $neg(?right)): return $!int{ 1 };
    case %(expr ?type ${$add(?left, ?right)}):
      return $!int{ ${type.repr() == "(int)" ? 2 : 3} };
  }
  return $!int{ 0 };
}
macro Expression $kind(Expr $code) => $classify($code);

int main(void) {
  int x = 1, y = 2;
  double z = 1.5;
  printf("%d %d %d %d\n", $kind(x + -y), $kind(x + y), $kind(z + y),
         $kind(x * y));
  return 0;
}

1 2 3 0
```

A named `case` whose arguments are all binders, such as `case $add(?left, ?right):`, keeps the recognition described above, including pending invocations. One with a nested pattern among its arguments, such as `case $add(?left, $neg(?right)):`, uses the derived pattern.

Hygiene and generated names

An expansion means what the same code would mean if it were written where the expansion lands. There are two exceptions. The names its body declares are private to that expansion. A name the macro lists with `using` keeps the file-scope declaration where the macro is written.

Arguments are parsed and typed where the invocation is written, and the names a caller writes in them also resolve where the expansion lands. A `Name` argument is a spelling. A `Name` hole can declare a new caller-visible variable or refer to an existing one. When the expansion declares the caller's `Name`, the caller's other uses of that spelling refer to that declaration. `$repeat(k, 3, k * 2, total)` below shows this. A name the body writes is private, so it never supplies a caller's name.

A declaration written in a body receives a fresh binding identity and a private generated C spelling for each expansion. The spelling of a file-scope declaration with external or no linkage also names its unit's file name and the outermost invocation, so units that include one another can expand the same macro. A field keeps its spelling, because C scopes it to its aggregate, and an anonymous aggregate is a distinct type in each expansion.

A literal name that a body passes to a nested macro's `Name` hole is also the body's own name. A declaration the nested macro makes from it belongs to this expansion, and the body's other references to that spelling reach it. A reference that no such declaration reaches resolves where the expansion lands. A local macro or an anonymous macro written inside a body shares the body's names the same way: its literal reference reaches a private declaration of the enclosing body.

A name the body reads without declaring resolves where the expansion lands, like any name written there: a caller's declaration of that spelling supplies it, including a local, and a local macro reads the locals around it. A name that nothing declares anywhere is left to C, as a native macro such as `errno` is. To read a caller's binding of a name by design, take the name as a `Name` argument.

```
macro Stmt $repeat(Name $i, Expr $count, Expr $value, Name $sum) {
  for (int $i = 0; $i < $count; $i++) $sum += $value;
}

macro Stmt $print_doubles() {
  int total = 0;
  $repeat(k, 3, k * 2, total);
  printf("%d\n", total);
}

macro Stmt $add_tens(Name $target) {
  $repeat(k, 3, 10, $target);
```

```
}

int main(void) {
  int k = 100;
  $print_doubles();
  $add_tens(k);
  printf("%d\n", k);
  return 0;
}

6
130
```

In `$print_doubles`, `k * 2` reads the loop variable that `$repeat` declares from the body's literal `k`. In `$add_tens`, the caller's `k` arrives through `$target`; the loop variable comes from `$add_tens`'s own literal `k`, so the assignment reaches the caller's variable.

Keeping a file-scope name with `using`

A macro keeps a file-scope declaration only when it lists the name with `using`. A braced body lists names as its first item, `using name, other;`. Any macro may write the list after its signature instead, and an arrow body must:

```
static int scale = 3;

macro Expression $scaled(Expr $value) using scale => $value * scale;
macro Expression $landed(Expr $value) => $value * scale;

int main(void) {
  int scale = 10;
  printf("%d %d %d\n", $scaled(2), $landed(2), scale);
  return 0;
}

6 20 10
```

Each listed name is a function, object, or enumerator. A file-scope declaration in the macro's file or a header it includes supplies it. A listed name may also be a native name, such as a C library function, that no x2c declaration supplies. File scope is the whole file, so the declaration may follow the macro. The expansion reads the declaration with its type. A caller's local of the same spelling receives a separate generated C spelling, so the expansion still reaches the file-scope name. Only the locals visible where the expansion lands are respelled; locals of other scopes keep their C spellings for the native macros that name them. A listed name resolves to that declaration in every unit that applies the macro, including through a `Macro` value. A name with internal linkage, such as a `static` function, is reachable only from its own unit, so a macro applied in other units lists nonstatic names, or `module` names within a module unit. Type names cannot be listed; a type name in a body resolves where the expansion lands, in a declaration, a cast, `sizeof`, and `_Alignof` alike.

`using name` keeps a declaration; `using $name` (below) creates a fresh private name; a lambda's `using &name` shares a captured local by reference.

An anonymous macro keeps one kind of value from where it is written: a `Macro` value that its body applies. Templates compose by value, so a later application applies the same children.

A `case` on a macro recognizes a name the macro lists with `using` only by its file-scope declaration. It recognizes any other free name by its spelling, because the code the macro builds binds that name where it lands.

A `struct`, `union`, or `enum` tag that a body defines or declares is private to each expansion, like a typedef, and so are the enumerators it lists. Tags are a separate namespace, so `struct tm tm;` names the outside tag and a private variable. Declarations in nested blocks may shadow same-spelled body locals and tags; references use the nearest lexical declaration. A tag the body only references keeps its outside meaning. To publish a tag, spell it through a `Name` hole or an `$x2c_ident` call:

```
#include "meta.x"
macro Unit $handler_type(name $tag) {
  struct $tag { int depth; };
  typedef struct $x2c_ident("Frame") { int line; } $x2c_ident("Frame");
}
```

```
$handler_type(Handler);
struct Handler handler = {1};
struct Frame frame = {2};
```

Most generated declarations need no directive: writing `int temporary` or a destructuring `Var (key, value)` in a body makes both that declaration and its literal references hygienic. When compile-time Lisp needs a private name before an ordinary declaration can introduce it, place `using $name, $other;` at the start of the braced body. Each name is a compiler-allocated singular `Name` hole. Every occurrence within one expansion receives the same binding, while different expansions receive different bindings. A generated name may not duplicate an argument hole or another generated name and takes no kind annotation or `@` prefix. The directives must precede all other body items.

A body marks an exact public spelling with a call such as `$x2c_ident("Frame")`, which `meta.x` declares; compile-time Lisp spells it `$(x2c.ident "Frame")`. A body that declares a public parameter writes the call at the declaration and at each reference, as in `int $function(int *$x2c_ident("values"))` with `$x2c_ident("values")[0]` in its body. In a declarator slot that spelling creates a declaration and is normally rejected if it is already bound in the same target scope. The one completion case is a function definition produced by a `Unit` macro after a same-scope prototype, including one earlier in that expansion. Canonical return and parameter types, qualifiers, linkage, and any method owner/member identity must match exactly. The definition reuses that prototype's binding once; a mismatch, a second definition, or a collision with another binding kind is rejected at the invocation. A nested scope may still shadow an outer declaration normally. In an identifier expression the spelling resolves to the existing declaration, with its type, and an unknown one is rejected. A bare Lisp `String` does not become a name when returned into a name-capable position.

The compiler parses and types arguments before expanding a macro. It then matches syntax, evaluates compile-time Lisp, and binds the generated syntax. Generated declarations become visible to following source only if expansion succeeds; a failed expansion leaves no provisional symbols or bindings. Expansion is limited to 64 nested applications and 10,000 applications per translation unit; an identical recursive application is rejected immediately. Templates without nested applications or definitions do not count toward the unit limit if each hole occurs at most once. Statements authored by a macro are attributed to the invocation site; captured statements retain their original source locations.

Operators in a macro expansion use the same protocol dispatch as operators written directly, including derived `==`, `!=`, and relational operations.

Compile-time Lisp and imports

Use `$helper(...)` to execute an x2c meta function. The `$(...)` form enters compile-time Lisp for Lisp interoperability, imports and session setup. At top level its result is discarded. Inside a macro body, `$name` refers to the hole's exact AST, and the result is inserted according to the recorded body position. A syntax `List` inserts syntax, a scalar becomes an expression literal, a semantic `Type List` fills a type slot, and a syntax sequence splices only at a sequence splice point. `String`s are values, never source text to be reparsed.

In a percent literal, `${$helper(...)}` and `${$(...)}` reach compile-time code; see percent literals.

```
$(import "helpers.xlisp")
#include "project-macros.x"

macro Expression $computed(Expr $value) => $(car (list $value));
```

`.xlisp` files execute in the translation unit's Lisp session. `$(import ...)` is for Lisp files; ordinary `.x` modules containing macros and meta functions use `#include`. Lisp import paths are relative to the importing file, canonicalized, loaded once, and checked for cycles. Direct Lisp file operations remain available but are not tracked as compiler dependencies.

A free name in a Lisp `lambda` or `defmacro` body reads the definitions the body was written next to, and then the session's globals. The environment the call was written in is not part of that chain, so a caller that binds the same name changes nothing about what the body reads. A lambda written inside a binding form reads that form's bindings and keeps the values it was made with. A macro that needs a value from its caller takes it as an argument.

A `let` binding is not visible inside its own initializer. A lambda a `let` binds therefore cannot call itself by that name, and the self-call reports `(unbound (name h))`. Write such a helper as a `defun`.

```
(let ((h (lambda (n) (h n)))) (h 3))   error: (unbound (name h))
(defun h (n) (h n))                    reads its own name
```

The session that translates a unit inherits the compile-time library and cannot replace one of its definitions. `def` on an inherited name raises `(bad-state (operation "def") (reason "inherited") (name NAME))`, which the compiler reports as a failed compile-time evaluation at the defining form. This covers `defun` and `defmacro`, which are `def`. The library defines many ordinary words, including `filter`, `last`, `map`, `search`, `len` and `apply`, so a macro file needs its own names for its own definitions. A unit's own definitions live in its own session, so one unit never changes what another reads. Lisp definitions treat an inherited binding as final. The host API `Lisp.set_global` can shadow an inherited name in the child's own bindings without changing the ancestor. Lisp `def` still rejects that name, including after a host binding shadows it.

`eval` is an ordinary procedure. The form it is given is evaluated in the session's globals, not in the bindings around the call, so `(let ((z 7)) (eval (quote (add z 1))))` reports `(unbound (name z))`. Build the form with the value in it when a local has to reach `eval`.

Compile-time Lisp is trusted code. It runs with the compiler user's authority, including the existing native bindings and file operations; there is no sandbox. Lisp may also construct canonical AST `Lists` directly; see macro-visible syntax.

The compiler supplies these contextual Lisp operations:

```
(Code.type syntax)
(Code.value literal)
(Code.source_text syntax)
(Code.binding_spelling syntax)
(Code.name function)
(Code.parameter function name)
(Code.body function)
(Code.arguments parameters)
(Type.fields type)
(Type.layout type)
(Type.resolve type)
(Type.is_value type)
(Type.parts type)
(Type.parameters type)
(Type.return_type type)
(Type.element type)
(Type.members type)
(Type.resolve_member type name call)
(Type.tag_name name)
(Type.reverse_name base participant)
(x2c.ident spelling)
(x2c.binding.fresh role)
(x2c.binding.for key role)
(x2c.literal.string value)
(x2c.literal.int value)
(x2c.literal.symbol value)
(x2c.embed.text path)
(x2c.invocation.file)
(x2c.invocation.line)
(x2c.invocation.column)
(x2c.enclosing what)
(x2c.place where code)
(x2c.diagnostic.fail message notes)
(x2c.diagnostic.fail.at node category message notes)

(x2c.diagnostic.warn message notes)
```

An operation on captured code or a semantic type is a method of `Code` or `Type`, spelled `Code.member` or `Type.member`. Any other supported operation is spelled `x2c.<noun>.<verb>`. A name beginning `_x2c.` is a compiler internal with no compatibility promise. Every `Code` and `Type` method in `lib/meta.x` has this Lisp form, including those not listed here.

`Code.type` returns the canonical semantic `Type` for supported typed syntax. During expansion, syntax a template left untyped is resolved where the expansion stands, so a template's locals, casts, and method results report their declared types.

`Code.binding_spelling` accepts a name hole's valid identifier `String` or compiler-issued identifier and binding syntax, and returns its source spelling without exposing the numeric identity. Unknown, malformed, and forged binding identities are rejected.

`Code.source_text` returns the exact source spelling of one complete captured macro argument or decorator target, including its interior whitespace, comments, parentheses, newlines, and literal escapes. Imported macros still read the caller's source, and syntax forwarded through another macro retains where it was written. Constructed syntax, derived subtrees, name values, and calls outside expansion are rejected. Standard Lisp `repr` renders a canonical AST rather than its source spelling.

`x2c.diagnostic.fail` reports a macro diagnostic at its source location with `String` notes and does not return.

`x2c.ident` returns the tagged name value described above and accepts a `String` that is a valid identifier spelling.

`x2c.binding.fresh` returns a new binding whose C spelling is built from the identifier `role`, as for the compiler's own generated declarations. Code that the macro places can declare the binding, and its result can refer to it. `x2c.binding.for` returns the binding the unit gave `key`, or the first time a new one named from `role`, so code placed once under `(unit-support KEY)` and each result that refers to it name the same binding.

`x2c.invocation.file`, `.line`, and `.column` return the caller's x2c source location while a macro body is expanding. Lines and columns are one-based. For an included macro, they still give the caller's location. For example, `$(x2c.invocation.line)` produces an integer expression for the invocation line.

`Code.name`, `Code.parameter`, and `Code.body` inspect a decorator's captured function. `name` returns its free-function or dotted method name as written in source, `parameter` resolves a named parameter to bound identifier syntax, and `body` returns the block-item sequence for an explicit `@` splice. Use `Code.type` on a resolved parameter when its canonical `Type` is needed. `Code.arguments` accepts either a `(params ...)` node or a `List` of parameter nodes and returns their bound identifier expressions. These operations do not construct or mutate functions.

`Type.resolve_member` performs ordinary member lookup for a `Type` and identifier `String`, as `.` does. It returns `(field ACCESS TYPE)` for a field, `(method BINDING SIGNATURE)` for a method when `call` is nonzero, `(ambiguous PACKAGE...)` when imported packages each provide the method, or `nil`. A method's callee expression is `(expr SIGNATURE (ident BINDING))`; callers supply arguments according to its function `Type`. It does not search delegate fields.

`Type.fields` resolves a typedef or qualified `Type` to a complete struct or union and returns its named fields in source order as `(("name" DECLARED_TYPE) ...)`. Unnamed fields are omitted; incomplete and non-aggregate `Type`s are rejected. Each declared `Type` retains pointer, array, qualifier, and bitfield modifiers.

`Type.resolve` follows the ordinary typedef chain and returns its canonical type representation. `Type.layout` returns the ordered field records of that representation, including unnamed members and padding. Each record is `(NAME DECLARED_TYPE)`; an unnamed field has an empty name. Use `fields` for named-member access and `layout` when every declared field affects a decision. `Type.parts` separates a semantic type into its declaration base and declarator modifiers as `(BASE MODIFIERS)`. `Type.parameters` and `Type.return_type` split a function type into its parameter types and its result. `Type.element` returns a pointer or array element type, and `Type.is_integral` and `Type.is_pointer`, from `lib/type.x`, classify a type without resolving it.

`Type.protocol_member` returns the protocol callable selected for member `name` of a type, or `nil` when none is available in the current function.

`Code.value` returns a captured constant's value: for a literal, a `String`, a number, or a `Symbol`, according to the literal. It is the inverse of `x2c.literal.string`, which builds a literal from a value. A literal whose text the parser has already consumed, such as a multi-line interpolated `%"..."`, is rejected; read its spelling with `Code.source_text` instead.

`Type.members` returns an enum's members in source order as `(("NAME" VALUE) ...)`. `VALUE` is one of these:

- `nil` when the member takes its position's value;
- the literal's exact spelling as a `String` when the initializer is one literal, including its base, suffix, and character quotes;

- otherwise the initializer's expression node, which a generator can splice or walk.

Test it with `string?` when both kinds can occur. A non-enum `Type` is rejected.

The Lisp operation `x2c.diagnostic.warn` reports a macro warning at the invocation with `String` notes and returns `nil`, so expansion continues.

`x2c.enclosing` answers the innermost syntax of one kind around the invocation, or `nil` when there is none. `<declarator>` is the initialized declarator whose initializer directly holds the invocation, answered as its one-declarator declaration `(declare BASE (bindings (bind NAME MODIFIERS)))`; `BASE` keeps the storage class, and `Code.type` reads its declared type. A block item begun inside that initializer, as in a statement expression, hides the declarator. `<statement>` is the enclosing block item, answered as its origin anchor `(at N (seq))`, which `x2c.diagnostic.fail.at` reports at. `<function>` is the enclosing function definition with the empty body `(seq)`, which `Code.name`, `Code.parameter`, and `Code.body` read. `<unit>` is a `String` expression holding the unit's path. Another kind is rejected.

`x2c.place` contributes code beyond the expansion's result and returns `nil`. It applies under the expansion's semantic transaction, as the effects of a code-value carrier do. Placed code that declares a name the result uses takes it from `x2c.binding.fresh`. `WHERE` is one of these Lists:

- `(after-statement)` places code after the initialized declarator that `x2c.enclosing` answers, or otherwise after the enclosing block item. A declaration split after the declarator keeps the declarators before it in their own declaration, its shared inline type, and initialization order. The placing expansion's result must be that declarator's complete initializer, through parentheses and typed expression shells. Otherwise, or when the declarator is not an item of a block, as in a `for` header, the compiler reports `managed initializer requires a complete block-local initializer` at the block item. Code placed outside any block item is rejected.
- `(block-exit)` runs code on every transfer that leaves the enclosing block after that declarator or block item, as `defer CODE` placed there with `(after-statement)` runs, under the same rules.
- `(function-entry)` places code at the entry of the enclosing function definition, after its parameters and before its first statement, in placement order. It applies to a macro, which runs while the parser reads the body, and to a rewrite rule, which runs while the body lowers. Code a macro places from a lambda body enters the function that holds the lambda; code a rule places there enters the lambda's own function. Code placed outside any function is rejected.
- `(unit-support)` adds declarations among the unit's file-scope support code; `(unit-support KEY)` adds them only for the first placement under `KEY`. Such code takes the names it shares with each result from `x2c.binding.for`.
- `(unit-init)` adds statements to the `finish` area of the unit's initialization, and `(unit-init AREA)` to the `protocol`, `prepare`, `statics`, or `finish` area, which run in that order.

Any other `WHERE` is rejected as an unknown placement.

`Type.is_value` recognizes numeric scalars and enums, Symbol, Var, Atom, String, List, and their typedef aliases. Pointer-shaped runtime handles such as Array are not classified as values by this operation. `Type.tag_name` returns a round-tripping compact Symbol from the full type-name String and the current owning source file, relative to the compiler root when applicable. This keeps private types with the same spelling in different files distinct. Descriptor registration still checks collisions and capacity; a compact tag does not establish type equality.

`Type.reverse_name` returns the conventional reverse-converter spelling for base and participant name Strings, using the protocol registry's package naming rules. For example, base `"Var"` and a registered package participant `"geometry__Point"` produce `"geometry__Var_point"`.

Lisp has no quotation, so the Lisp library builds the literal a `$(...)` form returns for a value. `x2c.literal.string` turns a compile-time `String` into a runtime `String` literal expression, and `x2c.literal.int` and `x2c.literal.symbol` do the same for a number and a `Symbol`. Other code a Lisp form returns is a quasiquoted List.

`x2c.embed.text` reads a regular text file exactly, returns its contents as a compile-time `String`, and records its canonical path as a translation dependency. It accepts either a Lisp `String` path, resolved relative to the file containing the Lisp form, or complete captured `String`-literal syntax, resolved relative to the caller file where the literal was written. Absolute paths remain absolute. A zero-byte file is valid. Invalid paths, non-regular

files, read failures, embedded NUL bytes, and `String` size overflow are diagnosed. Runtime embedding remains explicit through `x2c.literal.string`.

Names under `x2c.*` or `_x2c.*` with a component beginning _ are private implementation details.

Sharing compile-time definitions

Keep runtime code and compile-time definitions in the same ordinary source module:

```
#include "geometry.x"
```

Nonstatic macros, decorators, keyword aliases, and meta functions become available at the include line. They also reach files that include this one. Each canonical include path contributes once per translation unit, including when an include cycle reaches a file again.

Public inline bodies can use ordinary function signatures declared later in an include cycle. If a declaration producer creates a required signature, place that production before the include that needs it. Looking ahead does not execute later Lisp forms or declaration producers.

A definition marked `static` stays in its declaring source file. For example, `static macro Expression $helper(...)` declares a private macro, and `meta static int helper(int value)` declares a private meta function. Nonstatic top-level Lisp definitions and Lisp imports also follow the ordinary include interface; a `static $(...)` form stays in its file.

Definition order determines which macro or alias later source uses. Macro bodies retain their definition-time syntax; free runtime names follow the expansion-site rules under macro hygiene.

`meta` functions

A function definition may be marked `meta`. The marker precedes the whole declaration, ahead of any storage class:

```
meta int poly(int n) => n * n + 3 * n + 1;
meta static String label(String stem, int n) => %"$stem-$n";

int main(void) {
  printf("%d %s\n", $poly(7), $label("item", 2));
  return 0;
}
```

```
71 item-2
```

`meta` is contextual. It marks a function definition, a bodyless function prototype, a native function definition, an initialized file-static value, or a protocol adoption. Everywhere else it is an ordinary identifier, including as a file-scope name, an assignment target, and a struct field. Types need no marker: compile-time code can use any type the compiler sees.

A bodyless `meta` prototype declares a native function that the compiler itself links, such as `meta double sin(double);`. Compile-time code then calls the compiler's copy. The declared signature must match that function exactly. A function the compiler does not link can come from a loaded native module. A prototype for a function that neither supplies is accepted, and compile-time code that calls it is diagnosed. `meta native` before a function definition declares that function native in the same way, and its body is compiled only for the program. See native C functions.

A `meta` protocol adoption, such as `meta protocol Iter(List);`, adopts the protocol and declares each of that conformance's witnesses the way a bodyless `meta` prototype would. Availability stays explicit for each conformance: a protocol body cannot be marked, and a witness the compiler does not link is not available.

Marking a function `meta` has three consequences.

- The function joins the project meta module. Before translation, the compiler emits the bodied `meta` functions the inputs reach through the ordinary backend, compiles them with the host C compiler (`--meta-cc`), and links them into one cached helper program. Each `$` call runs in that helper; see How a meta function runs. It is callable during translation by `$name(args)` from an x2c body, and Lisp code can call it with `$(name args)`. Inside a `meta` body, `$f(x)` is an ordinary call of `f`, because the whole body runs at compile time.

- The function is also emitted as C and behaves as an ordinary function at run time, unless it reaches a compiler query or a source-template constructor, directly or through another meta function. Such a function has no runtime form and none is emitted for it.
- A call without `$` always calls the emitted function, even with constant arguments.

A dollar-prefixed function call requires compile-time evaluation. Each argument is a constant, arithmetic over constants, captured macro syntax, or another dollar call; any other expression, such as a program variable, is diagnosed. Visible macros retain precedence over meta functions with the same name. In a macro body, a captured hole passed directly to a `$` call passes its code. The runtime expression is not evaluated.

A directly forwarded hole retains its declared capture kind, including `Function`, and complete captures retain their source text and path context. Inside a macro template, an explicit call can supply a type, generated name or syntax sequence as well as an expression; the ordinary slot binder checks the returned value for that position.

Inside meta bodies, Unit and Stmt source macros used in expression positions construct deferred invocation Lists from computed arguments. Normal binding expands them when inserted into a program. Existing Expression macros still expand normally. See source templates.

A `meta` body is ordinary x2c compiled as C, so every construct the unit can compile runs at compile time with its C meaning; see C objects during compilation. A body that lets the address of its own storage outlive the call is rejected where the address leaves.

Meta functions may pass and return represented values such as wide integers, floating values, Strings and collections during compile-time execution. Explicit `$name(args)` insertion, and the optional `$(name args)` Lisp form, accept numeric values with their native Var family, computed strings as C string literals, Symbols, identifiers and nonempty expression-code Lists. Arrays and Maps are reconstructed as fresh runtime collections, including nested collections; cycles and shared mutable collections are rejected. Callable values and compiler addresses cannot be materialized directly. A parameter or result with no Var form, such as a struct, a C `bool` or a pointer, cannot cross a dollar call. This boundary does not limit internal returns to `int`; see results.

Compile-time objects belong to the compiler. A file-scope variable that a `meta` body reads is the compiled group's own instance, initialized from the same source initializer; it is never the program's variable. A `meta static` value lives in its provider's compiled group, and the helper re-initializes it for each translation unit, as described below. The project helper runs mutable `meta static` initializers when a compile-time call first reaches their provider in each translation unit. Including an unused provider does not run those initializers.

A `meta` body runs as native code, with these consequences:

- A `meta` function defined by a macro expansion or a `$` call is an error; one written in an ordinary `.x` file joins the project meta module.
- A project `meta` body receives compiler facts as arguments and can query the compiler while its helper call waits.
- A raise in a body is a diagnostic at the `$` call. A crash, exit, or stack overflow ends the helper and is reported at the call; the compiler continues. The Lisp step and depth budgets do not apply to a native body; a call longer than `X2C_META_TIMEOUT` seconds (60 by default) is stopped instead.
-

A nonstatic `meta` function in an included `.x` module is callable during translation and has an ordinary runtime declaration. Its provider source contains the runtime definition. Repeated includes contribute the public declaration once.

See Meta Functions for the authoring guide.

The same operations from x2c

Every public operation above is also declared in x2c, in the optional module `lib/meta.x`, so a `meta` function that includes it calls the compiler directly and a macro's implementation does not have to be written in Lisp. The declarations are the signatures; the semantics are the ones described above. A project `meta` function's queries, such as `Code.type` and `Type.fields`, ask the compiler while the call waits; see What the compiler answers.

The compiler binds each operation under its x2c name and derives the Lisp name from it. A method keeps its spelling: `Code.type` and `Type.is_value` are the same names in Lisp, and a predicate method answers `int`, 1 or 0, in both. In an `x2c_` helper each _ becomes `.`: `x2c_invocation_line` is `x2c.invocation.line`, and `x2c_ident`

is `x2c.ident`. Where the Lisp library defines the derived name itself, that definition stands. x2c code builds literals with typed quotations, so `x2c.literal.string`, `x2c.literal.int`, and `x2c.literal.symbol` have no x2c declaration.

The operations are bodyless `meta` prototypes supplied by the compiler itself, which generates its table of them from those declarations.

`x2c_meta_definition_hashes()` returns a `Map` with one entry for each function or initialized file-static value the unit has defined so far. The key is the definition's name. The value is a String hash of the definition's code tokens, from the first token after any `meta` marker to the end of its body or initializer. The compiler carries native copies of the `meta` functions in the source modules it ships (`src/linked-meta.x`, generated from them) and records their hashes with it. An included `meta` definition runs as its native copy when its hash matches the copy's. The hash of every unit definition it reaches through references must match too. An edited definition, or one that reaches an edited definition, is compiled with the unit's group like any other.

A `meta` function that makes an explicit `$` call or uses a source-template constructor is also compile-time-only. A run-time call to a compile-time-only function is diagnosed where it is written. The diagnostic gives the function and the reason.

Compiler facilities load before author imports, and macro code cannot redefine them.

Standard Lisp exposes the `List` matcher of Pattern Matching as six names. `match` tests a whole subject. It returns association-list bindings, `nil` on failure, and truthy `(())` for a match with no named binders. A pattern describes a `List` shape, so a subject that is not a `List` fails without raising. `bound` reads one binder out of that alist. `search` finds the first matching subtree. `match-replace` and `search-replace` rewrite the whole subject or the first matching subtree from a template. `match-case` tries each clause's pattern in order and evaluates the first body whose pattern matched, with that pattern's binders in scope as variables; a final `(else body)` clause runs when nothing matched.

```
(match-case form
  ((add ?a ?b) (+ ?a ?b))
  ((neg ?a)    (- 0 ?a))
  (else        0))
```

A clause's pattern is quoted implicitly and written as `List` syntax, with the same `?binder` and `*binder` spellings. A binder keeps its `?` where the body reads it. Without an `else` clause an exhausted `match-case` is `nil`.

Compile-time Lisp also reaches the core value types directly. A Lisp value in a session is an x2c `Var`: a Lisp list is a `List`, a Lisp string is a `String`, and a quoted name is a `Symbol`. The operations below are the library's own, so a macro and the program it generates compute with the same `List`, `String`, `Map`, and `Array`.

```
List.getindex  List.last     List.index    List.contains
List.get       List.assoc    List.array    List.sort

Array.new      Array.len     Array.push    Array.getindex  Array.setindex
Array.take_last Array.shift  Array.unshift Array.insert    Array.remove
Array.find     Array.contains Array.count  Array.join      Array.list

Map.new        Map.len       Map.get       Map.getindex    Map.setindex
Map.contains   Map.del       Map.getdefault Map.setdefault Map.list

String.len     String.find   String.rfind  String.count    String.contains
String.startswith String.endswith String.getindex String.getslice
String.add     String.lower  String.upper  String.capitalize
String.repeat  String.replace String.join  String.split    String.split_lines
String.partition String.rpartition String.find_all
String.remove_prefix String.remove_suffix String.escape String.unescape

Var.tag        Var.kind      Var.is        Var.parse       Var.convert
Var.integer    Var.floating  Var.is_null

Symbol.len     Symbol.str    Symbol.compare
```

Each name takes the arguments its library entry documents and behaves identically. `Var.parse` reads a `String` as an `int`, `double`, `string`, `symbol`, or `char`, and `Var.convert` changes a value's numeric tag, which is how compile-

time code narrows a `double` to an integer. `Map.list` returns a map's entries as `(key value)` pairs, because a `Map` cursor has no Lisp representation.

The Lisp core keeps its own spellings for cons-cell and text work. `car`, `cdr`, `cons`, `length`, `reverse`, `string-append`, `substring`, `lower`, `str`, and `repr` are unchanged and reach the same code. Note that Lisp's `assoc` returns the matching pair while `List.assoc` returns its second value, as the library documents.

An operation whose parameters or result have no `Var` representation is not available, which excludes `Iter` cursors, pointer out-parameters, variadic entry points, and `char *` parameters. A program embedding its own session loads the same names from `etc/lisp-values.xlisp`.

Macro-visible syntax

Lisp sees the canonical parsed, typed `List` AST used by `--dump-ast`, with source `(at ID NODE)` wrappers removed. Tags and semantic `Types` remain `Lists`. Binding records are visible so syntax can be preserved and compared, but their numeric IDs are opaque and must not be forged. Use `Code.binding_spelling` when text is required.

The compiler records where syntax was written separately from its AST value. `Code.source_text` reads it, as described above. Diagnostics retain the definition, invocation, import, and generated ancestry even after a macro returns a new `List`. Returned syntax must be valid for its expression, field, enumerator, block-item, or translation-unit position. A `meta` call written as a whole expression statement holds that statement's block-item or statement position, so it may return a statement as well as an expression. This holds in an ordinary function and in an expansion alike, such as the body of `macro Stmt $same(Stmt $code) => $same_of($code);`. A `meta` call inside a larger expression holds an expression position. Inside an expansion every `List` result is code. Elsewhere a `List` result is code when it is an expression node, a macro application, a quotation, or an identifier; any other `List` remains a runtime value. Protocol declarations, macro definitions, preprocessor nodes, and other compile-time-only source items are retained at translation-unit position and apply their source-order effect. The compiler rejects forms that are malformed or invalid in that position. Lisp may construct canonical AST `Lists` directly, including static declarations. The compiler accepts a valid structure whether it came from the parser, a `Match` capture, a template, or handwritten Lisp. It does not verify every type or binding annotation in a constructed `List` or reject it because of its origin. Like C pointer operations, this metaprogramming is unchecked. Prefer literal templates and the contextual `x2c.*` operations when you want the compiler to construct syntax for you.

Named types and declaration production

`NamedType` captures `NAME TYPE;` or the forward form `NAME;`. The name comes first, with no equals sign. `{ FIELDS }` abbreviates a value struct; `{ FIELDS } *` declares a pointer representation, with `struct` optional before the braces. Qualifiers, fields, arrays, and pointers use the ordinary type and declarator grammar. The name is reserved before its fields are parsed, and the complete definition supplies its representation. A forward declaration does not imply a pointer. Layout must be complete wherever the ordinary type rules require it.

The captured form is `(named-type NAME TYPE)`, where NAME is a String and TYPE is the complete canonical type syntax; a forward uses an empty TYPE. This form is constructible by any macro or Lisp producer. Binding it publishes the same ordinary typedef and aggregate declarations as the captured source.

A `Declaration` macro, or a `NamedType` decorator, produces declarations once while the owning source's public declarations are collected. Its result is retained for full binding; the producer is not evaluated again to obtain its bodies. Nested declaration producers share this rule. Static publication and ordinary dependency invalidation apply to the retained result. Imported and cached declarations publish the selected signatures of their owning source.

The canonical container is `(declaration-bundle (rows ITEM ...))`. Rows may include ordinary top-level syntax and these constructible forms:

- `(default FUNCTION)` supplies a function candidate. An ordinary declaration of that exact function in the owning source wins, including a later one. Only the selected body is bound. A candidate produced by a static type declaration has static linkage. Two ordinary definitions still conflict.

- `(declaration-recipe CALLBACK ARGUMENTS)` defers a Lisp producer until the owning source's declarations are available. It is evaluated once, and its declarations join the same bundle.
- `(default-forward CHILD PARENT MEMBER FALLBACK)` selects an ordinary parent method after signatures are collected and supplies a child forwarding method. FALLBACK is an optional function candidate when no parent applies.
- `(syntax-recipe CALLBACK ARGUMENTS)` supplies syntax when a retained body is bound, allowing it to use the selected method signatures.

Retained bodies also carry the usual macro bindings and source locations for diagnostics. A consumer cannot replace a provider's exported default by defining another function with its name.

Class declarations

The shipped `class` keyword aliases the `$class` NamedType decorator. See Classes and System Macros for the complete default-method inventory, construction, and lifetime examples. A class preserves its explicit representation and ordinary typedef ancestry. It adds replaceable methods through declaration defaults. An explicit `new` suppresses its generated constructor and init requirement. Derived classes forward the nearest applicable constructor; variadic forwarding requires an explicit constructor.

An aggregate class gets a generated `new` when it defines one of these initializers:

- `void T.init(T &, ...)` for a value, which also accepts the existing `T *` receiver;
- `void T.init(T, ...)` or `int T.init(T, ...)` for a heap pointer.

That `new` takes the parameters after the receiver. It calls `init` on zero-initialized storage, which a heap class obtains from its replaceable `T.alloc`. A zero result from an `int` initializer releases that storage and makes `new` return NULL. Otherwise flat value fields produce positional constructors in declaration order, with unnamed bitfield padding omitted, and non-flat or resource-containing aggregates require `init`. Heap defaults allocate through Scope and provide early `free`, which accepts NULL and calls `void T.drop(T)` first when the class defines it; they do not otherwise own fields. Scalar and derived aliases retain their ordinary Var representation. Heap classes box identity; aggregate values box a Scope-owned copy and require compatible equal/hash operations, generated for supported value fields.

`str` and `repr` are independently replaceable. Aggregate value str delegates to repr; heap str prints identity. Generated repr traverses printable fields and uses addresses for opaque pointers and fixed arrays. Repeated identities on the active rendering path print their pointer form. Descriptor registration retains the runtime's worker-start freeze rule. Overflow descriptors extend the compact descriptor table's capacity.

Managed initializers

`$auto(value)` is an ordinary expression macro. Its expansion asks `x2c.enclosing` for the initialized declarator, places a release after that declarator with `x2c.place`, and returns `value` unchanged. Forwarding through another macro or parentheses keeps that result the complete initializer. When the declaration finishes, the compiler checks that the result is the declarator's complete initializer. Then it binds the release. The release requires automatic storage and Cleanup participation. It saves the initialized value in a private const local and defers that saved value's `cleanup` call. The declaration keeps its type, binding identity, initialization conversions, and enclosing scope.

The declared type must participate in `Cleanup(T)`, whose member is `void T.cleanup(T)`, directly or through its ordinary typedef ancestry. Initializers run once in source order. Each successful initializer registers its cleanup before the following initializer runs, including within a compound declaration. Cleanup releases the acquired value even if the declared binding is reassigned. Reassignment does not release that value early, and returning or storing an alias does not cancel cleanup. An argument that is itself an `$auto` form, or a plain name of a local that an earlier `$auto` manages, is rejected, since the value would be released twice. Use an explicit `defer` when cleanup must instead read the binding's value at exit, such as a lazily filled Scope slot or a staging buffer cleared after ownership transfer.

Static, external, or threaded storage, field initializers, for-header declarations, assignments, returns, call arguments, and forms nested inside operators are outside this enclosing-block position. There is no runtime expression helper or expression-exit cleanup.

Checked foreign aliases

The reserved target macro `$x2c.foreign.alias` binds one top-level X2C function name directly to an ABI-compatible C function:

```
$x2c.foreign.alias(fclose)
inline int File.close(File file);
```

The target must be one fixed-arity function declaration without a body or initializer, and the argument must be one direct C identifier. Static aliases remain private; non-static aliases are published in the generated header like any other declaration. The generated C uses `_Static_assert` and `_Generic` to check the exact function-pointer type before defining the lowered X2C name as the native name. Calls and address-taking therefore use the native function itself; no wrapper object, thunk, argument mapping, or initializer is emitted. Variadic aliases are rejected.

Typed callback adapters

The reserved expression macro `$x2c.callback.adapt` adapts one direct function to an explicitly named callback typedef:

```
typedef String (*StringCallback)(Var);

static StringCallback array_string =
  $x2c.callback.adapt(StringCallback, Array.str);
```

The target must name a fixed, non-variadic function-pointer typedef. The source must be a direct free-function or `Type.method` designator, not a function-pointer variable, lambda, bound receiver, or conditional expression. Target and source must have the same arity and exact non-`void` return type. Parameters may be exact, or a target `Var` parameter may be extracted to the pointer/typedef owner or `Symbol` required by the source. The adapter performs no numeric or general coercion and does not insert, drop, reorder, or default arguments.

The compiler emits one translation-unit-local, statically typed thunk for each source/target pair and reuses it within that translation unit. The thunk has a prototype before any file-scope initializer that references it, and no incompatible function-pointer cast is emitted. The expression therefore suits private descriptor tables and other fixed C callback positions.

Values and literals

The tagged value model specified here is introduced gradually in values and Var.

Array and Map literals

In operand position, a bare `[` begins an `Array` literal. Its comma-separated elements are ordinary x2c expressions, each converted to `Var`, and a trailing comma is accepted:

```
int n = 4;
Array values = [1, n * 10, "text", <sym>, [n]];
```

A `{` whose first entry is a Map entry begins a `Map` literal. An entry is `KEY: VALUE`, or an `Entry` macro invocation. A key that is a bare identifier is an `Atom`, with the same spelling rules as a bare collection `Atom`. Every other key, and every value, is an ordinary x2c expression. Parenthesize an expression key that is a single identifier:

```
int n = 4;
String name = "ada";
Map ages = {ada: 36, grace: n + 41, "text": 1, <sym>: 2, (name): 3};
```

A Map entry is identified by its first `:` that is outside nested brackets and is not part of a `?:` conditional. A brace without such an entry remains a C initializer. Inside an initializer, a bracketed index followed by `=`, `.`, or `[` remains a designator; any other bracket is an `Array` literal. Both literals build a fresh object at each evaluation, as `%[]` and `%{}` do, and a declared typed `Array` or `Map` family builds its own representation. Nested `[...]` and `{...}` are evaluated literals of the same kinds; nested `List` data is written `%(...)`.

Elements, keys, and values are evaluated once each, left to right in source order, as in the percent forms below.

`[]` is a fresh empty `Array`. The empty brace `{}` is a fresh empty `Map` when its destination is a `Map`, a `Var`, an alias of either, or a type that converts from `Map`. The same rule applies to `Array`. For any other destination, `{}` is the native zero initializer. An element or value of a bare `[...]` or `{...}` literal is the exception: there `{}` is an empty `Map`, so `[{}]` holds one Map. Outside a declaration initializer, a brace that remains an initializer becomes a compound literal of its destination, as an argument, a `return` value, an assignment, or a `?:` arm: `Map m = ready ? {} : NULL;` builds a Map, `Var v = ready ? {} : NULL;` holds a Map when `ready`, and `return {3, 4};` in a function returning a struct builds that struct. An anonymous struct or union destination has no compound-literal spelling and is rejected.

The percent forms `%[...]` and `%{...}` below keep their quoted grammar. The bare forms differ only in evaluating elements, values, and non-identifier keys.

Percent literals, quote, and unquote

In operand position, `%` followed by a literal delimiter is the quoting sigil. The delimiter selects the literal grammar for the contents. Here, quoting means a change of parser context. Some percent forms evaluate their contents. Binary `%` remains the modulo operator, and `%!` remains the lambda-literal prefix.

`%"`, `%[`, and `%<<` open literals after any token. Modulo by a string literal is invalid C, and neither `[` nor `<<` begins an expression, so a cast may precede these literals: `return (Path) %"$base/$name";` quotes. A cast may precede a bare `[...]` Array literal as well: `(Var) [1, 2]`.

In `%(`, `%{`, and `%!` after an operand, including a closing parenthesis or brace, the `%` is the modulo operator, except where a statement begins:

- after the parenthesized condition of `if`, `while`, `for`, or `switch`, so `if (dirty) %(git stash).job().run();` quotes;
- after the `}` of a compound statement or declaration body, so a `%(...)` statement may follow the closing brace of a `foreach` body.

A brace counts as a body when the token before its `{` is one of these:

- `;`, `:`, `{`, `}`, `]`, or an identifier;
- `else`, `do`, `try`, `finally`, or `defer`;
- a `)` that closes a control condition, a `match` subject, or a parenthesized list after an identifier, as in a function header, `foreach`, `with`, or a decorator.

A compound literal or initializer brace follows an operator or a cast, so `%` after it stays modulo: `(int){9} %(4)` is `1`.

The tokenizer has no type information, so it cannot distinguish a cast from a parenthesized operand. After any other `)`, `%(`, `%{`, and `%!` are modulo, as in `(a) %(b)`. Parenthesize such a literal after a cast: `(List) (%(echo done))`.

The collection and string literal forms are:

Form	Static type	Element syntax
`%(a b c)`	`List`	quoted values separated by whitespace
`%<<a b c>>`	`SymbolSet`	literal compact `Symbol`s in dense order
`%[a, b, c]`	`Array`	quoted values separated by commas
`%{a: b, c: d}`	`Map`	quoted key/value pairs
`%"text"`	`String`	quoted text plus unquote

The static type in that table is the default. The declared target may instead be one of the optional typed families: a packed `Array` from `typed-array.x`, a packed `Map` from `typed-map.x`, or a typed cons chain from `typed-list.x`. The literal then builds that representation directly, and an element that does not convert raises `<no-convert>`. A null source stays null.

The empty forms are `%()`, `%<<>>`, `%[]`, `%{}`, and `%""`. Each `%[]` or `%{}` evaluation creates a fresh allocated object that can be mutated immediately; null is not an empty `Array` or `Map`.

Inside a `List`, `Array`, or `Map`, a bare spelling is a case-sensitive `Atom`. It is a compact `Symbol` when `Symbol` encoding reproduces the spelling exactly; otherwise it uses `<lsym>`. Consequently, `%(name)`, `%[name]`, and the key in `%{name: 1}` hold the same value as an explicit `<name>`, and the angles are redundant. Use angles when bare

collection syntax would read the text as another value, as in `<1>` for the `Symbol` `1` rather than the integer `1`, or when compact `Symbol` representation is required. The empty `Symbol` must likewise be written `<"">` because a bare `Atom` cannot be empty. An angle spelling that does not fit is rejected instead of falling back to `<lsym>`.

Integer, floating, signed numeric, and character literals retain their concrete types. A signed integer token is one value, not C's unary minus: it selects the first signed type among its suffix's choices that holds the signed value, in any radix, so `%(-0x80000001)` is the `long` value −2147483649 and `%(-2147483648)` is an `int`. A `u` suffix keeps the unsigned type C would choose and its modular negation, so `%(-1u)` is the `unsigned` value 4294967295.

`$name` inserts one identifier expression into a quoted collection. `${expression}` inserts one arbitrary x2c expression, including calls, member and index expressions, operators, casts, compound literals, `NULL`, enum constants, and comma expressions. `@name` and `@{expression}` splice a `List` into a surrounding `List`; Arrays and Maps have no splice form. The inserted and spliced expressions of one `%()`, `%[]`, `%{}`, or `%""` literal are evaluated once each, left to right in source order. The arguments of an ordinary function call keep C's unspecified order. The identifier forms are the short versions of the braced expression forms. The collections chapter walks through building and using each one.

For `Lists` that will be evaluated as Lisp, `'`, `` ` ``, `,`, and `,@` are the short forms of `quote`, `quasiquote`, `unquote`, and `unquote-splicing`. Each wraps the one element that follows it, wherever it appears, so `%(...)` and the Lisp reader construct the same `List` data from the same text. Reader punctuation also ends a bare `Atom`: `%(a,b)` is `a` followed by `(unquote b)`. These spellings do not replace x2c's `$` and `@`: those still insert or splice x2c values while the `List` itself is being constructed. A `@` followed by whitespace, `)`, or `=` is ordinary `@` or `@=` atom data. `%(op @ a b)` remains a List of atoms; it does not authorize matrix operator syntax.

`%<<...>>` is an immutable ordered `SymbolSet`. Bare entries are compact `Symbols` rather than `Atoms`, and their source positions are their numeric indexes. Entries must be literal spellings; interpolation, splicing, and runtime expressions are not accepted. Every entry must survive compact `Symbol` encoding exactly. Duplicate encoded `Symbols` are a compile-time error. `SymbolSet.index` returns the source-order index or `-1`, `contains` tests membership, `getindex` performs the reverse mapping, and the `Iter` protocol traverses members in source order. The compiler emits the perfect hash and ordered membership table as static bytes, with no runtime construction or allocation.

Inside `%()`, `%[]`, `%{}`, and `%""`, `$` is the unquoting sigil. It leaves the selected literal grammar, parses and evaluates one x2c expression, inserts the converted result, and returns to the literal grammar. In `%()`, `@` makes the same crossing but splices the evaluated `List`'s elements. Collection insertion converts to the representation required by the literal; `String` insertion accepts `String`, `Symbol`, declared converters, `Var`, aliases of `Var`, and supported numeric values under the existing conversion rules. The shipped component `src/components/interpolation.x` joins a `%""` literal's parts, a translator that `$rewrite(%(expr ("String") (segments *)))` registers on the literal's `segments` head. A program's rule for `String.join` receives only the calls that the program writes, including one inside a `$` insertion. It does not receive the call that the shipped rule writes.

The braces are part of the unquote and contain one complete x2c expression, including nested calls, indexing, casts, assignments, conditionals, comma expressions, and nested literals. In a collection, `$name(...)` instead inserts `$name` and then reads `(...)` as a nested symbolic `List`. In `%""`, the parentheses after `$name` are text. A runtime call in any quoted form is therefore `${name(...)}`. In `%""` and in a quoted `"..."` element, `$$` and `\$` each produce one literal `$`: `%"a$$"` is `a$`.

Nested `(...)`, `[...]`, and `{...}` remain quoted `List`, `Array`, and `Map` data, and nested `"..."` selects the x2c `String` grammar with its interpolation and escape rules. None repeats the `%` sigil, because a bare `%` inside quoted collection data is an `Atom`: `%(k %"txt")` reads as the three elements `k`, `%`, and `"txt"`. `${"text"}` instead inserts an ordinary C string expression. Unescaped comma, colon, `]`, and `}` terminate collection Atoms; backslash escaping or angle syntax expresses those bytes as data. `%<<...>>` accepts literal `Symbols` only and permits no interpolation or splicing.

`$(...)` remains compile-time Lisp in ordinary x2c code. Inside a quoted collection, `${$(+ 40 2)}` first unquotes into x2c and then enters compile-time Lisp. `$name(...)` remains a macro invocation in x2c. These parser contexts do not change Lisp `quote` or `quasiquote`.

Percent strings are byte strings: they may span physical lines and their `\\x`, `\\u`, and `\\U` escapes consume one or two hexadecimal digits. Ordinary C string and character literals instead follow C escape widths, including

exactly four digits for `\\u`, eight for `\\U`, and one or more for `\\x`; an unescaped newline does not continue an ordinary C literal.

Adjacent ordinary C string literals form one expression, including across comments or newlines. Their separate escape boundaries are preserved: `"\\x41" "B"` contains `A` followed by `B`. The result keeps ordinary C string typing and converts to `String` when its context requires it. A context whose type is a `class` or `typedef` alias reaching `String`, such as `class Path String;`, requires the same conversion. A literal used as the receiver of a method with no `char *` definition converts to `String` first, so `"hello".len()` is `5`. `foreach` iterates such a literal as that `String`, and a raise detail accepts it as an immutable `String`. In these three positions a parenthesized literal, a `?:` whose arms are both literals, and an object-like macro defined to a string literal, `#define NAME "x"`, are literals too, so `(ready ? "on" : "off").len()` and `NAME.len()` work. A macro name is a literal only while every definition it has is a string literal and no `#undef` has dropped it. A name written next to a literal is one of the adjacent words when it is such a macro or when this unit cannot resolve it, as a header's `PRId64` is, so `printf("%" PRId64 "\n", count)` works. This adjacency rule does not combine percent strings or change quoted collection syntax.

Immutable literal construction is cached for the process lifetime. This includes an ordinary C string literal when its context promotes it to `String`, as in `String name = "x2c";`; a dynamic `char *` or `char[]` value still converts when the expression is evaluated. The ordinary literal keeps its exact C escape spelling and decoding, and does not acquire percent-string escape rules.

Cached `String`s, boxed `Var` values, and canonical `List` graphs are allocated eagerly during translation-unit initialization and retained for the process lifetime. A cached literal in a public inline function uses private storage in each C translation unit that includes the generated header; canonicalizers still make equal `String` and `List` values share their value identity. A guarded call on the inline entry is the fallback on hosts where eager constructors do not run. Therefore, an allocation failure for a cached literal can occur before `main` runs.

Caching does not change the other identity rules: `List`s and non-empty `String`s are canonical when constructed through their canonicalizers; `Array`s and `Map`s are mutable identity-bearing objects and are constructed at each evaluation. Non-empty `Array` and `Map` literals use counted construction.

File-static `String`, `List`, `Array`, and `Map` declarations may use their percent literals directly. A file-static `Var` may likewise use one of those literal values:

```
static String child = %"$root/child";
static String root = %"root";
static List names = %($root $child);
static Var literal_index = %{root: $root, child: $child};
```

The compiler leaves each C declaration zero-initialized and moves its runtime assignment into the translation unit's guarded initializer. Literal caches run first, followed by the assignments in dependency order; a declaration may therefore depend on a later file-static declaration. Independent assignments retain source order. Every object referenced by one of these initializers must itself be file-static, and a dependency cycle is a compile-time error. Native initializer operands expand at the original source position: later macro definitions do not change them, `__COUNTER__` keeps source order, and explicit inline tags remain visible at file scope. Native macro invocations retain their normal stringizing and token-pasting rules. Tags hidden inside an opaque native macro remain subject to native C scope.

A function-local `static` declaration with a runtime-valued initializer runs once when control first reaches it. This includes aggregates containing `String` or `Var` values, function calls, and values from function arguments. Concurrent calls wait for the initializing call to publish the whole object. If initialization raises an Error, a later call retries; side effects already performed by the initializer remain. Recursive initialization raises `bad-state`. This includes a cycle between initializing threads. A `static threaded` declaration follows the same rule separately in each thread. Native constant initializers retain native C static storage; a function name, including a `$x2c.callback.adapt` result, is constant.

Runtime-initialized local objects preserve their declared type, qualifiers, array shape, and address across calls. Their storage lasts until process shutdown, or thread teardown for `threaded` objects. This storage does not extend the lifetime of values it refers to: an initializer allocating a Map inside a temporary Scope still gives that Map the ordinary Scope lifetime. Use an owner that outlives every use of the stored value. Failed initialization

retains the reserved address; the next attempt starts with zeroed storage. Only successful initialization publishes the value, and retry does not change referent ownership.

A `goto` or switch dispatch cannot bypass a runtime static declaration and enter its remaining block. Put the declaration before the switch, or put it inside a case's own block. A nested switch reached after initialization is valid.

Native macro calls can be runtime initializers. A bare unknown native macro name, however, retains native C initializer rules: its expansion may be either a constant or a call, which x2c does not inspect. Use an explicit function call when a runtime expansion needs first-use initialization. Native macro bodies also cannot name a lowered local object implicitly; pass the object as a macro argument so its ordinary bound expression is preserved.

Symbols

`<name>` and `<"punctuated name">` produce a 64-bit immediate `Symbol`. `Symbols` are encoded, not allocated or entered in an intern table.

The encoding is selected from the payload:

- the restricted 5-bit alphabet preserves at most ten characters;
- the general 7-bit encoding preserves at most seven characters.

A source literal is accepted only when decoding the encoded value reproduces its exact spelling. Case folding, _/- folding, and truncation are compile errors in literals. `Symbol.new` remains the runtime encoding API and truncates or normalizes dynamic input to the selected capacity. `Symbol.try_new` instead reports whether dynamic input has an exact compact representation and leaves its output untouched when it does not. Equality compares the encoded value.

`Symbols` also name outcomes and states. For example, `Lisp.read` returns `<value>` or `<eof>`. It raises `<incomplete>` or `<malformed>`. `Error` causes such as `<bad-sig>`, `<no-symbol>`, `<bad-arity>`, `<bad-types>`, and `<bad-result>` are `Symbols` too. Numeric enums remain appropriate when their values are indexes, packed fields, arithmetic inputs, or external numeric encodings.

For Lisp input, `<incomplete>` means more source can complete the current form: an open list, string, block comment, or symbol literal, a reader prefix without its form, or an escape cut off by end of input. `<malformed>` means the bytes already prove the form invalid, such as a bad escape, raw string newline, malformed number, invalid closed symbol literal, or stray closing parenthesis. The shared `Tokenizer` classifies lexemes; `Lisp.read` checks structural form balance and preserves the form-start cursor on either failure.

Atoms

`Atom` is the canonical exact-name type used by `List` literals and Lisp. It is `Var`-shaped and has two representations:

- an immediate compact `<symbol>` when encoding and decoding reproduces the exact spelling bytes;
- a private `<lsym>` whose payload is the canonical `String` pointer otherwise.

`Atom.intern` is the only function that maps a spelling to a representation. Repeated construction of one spelling has identical `Var` bits. Long `Atom` equality therefore completes in the existing raw-`Var` fast path, and its hash is the `String`'s cached hash. `Atom.str` returns the exact bytes, and `Atom` representation escapes delimiters, numeric-looking prefixes, comment openers, whitespace, controls, and backslashes so both `%()` and `Lisp.read` recover the same value.

Long `Atoms` follow the canonical `String` pool lifetime, which is described in scopes and lifetime. Promoting a `List` also promotes long `Atom` payloads contained in it; a standalone long `Atom` that must escape a child `String` pool can promote its `Atom.str` before that pool is released.

For a side-by-side introduction to the two types, see symbols and atoms.

`Var`, Null, and `void`

`Var` is the tagged value used by heterogeneous collections and generic runtime APIs. Supported native source types round-trip through their matching `Var` tags. `long`, `unsigned long`, `long long`, `unsigned long long`, and `long double` use immutable scope-owned boxes while `Var` itself remains eight bytes. Their tags are `<long>`,

<ulong>, <llong>, <ullong>, and <ldouble> respectively; the runtime derives their widths from those C types. Explicit i48/u48 tags complete the numeric family set. The same names with * or ** identify their pointer families.

Raw Null is the all-zero Var. A typed empty String and nil/List preserve their type tag when boxed. Arrays and Maps instead require allocated objects even when empty; boxing a null pointer with either value tag is an invariant violation. void is the all-ones terminal/tombstone value used by APIs for exhaustion or absence. Status-bearing Iter and Map APIs separate success from payload bits. Iter.next, Map.get, and Map.del use void for exhaustion or absence. Malformed representations, including null wide boxes and null Array or Map values, are rejected. Every accepted nonnull pointer must name a live object established by its constructor. Collection construction, Array.push, and Map.set reject void; raw Null is valid collection data.

Lowercase void is contextual. In a type position it retains C's no-result type, including function returns, (void) parameter lists, void *, casts, and sizeof(void). In an expression position it is a Var literal for the all-ones sentinel. Parentheses do not change that distinction: (void) is a parenthesized sentinel expression when it stands alone, while (void) call() and (void *) pointer are casts because an operand follows the complete type.

Two sentinel operands compare equal with == and identical with ===; != and !== are their exact inverses. A sentinel and any ordinary value compare unequal. Var.equal, Var.fallback_equal, Var.same, and the four equality and identity operations accepted by Var.binary follow those rules. Var.str, Var.repr, Var.write_str, and Var.write_repr all render the sentinel as lowercase void.

Hashing, ordering, truthiness, iteration, conversion, arithmetic, compound updates, and increment or decrement raise <void-op> when they receive it.

Membership with `in`

The contextual in operator tests membership in a collection. It is the operator spelling of the receiver's contains member, selected through the Var(T) protocol, and it compiles to that same call:

```
Map ages = {ada: 36};
List names = %(ada grace);
Array counts = [1, 2, 3];
String text = "hello";

int keyed = <ada> in ages;
int element = <grace> in names;
int value = 2 in counts;
int substring = "ell" in text;
```

A Map tests its keys, a List or Array its elements, and a String a substring. The left operand converts to the member's parameter type, so a Var holding the key works as well as a literal. A receiver whose type does not implement a contains member is rejected with operator 'in' requires an implemented contains member. The member comes from a protocol the receiver adopts: Var(T) declares one, and the one-member Contains(T) protocol gives membership to a type that holds members without boxing them, as SymbolSet does. There is no not in spelling; write !(name in ages).

in is a keyword only between two operands, so struct buffer *in and int in = 0 remain ordinary names.

Exact Var-tag tests with `is` and `is not`

The contextual is operator tests a Var value against a source type or an exact tag Symbol. Add not after is to invert the result without wrapping the test in !():

```
Var value = 42;

int integer = value is int;
int byte = value is u8;
int sequence = value is List;
int exact = value is <i32>;
int pointer = value is (Var *);
int absent = value is void;
int not_byte = value is not u8;
int present = value is not void;
```

The left operand must have static type `Var` or a file-scope alias of `Var`. The result is `int`. The operator has relational precedence, and its left operand is evaluated exactly once. A `Symbol` expression on the right is also evaluated exactly once. A type selector is compile-time syntax and is not evaluated. `is not` has the same precedence and evaluation rules, then logically negates the exact-tag result. `is` and `not` remain ordinary identifiers in all other positions, so existing variables with either name and calls such as `value.is(<i32>)` are unchanged.

The test compares exact `Var` tags. It does not test numeric convertibility, protocol ancestry, or C type identity. Qualifiers and aliases are resolved to the tag family used by boxing, which means source types that share one tag cannot be distinguished. For example, `char` and `signed char` both select `<i8>`, while `int`, `int32_t`, and `i32` all select `<i32>`.

The right operand may be either a type or a `Symbol` expression. Accepted types include a supported scalar or system numeric typedef, a builtin runtime type such as `String`, `List`, `Symbol`, or `Var`, a visible type with a declared `protocol Var(T)` conversion, or a package-qualified form of one of those types. Supported pointer and double-pointer types are also accepted, but declarator-shaped types must be parenthesized:

```
static int pointer_tags(void) {
  int number = 0;
  int *pointer = &number;
  int **double_pointer = &pointer;
  Var pointer_value = pointer;
  Var double_pointer_value = double_pointer;
  return pointer_value is (int *) &&
         double_pointer_value is (int **);
}
```

An unparenthesized pointer selector is rejected with the parenthesized spelling. `Arrays`, function types, unsupported pointer depths, unknown types, and aggregates without a `Var(T)` adoption have no tag to select and are rejected. Enum selectors are rejected because enum values box as the shared `<i32>` family and retain no nominal enum identity.

A literal or computed `Symbol` is used directly as the exact tag selector:

```
Var value = 42;
Symbol wanted = <i32>;

int literal = value is <i32>;
int computed = value is wanted;
```

An unregistered `Symbol` cannot match a valid `Var` tag, so the result is false. No other expression type is accepted on the right: numeric, pointer, object, and `Var`-valued expressions do not stand for types.

Lowercase `void` is the only selector that is both a type spelling and a value. It tests the all-ones absence sentinel through `Var.is_void`: `void is void` is true, while the sentinel does not match any ordinary tag or type. Raw `Null` has the pointer-family `<p48>` tag, so it matches `(void *)` and does not match `void`; there is no `Null` selector.

A `Type` hole or a `Symbol`-valued `Expr` hole in an `Expression` macro may occupy the selector position. An expanded type is resolved in the invocation context, including pointer types:

```
macro Expression $has_type(Expr $value, Type $T) => $value is $T;
macro Expression $has_tag(Expr $value, Expr $tag) => $value is $tag;

static int has_pointer_type(void) {
  int number = 0;
  int *pointer = &number;
  Var value = pointer;
  return $has_type(value, int *) && $has_tag(value, <i32*>);
}
```

Dynamic numeric conversion

The public numeric conversion domain contains all 15 `Var` families: `i8/u8`, `i16/u16`, `i32/u32`, `i48/u48`, `long/ulong`, `llong/ullong`, and `f32/f64/ldouble`. `Var.convert(value, tag)` returns the converted value and raises the specific `Error` cause when no conversion exists, the target is invalid, the value is out of range, or the `Var` encoding is invalid. Compiler-inserted `Var`-to-native numeric conversion passes through the same function before using the target extractor. The scalar-named readers `Var.int`, `Var.double`, and their siblings apply the

same rules directly. A matching tag reads its payload, and any other numeric tag converts through `Var.convert`. The raw payload readers `Var.integer` and `Var.floating` and the wide `*_value` functions remain exact low-level decoders.

Integer-to-integer conversion never uses a floating intermediary. It retains the destination-width low bits; signed destinations interpret those bits as two's-complement. Floating-to-integer conversion truncates toward zero when the truncated result is representable. NaN, infinities, and out-of-range results report `<conv-range>`. Integer-to-floating and floating narrowing use normal host floating-point behavior.

These numeric rules do not define conversions between nonnumeric tags. `String`, `Symbol`, collection, pointer, and custom-object values use their declared conversions. Numeric-to-`String` interpolation boxes through the verified numeric tags and renders with `Var.str`. A structurally valid nonnumeric `Var` may convert to its own tag unchanged; conversions to other tags must be defined separately.

A printf-family call with a static format can also format a `Var`. Numeric format specifiers convert it to the required promoted C type through `Var.convert`; `%s` uses `Var.str`. The recognized families are `printf`, `fprintf`, `sprintf`, `snprintf`, `String.printf`, `File.printf`, and `Buffer.printf`. A call with a `Var` variadic argument requires one direct static format literal. Automatic lowering covers numeric conversions, `%c`, `%s`, `*` width and precision, and the `hh`, `h`, `l`, `ll`, and `L` modifiers. Other modifiers, pointer and count conversions, wide strings and characters, and positional formats require explicit native arguments. This lowering is the compiler component `src/components/printf-format.x`, which registers a `$rewrite` call pattern for each family member. A project can register its own call pattern the same way, for example for its own logging function.

Dynamic truthiness and binary operators

When a condition has static type `Var` or a file-scope alias of `Var`, the compiler applies the `Var.truth` base member in `if`, `while`, `do`, classic `for`, `?:`, and unary `!`. A static protocol participant uses its eligible `truth` member: an implementation, native alias, or base default. The compiler applies the member independently to operands of `&&` and `||`, leaving C's short-circuit evaluation intact.

Truthiness is false for numeric or `Symbol` zero, `Null` and null pointers, canonical empty `String`/`List`, and empty `Array`, `Map`, `Block`, `Bytes`, or `Buffer` values. It is true for nonzero values, NaN, infinities, nonempty containers, and other nonnull objects. A nonnull `Iter` is true even when exhausted; the test does not probe its producer.

If either operand of `+`, `-`, `*`, `/`, `%`, `<<`, `>>`, `&`, `^`, or `|` has static `Var` identity and the other operand is `Var` or statically numeric, both operands convert to `Var` and the result is `Var`. A nonnumeric operand is rejected at compile time if its type is known statically, or at runtime if it is held in a `Var`. Native-only expressions remain native C. Integer operations promote their operands to a common integer type. Arithmetic and left shift retain the low bits that fit that type; signed results interpret those bits as two's-complement. Signed right shift fills the sign bits. A negative shift count or a count at least the promoted left width fails, as does integer division or remainder by zero. `String`-tagged `Var` values additionally accept `+` with a `String` or raw C string operand. The result is a canonical `String`; no other runtime tag is stringified implicitly.

Floating operands support `+`, `-`, `*`, and `/` in the widest participating floating family. Host floating behavior includes infinities or NaN from floating division by zero. Remainder, shifts, and bitwise operators reject a floating operand. Direct `Var` lvalues support prefix and postfix `++`/`--`; prefix returns the updated `Var` and postfix returns its original value. Unary `-` uses an eligible `neg` protocol member for a static participant or the `Var` base member for a dynamic value. Comparisons dispatch separately. For a static participant, `==`/`!=` use an eligible `equal` member, relational punctuation derives from `compare`, and `===`/`!==` remain unconditional identity tests. Eligible means implemented, native, or a base default. When the operands of a protocol operator are two different typedef names that share an ancestor, such as an alias of `String` and a `String`, or two aliases of `String`, the operator uses the eligible member of their nearest shared ancestor. Equality, identity, and total ordering do not become binary-arithmetic operations.

A comparison with a `Var` operand converts both operands to `Var`. A comparison of two numeric `Var`s behaves as the same comparison between their native values in C, after the usual arithmetic conversions. So `3 == 3L` and `-1 == 4294967295u` hold, `-1 < 1u` does not, and a NaN is unequal to and unordered with every value. Other `Var` operands compare with `Var.equal`, and relational punctuation orders them with `Var.compare`. `===` on `Var` operands compares their bits, and two numbers are identical when they have the same type and the same

bits. So `NaN === NaN` holds and `+0.0 === -0.0` does not. `Map` keys compare with `Var.equal` and sorting uses `Var.compare`, so both keep numeric types apart.

For `==`, `!=`, `<`, `>`, `<=`, and `>=`, a C string literal opposite an operand of static type `String`, or an alias reaching `String`, converts to that type before ordinary protocol comparison. This applies in either operand order, through parentheses around the literal, and to a `?:` whose arms are both literals. Other C pointer expressions, including variables and casts, retain their native comparison behavior. `===` and `!==` do not perform this literal conversion.

For `+`, when each operand is a `String`, an alias reaching `String`, a `char` array, a `char *`, or a `const char *`, both convert to `String` and the result is the `String` that `String.add` returns, so `path + ".o"` concatenates.

The shipped component `src/components/dynamic-operators.x` implements these lowerings. It also implements the dynamic compound assignments and increments below, and unary fallback rejection after protocol resolution. It registers each translator with `$rewrite` and a Match pattern typed `Var`, such as `%(expr ("Var") (op (!or + - (!quote *) / % ...) ? ?))`, which keys the rule by each operator the pattern names and `Var`. A pattern that types the operation or an operand `Var`, or an alias of it, takes the dynamic operations. They run after protocol operators resolve, only when an operand has `Var` identity, so native operands never reach them; each operation arrives typed `Var`. A program's rule registered for the same operator is tried first. The holes in a macro pattern select the operands that its rule receives, so a rule for `$!Money{...}` with `typedef Var Money` leaves other `Var` operands to the shipped rules. The same order applies to every shipped component: a program's rule for the same head is tried first.

Direct runtime calls to `Var.binary` additionally accept comparisons and eager `&&`/`||`. The compiler does not use that eager logical path. It converts each `Var` operand at its original C short-circuit position.

Protocol-backed direct updates

A direct participant lvalue supports `+=`, `-=`, `*=`, `/=`, or `%=` when its protocol-resolved `add`, `sub`, `mul`, `div`, or `mod` member has signature `Participant member(Participant, RHS)`. The right operand is converted to `RHS`. Prefix and postfix `++`/`--` use `add` or `sub` with the integer `1` converted to `RHS`.

The lvalue is evaluated once. Its current value is passed to the member once, and the returned `Participant` is stored once. Compound assignment and prefix forms return the stored value; postfix forms return the original value.

Dynamic compound assignment

Native-only compound assignment remains native C. If a numeric compound assignment involves `Var`, the compiler takes the address of an addressable lvalue once. The runtime loads its current value, performs the dynamic operation, converts the result to the declared target family, and stores once only after every step succeeds. The expression result is that converted value. A failed conversion, divide, remainder, or shift leaves the target unchanged.

This rule covers native scalar numeric and `Var` lvalues, including C indexing and member access. The compiler selects a typed adapter from the resolved storage family; plain `char` and `signed char` remain distinct storage types even though both box as `<i8>`. Enum and bitfield targets are excluded.

`String`-tagged `Var` values accept `+=` with `String` or raw C string operands. Native `String` lvalues accept the same `+=` spelling as concatenate-and-rebind. Concatenation finishes before the binding is changed.

`Array` and `Map` elements support all ten numeric compound operators and prefix/postfix `++`/`--`. The runtime performs the element read, operation, stored-tag conversion, and write in one call. A missing `Map` key or out-of-range `Array` index fails without insertion, growth, or mutation. Numeric `+=` is the `Map` exception. A missing destination key is inserted with the right-hand side as its initial value, as though the prior value were numeric zero, and the inserted value keeps the right-hand side's numeric tag. The read, modify, and write happen in one call; that is not a thread-safety guarantee.

Additive initialization does not make `void` numeric. A missing or `void` right-hand side still fails, as do nonnumeric `+=` and every other compound operator when the destination key is absent. Successful insertion is a structural `Map` change and may invalidate outstanding traversal state.

A direct, optionally parenthesized `Array` or `Map` indexed right-hand side uses one typed cross-container helper. The source value is captured before the destination is committed, including same-container and same-slot cases.

A cast or larger expression reads the source and then performs one destination update. Each base, selector, and value expression is evaluated once, and C's operand evaluation order remains unchanged.

Dynamic conversion, operators, and updates do not have parallel error-code APIs. A failure raises its specific cause through the ambient `Error` channel. Each of these causes is a shared cause, so it never returns to the operation. Only a matching filtered `catch` receives it, and an interrupted update leaves its target unchanged.

The `try_*` prefix remains for status results whose payload alone cannot represent every successful result. Examples include `Map.try_get` for presence, `Iter.try_next` for exhaustion, `String.try_long` for parse success, and match/search operations for no-match. Those integer results answer whether a result exists or an operation applies. Failure causes still go through the ambient `Error` channel.

Indexing and slicing

Native C arrays and pointers retain C indexing. `Array`, `List`, `String`, and `Map` also define indexed access.

A typedef of one of these four collections inherits its bracket reads. An exact `getindex` method on the alias takes precedence over the inherited getter. A typedef of an ordinary C pointer keeps native indexing unless it defines its own `getindex` method. Indexed writes and updates continue to use the collection's mutation operations; a read override does not replace them.

- `Array`, `List`, and `String` accept negative indices.
- An `Array`, `List`, or `String` index is an `int`; a wider value is outside the supported domain and converts as any `Var`-to-`int` conversion does.
- Slice bounds and steps share that `int` domain; a wider value is truncated, so a step of `2^32` becomes `0`.
- An out-of-range `Array` or `List` read returns `void`.
- An out-of-range `String` read returns `-1`; its indexed result is a byte represented as an `int`, not a one-byte `String`.
- A missing `Map` key returns `void` through `get` and bracket reads; `Map.try_get` reports presence separately.
- `Array` and `Map` indexed assignment are supported.
- `Array` and `Map` indexed compound assignment and prefix/postfix `++`/`--` are supported as single-call updates.
- Numeric `map[key] += value` initializes an absent key from `value`; other indexed updates require an existing destination.
- `List` indexed assignment is not supported.
- `String` indexed assignment is not supported: `String` is immutable, and an in-place write would corrupt every equal `String` sharing its canonical storage. Use the copy-producing `String.withindex` result, or bind a transient `String.malloc` buffer to a `char *` and write that natively.

The optional typed families have different indexing rules:

- A packed `Array` from `typed-array.x` has a raw bracket: reads, writes, and compound updates index a concrete element pointer with no null test, no bounds test, and no negative-index normalization. An index outside the elements is undefined as it is for the equivalent C pointer. `try_get` is the checked read, and it does normalize a negative index.
- A packed `Map` from `typed-map.x` adopts `protocol Var`, so its `getindex`, `setindex`, `updateindex`, and `postfixindex` members provide bracket reads, writes, compound updates, and numeric postfix updates.
- A typed cons chain from `typed-list.x`, such as `ListInt`, is a `List` typedef. Its bracket reads behave as `List` reads, including negative indices, and it rejects indexed updates as `List` does.

`Array`, `List`, and `String` support `value[start:stop:step]`. Bounds and step may be omitted. Negative bounds and reverse steps are normalized consistently. A zero step is invalid.

`List` indexed updates and `String` character updates remain unsupported. `Array`-level and `Map`-level `+=` are also unsupported; update an element instead of the container binding.

Iterator destination omission

Iterator sources and lazy operations take a final `Iter` destination pointer. That argument may be omitted for calls nested in an iterator expression that is consumed immediately by `try_next`, `next`, `list`, `array`, `foldl`, `any`, `all`, `find`, `count`, `sum`, `product`, `min`, `max`, or `foreach`.

For every missing destination, x2c passes a distinct zero-initialized `struct Iter` compound literal. Its automatic lifetime is the enclosing block, so every source remains valid while the final consumer runs. Explicit final destinations are preserved, including a mixture of explicit and omitted destinations in one chain. Calls nested in iterator-valued arguments, such as both inputs to `zip` or `map2`, are completed recursively.

Only iterator destinations can be omitted. x2c has no general default arguments. An iterator assigned to a local, returned from a function, boxed, or passed to an arbitrary call still requires explicit storage at every stage. `unzip` also remains explicit because its two result iterators share one caller-owned `UnzipShared` buffer.

Lambdas

The literal forms are `%!(parameters) => expression` and `%!(parameters) => { statements }`. An optional `using &name, &other` clause between the parameters and `=>` captures those surrounding bindings by reference.

An expression body parses through assignment precedence. An unparenthesized comma separates arguments of an enclosing call; write `(first, second)` when the comma expression itself is the lambda body.

Current guarantees:

- an expression body produces its value; a native C `void` expression runs once and produces `void`;
- a block body accepts statements, including declarations, control flow, `defer`, and nested lambdas. `return expression;` converts the result to `Var`; bare `return;` and fallthrough produce `void`;
- bare parameters are `Var` values;
- nullary and multi-parameter forms work;
- unlisted automatic values referenced by the body become read-only snapshots when the lambda is created, provided their types convert to and from `Var`;
- assignment, updates, taking a captured binding's address, and passing it by reference require `using &name`. Explicit readers and writers share the original binding; adding another lambda cannot change a snapshot;
- globals remain directly accessible.

```
int value = 1;
Func read = %!() => value;
Func bump = %!() using &value => ++value;
value = 2;
printf("snapshot=%ld shared=%ld\n", read().integer(), bump().integer());
// snapshot=1 shared=3
```

Clause names resolve in the surrounding scope. Parameters and local variables still shadow them. Repeated entries are redundant, and an entry unused by the body creates no capture.

Typed parameter syntax retains the complete parameter declarator. A parameter such as `int &value` aliases its caller's lvalue just as it does in a function; a dynamic `Func` call checks the source type and qualifiers before passing the address to its generated adapter. Value parameters are evaluated once and boxed, while reference parameters take the lvalue's address without first reading it.

A noncapturing lambda remains a generated C function and can adapt to a supported C callback type, including `int (*)(int)`, in an initializer, assignment, argument, or return. A local typedef of the callback type has the same behavior. Parentheses around the lambda preserve this adaptation. Where a `Func` is expected, including a `(Func)` cast, a fixed nonvariadic function or function-pointer value converts implicitly when the `Func` conversion supports its parameters and result. A lambda after a cast needs its own parentheses, as in `(Func) (%!(int a) => a * 13)`, because `%!` after `)` reads as modulo. Value parameters and results need a lossless `Var` conversion, and reference parameters retain their typed lvalue address. A pointer parameter whose type has no `Var` tag of its own, such as `const char *` or `struct timespec *`, accepts any pointer or `String` argument. A struct parameter taken by value receives a `<p48>` pointer to the bytes to copy, and a struct result arrives as a `<p48>` pointer to a copy of its bytes in the active `Scope`. Native code passes such an address as a `void *`, because a struct pointer has no lossless `Var` conversion of its own. A direct function or noncapturing lambda has one file-static `Func`

handle that every conversion reuses. The handle lives for the program; operations such as `Lisp.bind` that take ownership of a `Func` leave it in place. A function-pointer expression is evaluated once and its exact pointer is copied into a new `Func`; later assignment to the source pointer does not retarget it. A null function pointer instead produces null `Func` without allocating a binding. Variadic functions do not convert implicitly; use the advanced `Func.new_rest` API when a native operation consumes a rest `List`.

A capturing lambda has static type `Func`; calling it directly returns `Var`. The same is true after a noncapturing lambda converts to `Func`. Successful effect-only calls and explicit `void` results produce `void` through either route. `Func.apply` preserves that no-value result. A value parameter declared as `Var` may receive it; a concrete typed value parameter rejects it. Rest bindings reject it before constructing their argument `List`. A `Func` cannot be passed as a context-free C callback because that ABI has nowhere to carry its captured values. `List`, `Array`, `String`, and `Iter` higher-order methods accept `Func` instead, so they can use captured callbacks without a second API.

Each snapshot conversion executes once, in body first-use order, when the `Func` is constructed. A reference capture uses a shared typed cell allocated where the original binding is declared, preserving initializer order and single evaluation. Calling the `Func` allocates no capture storage.

A default capture of an outer reference parameter snapshots its current referent. `using &name` retains the caller's alias without another cell. Reference captures retain the source's qualifiers: capturing a `const` object by reference does not make it writable.

Nested lambdas capture from their enclosing lexical environment. Sharing an original variable requires `using &name` at every enclosing lambda. An inner reference clause cannot reach through an enclosing snapshot. A nested snapshot records the value when that inner lambda is constructed.

Capturing a pointer, `Array`, or `Map` copies its pointer value. The binding is read-only, but its pointed-to contents remain mutable; later rebinding the original variable does not retarget the capture. An aggregate without `Var` conversions cannot be captured by value; capture a pointer or explicitly capture the aggregate by reference.

Captured values and cells remain valid only as long as their owning `Scope`. Caller objects and pointer-backed contents keep their existing lifetimes; reference capture extends neither and adds no synchronization. Reused direct function handles have file-static lifetime.

Statements

C control flow

x2c accepts C-style `if`, `switch`, `while`, `do`, classic `for`, labels, `goto`, `return`, `break`, and `continue` statements.

A `switch` whose selector has static type `String`, or an alias of `String`, selects its case by content. Each case label of that switch is a string literal, including adjacent literals and a native macro that expands to one; any other label is an error. A label matches when `selector == "label"` holds under the ordinary `String` comparison, so `case ""` matches the empty `String`, which is null. The selector is evaluated once. Adjacent labels, `default` in any position, fallthrough, `break`, an enclosing loop's `continue`, nested switches, and labels inside nested blocks behave as in an integer switch. A `char *`, `const char *`, or `Var` selector keeps C's integer `switch`.

```
typedef String Color;

int classify(Color color) {
  switch (color) {
    case "red": return 1;
    case "green":
    case "blue": return 2;
    case "": return 3;
    default: return 0;
  }
}

int main(void) {
  printf("%d %d %d %d\n", classify("red"), classify("blue"), classify(NULL),
         classify("purple"));
  return 0;
}
```

```
1 2 3 0
```

System block decorators

The built-in macro module installs `$scope`, `$let`, and `$lock` without a per-source import. They retain the normal runtime declaration requirements and macro collision policy. None has a bare keyword alias.

`$scope()` retains one region around its following statement and defers release inside an inner block containing that statement. `$scope(pointer)` evaluates the Scope-pointer expression once, pushes that destination, and uses the same inner-block placement for a deferred pop. Pop restores the previous destination without destroying the selected Scope. More than one argument is an invocation error. A decorator around a loop creates one region; a decorator around its body creates one per iteration.

`$let(place, value)` captures the address of the place once, saves its value, registers restoration, assigns the new value once, and runs its body. The place must be addressable and assignable, and its storage must outlive the body. Restoration uses the captured address even when later changes would cause the original expression to name different storage.

`$lock(mutex)` evaluates a Mutex expression once, calls `lock`, then defers `unlock` around its body. A failed acquisition registers no unlock. These forms add ordinary blocks and defers and no hidden loops.

See Classes and System Macros for examples and expansions.

With

`with expression [as name] { ... }` gives an expression a short lexical name. The name defaults to `_`:

```
Point point = { 0 };
with point {
  _.x = 1;
  _.y = 2;
  _.z = 3;
}
```

The expression may be any x2c expression. Each use acts as a parenthesized copy of that expression. A call used twice is called twice, and an unused expression is not evaluated.

The name is available only in the required braced body and only in expression positions. It does not replace field names, labels, declaration names, type syntax, or literal atoms. A declaration of the same name shadows it from that declaration onward.

`with` nests lexically. An inner use of the same name shadows the outer one, and the outer meaning resumes after the inner block. Naming the outer expression keeps it available through an inner default shorthand:

```
with pair.left as left {
  with pair.right {
    _.x = left.x;
  }
}
```

At the beginning of a statement, `with` always introduces this construct. It remains a contextual identifier in declarations and other expression positions. A statement that calls a function named `with` uses the spelling `(with)(arguments);`.

Foreach

`foreach (declaration in expression) statement` evaluates the expression once and visits its elements. A comma may take the place of `in`. The declaration has no trailing semicolon and may bind one name or destructure a key and value:

```
foreach (int value in values) total += value;
foreach (int key in map.keys()) printf("%d\n", key);
foreach (Var (key, value) in map)
  printf("%ld=%ld\n", key.integer(), value.integer());
foreach (int value, values) total -= value;
```

Cursor-backed collections use their typed `try_next` member directly. Other values convert to `Iter`, and each yielded `Var` converts to the declared loop type when that conversion exists. Iteration covers the loop forms and the `Iter` protocol in tutorial order.

Foreach uses `Iter.try_next`. A boxed `Var` without a registered iterator adapter produces an already-exhausted iterator. A receiver with a statically known type and no visible exact or inherited `Iter` conformance is rejected as not iterable.

Match

`match (expression) { case pattern: statement ... default: statement }` evaluates its subject once and executes the first matching case. Pattern matching introduces the pattern language with worked examples.

Patterns use `List`-literal notation. `?` and `*` are anonymous wildcards. Named binders are `?IDENT` and `*IDENT`, where `IDENT` is `[A-Za-z_][A-Za-z0-9_]*`. Binder keys are canonical `Atoms`, so long and case-sensitive identifiers work in recursive, prepared, and compiled match paths. A sigil-leading `Atom` with any other suffix is a malformed pattern: recursive matching fails, `MatchPlan` reports `binder-name`, and compiled `match` syntax reports a positioned diagnostic.

`Match` operators and predicate names are compact `Symbols`. The `?binder?`, `*binder?`, and `!op?` spellings are reserved control vocabulary only in their `!is` predicate operand positions; they are not named binders. Nested `List` patterns, literal comparison, default selection, and the `!not`, `!or`, `!and`, `!set`, `!quote`, and `!is` forms are supported. `break` exits the `match`; `continue` targets an enclosing loop.

Runtime-built patterns may interpolate an interned `Match` operator in head position and retain the literal operator's semantics. Source `match` arms require literal operators so the compiler can prove binder availability. Named binders inside `!not` operands are not definitely assigned; binders under `!or` or membership-style `!set` must occur in every alternative; `!quote` is opaque. Arm binders are semantic `Var` or `List` locals and support method syntax.

`?(typename name)` declares a native typed capture. Its exact `Var` tag must match the tag tested by `value is typename`; mismatches fail the pattern without conversion. The type applies to every unquoted occurrence of that binder in the arm, including `?name` and alternative branches. Repeated names retain their equality constraint. The shorthand lowers to existing `!is` predicates and ordinary local declarations; explicit `!is` captures remain `Var` locals. The opener `?(` is adjacent; `? (String text)` is a wildcard followed by a sublist pattern.

Inside `?(`, `typename` is ordinary source type syntax rather than literal List data: write `?(String text)` or `?(List payload)`. The compiler stores named types such as `String` as Strings in type syntax. The type List `%("String")` therefore differs from `%(String)`, which contains an `Atom`; see macro values. Typedefs use their underlying `Var` tag, so `?(Type type)`, with `typedef List Type`, tests for a List payload without validating its contents as type syntax. The tag test precedes the native local declaration.

An arm may place `if (expression)` before its colon. The expression runs after matching, with captures visible, and uses ordinary truth conversion. False tries the next arm; errors propagate normally. A successful guard runs the body once.

The shipped component `src/components/pattern-match.x` lowers `match`. `$rewrite($matched)` registers it on the `match` statement head; a rule a program registers there runs first. A program's `switch` rule receives the statements in the arms. It does not receive the `switch` that the lowering writes.

Errors and cleanup

`try` requires a following filtered `catch`, `finally`, or both.

`raise %(CODE (KEY VALUE)...);` records one structured `Error`. Every `KEY` is a bare `Symbol` read by this syntax, and `CODE` is either a bare `Symbol` or `$name`, a variable converted to `Symbol`; each `VALUE` is one expression. Values are restricted recursively to `Null`/`nil`, numeric and enum values, `Symbols`, `Atoms`, `Strings`, and `Lists` of permitted values. `void`, pointers, mutable containers, resources, custom objects, and other identity-bearing values are invalid. A value whose own type is invalid is a compiler error; invalid contents of a `List` or `Var` value reach `Error`'s `<bad-types>` raw floor when the `raise` runs. The shipped component `src/components/raise.x` performs the conversions and the check. `$rewrite($raised)` registers on the `raise` statement head.

The compiler evaluates the converted code and values once each, left to right, and builds their literals when the `raise` runs. A `try` may have adjacent `catch` arms:

```
try block.reserve(n);
catch %(alloc-fail * (bytes ?count) *): return 0;
catch %(bad-arg *rest): report(rest);
catch: return -1;
```

The newest `Error` is matched as `%(CODE @DETAIL)`. Filtered arms use the same pattern and binder rules as `match`; `?name` declares a `Var`, `*name` declares a `List`, and the first matching arm runs. The code position takes a bare `Symbol`, a binder, or any single pattern, so `%((!or ?code bad-enc conv-range) *cause)` selects either code and binds it. A bare `catch:` is the optional default and must be last. Every filter expression is evaluated once when the `try` is entered.

Selecting a filtered arm consumes the errors accumulated since that arm was registered and transfers through each intervening cleanup frame. Its `finally` and `defer` cleanup runs before the selected arm. A `finally` body may not define a label, because its statements are repeated on each path that leaves the region. The transferring registration is removed before the arm executes, so raising a replacement `Error` continues outward instead of re-entering the same arm. When no arm matches, the `Error` continues outward unchanged. Catch bindings are borrowed through the selected arm and must be copied with `Error.snapshot` to outlive it.

Every cause in the shared table never returns to its raising call. `Error` keeps their policies at `<abort>` and observing handlers cannot consume them; only a matching filtered `catch` transfers control. For a literal `raise` of one of them, generated C places `__builtin_unreachable()` after the runtime `raise` call so C control flow has the same rule.

These are the shared causes. The policy of each is `<abort>`, and a process cannot change it:

Code	Cause
`<alloc-fail>`	allocation failed
`<size-limit>`	a size, length, or capacity is unrepresentable
`<invariant>`	an impossible internal state was reached
`<bad-state>`	a valid object is in the wrong lifecycle state
`<init-fail>`	process-owner initialization failed
`<bad-arg>`	an API precondition failed with no better code
`<bad-enc>`	a `Var` encoding is invalid
`<void-op>`	`void` was used as a dynamic value
`<bad-types>`	operand types do not support the operation
`<bad-target>`	a requested conversion target is invalid
`<no-convert>`	no requested conversion exists
`<conv-range>`	conversion exists but the value is out of range
`<bad-op>`	an operator is unsupported
`<div-zero>`	integer division or remainder by zero
`<bad-shift>`	a shift count is invalid
`<no-member>`	a dynamic value has no such member
`<bad-sig>`	a native or Lisp callable signature is invalid
`<bad-arity>`	a callable received the wrong argument count
`<no-symbol>`	a native symbol cannot be resolved
`<bad-result>`	a native result cannot be represented
`<not-call>`	a value cannot be called
`<call-stack>`	evaluation exceeds its call budget or stack depth
`<interrupt>`	the evaluation session has been interrupted
`<unbound>`	a program name has no binding

Code	Cause
`<malformed>`	external syntax or serialized input is invalid
`<incomplete>`	external input needs more data
`<format>`	formatting failed without output
`<not-found>`	an external resource does not exist
`<io-fail>`	another external I/O operation failed
`<cmd-fail>`	a command or pipeline exited with a nonzero status
`<join-fail>`	a worker transferred captured errors at join

Unknown codes also default to `<abort>`. A process may set a policy only for unknown and user-defined codes.

The `defer` statement schedules the statement for the current block exit. Multiple `defer` statements unwind in last-in, first-out order. Cleanup runs for normal scope exit, `return`, `goto`, and `Error` transfer to a selected `catch`, all the way out to the function boundary. It does not run when an uncaught `Error` aborts the program, or when native code calls `exit` or `longjmp`. `break` and `continue` unwind cleanups only up to the innermost enclosing loop or `switch` (whichever the jump targets) and stop there; a `defer` or cleanup frame registered outside that boundary is left for a later normal exit, `return`, or `Error` transfer to reach. The errors and cleanup chapter shows how the two are combined.

A `return` expression is evaluated and saved before its cleanups run; the saved result is returned afterward. A `goto` within the same cleanup ancestry runs no cleanup. An outward `goto` runs every cleanup region it exits. A `goto` into a protected cleanup region, or into a sibling protected region, is rejected at compile time, and so is a `case` or `default` label that would enter a region its `switch` body opened before it.

Generated `Error` transfer preserves the automatic locals and parameters a protected body modifies under the repository's optimized build, whether the body assigns them by name or writes through a pointer it holds the address in. The compiler supplies the required volatile C representation for those values and for the transfer state its frames carry; source code does not need optimization-specific qualifiers for ordinary assignments in `try`, `catch`, or `finally`. `Error` transfer does not restore the process signal mask; code that changes a signal mask must restore it.

Type-owned initialization

A translation unit may define one top-level initializer with the exact signature `void TYPE.initialize(void)`, where `TYPE` is a typedef name. The compiler calls it lazily at every non-static function boundary and guards it so direct or recursive calls still execute its body once. Static functions are internal helpers. They trust a non-static entry point or the initializer that called them and do not repeat the generated guard.

Compiler-owned literal and static-runtime initialization runs after the guard is set and before the method body. This makes calls from the initializer back into static helpers in the same translation unit safe. Translation units that need only literal caching retain a private synthetic initializer.

When the initializer is defined only inside conditional groups, the non-static functions call a private synthetic initializer instead. It repeats the conditions around the initializer's definition to call the initializer where it is compiled, and otherwise runs the compiler-owned initialization itself. It sits before the unit's first function, so a `#define` or `#undef` between that function and the initializer's definition that changes those conditions makes the two disagree. The initializer's body then never runs, or the generated C calls an initializer that was compiled out.

Other initialization statements and runtime-valued static assignments belong in the method body. Eligible file-static percent literals use the generated sequence described under Values and literals. Top-level decorators are rejected.

Type-owned shutdown

A translation unit may likewise define one top-level shutdown function with the exact signature `void TYPE.shutdown(void)`. The compiler registers it once through `Scope.shutdown_hook` when that unit initializes. When the unit also defines `TYPE.initialize`, that guarded function registers it after its complete generated initialization sequence; otherwise the unit's private synthetic initializer registers it.

Shutdown hooks run in reverse registration order. A type initializer that acquires process-lifetime resources therefore registers its matching shutdown only after acquisition succeeds. Calling `TYPE.shutdown` directly still passes through the unit's initialization guard.

Types and conversions

x2c accepts normal C type forms plus runtime types such as `Var`, `List`, `Array`, `Map`, `String`, `Symbol`, and `Iter`.

Typedef declarations are supported at file scope and inside compound statements. Local aliases follow lexical scope and shadowing. Their uses are resolved while that scope is available, preserving the existing meaning of file-scope types, including their methods and converters. Local typedefs stay inside their C blocks and are not exported. A locally defined aggregate also stays in its block; it cannot supply a type to a lambda helper lifted outside that block.

Direct and chained file-scope aliases of `Var` behave like `Var` in boxing, extraction, initialization, assignment, function arguments, returns, and comparisons. Declarations and signatures retain the source alias. This rule is specific to `Var`; other file-scope typedefs gain only the methods and converters declared by their own ancestry.

A typedef and the type it names may substitute for each other in either direction. An `Array` can therefore be used as a `Block`, and an `Ast` as a `List`, without conversion. Two typedefs of a common type cannot substitute for each other this way: they may describe different contents despite sharing a C representation. Thus `ArrayInt counts = someArrayDbl;` is a type error whose message gives the required sibling converter, `ArrayDbl.arrayint`. The existing `Array.arrayint` instead converts from the common Array ancestor. A cast still permits reinterpretation.

The compiler warns with code `conversion` when source spells a conversion that its destination already performs. Translation continues.

A cast is reported when its operand already has the cast type, qualifiers included. The comparison uses the declared x2c type, so casts between a typedef and its underlying type, or between two typedefs of one C type, are not reported. Neither are `(void)` casts, or casts of an operand whose C type x2c does not track, such as a pointer difference or a character constant. A cast of a C string literal is not reported. The cast keeps the literal native where x2c would otherwise promote it to a `String`.

A converter call is reported when its destination would make the same call. A converter is a method that takes only its receiver and is named for its result, with `str` for `String`. The destinations are the value of an initializer, an assignment, a `return`, a declared call argument, an interpolation hole, and a `Var` value that a printf-family format consumes. So `String name = sym.str();`, `f(x.var())`, and `printf("%s", v.str())` are reported. A printf-family value is a destination only where the format is a literal the compiler reads; a computed format, such as a `const char *` variable or a name an object macro defines, converts nothing. The destination makes the same call when either side is `Var`, both types share one C type, or the receiver declares the converter. A method receiver, an operator operand, and a parameter with a qualifier the result lacks, such as `const char *`, are not destinations.

These explicit calls differ from the implicit crossing and are not reported:

- `.str()` on a `Var` at a typed destination, and any other converter call on a `Var` in an interpolation hole or a printf-family value. Holes and formats render a `Var` through `Var.str`, which displays any value. The implicit crossing to `String` displays any value but `void` and `Null`, which it reads as NULL, while `.string()` reads the String payload and raises `<no-convert>` for another tag.
- An explicit reader such as `.list()` or `.map()` at a destination of its type. The implicit crossing reads the payload or NULL through `Var.list_payload` and its relatives, while the explicit reader raises `<no-convert>` for another tag.
- A `Var` read into a numeric type other than `Symbol`, such as `v.int()` or `v.integer()`. The implicit crossing calls `Var.convert` and then reads the result. `Var.int` already converts, and `Var.integer` returns `0` where `Var.convert` raises.
- A reader other than `str` into `String` on another type, such as `File.string`. The implicit crossing calls the type's `str` converter.
- A call bound from macro-constructed syntax, and a call inside the function that implements the crossing, such as `value.list()` inside `Var.row` for a `List` typedef `Row`.

Supported primitive C specifiers are order-independent and normalize to one compiler spelling. For example, `double long` becomes `long double`, and `long unsigned long` becomes `unsigned long long`. Invalid combinations such as `long float` are structured type errors at the original source token. Numeric literal radix, magnitude, and suffix determine the annotated native family; integer promotions and the usual arithmetic conversions then operate on that canonical identity. A literal beyond the supported native integer families is a structured type error.

Conversions remain context-sensitive outside the numeric domain. A source expression may be boxed to `Var`, converted through an established target conversion, or rejected. Scalar typedefs retain their declared annotation but use the underlying scalar family for arithmetic and conversion to or from `Var`.

A converter is the source method named for its target: `Source.target` for a plain target and `Source.str` for `String`. Lookup uses the source type's exact converter when it declares one; otherwise it walks the source's typedef chain and uses the nearest converter declaration. Thus a `typedef List Row` may be assigned or interpolated as a `String` through `List.str` without a forwarding `Row.str`. The generated call still names `List_str`, and direct C calls retain the converter's declared ABI. Lookup stops without a call when the target itself occurs in the chain, since typedef identity already handles that crossing.

Type qualifiers take part in that decision. `const`, `volatile`, and `restrict` are retained on the declared type of a function result, a global object, a record field, and a local, including one imported from a C header. Handing such a value to a target that is the same type without one of those qualifiers is a structured type error, because that conversion passes the same address on unchanged. A conversion that copies through a converter is unaffected: `const char *` still becomes a `String` through `String_new`. A qualifier on the value being copied, as opposed to what it points at, is not part of the comparison, so `char *const` converts to `char *`.

A pointer to `void` is held to the same rule even though its base type differs from the source's, because it also passes the same address on unchanged. `void *` therefore does not accept a `const char *`; `const void *` does.

Host preprocessing

When preprocessing is required, x2c passes the source pathname and every include directory to the host `cc` as separate argv elements. x2c does not interpret spaces or shell metacharacters in these paths. The host process receives the source path, so quoted includes and its diagnostics retain source-file context.

The active preprocessed stream is shallow-parsed only to establish the global environment. Full parsing, diagnostics, and emitted source still come from the original token stream. Directives remain AST nodes in source order at top level and inside compound statements. This includes a trailing directive before `}`; a directive between `if (...)`, `else`, `while (...)`, `for (...)`, `do`, `switch (...)`, `defer`, `try`, `catch ...:`, `finally`, a match arm's `:`, or a statement macro such as `foreach` and the statement that follows it; and a directive before the `else`, do-while `while`, `catch`, or `finally` that continues a statement. A conditional group opened there also takes the statement of each later arm and the closing directive, so `foreach` keeps a group of one statement per arm inside its loop. A later statement in the same arm follows the governed statement, as in C. Directives between match arms stay between those arms, so an arm inside a conditional group is present exactly when C compiles the branch that holds it.

An include that x2c cannot resolve remains in the emitted C. Translation without host preprocessing can therefore succeed with an active missing header; native compilation rejects it. Explicit host preprocessing also rejects an active missing header. A missing include in an inactive branch such as `#if 0` does not prevent compilation.

x2c does not fully preprocess its source. The original tokens must still form syntax that x2c can parse. Macro expansion that supplies grammar, or inactive branches containing otherwise unparseable source, can require adjustment even when the host C compiler accepts them.

A C source file renamed from `.c` to `.x` is an x2c translation unit. A C header stays a `.h` file reached with `#include`; x2c collects its declarations and the native compiler reads it from the generated C.

Every branch of a conditional is parsed except the branches C never compiles. x2c output is compiled as C by a GNU-style compiler. Therefore x2c skips these branches:

- the branch under `#ifdef __cplusplus`, or under `#if defined(__cplusplus)` alone or first in a `&&` conjunction with no `||`;
- the same forms for `_MSC_VER`;
- the `#else` branch of `#ifndef __cplusplus` or `#if !defined(__cplusplus)`;
- `#if 0`.

Their tokens are trivia. The directives around them stay in place and are emitted. A compiler built for a host other than Windows treats `_WIN32`, `_WIN64`, and `__CYGWIN__` the same way. An `#include` in a skipped branch is not read. A function defined in two arms of one conditional is one definition. At file scope, `extern "C" {` opens a linkage group and a `}` closes it, even with an `#include` between them; the group's declarations are file-scope declarations, and its braces are not emitted. Any string literal is accepted as the linkage name. `extern "C"` before a single declaration is read as `extern`.

A unit or a collected header may place a macro before or among a declaration's specifiers. The macro's definition determines what the name contributes: nothing for an empty body or an attribute (`#define RLAPI`, `#define WEAK __attribute__((weak))`), its storage class for `static`, `extern`, or `inline` (`#define JSMN_API static`), builtin type words for a body such as `signed int`, and, for a body that is the name of another prefix macro, that macro's reading. A function-like macro whose body is its parameter amid such prefixes wraps a type: in `CJSON_PUBLIC(const char *) f(void);` the type is the one inside the parentheses. A macro whose words are a storage class reads as that storage after the type as well, as in `int LOCAL f(void)`. Where a name is defined differently in several conditional branches, a `static` definition takes precedence, then any prefix without a qualifier, then a prefix with one, such as zlib's `z_const`, which is `const` in one branch and nothing in another. A name defined to any other text is a typedef name. In a collected header, a name with no visible definition directly before a builtin type word, qualifier, storage class, or `inline`, as in `API int f(void);` with `API` defined on the C compile line, contributes nothing.

A GNU attribute or a function-like attribute macro after a declarator or parameter, `int a __attribute__((unused)) = 1, b = 2;` or `int f(int x __attribute__((unused)));`, is kept as text after that declarator and applies to it alone, as in C. An attribute on the prototype of a function the unit defines is written before the type on both the prototype and the definition x2c generates. Attributes are not part of the signature compared between a prototype and its definition. A definition without `static` after a `static` prototype keeps the prototype's internal linkage. `__inline`, `__inline__`, `__restrict`, and `__restrict__` are the standard keywords. A collected header may also place attributes after `struct`, `union`, or `enum` and after an aggregate's closing brace, as in `struct S { char c; int i; } __attribute__((packed));`. Collection reads past them, and C applies them. A unit does not accept these two positions.

`in` and `match` are keywords only where their x2c forms can occur: `in` between two operands, and `match` as `match (...)` followed by `case` or `{`. Elsewhere both are identifiers. A field of a foreign struct whose type x2c cannot resolve can be indexed; the C compiler alone types the result.

A public `struct tag { ... } name;` publishes the tag body and `extern struct tag name;` in the header and defines `name` in the source. A public prototype that uses `struct tag` is preceded in the header by `struct tag;` when the header has not declared the tag. A function body that follows the declarations keeps the conditional arm it was written in. It is emitted before a later `#undef` and before any conditional group that contains one, so the macro definitions in effect where it was written apply to it.

A failed host preprocess prints its captured stderr, reports a structured x2c driver diagnostic at the first directive, and exits nonzero. Partial host stdout is never parsed after failure. `--no-cpp` bypasses this discovery pass; `--dump-cpp-text` exposes the successful host output.

Diagnostics

Command-line options, including the dump flags that expose an individual phase and the include and output directory switches, are listed in compiler options.

Translation stops a unit after 20 errors by default. Compiler diagnostics describes `--max-errors` and the line that reports the stop. Parsing recovers at top-level declarations. A rejected declaration is skipped whole and parsing resumes at the next one, so independent errors in separate declarations are reported together in source order, and one declaration contributes at most one error. Tokenization, symbol collection, transformation, generation,

and emission stop at their first error. Warnings do not count toward the bound, and a report identical to an earlier one is not repeated. `--diagnostics-file` writes the same diagnostics as JSON Lines; the command-line reference describes the fields.

Locations retain `file`, one-based `line` and `column`, token `length`, and absolute byte `position`. The source renderer uses `length` for the caret width.

Region warnings

Three codes come from the region check described in Scopes and Lifetime. Each is a warning: translation continues and the program still compiles.

Code	Reported for
`region`	A value allocated inside a region is reachable after the region ends. The message gives the way the value leaves, and the note gives the line that opened the region.
`after-free`	A local is read after a call freed it, such as `Scope.free`, `Scope.realloc`, `Array.free`, `Array.list_free`, or a `cleanup`.
`bad-free`	`Scope.free` or `Scope.realloc` is given a literal, a local's address, or other storage no Scope allocator returned.

The check analyzes regions lexically and summarizes each function within its own unit, so a unit gets the same warnings whatever else is translated with it. For calls into another unit, the check uses only a fixed table of runtime operations. It does not cover storage from plain `malloc` or a C library, raw pointer arithmetic, values reached through a field of a stack `struct`, callbacks, or `Context` regions.

Syntax specifications

x2c's syntax is specified here as a token grammar plus lexical transition rules that describe the current parser. The first extraction matched revision `48715bde` on `dev`. Parser changes update these pages and `etc/syntax/` by hand. `tools/syntax-spec check` verifies grammar references and reports generated files that differ from their data. It does not compare the data with the parser, and no gate runs it. The language reference supplies the meaning and constraints of the forms.

- Source grammar: EBNF, precedence, contextual decisions, and macro extension contracts. The EBNF file is the same grammar included in that chapter, generated from `etc/syntax/grammar.x`.
- Lexical specification: byte classes, ordered recognition, numeric and escape rules, nested modes, token reclassification, layout, and failure behavior.
- Syntax as data: ordinary x2c Lists, their algebra, and a tool for generation, reference checks, dependency graphs, and lexical mode graphs.
- x2c syntax map: the additions beyond C11, grouped for onboarding and linked to the specification data.

The grammar describes source after tokenization and indentation rewriting. It is not a claim that a context-free recognizer can decide whether an x2c program is valid. Names, types, macro signatures, and preprocessor state participate in parsing. The grammar marks these dependencies as external recognizers.

Catalog of existing material

This catalog lists the existing sources of syntax rules. Paths below are repository-relative. The table gives function names because line numbers change.

Material	Location	What it specifies	Limits
Language reference	`docs/src/reference/language.md`	User-facing forms, examples, semantic constraints; a small protocol grammar under Protocols	No complete grammar or lexical rule inventory
Syntax guide	`docs/src/guide/{prod-c,indentation,macros,match,collections,symbols,protocols}.md`	Explains, examples, and paths of individual forms	Examples do not define all accepted token sequences
Declaration parser	`src/parse/parse.x`	Recursive-descent rules for units, imports, types, declarations, declarators, parameters, fields, enumerators	Mixes recognition with name binding, collection, and semantic work
Expression parser and precedence ledger	`src/expressions/expressions.x`, `src/operator-ledger.x`	Expression productions, ambiguity decisions, complete binary operator precedence rows	Depends on type and macro state
Statement parser	`src/statements.x`	Statements, blocks, match/catch arms, directive-aware governed statements	Macro-produced statements also enter binding
Literal and lambda parsers	`src/literals/literals.x`, `src/lambdas.x`	Quoted data, interpolation, patterns, lambdas and captures	Token modes determine which spellings reach them
Macro parser and built-ins	`src/macros/macros.x`, `etc/builtin-macros.x`	Category table, definitions, invocations, quotations, typed holes, keyword aliases; built-in `class` and `foreach`	Grammar is extended by visible definitions; libraries such as `lib/system-macros.x` add more
Protocol parser	`src/protocol/protocol.x`	Protocol definitions, associated types, members, adoptions	Witness resolution and type constraints are separate
AST source patterns	`src/grammar.x`	Executable source-shaped macro patterns and AST accessors used by lowering	Despite the filename, not a complete source grammar
Character scanners	`lib/scan.x`	Allocation-free byte recognizers, keywords, longest operator	A scanner in isolation does not specify tokenizer dispatch or context

Material	Location	What it specifies	Limits
		prefix, literal escapes, failure statuses	
Tokenizer and layout	`lib/tokenizer.x`	Ordered scanning, mode stack, operand context, keyword reclassification, indentation rewrite, positions	Contextual and stateful, not a single token regex table
Preprocessor handling	`src/preprocess.x`, `src/frontend.x`	Directive handling, conditional groups, configured units and host-preprocessed imports	Host C preprocessing has its own language and target configuration
Compiler token preparation	`src/compiler/parsing/tokens.x` (`Compiler.tokenize`, `_retag_keywords`)	Phase ordering and contextual `in`/`match` token types	Runs after raw tokenization and conditional scanning
Lisp reader	`lib/lisp/lisp.x` (`LispReader`)	Lists, reader prefixes and atomic forms over Lisp-mode tokens	Evaluation and compile-time operations are not source grammar
Generated API pages	`docs/src/internals/compiler-api/`, notably `parse.md` and `grammar.md`	Public signatures and source doc comments	Derived API documentation, not an independent grammar
Architecture and source map	`docs/src/internals/{architecture,implementation-map}.md`	Phase ordering and syntax owners	Describes organization, not acceptance rules
Scanner/tokenizer tests	`unittest/test-scan.x`, `unittest/test-tokenizer.x`	Executable recognition, boundary, mode and layout cases	Finite observations, not a formal definition
Compiler fixtures	`unittest/compiler-fixtures/`	Token dumps, AST/translation outputs, diagnostics, successful and rejected programs	Expectations cover selected cases, not all derivations
Editor grammars	`etc/vsc-extension/syntaxes/{x2c,x2c-lisp}.tmLanguage.json`	TextMate regular expressions and highlighting contexts	Approximate; cannot decide acceptance or reproduce binding
Editor grammar tests	`etc/vsc-extension/test/current-grammar.test.x`	Expected highlighting scopes	Contains syntax-shaped samples that are not accepted programs; not an acceptance corpus
Bootstrap copies	`bootstrap/lib/scan.{c,h}`, `bootstrap/lib/tokenizer.{c,h}`	Generated C implementation of lexical rules	Derived copies, not independent owners
Executable examples	`examples/manifest.txt` and its programs	Working uses of the language with recorded checks	Demonstrations, not exhaustive syntax rules

Status and boundaries

The extraction has three layers:

1. Byte recognition and token transitions in the lexical specification.
2. Token productions in `etc/syntax/grammar.x`, rendered as `x2c.ebnf`, with explicit external recognizers.
3. Context and semantic constraints in the grammar chapter and language reference.

This specification is written and reviewed by hand. No parser is generated from it, and nothing proves it equal to the compiler. Named external recognizers cover the macro syntax, which definitions extend, and host-specific syntax. The lexical chapter identifies observed implementation edges separately from the supported spelling described by the book.

C translation constraints, preprocessing expressions, type checking, ownership, overload/protocol resolution, macro evaluation, AST lowering, and emitted C are outside the source grammar. The host compiler still checks the generated C. Resource limits and diagnostic wording are not productions.

Source grammar

The EBNF file, generated from ordinary List data, describes x2c's current source forms. It consumes tokens after the lexical transformations. A parser generator cannot use it directly, because some rules depend on context. The language reference defines semantics and restrictions.

Notation and entry points

`A = B ;` defines a production. `,` concatenates, `|` selects alternatives, `[ ... ]` is optional, `{ ... }` repeats zero or more times, and parentheses group. Double-quoted strings denote terminal token spellings. Backslash escapes a quotation mark, apostrophe, or backslash inside a terminal string. `(* ... *)` is a comment. `? description ?` is an external recognizer, whose contract is stated below or in the lexical specification.

A quoted keyword can mean a token kind or a contextual identifier spelling. For example, `is` is initially an identifier; `inline` includes normalized GNU spellings. Productions do not accept a token of an incompatible mode merely because its text matches a terminal. Trivia can separate code tokens unless the lexical specification requires adjacent bytes for one token.

Alternatives are not a PEG priority list. The contextual rules below select ambiguous alternatives before semantic checks. Repetition in the binary expression ladder folds left; assignment and conditional tails recurse right. The grammar permits some syntactic structures whose types or placement later fail. It does not enumerate invalid programs by duplicating the type system.

The ordinary entry is `translation-unit`. `source-unit` also names script input. After frontend shebang handling, a script with an explicit `main` uses ordinary file scope. A script without `main` retains file-level functions, types, imports, macros, and explicit static declarations, while its executable block items become the implicit script body. This partition uses `Compiler.script_statement_starts`, `_declaration_stays`, and `defines_main`, defined in `src/parse/units.x`. File-level functions cannot refer to script locals.

Module blocks are expanded by `src/collect/collect.x` before `translation-unit` parsing. `module-block` is spliced before parsing and is not a `unit-item`. It contains quoted member paths, separated by commas, with an optional trailing comma. Members are inserted in order from the containing file's companion directory, `NAME/` beside `NAME.x`; each path must resolve inside it. Nested module blocks are expanded too. Each file permits one file-scope block and must use default symbol collection. CPP and live-symbol modes do not support member splicing. The `module` storage class is described in D4.

Declaration contracts

D1. Names and declaration starts

`Compiler.test_declaration` uses storage/type keywords, native prefix macros, typedef bindings, package aliases, and template-hole kinds. A known non-typedef object name does not start a declaration. An unknown identifier can start one when the following token looks like a declarator. Thus the grammar does not assume a lexer token named `TYPE_NAME`. Import aliases fold through the current symbol table; local bindings can shadow them. An imported package string must decode to an identifier, and each imported member must exist.

`identifier` normally consumes `ident`. Member selection, method names, and declarator names also accept identifier-shaped keyword text where their parser explicitly does so. In `macro-name`, the component after `$` must be an `ident`; each component after a `.` may also be keyword text, so `$a.if` and `$a.while.for` are macro names. `method-owner` is the owner recognized by `_complex_identifier`: a built-in type word or known typedef, including package-qualified type names. `Self` is resolved as a contextual type. Type existence and a valid scalar combination remain semantic checks. The scalar-word production does not include `_Bool`: the current parser handles names outside that list through its named-type path.

D2. Declaration commas and parentheses

In a declaration row, a comma starts a new group only when `_group_comma` recognizes a declaration start with a following declarator. Otherwise it joins declarators of the current type. For example, `int i, T;` declares two integers, whereas `int i, T value;` can introduce a new group. Within parentheses, `_parameters_follow` distinguishes an abstract function declarator from a parenthesized name using the current typedef environment.

A declarator can have no direct name: this represents an abstract declarator, an unnamed bit-field, or a type-only declaration, as allowed by its context. Function declarator parameter lists are nonempty: `(void)` is the no-argument form. Calls and lambdas separately permit `()`. Array dimensions use an expression; bit-field widths use a primary expression, which can be grouped. A full `type-operand` can contain abstract declarators; `type-name` only contains qualifiers, a specifier, and pointer/reference modifiers. These are different entry points, notably in casts versus `_Generic` associations.

D3. Native declarations

The storage parser permits one ordinary storage class and one `threaded`, with `inline` and recognized attributes. `typedef` has its own declaration path and cannot appear as trailing storage. Known native macros can supply storage, qualifiers, types, annotation invocations, or a wrapper around a type. Shallow C-header collection can ignore an unseen prefix before a declaration keyword. `prefix-macro` stands for that operation. It accepts only recorded native macros.

`attribute` retains the balanced source text of `__attribute__(...)` or a known annotation macro. Placement follows `_attribute`, `_storage_class`, and `_declarator_suffix`; tag attributes handled by `_skip_attributes` belong to shallow imported-header collection. The EBNF shows their structural positions, not a grammar for GNU attribute arguments. Reference placement, bit-field legality, storage compatibility, and initializer conversion are outside these productions.

Ordinary aggregate bodies enter the field parser even when the next token is `}`; the grammar therefore requires a field item. Named-type record bodies explicitly permit an empty body. A protocol definition's participant is a binder; an adoption's participant is a type. Protocol members must be single function declarations owned by that binder. Associated types precede members; `as` and `tag` are mutually exclusive Var-adoption modifiers.

D4. Interface modifiers

`private-interface` is the top-level visibility dispatch before declaration parsing. A leading `static` on a typedef, protocol body, class, or top-level Lisp form also controls source visibility. `meta native` applies to function interfaces. Protocol adoptions permit `meta` followed by optional `static`, or `static` alone; protocol bodies reject `meta`. A file-scope `module-declaration` is the word `module` followed by an identifier, a type keyword, a qualifier, `inline`, or `typedef`. It parses as the same declaration with `static` in place of `module`. See the reference for the public/private and compile-time rules. A visible binding named `module` disables the `module-declaration` form.

Expression contracts

E1. Casts, selectors, and braces

A parenthesized declaration start selects a cast only when the token after `)` can start its operand. `_cast_operand_follows` recognizes primary starts, `sizeof`, prefix update, and unary operators. Otherwise parentheses resume a grouped expression. At statement start, named parameters in the parentheses select a typed destructuring declaration; one anonymous parameter can instead be a cast.

After `sizeof`, a parenthesized declaration start is a type operand that ends at its `)`, unless `{` follows the `)`. Then the operand is a compound literal with its postfix continuation: `sizeof (int[]){1, 2, 3}` measures the array, and `sizeof (P){1, 2}.y` measures the member. Any other operand is a `unary` expression, so a parenthesized expression keeps its postfix continuation: `sizeof(a)[0]` measures an element.

`is [not]` shares relational precedence. Its right operand is a type when `_is_type_selector_start` recognizes a declaration, `Void`, or an unbound identifier; otherwise it parses one cast expression for the Symbol selector. An explicit pointer type requires parentheses. Arrays and function types are not accepted as direct `is` type selectors.

A brace in expression position is a Map when `_brace_starts_map` sees an Entry macro or the first non-conditional colon before a top-level comma or terminator. Otherwise it is an initializer. Consequently `{}` initially parses as an empty composite; its destination can establish Map meaning. The productions alone do not select `map` for every brace expression.

Immediately after `(`, a leading brace with a top-level semicolon selects a statement expression. A template hole can also select it when there is no top-level comma. Otherwise the brace remains data. Bare statements in parentheses without braces do not form a statement expression.

E2. Precedence

From lowest to highest:

Level	Operators	Association
Comma	`,`	Left
Assignment	`= += -= *= /= %= <<= >>= &= ^= |=`	Right
Conditional	`? :`	Right
Binary 1	`||`	Left
Binary 2	`&&`	Left
Binary 3	`|`	Left
Binary 4	`^`	Left
Binary 5	`&`	Left
Binary 6	`== != === !==`	Left
Binary 7	`< <= > >= in is` (`is not` included)	Left
Binary 8	`<< >>`	Left
Binary 9	`+ -`	Left
Binary 10	`* / %`	Left
Cast/unary	Cast, prefix update, address, dereference, sign, `~ ! sizeof`	Prefix nesting
Postfix	Call, index, slice, member selection, postfix update	Left

`src/operator-ledger.x` defines the binary rows except the contextual `is` handler in `src/expressions/expressions.x`. Assignment parses a conditional left side; whether that side is assignable is a semantic question. An expression-bodied function uses a full expression, while a lambda expression body uses an assignment expression. Calls and literal element lists permit a trailing comma. Meta calls such as `$name(a, b,)` and macro invocations reject it.

Form	Operand or target
`in`	Shift expression at relational precedence
`is [not]`	Contextual type selector or one cast expression
Lambda => body	Assignment expression, or raw braced block
Function => body	Full comma expression
Expression macro body	Full comma expression
Expression decorator target	One cast expression

E3. Special primaries

`_Generic`, `va_arg`, and `offsetof` are recognized by identifier spelling in `_parse_ident_primary`. Adjacent C strings join. A following identifier joins that sequence when it is a recorded native string macro or is unbound, except `in`; a bound runtime name does not. This is the external `native-string-word` recognizer.

Statement, directive, and pattern contracts

G1. Positions and directives

A block accepts declarations and statements; a governed position requires one statement. A macro expansion yielding multiple block items must respect that difference. `parse_governed` retains preprocessor groups while parsing one governed statement per applicable arm. `src/preprocess.x` selects conditional groups; this grammar treats a retained directive as one token.

`else` belongs to the nearest eligible `if`. Match defaults and unfiltered catches must be last in their applicable arm sequence. A label, `case`, or `default` is a parsed row by itself. `do ... while (...)` does not consume a semicolon; a conventional following `;` is an empty statement. This matters when comparing parser consumption, even though emitted C uses its required do-while punctuation. `case` consumes a full expression, including a comma expression, but not a GNU `...` range. The statement parser also accepts a `%{` token as a compound-statement opener; its contents still arrive from Map lexical mode, so it is not a general substitute for a code-mode `{`.

P1. Patterns and error payloads

A match pattern yields a List pattern, which can be static or supplied at runtime. Besides a List expression, a visible macro can supply a derived pattern. Its arguments are `?`/`*` binders with optional identifiers, nested macro patterns, or List patterns. `${$macro(...)}` embeds a derived pattern inside a List; in an `(expr TYPE CONTENT)` pattern its position can request only the content. Typed captures use `?(Type name)` and the type's Var tag. `src/literals/literals.x` and `src/macros/macros.x` implement these rules. `lib/match/match.x` defines what a pattern means.

Raise and catch payloads have narrower rules than an arbitrary List. Raise detail keys are bare exact Symbols; each detail contains one value. Catch details are a `*`-prefixed pattern or `(key pattern)` pair. A pair's value/pattern cannot be an outer splice, and a raise/catch code is required. A catch code may instead be a nested List pattern such as `(!or io eof)`. Raise causes and bare catch causes and detail keys must round-trip through the compact Symbol representation. Template slots can produce the corresponding expression or argument rows; they do not create a separate unchecked payload grammar.

Literal and Lisp contracts

L1. Tokens versus bytes

In data mode, bare `[`, `{`, and `"` emit the same opener kinds as code `%[`, `%{`, and `%"`. The EBNF uses these normalized kinds for nested data. Bare `(` remains the nested-list opener. List elements are separated by token boundaries. A comma there is a reader prefix. Array/Map entries use commas. Arrays and Maps accept value insertion, while List `@` additionally splices. An inserted value expression uses code mode. Literal meaning, caching, Symbol limits, and pattern binders remain subject to the reference's rules.

A percent List containing just one reader-prefixed form returns that form rather than adding another surrounding list. Quoted Map entries can contain `${...}` with either a computed key followed by `:` or an Entry macro/slot. A String interpolation uses `$name` or `${expression}`; the latter also supports macro sequence handling where the template parser permits it.

`lisp-escape` records the contents inside `$(`, whose outer parentheses also form the compile-time Lisp call/list. Templates use `@(form)` to splice a Lisp result at a sequence slot. The ordinary Lisp reader supports lists, four reader prefixes, and atomic values; it has no special dotted-pair production. Compiler templates add hole capture before evaluation. The EBNF covers reading; it does not specify Lisp evaluation, imports, or the available compile-time operations.

Macro extension contracts

The macro mechanism makes a fixed enumeration of every source spelling impossible. These contracts specify how visible definitions extend the productions, without treating extension contents as unrestricted text.

M1. Arguments and positions

Global macro definition names and keyword alias targets require `$`; `@` cannot replace that prefix. Dotted macro names permit keyword-shaped components after the first component, such as `$error.arg.void`. A scalar macro expression uses `$name(...)`, a local macro name, or a keyword alias; `@name(...)` is not a scalar expression.

Look up the macro or keyword alias in the active compiler state, then parse each argument using its signature category. Invocation arguments reject a trailing comma. The first `Param` argument opens one scope shared by the remaining arguments. An untyped hole initially reads an assignment expression. A final sequence parameter repeats its category with commas; it can capture zero elements. After a `Decl` argument, `in` may replace the separating comma, which is how `foreach (T value in collection)` uses the ordinary macro parser.

Hole kind	Argument recognizer
`Expr`, `Expression`, untyped	Assignment expression
`Type`	Type name
`NamedType`	Named-type declaration, including its semicolon
`Decl`	One non-function, non-typedef declaration; destructuring can omit its initializer
`DeclaratorRow`	One declarator with optional initializer; base type supplied at expansion
`Param`	Parameter declaration or ellipsis
`Name`	Identifier spelling
`Literal`	Atomic literal
`Stmt`	Block item
`Field`	Field item
`Entry`	Map entry
`Enumerator`	Enumerator
`MatchRow`	One match arm
`Unit`	Top-level item
`Function`	Function definition
`Catch`, `Captures`	Structured template sequence positions; not generic invocation arguments

Result/argument kind spellings are case-insensitive; EBNF lists their usual capitalization. `macro_categories` in `src/macros/macros.x` is the complete table. Expression, statement, field, entry, enumerator, unit, declaration, and decorator results must be applied in compatible positions. Local macro values and keyword aliases use their bound signatures too. `class` and `foreach` are built-in aliases, whose definitions live in `etc/builtin-macros.x`. The remaining built-ins and library macros need no new core productions. A bare visible `$name` can also denote a Macro value without invoking it. A direct non-decorator invocation consumes `;` in unit, block, statement, and field positions. Enumerator and Entry invocations use their enclosing comma list instead. Anonymous open arrow statements omit that terminator.

M2. Definitions and bodies

A global definition uses a `$` name; a named local definition omits `$`; an anonymous macro expression omits the name. An Expression result, or expression-target Decorator, requires an arrow expression body. A Stmt result uses a braced block-item template or an arrow statement (a Stmt macro/decorator invocation or an expression statement). Other results use a braced category-specific sequence, optionally preceded by `=>`. A named expression body normally ends with `;`; an anonymous arrow body omits that terminator, which belongs to the enclosing expression. The implementation also accepts a legacy parenthesized expression body after the arrow without its own semicolon when the next token cannot extend that expression. A following semicolon selects the canonical form. See `_legacy_expression_body` and `_expression_body`.

A braced template can begin with `using` lines terminated by semicolons. `using $name` declares fresh names; `using name` retains a file-level binding. `body-using` describes those leading lines. A signature can also have a `using` clause. Sequence parameters must be last. Local definitions cannot produce Unit/Declaration results or decorate Function/Unit/NamedType targets. These restrictions are enforced by `Definition` in `src/macros/macros.x`.

M3. Decorators

The first signature parameter is the target; invocation arguments supply the remaining parameters and the following source supplies the target category. Targets can be Expr, Stmt, Field, Unit, Function, or NamedType. An identifier alias omits parentheses when there are no ordinary arguments; a `$` invocation still has its argument parentheses. Unit/Function/NamedType decorators therefore extend file scope as well as block/expression positions. Indentation syntax removes the leading `@` of `@$decorator(...)` before parsing; see the lexical specification. Brace-source decorators compose directly before their target. A decorator invocation consumes its following target and must not end with a semicolon. At file scope, a `Stmt` decorator applied to a function definition captures

and decorates the function body. This adaptation does not permit arbitrary Unit targets for `Stmt` decorators. A naked file-scope `@` is rejected. Target placement and alias shadowing follow `try_parse_macro_target_at`, .

M4. Quotations

`$!(expression)` quotes an expression; `$!{...}` quotes block items. `$!Kind{...}` selects a result kind, Type, or Param. A typed quotation `$!T{expression}` or `$!(type){expression}` uses a type operand; the parenthesized form also accepts `$name` or `${expression}` carrying a type. Category names take priority over an ordinary type spelling. The quotation captures local `$name` references and braced value holes according to `parse_macro_quotation`. Typed quotations have context and nesting restrictions described in the reference.

M5. Slots and sequences

Inside a template, `$name` inserts a declared hole in a category-compatible position, and `@name` splices a sequence. `${expression}` and `$(...)` supply computed syntax in the applicable slot; meta calls use the same binding operations. `@producer(...)` and `@(form)` can supply sequences in positions that accept a sequence slot. Shared invocation lookup recognizing `@` does not permit scalar `@` calls or `@` definition names. The parser selects the slot's role (expression, type, name, parameter, declarator row, field, entry, enumerator, match row, catch, captures, block, or unit) and whether a sequence is permitted. In a meta call, a whole `$name` hole passes its captured value; a whole sequence hole passes one List, rather than splicing call arguments.

Lambda templates accept `&$name` capture rows as well as one Captures slot. Quoted Arrays accept `${@sequence}` in an argument-sequence position. Quoted Maps accept a bare Entry slot or a slot inside `${...}`. These slots require a compatible template context. Call arguments, initializer elements, quoted Array elements, and raise details also have argument-sequence positions. A sequence slot cannot supply several statements where one statement is required.

Complete production inventory

The following is included directly from the standalone file.

```
(* Generated by tools/syntax-spec write from etc/syntax/grammar.x.
   Read grammar.md and lexical.md for contextual contracts.
   Strings are token spellings; ? ... ? marks external recognition. *)

source-unit = ( translation-unit | script-unit ) ;
script-unit = ? script partition described in grammar.md entry points ? ;
translation-unit = { unit-item } ;
module-block = ( "module" , "{" , [ ( c-string , { ( "," , c-string ) } , [ "," ] ) ] , "}" ) ;
private-interface = ? static typedef, protocol body, class, or Lisp interface item; D4 ? ;
module-declaration = ? module before a file-scope declaration, read as static; D4 ? ;
unit-item = ( directive |
    static-assert |
    import-declaration |
    protocol-form |
    macro-definition |
    keyword-definition |
    private-interface |
    module-declaration |
    lisp-escape |
    linkage-group |
    declaration-definition |
    unit-extension ) ;
linkage-group = ( "extern" , c-string , "{" , { unit-item } , "}" ) ;
declaration-definition = ( [ ( "meta" , [ "native" ] ) ] , declaration-row , ( ";" | function-
body ) ) ;
function-body = ( raw-block | ( "=" , ">" , expression , ";" ) ) ;
import-declaration = ( "import" ,
    c-string ,
    [ ( "as" , identifier ) ] ,
    [ ( "with" , import-member , { ( "," , import-member ) } ) ] ,
    ";" ) ;
import-member = ( identifier , [ ( "as" , identifier ) ] ) ;
static-assert = ( "_Static_assert" , "(" , assignment , "," , assignment , ")" , ";" ) ;
declaration-row = ( declaration-group , { ( "," , declaration-group ) } ) ;
declaration-group = ( typedef-declaration |
    ( specifiers , declarator-list ) |
```

```
    ( specifiers , destructuring-targets , "=" , assignment ) ) ;
typedef-declaration = ( "typedef" , { qualifier } , type-specifier , declarator-list ) ;
specifiers = ( { declaration-prefix } , { qualifier } , type-specifier , { trailing-storage } ) ;
declaration-prefix = ( storage | "inline" | "_Noreturn" | attribute | prefix-macro ) ;
trailing-storage = ( storage | "inline" | prefix-macro ) ;
storage = ( ( "extern" , [ c-string ] ) | "static" | "auto" | "register" | "threaded" ) ;
qualifier = ( "const" | "restrict" | "volatile" ) ;
type-specifier = ( scalar-specifiers | "Self" | aggregate | enumeration | type-reference | type-
slot ) ;
scalar-specifiers = ( scalar-word , { scalar-word } ) ;
scalar-word = ( "void" | "char" | "short" | "int" | "long" | "float" | "double" | "signed" |
"unsigned" ) ;
type-reference = ( identifier , [ ( "." , identifier ) ] ) ;
type-name = ( { qualifier } , type-specifier , { pointer-part } ) ;
type-operand = declaration-group ;
declarator-list = ( init-declarator , { ( "," , init-declarator ) } ) ;
init-declarator = ( ( declarator | declarator-row-slot ) , [ ( "=" , assignment ) ] ) ;
declarator = ( { pointer-part } , [ direct-declarator ] , { declarator-suffix } ) ;
pointer-part = ( "*" | "^" | ( "&" , [ "?" ] ) | qualifier | qualifier-macro ) ;
direct-declarator = ( declaration-name | ( "(" , declarator , ")" ) ) ;
declaration-name = ( identifier | method-name | name-slot ) ;
method-name = ( method-owner , "." , identifier ) ;
method-owner = ? contextual type owner under D1 ? ;
declarator-suffix = ( ( "[" , [ expression ] , "]" ) |
    ( "(" , parameter-list , ")" ) |
    ( ":" , primary ) |
    attribute ) ;
parameter-list = ( parameter , { ( "," , parameter ) } ) ;
parameter = ( ( { qualifier } , type-specifier , declarator ) | "..." | parameter-slot ) ;
destructuring-targets = ( "(" , identifier , "," , identifier , { ( "," , identifier ) } ,
")" ) ;
typed-destructuring = ( "(" , parameter-list , ")" , "=" , assignment , ";" ) ;
aggregate = ( ( "struct" | "union" ) ,
    { attribute } ,
    [ declaration-name ] ,
    [ ( "{" , field , { field } , "}" , { attribute } ) ] ) ;
field = ( ( [ "delegate" ] , declaration-row , ";" ) | static-assert | field-extension ) ;
enumeration = ( "enum" ,
    { attribute } ,
    [ declaration-name ] ,
    [ ( "{" , [ ( enumerator , { ( "," , enumerator ) } , [ "," ] ) ] , "}" ,
{ attribute } ) ] ) ;
enumerator = ( ( declaration-name , [ ( "=" , conditional ) ] ) | enumerator-extension ) ;
named-type = ( ( identifier ,
    ( ";" |
    ( { qualifier } ,
    ( type-specifier | ( [ ( "struct" | "union" ) ] , "{" , { field } , "}" ) ) ,
    abstract-declarator ,
    ";" ) ) ) |
    named-type-slot ) ;
abstract-declarator = ? declarator with no declared name ? ;
attribute = ? balanced GNU attribute or known annotation-macro invocation ? ;
prefix-macro = ? known native declaration-prefix macro; contract D3 ? ;
qualifier-macro = ? known native macro supplying qualifiers ? ;
protocol-form = ( ( "protocol" ,
    type-name ,
    "(" ,
    identifier ,
    ")" ,
    "{" ,
    { associated-type } ,
    { protocol-member } ,
    "}" ) |
    ( [ "meta" ] ,
    [ "static" ] ,
    "protocol" ,
    type-name ,
    "(" ,
    type-name ,
    ")" ,
    [ ( ( "as" , type-name ) | ( "tag" , ( atomic | expression-slot ) ) ) ] ,
    ";" ) ) ;
associated-type = ( "associated" , identifier , "=" , type-name , ";" ) ;
```

```
protocol-member = ( specifiers , declarator , [ ( "=" , identifier ) ] , ";" ) ;
raw-block = ( "{" , { block-item } , "}" ) ;
block = ( ( "{" | "%{" ) , { block-item } , "}" ) ;
block-item = ( directive |
    static-assert |
    ( declaration-row , ";" ) |
    local-macro-definition |
    statement |
    block-extension ) ;
statement = ( block |
    ";" |
    ( expression , ";" ) |
    typed-destructuring |
    ( ( identifier | name-slot ) , ":" ) |
    ( "case" , expression , ":" ) |
    ( "default" , ":" ) |
    ( "if" , "(" , expression , ")" , governed , [ ( "else" , governed ) ] ) |
    ( "while" , "(" , expression , ")" , governed ) |
    ( "do" , governed , "while" , "(" , expression , ")" ) |
    ( "for" ,
    "(" ,
    [ for-init ] ,
    ";" ,
    [ expression ] ,
    ";" ,
    [ expression ] ,
    ")" ,
    governed ) |
    ( "switch" , "(" , expression , ")" , governed ) |
    ( "return" , [ expression ] , ";" ) |
    ( ( "break" | "continue" ) , ";" ) |
    ( "goto" , declaration-name , ";" ) |
    ( "defer" , governed ) |
    with-statement |
    match-statement |
    try-statement |
    raise-statement |
    statement-extension ) ;
for-init = ( type-operand | expression ) ;
governed = ? one statement, with directive handling G1 ? ;
with-statement = ( "with" , expression , [ ( "as" , identifier ) ] , raw-block ) ;
match-statement = ( "match" ,
    "(" ,
    expression ,
    ")" ,
    ( ( "{" , { ( directive | match-row ) } , "}" ) | match-row ) ) ;
match-row = ( ( ( ( "case" , pattern ) | "default" ) ,
    [ ( "if" , "(" , expression , ")" ) ] ,
    ":" ,
    governed ) |
    match-row-slot ) ;
pattern = ? expression yielding a List pattern, or macro pattern P1 ? ;
try-statement = ( "try" ,
    governed ,
    ( ( catch-arm , { catch-arm } , [ ( "finally" , governed ) ] ) | ( "finally" ,
governed ) ) ) ;
catch-arm = ( "catch" , ( ( [ catch-payload ] , ":" , governed ) | catch-slot ) ) ;
catch-payload = ( "%(" , list-element , { catch-detail } , ")" ) ;
catch-detail = ( sequence-pattern | ( "(" , bare-symbol , list-element , ")" ) ) ;
sequence-pattern = ? list-mode atom beginning with '*' ? ;
raise-statement = ( "raise" ,
    "%(" ,
    ( bare-symbol | insertion | expression-slot ) ,
    { ( raise-detail | argument-slot ) } ,
    ")" ,
    ";" ) ;
raise-detail = ( "(" , ( bare-symbol | expression-slot ) , list-element , ")" ) ;
bare-symbol = ? bare lit-atom whose decoded spelling is an exact Symbol ? ;
expression = ( assignment , { ( "," , assignment ) } ) ;
assignment = ( conditional , [ ( assignment-op , assignment ) ] ) ;
assignment-op = ( "=" | "+=" | "-=" | "*=" | "/=" | "%=" | "<<=" | ">>=" | "&=" | "^=" | "|=" ) ;
conditional = ( logical-or , [ ( "?" , expression , ":" , conditional ) ] ) ;
logical-or = ( logical-and , { ( "||" , logical-and ) } ) ;
```

```
logical-and = ( bitwise-or , { ( "&&" , bitwise-or ) } ) ;
bitwise-or = ( bitwise-xor , { ( "|" , bitwise-xor ) } ) ;
bitwise-xor = ( bitwise-and , { ( "^" , bitwise-and ) } ) ;
bitwise-and = ( equality , { ( "&" , equality ) } ) ;
equality = ( relational , { ( ( "==" | "!=" | "===" | "!==" ) , relational ) } ) ;
relational = ( shift ,
    { ( ( ( "<" | "<=" | ">" | ">=" | "in" ) , shift ) | ( "is" , [ "not" ] , type-
selector ) ) } ) ;
type-selector = ( type-name | ( "(" , type-name , ")" ) | cast ) ;
shift = ( additive , { ( ( "<<" | ">>" ) , additive ) } ) ;
additive = ( multiplicative , { ( ( "+" | "-" ) , multiplicative ) } ) ;
multiplicative = ( cast , { ( ( "*" | "/" | "%" ) , cast ) } ) ;
cast = ( ( cast-head , cast ) | unary ) ;
cast-head = ? parenthesized type operand with cast-follow lookahead; E1 ? ;
unary = ( ( ( "++" | "--" ) , unary ) |
    ( ( "&" | "*" | "+" | "-" | "~" | "!" ) , cast ) |
    ( "sizeof" ,
    ( ( "(" , type-operand , ")" , initializer , { postfix-part } ) |
    ( "(" , type-operand , ")" ) |
    type-operand |
    unary ) ) |
    postfix ) ;
postfix = ( primary , { postfix-part } ) ;
postfix-part = ( ( "[" , expression , "]" ) |
    ( "[" , [ expression ] , ":" , [ expression ] , [ ( ":" , [ expression ] ) ] , "]" ) |
    ( "(" , [ arguments ] , ")" ) |
    ( ( "." | "->" ) , member-name ) |
    "++" |
    "--" ) ;
arguments = ( argument , { ( "," , argument ) } , [ "," ] ) ;
argument = ( assignment | argument-slot ) ;
member-name = ( identifier | name-slot ) ;
primary = ( atomic |
    c-string-run |
    identifier |
    ( "(" , expression , ")" ) |
    statement-expression |
    initializer |
    array |
    map |
    quoted-list |
    quoted-array |
    quoted-map |
    percent-string |
    symbol-set |
    lambda |
    generic |
    va-arg |
    offsetof |
    lisp-escape |
    macro-expression |
    quotation |
    expression-slot ) ;
atomic = ( integer | floating | character | c-string | symbol | atom | "void" ) ;
c-string-run = ( c-string , { ( c-string | native-string-word ) } ) ;
native-string-word = ? adjacent native macro or unbound word accepted by E3 ? ;
statement-expression = ( "(" , raw-block , ")" ) ;
initializer = ( "{" , [ ( initializer-item , { ( "," , initializer-item ) } , [ "," ] ) ] ,
"}" ) ;
initializer-item = ( assignment | argument-slot | ( designator , { designator } , "=" ,
assignment ) ) ;
designator = ( ( "." , identifier ) | ( "[" , assignment , "]" ) ) ;
array = ( "[" , [ ( assignment , { ( "," , assignment ) } , [ "," ] ) ] , "]" ) ;
map = ( "{" , [ ( map-entry , { ( "," , map-entry ) } , [ "," ] ) ] , "}" ) ;
map-entry = ( ( ( identifier | assignment ) , ":" , assignment ) | entry-extension ) ;
generic = ( "_Generic" ,
    "(" ,
    assignment ,
    { ( "," , ( type-name | "default" ) , ":" , assignment ) } ,
    ")" ) ;
va-arg = ( "va_arg" , "(" , assignment , "," , type-operand , ")" ) ;
offsetof = ( "offsetof" ,
    "(" ,
```

```
    type-name ,
    "," ,
    identifier ,
    { ( ( "." , identifier ) | ( "[" , expression , "]" ) ) } ,
    ")" ) ;
lambda = ( "%!" ,
    "(" ,
    [ ( parameter-list | bare-parameters ) ] ,
    ")" ,
    [ ( "using" , capture-list ) ] ,
    "=" ,
    ">" ,
    ( raw-block | assignment ) ) ;
bare-parameters = ( identifier , { ( "," , identifier ) } ) ;
capture-list = ( ( "&" , capture-name , { ( "," , "&" , capture-name ) } ) | captures-slot ) ;
capture-name = ( identifier | name-slot ) ;
quoted-list = ( "%(" , { list-element } , ")" ) ;
nested-list = ( "(" , { list-element } , ")" ) ;
list-element = ( literal-element | insertion | splice | ( reader-prefix , list-element ) ) ;
literal-element = ( atomic | nested-list | quoted-array | quoted-map | percent-string | typed-
capture ) ;
insertion = ( ( "$" , identifier ) | ( "${" , expression , "}" ) ) ;
splice = ( ( "@" , identifier ) | ( "@{" , expression , "}" ) ) ;
reader-prefix = ( "\'" | "`" | "," | ",@" ) ;
typed-capture = ( "?(" , type-name , identifier , ")" ) ;
quoted-array = ( "%[" , [ ( quoted-array-item , { ( "," , quoted-array-item ) } , [ "," ] ) ] ,
"]" ) ;
quoted-array-item = ( data-element | argument-slot | ( "${" , argument-slot , "}" ) ) ;
data-element = ( literal-element | insertion ) ;
quoted-map = ( "%{" , [ ( quoted-entry , { ( "," , quoted-entry ) } , [ "," ] ) ] , "}" ) ;
quoted-entry = ( ( data-element , ":" , data-element ) | entry-extension | quoted-entry-
extension ) ;
quoted-entry-extension = ( "${" , entry-extension , "}" ) ;
percent-string = ( "%\"" , { ( string-segment | insertion ) } , "\"" ) ;
symbol-set = ( "%<<" , { symbol-set-member } , ">>" ) ;
symbol-set-member = ? symbol-set-mode lit-atom or lit-symbol ? ;
unit-extension = ( class-declaration | unit-macro | decorator ) ;
class-declaration = ( "class" , named-type ) ;
statement-extension = ( foreach-statement | statement-macro | decorator ) ;
foreach-statement = ( "foreach" , "(" , declaration-argument , ( "in" | "," ) , assignment ,
")" , governed ) ;
declaration-argument = ( specifiers , ( init-declarator | destructuring-targets ) ) ;
keyword-definition = ( [ "static" ] , "keyword" , identifier , macro-name , ";" ) ;
macro-name = ( "$" , identifier , { ( "." , identifier ) } ) ;
macro-definition = ( [ "static" ] , "macro" , result-kind , macro-name , macro-signature , macro-
body ) ;
local-macro-definition = ( "macro" , result-kind , identifier , macro-signature , macro-body ) ;
anonymous-macro = ( "macro" , result-kind , macro-signature , macro-body ) ;
macro-expression = ( macro-call | local-macro-call | anonymous-macro | meta-call | macro-name ) ;
macro-signature = ( "(" ,
    [ ( hole-parameter , { ( "," , hole-parameter ) } ) ] ,
    ")" ,
    [ ( "using" , using-list ) ] ) ;
hole-parameter = ( [ hole-kind ] , ( "$" | "@" ) , identifier ) ;
using-list = ( ( "$" , identifier , { ( "," , "$" , identifier ) } ) |
    ( identifier , { ( "," , identifier ) } ) ) ;
body-using = ( "using" , using-list , ";" ) ;
macro-body = ? body selected by result/target kind; braced bodies start with body-using lines;
M2 ? ;
macro-call = ( macro-name , "(" , macro-arguments , ")" ) ;
local-macro-call = ( identifier , "(" , macro-arguments , ")" ) ;
meta-call = ( macro-name , "(" , [ ( meta-argument , { ( "," , meta-argument ) } ) ] , ")" ) ;
meta-argument = ( assignment | meta-value-hole ) ;
meta-value-hole = ? whole template hole passed as a value, under M5 ? ;
macro-arguments = ? signature-directed argument sequence; contract M1 ? ;
decorator = ? visible decorator invocation followed by its target; M3 ? ;
quotation = ( "$" ,
    "!" ,
    ( ( "(" , expression , ")" ) |
    ( [ quotation-kind ] , "{" , quotation-items , "}" ) |
    ( quotation-type , "{" , expression , "}" ) ) ) ;
quotation-kind = ( result-kind | "Type" | "Param" ) ;
quotation-type = ? type operand or computed type under M4 ? ;
```

```
quotation-items = ? contents selected by quotation kind, with ${expr} and @{expr} holes; M4 ? ;
result-kind = ( "Expr" |
    "Expression" |
    "Stmt" |
    "Field" |
    "Entry" |
    "Enumerator" |
    "Unit" |
    "Declaration" |
    "Decorator" ) ;
hole-kind = ( "Expr" |
    "Expression" |
    "Stmt" |
    "Field" |
    "Entry" |
    "Enumerator" |
    "Unit" |
    "Function" |
    "NamedType" |
    "Type" |
    "Decl" |
    "DeclaratorRow" |
    "Name" |
    "Literal" |
    "Param" |
    "Catch" |
    "Captures" |
    "MatchRow" ) ;
expression-slot = ? expression slot under M5 ? ;
argument-slot = ? argument sequence slot: @name, @call(...), or @(form); M5 ? ;
declarator-row-slot = ? declarator row slot under M5 ? ;
named-type-slot = ? named type slot under M5 ? ;
type-slot = ? type slot under M5 ? ;
name-slot = ? name slot under M5 ? ;
parameter-slot = ? parameter slot under M5 ? ;
captures-slot = ? capture slot under M5 ? ;
catch-slot = ? catch-arm slot under M5 ? ;
match-row-slot = ? match-row slot under M5 ? ;
unit-macro = ? unit-position macro/slot under M1-M5 ? ;
block-extension = ? block-position macro/slot under M1-M5 ? ;
statement-macro = ? statement-position macro/slot under M1-M5 ? ;
field-extension = ? field-position macro/slot under M1-M5 ? ;
enumerator-extension = ? enumerator-position macro/slot under M1-M5 ? ;
entry-extension = ? map-entry-position macro/slot under M1-M5 ? ;
lisp-escape = ( "$(" , { lisp-form } , ")" ) ;
lisp-form = ( lisp-atom | ( "(" , { lisp-form } , ")" ) | ( reader-prefix , lisp-form ) | lisp-
hole ) ;
lisp-atom = ( integer | floating | c-string | symbol | lisp-identifier ) ;
lisp-hole = ? template hole in compiler Lisp; M5 ? ;
lisp-identifier = ? ident token in Lisp mode ? ;
identifier = ? identifier accepted in the current name position; D1 ? ;
integer = ? lit-int ? ;
floating = ? lit-float ? ;
character = ? lit-char ? ;
c-string = ? lit-char* ? ;
symbol = ? lit-symbol ? ;
atom = ? lit-atom ? ;
string-segment = ? segment ? ;
directive = ? preproc token retained/selected by preprocessing; G1 ? ;
```

Coverage and validation limits

The inventory covers declarations, types, expressions, statements, literals, protocols, macro syntax, embedded Lisp reading, and script entry selection. External recognizers cover: native annotation contents, dynamic macro signatures, context-sensitive names, slots, and pattern recognition. The accompanying lexical specification covers raw source and layout spelling.

Source comparison, existing fixtures, and focused probes can establish specific correspondences. They do not prove that this grammar and the compiler accept exactly the same language. In particular, no independent

parser has consumed this EBNF over the entire repository corpus. Full semantic validity, imported C dialects, and arbitrary compile-time evaluation remain outside that claim.

Lexical specification

This is an operational specification of `lib/scan.x`, `lib/tokenizer.x`, and the keyword retagging in `src/compiler/parsing/tokens.x`. Rules are ordered where order affects recognition. The source grammar consumes their output. The same rules are available as structured List data, with a generated inventory.

File-scope `module` blocks are spliced before tokenizing; see module collection.

Input and output

Input is a NUL-terminated byte sequence. The first NUL ends scanning in every mode. There is no global UTF-8 validation pass. Identifiers use ASCII classes; quoted text and atoms can contain other bytes subject to their own delimiters. The separate `scan_utf8_length` helper is not such a pass.

A token is `(type, text, pos, len, line, col)`. `pos` is a zero-based byte offset; `len` is a byte count. Lines and columns start at one. LF increments the line and sets the next column to one. Every other byte, including CR and tab, advances the column by one. Token text is copied from the input.

`space` and `comment` tokens remain in physical storage. `Tokenizer.next` skips those two kinds, retaining `preproc`. Compiler token navigation also skips directives where appropriate, preserving their source and conditional context separately. End of input emits zero-width `eof`. An unclosed lexical mode alone does not make EOF a lexical error: the parser or Lisp reader diagnoses the missing delimiter.

Byte classes and trivia

In the productions below, quoted characters denote bytes. `|` is choice, `,` is concatenation, `[ ]` is optional, and `{ }` repeats zero or more times. Range descriptions name ASCII bytes. The productions do not override the ordered mode dispatch below.

```
letter = ? A-Z or a-z ? ;
digit = ? 0-9 ? ;
hex-digit = digit | ? A-F or a-f ? ;
octal-digit = ? 0-7 ? ;
binary-digit = "0" | "1" ;
identifier = ( letter | "_" ) , { letter | digit | "_" } ;
white-byte = ? SP HT CR VT FF LF ? ;
space = white-byte , { white-byte } ;
line-comment = "//" , { ? byte other than LF or NUL ? } ;
block-comment = "/*" , ? bytes through the first closing */ ? ;
```

Comments do not nest. A block comment without its closing delimiter reports `incomplete`. A line comment excludes its final LF and may end at NUL.

Where common-token dispatch applies, `#` starts a `preproc` token without a beginning-of-line requirement. The token ends before an uncontinued LF, or at NUL. Backslash-LF and backslash-CRLF are included and continue scanning. A backslash immediately before NUL is malformed. The scanner does not expand the directive or validate its directive-specific grammar.

Names, keywords, and punctuation

The initial keyword table is exactly:

```
associated auto break case catch char const continue default defer delegate
 do double else enum extern finally float for goto if import in inline int
 long match protocol raise register restrict return short signed sizeof
 static struct switch threaded try typedef union unsigned void volatile while
```

The scanner emits `in` and `match` as keywords. After scanning, the compiler retags each to `ident` unless its context is the operator or statement form; see compiler reclassification.

Matching is case-sensitive. The following spellings normalize token type while retaining original token text:

Source spelling	Token type
`__inline`, `__inline__`	`inline`
`__restrict`, `__restrict__`	`restrict`
`thread_local`, `_Thread_local`	`threaded`

Other words initially emit `ident`, including `is`, `not`, `with`, `as`, `tag`, `macro`, `keyword`, `using`, `meta`, `native`, `Self`, `class`, `foreach`, `module`, `_Static_assert`, `_Generic`, `offsetof`, and `va_arg`. Parser context or visible macro definitions give these words their roles. A named-reference sigil scans the immediately following identifier as `ident` without applying the keyword table.

The C operator scanner chooses the longest supported prefix from this set:

```
... >>= <<= === !==
-> ++ -- << >> <= >= == != && ||
+= -= *= /= %= &= ^= |=
- , ; : ! ? . ( ) [ ] { } * / & % ^ + < = > | ~ @
```

`=>` is two tokens: `=` then `>`. `$`, `$(`, `@(`, `${`, `@{`, `?(`, `%(`, `%[`, `%{`, `%<<`, `%"`, and `%!` are handled by mode dispatch, not by this operator scanner. The same bytes can therefore have different roles in different modes.

Numeric tokens

Numeric recognition first chooses a radix, then scans the selected form. The rules below differ from C in the cases listed.

1. An explicit `0x`/`0X`, `0b`/`0B`, or `0o`/`0O` selects hexadecimal, binary, or octal.
2. Otherwise, `0` immediately followed by `1` through `7` selects legacy octal, unless the run of decimal digits is followed by `.`, `e`, or `E`. Those three continuations select decimal floating recognition.
3. Every other numeric start selects decimal recognition.

After this dispatch, the following productions apply. A numeric token must end before a byte outside `letter | digit | "_"`; a bad suffix or radix continuation fails the token instead of splitting it into a number and name.

```
decimal-integer = digit , { digit } , integer-suffix ;
binary-integer = ( "0b" | "0B" ) , binary-digit , { binary-digit } ,
                 integer-suffix ;
octal-integer = ( "0o" | "0O" ) , octal-digit , { octal-digit } ,
                integer-suffix ;
legacy-octal = "0" , ? 1-7 ? , { octal-digit } , integer-suffix ;
hex-integer = ( "0x" | "0X" ) , hex-digit , { hex-digit } , integer-suffix ;
integer-suffix = [ unsigned , [ long , [ long ] ]
                 | long , [ long ] , [ unsigned ] ] ;
unsigned = "u" | "U" ;
long = "l" | "L" ;
float-suffix = [ "f" | "F" | "l" | "L" ] ;
exponent-digits = [ "+" | "-" ] , digit , { digit } ;
decimal-exponent = ( "e" | "E" ) , exponent-digits ;
decimal-float = ( digit , { digit } , "." , { digit } , [ decimal-exponent ]
                | "." , digit , { digit } , [ decimal-exponent ]
                | digit , { digit } , decimal-exponent ) , float-suffix ;
hex-float = ( "0x" | "0X" ) ,
            ( hex-digit , { hex-digit } , [ "." , { hex-digit } ]
            | "." , hex-digit , { hex-digit } ) ,
            ( "p" | "P" ) , exponent-digits , float-suffix ;
```

A hexadecimal point requires a `p`/`P` exponent. Integer suffix letters can mix case. `1f` is malformed; `1.f` is a float. `08`, `09`, and `00` take the decimal path, `018` is malformed, and `017e3` is decimal floating. The native C compiler still checks each spelling and its range.

The numeric helper accepts an optional leading sign. Code mode emits `+` and `-` as operators. List, Lisp, and quoted Array/Map modes invoke signed numeric recognition only when the sign immediately precedes a decimal digit. Thus `-.5` is an atom in those modes, while `-0.5` is a numeric token. An unsigned `.5` starts a number through common-token dispatch.

Quoted bytes and escapes

A C string starts and ends with `"`. Its contents are ordinary bytes other than NUL, raw CR/LF, quote, or backslash, plus accepted C escapes. NUL before the close reports `incomplete`; raw CR/LF reports `malformed`.

A C character literal contains exactly one unescaped byte or one accepted escape between apostrophes. Empty contents or multiple unescaped bytes fail. The scanner does not combine wide/UTF-prefixed strings into a single token: a prefix such as `L` is scanned separately as an identifier.

Escape after backslash	C string/character	Percent string or quoted Symbol
`a b f n r t v ' " ?` or backslash	One escaped byte spelling	Same
LF or CRLF	Source continuation	Source continuation
Lone CR	Rejected	Rejected
One to three octal digits	Three digits must start with `0` through `3`	Same
`x` or `X` and hex digits	One or more; consumes all following hex digits	One or two hex digits
`u` and hex digits	Exactly four	One or two
`U` and hex digits	Exactly eight	One or two
`$`	Rejected	Accepted only by the percent-string segment scanner
Any other escape	Rejected	Rejected

Widths describe recognition. Unicode scalar validity, decoded numeric range, and host compiler acceptance are separate questions. Percent-string decoding removes continuations, normalizes raw CR and CRLF to LF, and decodes escapes. See `_parse_text_segment` in `src/literals/literals.x`.

A simple Symbol spelling is `<` followed by one or more non-whitespace, non-NUL bytes, through the first `>`. A quoted spelling is `<"` followed by quoted-Symbol contents, then `">`. Quoted Symbols permit raw line breaks and x2c byte escapes. Compact-Symbol representability and exact round trips are checked later, by literal processing.

Atom boundaries

An ordinary List/Lisp atom stops before any of these unescaped boundaries: `(`, `)`, apostrophe, backtick, comma, `"`, `$`, `@`, `{`, `[`, whitespace, or a comment opener. Backslash quotes the next byte for boundary recognition; a final backslash reports `incomplete`. Quoted Array/Map atoms additionally stop before `:`, `]`, or `}`. Comma already ends an atom.

A bare SymbolSet atom stops before whitespace, `$`, `@`, or `>>`. Backslash quotes the next byte. A leading `"` instead scans a quoted Symbol spelling without angle brackets, so `>>` inside quotes is content. Other punctuation can belong to a bare set atom. Dispatch still claims a leading comment, digit, apostrophe, or `#` before the set-atom scanner runs.

Ordered scanning by mode

Maintain a stack with root mode `x2c` for source or `lisp` for a standalone Lisp reader. At each position, first test for NUL. Otherwise apply the first matching successful rule in the current mode. Recognition of a malformed prefix records failure. See failure edges for how scanning stops.

Common rules, in order of starting byte, recognize whitespace, `#` directives, `//` or `/*` comments, C characters at apostrophe, numbers at a digit, and numbers at `.` followed by a digit.

Mode	Ordered recognition
`x2c`, `x2c-par`	Common rules; `$(`/`@(` or `$`/`@` reference; percent forms; eligible angle Symbol; C string; identifier/keyword; C operator
`array`, `map`	Common rules; data prefixes; data punctuation; collection atom as `lit-atom`
`list`	Data prefixes; data punctuation; common rules; atom as `lit-atom`
`lisp`, `macro-lisp`	Lisp `$` reference; reader punctuation/signed number/Symbol; common rules; atom as `ident`
`symbol-set`	Common rules; `>>`; `$`/`@`; angle Symbol; set atom as `lit-atom`

Mode	Ordered recognition
`string`	`${` or `$` reference; closing `"`; segment scanner as `segment`

Data-prefix rules recognize `void` as a keyword only in quoted Array/Map mode, `?(` only in List mode, and `$` interpolation in all data modes. Array/Map modes recognize `@` prefixes for template sequence slots. List mode also recognizes runtime `@` splicing. A List `@` immediately followed by EOF, whitespace, `)`, or `=` instead emits the data atom `@` or `@=`.

Data punctuation recognizes nested `(`, strings, arrays, and maps; collection closers and colons; reader prefixes; angle Symbols; and a signed number when the sign precedes a digit. Comma is an element separator in Array/Map mode and a reader prefix elsewhere; `,@` is one reader-prefix token outside Array/Map mode. Apostrophe therefore starts a reader form in List/Lisp mode but a C character in Array/Map mode. Not every recognized reader token is accepted by every data parser.

In Lisp-shaped modes, `<` stays an atom prefix when its next byte is EOF, `=`, whitespace, or one of `()'`, backtick, or comma. Otherwise it attempts a Symbol literal. Standalone Lisp `$` scanning does not provide the data-mode `${expression}` escape; its reader rejects unsupported token kinds.

In String mode a segment stops before a closing quote or a single unescaped `$`. Inside a segment, `$$` is consumed as literal content. Dispatch handles a `$` at the beginning of a segment before calling the segment scanner; see the `string` row of the table above.

Mode transitions

A transition emits the opener/closer token as well as changing the stack. Other recognized tokens leave the mode unchanged.

Current mode	Input	Stack action / emitted kind
Code	`{`	Push `x2c`
Code	`}`	Pop if stack depth exceeds one
Code	`%(`, `%[`, `%{`, `%<<`, `%"`, `$(`/`@(`	Push `list`, `array`, `map`, `symbol-set`, `string`, `macro-lisp`, respectively
Code	`%!`	Emit lambda prefix; remain in code
`x2c-par`	`(` / `)`	Push `x2c-par` / pop
Data (`list`, `array`, `map`)	`(`	Push `list`
Data	`[`, `{`, `"`	Push `array`, `map`, `string`; emit `%[`, `%{`, `%"` kind, retaining original text/width
Data	`${`, `@{`	Push `x2c`
`list`	`?(`	Push `x2c-par`
`list`, `array`, `map`	Matching `)`, `]`, `}`	Pop corresponding mode
`symbol-set`	`>>`	Pop
`string`	`${` / `"`	Push `x2c` / pop
`macro-lisp`	`(` / `)`	Push `macro-lisp` / pop

Root `lisp` does not pop on `)`. The reader checks list structure. Normal code parentheses do not change lexical mode. Nested quoted data uses bare openers; writing a percent opener inside data is not the same transition.

Operand context for percent and angle literals

Let `previous` be the preceding token excluding spaces and comments. The tokenizer's operand-ending set is:

```
ident lit-int lit-float lit-char* lit-char lit-atom lit-symbol
) ] } " ++ --
```

`%(`, `%{`, and `%!` scan `%` as modulo after an operand, except after a control condition, a recognized statement block, or at a layout statement start. `%"`, `%[`, and `%<<` open literals regardless of the previous token. An eligible `%<` without the second `<` is malformed.

A control condition is a `)` whose matching ordinary `(` follows `if`, `while`, `for`, or `switch`. A statement block is a `}` whose matching ordinary `{` follows one of:

- `;`, `:`, `{`, `}`, `]`, an identifier, `else`, `do`, `try`, `finally`, or `defer`.
- `)` whose matching ordinary `(` follows an identifier, `match`, or one of the four control keywords above.

`<` may start a Symbol when no preceding operand exists. After an operand, it may do so following identifier `tag`, or following an operand then identifier `is` and optional identifier `not`.

For these two predicates only, a layout statement starts when the last stored token is whitespace containing LF, and the current column is no greater than the first significant token's column on the previous significant token's physical line. A deeper continuation retains operand context.

Indentation as a token transformation

Layout is enabled for `.xp` input by the compiler, or by exactly `#pragma indent` before code. Trailing whitespace is allowed on that directive; comments and other directives may precede it. The pass runs after raw scanning and before compiler keyword reclassification. Its state comprises logical lines, bracket depths, pending ternary colons, open indentation levels, closing-token strings, and whether each block is an enum.

Apply the following transformation in order:

1. Remove spaces/comments from the working index, retaining physical storage. A directive always forms a separate logical line. Otherwise split at a physical line break at bracket depth zero when the next line is no deeper, or the previous line ends with a non-ternary colon. A leading `.` continues the prior line. Track `?` and its `:` at depth zero.
2. Use the first statement's column as the base indentation. For each line, find the next non-directive line's indentation, or the base at EOF.
3. If the line ends with a non-ternary `:` and the next statement is deeper, open a block. Replace the colon with `{`, except that `case`, `default`, and `catch` retain `:` and add `{`. Push the deeper indentation.
4. For a control header, use the last depth-zero `if`, `while`, `for`, `foreach`, `switch`, or `match` before the colon, excluding dotted member names. Add condition parentheses if the condition is not already wrapped. `for` uses its written clause parentheses. An inline body uses the first non-ternary, non-label colon and removes that body separator.
5. An aggregate header contains `struct`, `union`, or `enum` and no ordinary parameter `(`. Its closer is `};`, except a leading `typedef` or `static typedef` uses `}`. Other block closers are `}`.
6. A line consisting of `do:` becomes a bare block if the first following non-directive line at the same or lower indentation is not a same-indent `while` trailer without a final colon. Otherwise retain `do`.
7. Append `;` to an ordinary statement line unless it already ends with `;`, is in an enum body, or is one whole `$(...)` or `@(form)` form. A whole-line `$name`, `@name`, or computed `@producer(...)` hole also takes no semicolon, including a hole after a control header's `)`. The leading `@` in `@$decorator(...)` becomes trivia and suppresses the semicolon. Directive lines receive no semicolon; `#pragma indent` becomes a comment.
8. Append stored closers while the next indentation is lower than the stack top. If a following statement matches no remaining level, report `indent`. Close all outstanding blocks at EOF.

The pass checks tabs after the last LF in the whitespace token immediately before a new statement line. Indentation uses spaces. New punctuation has zero width at its neighboring token's start or end. Reclassified source punctuation retains its source extent. See `_Layout` in `lib/tokenizer.x` for the transitions and the indentation guide for examples.

Compiler reclassification and preprocessing

After scanning/layout, `Compiler.tokenize` reports malformed tokens, calls `scan_conditionals`, then `_retag_keywords`. This last operation differs from the raw tokenizer's operand-context predicate.

- An `in` token retains that type only if the preceding significant kind is one of `ident`, the six atomic literal kinds, `)`, or `]`, and the next kind can begin an operand. That beginning set is `ident`, the six atomic literal kinds, `(`, `%(`, `%[`, `%{`, `$(`, `${`, `$`, `!`, `-`, `*`, &, ~, ++, or `--`. Otherwise it becomes `ident`. The parser can still recognize the spelling in the relevant position.

- For `match`, inspect the next significant token, skipping a parenthesized group if one follows. Keep `match` when the inspected token is `case` or `{`; otherwise change it to `ident`.

Here the six atomic literal kinds are `lit-int`, `lit-float`, `lit-char`, `lit-char*`, `lit-atom`, and `lit-symbol`.

Directives are not expanded by the tokenizer. Later conditional handling hides selected inactive branches and retains conditional structure. Lexical failures can therefore occur in inactive branches: scanning precedes that selection. Host preprocessing for imported declarations is a separate phase. A source shebang is handled by the frontend before ordinary lexing; it substitutes the scripting include while retaining subsequent line numbers. See host preprocessing.

Failure and implementation edges

Most scanner helpers return a positive matched length, zero for no match, or a negative failure, with some prefix preconditions enforced by `Error`. The individual interfaces matter: `scan_keyword` and `scan_c_operator` use `-1` for an unsupported start; `scan_white_space` also returns `-1` at NUL. Status-aware scanners distinguish `incomplete` from `malformed`; wrappers without a status preserve only `malformed`. The tokenizer records the first failure status and appends zero-width `error` and `eof` at the refused position. Physical storage can contain another sentinel pair or tokens after that first EOF because failure callbacks return zero to chained dispatch. `Tokenizer.next` stops at the first EOF. Layout failure uses `indent`. Consumers must not equate lexical success with balanced delimiters or a valid program.

Syntax as data

The source grammar and lexical rules are ordinary x2c `List` values. A tool can compile them, inspect their structure, and generate another representation without first parsing the prose specification.

Three modules under `etc/syntax/` hold the data:

Module	Accessor	Contents
`grammar.x`	`syntax_grammar()`	Entry point and 187 productions
`lexical.x`	`syntax_lexical()`	Byte patterns, ordered scanning, modes, transitions, predicates, token passes, failures
`extensions.x`	`syntax_extensions()`	Reviewed C11 comparison, feature groups, affected rule names, example spellings, documentation links

For example, an ordinary x2c script can consume the grammar:

```
#!/usr/bin/env -S x2c script
#include "etc/syntax/grammar.x"

List grammar = syntax_grammar();
printf("%s\n", grammar.repr());
```

Save this example at the repository root. `x2c script` builds and links its included local module. An ordinary `x2c build` invocation must list the module as a build input. The accessors construct descriptive values; the compiler's production parser and tokenizer do not consult them.

Grammar model

The root has the shape `(grammar (start NAME) (rules RULE...))`. Each rule is `(rule NAME EXPRESSION)`. Names are atoms and terminal spellings are Strings. Rule expressions use these forms:

Expression	Meaning
`(ref NAME)`	Recognize another named production
`(token "text")`	Recognize the token spelling in the applicable lexical context
`(seq EXPR...)`	Recognize every child in order
`(choice EXPR...)`	Recognize an alternative; this does not specify PEG priority
`(optional EXPR)`	Recognize zero or one occurrence
`(repeat 0 EXPR)`	Recognize zero or more occurrences

Expression	Meaning
`(external "contract")`	Delegate recognition to the named lexical or contextual contract

For example, the function-body rule contains both familiar blocks and the expression-body addition:

```
List rule = %(rule function-body
  (choice
    (ref block)
    (seq (token "=") (token ">") (ref expression) (token ";"))));
```

The EBNF is generated from this value. The grammar chapter explains binding, macro signatures, precedence, and other contextual contracts. An external leaf marks a rule that the grammar chapter describes in prose. A parser generator would need code for each one.

Lexical model

Character regular expressions cannot describe the lexer. A quoted List, for example, changes the significance of punctuation and signed numbers. Braced interpolation temporarily returns to code scanning.

The generated inventory exposes every record. The lexical specification explains the operations and boundary cases. The sections are:

Section	Meaning
`input`	Byte encoding, NUL termination, token fields, positions, and trivia
`classes`, `rules`	Named byte recognizers and their actions
`modes`	Ordered rule dispatch for each scanner mode
`transitions`	Source mode, opener/closer bytes, emitted spelling, stack push/pop
`predicates`	Conditions on surrounding tokens and named token sets
`passes`	Frontend preparation, scanning, layout, conditional processing, keyword retagging, parsing
`keywords`, `operators`	Exact scanner spellings and keyword normalization
`failures`	Scanner result conventions, status, emitted errors, and stopping behavior
`boundaries`	Literal decoding, Symbol representation, host preprocessing, native C acceptance
`sources`	Implementation owners used for the extraction

Byte-pattern primitives are `byte`, `bytes`, inclusive `range`, and `one-of-bytes`. `ref` refers to a class or rule. `seq`, `choice`, `optional`, and `repeat MINIMUM` compose patterns. `ordered` tries children in dispatch order; `until` records whether a closing delimiter is consumed; `except` excludes listed byte patterns or prefixes. Guards such as `when-prefix` and `when-mode` select the applicable pattern. They are structured records.

Actions record `emit`, consumed width, failure behavior, and `transition`. A transition's `push` stores a new mode; `pop` restores the previous stack mode. The transition table's `defaults` record applies unless a row overrides it. Predicates record Boolean conditions and token-history operations. Passes record ordered transformations and the state they use. `external` marks work done by other code.

This is a descriptive operational model. The tool validates named references and renders records; it does not interpret every lexical operation. Building an executable scanner from these records requires giving each operation an implementation and comparing its token streams with the existing tokenizer.

Existing consumers

Run these commands from the repository root with a built compiler:

```
builds/0/x2c script tools/syntax-spec summary
builds/0/x2c script tools/syntax-spec dump grammar
builds/0/x2c script tools/syntax-spec uses expression
builds/0/x2c script tools/syntax-spec graph quoted-list
builds/0/x2c script tools/syntax-spec modes
builds/0/x2c script tools/syntax-spec write
builds/0/x2c script tools/syntax-spec check
```

`dump` also accepts `lexical` and `extensions`. It prints the runtime List representation for inspection. `uses` finds direct callers of a production. `graph` emits a Mermaid production-dependency graph. `modes` emits a Mermaid graph of stack-push transitions; ordered dispatch and predicates still apply, and closing delimiters return to the previous stack mode.

`write` generates the EBNF, lexical inventory, and the table in the x2c syntax map. `check` validates references and reports stale generated outputs without changing them. These are optional commands; no recurring build or publication check was added.

The extension data lists the productions that x2c adds to or changes from C.

Further uses and limits

These values could support:

- Generate railroad diagrams for productions with no external leaves.
- Produce candidate token sequences from bounded derivations, then filter them through the real compiler and retain accepted and rejected cases.
- Compare scanner outputs against an interpreter of the lexical records.
- Generate highlighting tables from keyword and operator spellings.
- Select documentation examples and diagrams by extension family.

Lexical data inventory

Generated by `tools/syntax-spec write` from `etc/syntax/lexical.x`. See the lexical specification for explanations and the data model for notation and consumption.

input

```
(input (encoding bytes) (terminator 0) (utf8-validation none)
 (token-fields type text pos len line col)
 (position (pos 0 bytes) (len bytes) (line 1) (col 1 bytes))
 (advance (on 10 (increment line) (set col 1)) (otherwise (increment col)))
 (text copy) (trivia space comment) (retain preproc)
 (roots x2c lisp) (termination before-mode-dispatch))
```

classes

```
(classes (class letter (choice (range 65 90) (range 97 122)))
 (class digit (range 48 57))
 (class hex-digit (choice (ref digit) (range 65 70) (range 97 102)))
 (class octal-digit (range 48 55))
 (class binary-digit (choice (byte 48) (byte 49)))
 (class identifier-start (choice (ref letter) (byte 95)))
 (class identifier-rest (choice (ref identifier-start) (ref digit)))
 (class white-byte
  (choice (byte 32) (byte 9) (byte 13) (byte 11)
   (byte 12) (byte 10))) (class simple-escape (one-of-bytes "abfnrtv\'\"?\\"))
 (class atom-boundary (choice (one-of-bytes "()\'`,\"$@{[") (ref white-byte)))
 (class collection-boundary (one-of-bytes ",:]}"))
 (class set-boundary (choice (ref white-byte) (byte 36) (byte 64))))
```

rules

```
(rules
 (rule identifier (seq (ref identifier-start) (repeat 0 (ref identifier-rest))))
 (rule whitespace (repeat 1 (ref white-byte)) (emit space))
 (rule line-comment
  (seq (bytes "//") (until (choice (byte 10) (byte 0)) exclude))
  (emit comment))
 (rule block-comment (seq (bytes "/*") (until (bytes "*/") include))
  (nesting none) (on-nul incomplete) (emit comment))
 (rule preprocessor
  (seq (byte 35)
   (repeat 0 (ordered (ref continuation) (except (byte 10) (byte 0))))))
```

```
 (reject trailing-backslash) (emit preproc))
(rule continuation (seq (byte 92) (choice (byte 10) (seq (byte 13) (byte 10)))))
(rule unsigned-suffix (choice (bytes "u") (bytes "U")))
(rule long-suffix (choice (bytes "l") (bytes "L")))
(rule integer-suffix
 (optional
  (choice
   (seq (ref unsigned-suffix)
    (optional (seq (ref long-suffix) (optional (ref long-suffix)))))
   (seq (ref long-suffix) (optional (ref long-suffix))
    (optional (ref unsigned-suffix))))))
(rule float-suffix (optional (one-of-bytes "fFlL")))
(rule exponent-digits
 (seq (optional (one-of-bytes "+-")) (repeat 1 (ref digit))))
(rule decimal-exponent (seq (one-of-bytes "eE") (ref exponent-digits)))
(rule decimal-integer (seq (repeat 1 (ref digit)) (ref integer-suffix)))
(rule binary-integer
 (seq (byte 48) (one-of-bytes "bB") (repeat 1 (ref binary-digit))
  (ref integer-suffix)))
(rule octal-integer
 (seq (byte 48) (one-of-bytes "oO") (repeat 1 (ref octal-digit))
  (ref integer-suffix)))
(rule legacy-octal
 (seq (byte 48) (range 49 55) (repeat 0 (ref octal-digit))
  (ref integer-suffix)))
(rule hex-integer
 (seq (byte 48) (one-of-bytes "xX") (repeat 1 (ref hex-digit))
  (ref integer-suffix)))
(rule decimal-float
 (seq (choice
       (seq (repeat 1 (ref digit)) (byte 46) (repeat 0 (ref digit))
        (optional (ref decimal-exponent)))
       (seq (byte 46) (repeat 1 (ref digit)) (optional (ref decimal-exponent)))
       (seq (repeat 1 (ref digit)) (ref decimal-exponent)))
  (ref float-suffix)))
(rule hex-float
 (seq (byte 48) (one-of-bytes "xX")
  (choice
   (seq (repeat 1 (ref hex-digit))
    (optional (seq (byte 46) (repeat 0 (ref hex-digit)))))
   (seq (byte 46) (repeat 1 (ref hex-digit))))
  (one-of-bytes "pP") (ref exponent-digits)
  (ref float-suffix)))
(rule number
 (seq (optional (one-of-bytes "+-"))
  (select-radix
   (ordered
    (when-prefix (seq (byte 48) (one-of-bytes "xX"))
     (choice (ref hex-float) (ref hex-integer)))
    (when-prefix (seq (byte 48) (one-of-bytes "bB")) (ref binary-integer))
    (when-prefix (seq (byte 48) (one-of-bytes "oO")) (ref octal-integer))
    (when (and (prefix (seq (byte 48) (range 49 55)))
           (not (after-decimal-digits (one-of-bytes ".eE"))))
     (ref legacy-octal))
    (otherwise (choice (ref decimal-float) (ref decimal-integer))))))
 (require (not-followed-by (ref identifier-rest)))
 (emit-by-form (integer lit-int) (float lit-float))
 (failure malformed))
(rule octal-escape (seq (byte 92) (repeat-between 1 3 (ref octal-digit)))
 (greedy true) (when-count 3 (require-first (range 48 51))))
(rule byte-escape
 (seq (byte 92) (one-of-bytes "xXuU") (repeat-between 1 2 (ref hex-digit))))
(rule c-escape
 (ordered (seq (byte 92) (ref simple-escape))
  (ref continuation) (ref octal-escape)
  (seq (byte 92) (one-of-bytes "xX") (repeat 1 (ref hex-digit)))
  (seq (byte 92) (byte 117) (repeat-exact 4 (ref hex-digit)))
  (seq (byte 92) (byte 85) (repeat-exact 8 (ref hex-digit)))))
(rule x2c-escape
 (ordered (seq (byte 92) (ref simple-escape))
  (ref continuation) (ref octal-escape) (ref byte-escape)))
(rule c-string
 (seq (byte 34)
```

```
  (repeat 0
   (choice (ref c-escape)
    (except (byte 0) (byte 10) (byte 13) (byte 34)
     (byte 92)))) (byte 34)) (on-nul incomplete)
 (on-raw-newline malformed) (emit lit-char*))
(rule c-character
 (seq (byte 39)
  (choice (ref c-escape)
   (except (byte 0) (byte 10) (byte 13) (byte 39)
    (byte 92))) (byte 39)) (failure malformed)
 (emit lit-char))
(rule quoted-symbol-spelling
 (seq (byte 34)
  (repeat 0 (choice (ref x2c-escape) (except (byte 0) (byte 34) (byte 92))))
  (byte 34)) (on-nul incomplete))
(rule symbol
 (ordered (seq (byte 60) (ref quoted-symbol-spelling) (byte 62))
  (seq (byte 60) (repeat 1 (except (byte 0) (byte 62) (ref white-byte)))
   (byte 62))) (on-nul incomplete) (emit lit-symbol))
(rule atom-escape (seq (byte 92) (except (byte 0))) (on-nul incomplete))
(rule atom
 (repeat 1
  (ordered (ref atom-escape)
   (except (byte 0) (ref atom-boundary) (bytes "//")
    (bytes "/*")))))
(rule collection-atom (ref atom)
 (stop-before-unescaped (ref collection-boundary))
 (emit lit-atom))
(rule set-atom
 (ordered (ref quoted-symbol-spelling)
  (repeat 0
   (ordered (ref atom-escape) (except (byte 0) (ref set-boundary) (bytes ">>"))))))
 (emit lit-atom))
(rule string-segment
 (repeat 1
  (ordered (seq (byte 92) (byte 36)) (ref x2c-escape)
   (bytes "$$") (except (byte 0) (byte 34) (byte 36) (byte 92))))
 (stop-before (choice (byte 34) (byte 36)))
 (on-nul malformed) (emit segment))
(rule common
 (ordered (ref whitespace) (ref preprocessor)
  (ref line-comment) (ref block-comment) (ref c-character)
  (when-prefix (ref digit) (ref number))
  (when-prefix (seq (byte 46) (ref digit))
   (ref number))))
(rule named-reference (seq (one-of-bytes "$@") (optional (ref identifier)))
 (emit-sigil) (emit-identifier ident) (keyword-lookup false))
(rule code-reference
 (ordered (seq (bytes "$(") (transition embedded-lisp))
  (seq (bytes "@(") (transition embedded-lisp-splice))
  (when-prefix (one-of-bytes "$@") (ref named-reference))))
(rule percent-forms
 (ordered (seq (bytes "%\"") (transition percent-string))
  (seq (bytes "%[") (transition percent-array))
  (seq (bytes "%<<") (transition percent-set))
  (when (not (predicate percent-is-operator))
   (ordered (seq (bytes "%(") (transition percent-list))
    (seq (bytes "%{") (transition percent-map))
    (seq (bytes "%!") (emit "%!")) (seq (bytes "%<") (fail malformed))))))
(rule angle-symbol (when (predicate can-start-symbol) (ref symbol))
 (failure malformed))
(rule c-tokens
 (ordered (ref c-string) (seq (ref identifier) (emit-keyword-or-ident))
  (longest-prefix operators (emit-spelling) (apply-transitions))))
(rule data-prefix
 (ordered
  (when-mode array map
   (seq (bytes "void") (not-followed-by (ref identifier-rest))
    (emit void))) (when-mode list (seq (bytes "?(") (transition capture)))
  (when-mode list
   (when-prefix
    (seq (byte 64)
     (lookahead (choice (byte 0) (byte 61) (byte 41) (ref white-byte))))
```

```
    (seq (byte 64) (optional (byte 61)) (emit lit-atom))))
  (when-mode list array map
   (when-prefix (byte 36)
    (ordered (seq (bytes "${") (transition data-code))
     (ref named-reference))))
  (when-mode list array map
   (when-prefix (byte 64)
    (ordered (seq (bytes "@{") (transition list-splice))
     (ref named-reference))))))
(rule lisp-reference (when-prefix (byte 36) (ref named-reference)))
(rule lisp-symbol (when (predicate lisp-angle-start) (ref symbol)))
(rule signed-number
 (when-prefix (seq (one-of-bytes "+-") (ref digit))
  (ref number)))
(rule data-punctuation
 (ordered (seq (one-of-bytes "()") (emit-spelling) (apply-transitions))
  (when-mode list array map (seq (one-of-bytes "\"{[") (apply-transitions)))
  (when-mode lisp macro-lisp (ref c-string))
  (when-mode array map
   (seq (one-of-bytes "]}:,") (emit-spelling)
    (apply-transitions))) (seq (one-of-bytes "\'`") (emit-spelling))
  (when-mode list lisp macro-lisp
   (seq (byte 44) (optional (byte 64)) (emit-spelling)))
  (ref lisp-symbol) (ref signed-number)))
(rule list-atom (ref atom) (emit lit-atom))
(rule lisp-atom (ref atom) (emit ident))
(rule symbol-set-tokens
 (ordered (seq (bytes ">>") (transition close-set))
  (seq (one-of-bytes "$@") (emit-spelling))
  (ref symbol) (ref set-atom)) (failure malformed))
(rule string-tokens
 (ordered (seq (bytes "${") (transition string-code))
  (when-prefix (byte 36) (ref named-reference))
  (seq (byte 34) (transition close-string))
  (ref string-segment))))
```

modes

```
(modes
 (mode x2c (ordered common code-reference percent-forms angle-symbol c-tokens))
 (mode x2c-par
  (ordered common code-reference percent-forms
   angle-symbol c-tokens))
 (mode array (ordered common data-prefix data-punctuation collection-atom))
 (mode map (ordered common data-prefix data-punctuation collection-atom))
 (mode list (ordered data-prefix data-punctuation common list-atom))
 (mode lisp (ordered lisp-reference data-punctuation common lisp-atom))
 (mode macro-lisp (ordered lisp-reference data-punctuation common lisp-atom))
 (mode symbol-set (ordered common symbol-set-tokens))
 (mode string (ordered string-tokens)))
```

transitions

```
(transitions
 (defaults (emit spelling) (width input) (unlisted-mode-action unchanged))
 (transition code-brace (from x2c x2c-par) (on "{") (push x2c))
 (transition close-code (from x2c x2c-par)
  (on "}") (pop when-depth-exceeds-one))
 (transition percent-list (from x2c x2c-par)
  (on "%(") (push list))
 (transition percent-array (from x2c x2c-par)
  (on "%[") (push array))
 (transition percent-map (from x2c x2c-par)
  (on "%{") (push map))
 (transition percent-set (from x2c x2c-par)
  (on "%<<") (push symbol-set))
 (transition percent-string (from x2c x2c-par)
  (on "%\"") (push string))
 (transition embedded-lisp (from x2c x2c-par)
  (on "$(") (push macro-lisp))
```

```
 (transition embedded-lisp-splice (from x2c x2c-par)
  (on "@(") (push macro-lisp))
 (transition code-parenthesis (from x2c-par)
  (on "(") (push x2c-par))
 (transition close-code-parenthesis (from x2c-par)
  (on ")") (pop))
 (transition nested-list (from list array map)
  (on "(") (push list))
 (transition nested-array (from list array map)
  (on "[") (emit "%[") (push array))
 (transition nested-map (from list array map)
  (on "{") (emit "%{") (push map))
 (transition nested-string (from list array map)
  (on "\"") (emit "%\"") (push string))
 (transition data-code (from list array map)
  (on "${") (push x2c))
 (transition list-splice (from list array map)
  (on "@{") (push x2c))
 (transition capture (from list) (on "?(")
  (push x2c-par)) (transition close-list (from list) (on ")") (pop))
 (transition close-array (from array) (on "]") (pop))
 (transition close-map (from map) (on "}") (pop))
 (transition close-set (from symbol-set) (on ">>") (pop))
 (transition string-code (from string) (on "${") (push x2c))
 (transition close-string (from string) (on "\"") (pop))
 (transition nested-lisp (from macro-lisp) (on "(") (push macro-lisp))
 (transition close-lisp (from macro-lisp) (on ")") (pop)))
```

predicates

```
(predicates
 (predicate percent-is-operator
  (and (not (predicate layout-statement-start))
   (previous-kind-in operand-end) (not (predicate closes-control))
   (not (predicate closes-statement-block))))
 (predicate can-start-symbol
  (or (predicate layout-statement-start) (not (previous-kind-in operand-end))
   (previous-ident "tag")
   (previous-sequence (kind-in operand-end)
    (ident "is") (optional (ident "not")))))
 (predicate layout-statement-start
  (and layout-enabled (last-stored-kind space)
   (last-stored-text-contains (byte 10)) (previous-significant-exists)
   (column-at-most first-significant-on-previous-token-physical-line)))
 (predicate closes-control
  (and (previous-kind ")")
   (before-matching-ordinary-group-kind-in "if"
    "while" "for" "switch")))
 (predicate closes-statement-block
  (and (previous-kind "}")
   (or (before-matching-ordinary-group-kind-in
        ";" ":" "{" "}" "]" "ident" "else" "do" "try"
        "finally" "defer")
    (before-matching-ordinary-group (kind ")")
     (before-matching-ordinary-group-kind-in "ident"
      "match" "if" "while" "for" "switch")))))
 (predicate lisp-angle-start
  (and (prefix (byte 60))
   (not (next-byte-in (byte 0) (byte 61) (ref white-byte)
         (one-of-bytes "()\'`,")))))
 (token-set operand-end ident lit-int lit-float
  lit-char* lit-char lit-atom lit-symbol ")"
  "]" "}" "\"" "++" "--")
 (token-set compiler-operand-end ident lit-int
  lit-float lit-char* lit-char lit-atom lit-symbol
  ")" "]")
 (token-set compiler-operand-start ident lit-int
  lit-float lit-char* lit-char lit-atom lit-symbol
  "(" "%(" "%[" "%{" "$(" "${" "$" "!" "-"
  "*" "&" "~" "++" "--"))
```

passes

```
(passes
 (pass module-collection
  (external "src/collect/collect.x; active file-scope module blocks before tokenizing")
  (splice member-text-in-order)
  (resolve under-containing-file-companion-directory)
  (recursive true) (word "module" ident))
 (pass pipeline
  (ordered frontend-shebang module-collection
   scan layout lexical-diagnostics conditionals
   keyword-retag parser))
 (pass frontend-shebang (external "src/frontend.x")
  (replace-with scripting-include) (preserve subsequent-line-numbers))
 (pass layout
  (enabled-by (extension ".xp")
   (directive-before-code "#pragma indent" trailing-whitespace-only))
  (state logical-lines bracket-depth pending-ternary indents closers enums)
  (depth-open (text-ends-in "([{") (except-kind lit-char* lit-char segment))
  (depth-close (kind-in ")" "]" "}"))
  (ordered (index-significant (exclude space comment) (retain-physical true))
   (split-lines (always directive)
    (otherwise
     (and physical-line-break depth-zero (not (leading "."))
      (or no-deeper-indent previous-nonternary-colon))))
   (check-tabs (immediately-preceding space)
    (after-last-lf) (except directive) (fail indent))
   (base-indent first-statement-column)
   (open-block (when final-nonternary-colon next-statement-deeper)
    (replace ":" "{") (except (depth-zero-word case default catch) (append "{"))
    (push next-statement-indent))
   (wrap-control-condition
    (last-depth-zero-word if while for foreach
     switch match) (exclude dotted-member) (unless already-parenthesized)
    (for written-parentheses)
    (inline-body (first-depth-zero-nonternary-colon)
     (stop-at-label) (remove-if control-header-or-preceding-else)))
   (select-closer
    (aggregate (contains struct union enum) (no ordinary-parameter-open))
    (default "}") (aggregate "};") (leading-typedef "}")
    (leading-static-typedef "}"))
   (bare-do (line "do:")
    (unless next-same-or-lower-line-is-same-indent-while-trailer)
    (remove "do"))
   (append-semicolon
    (unless directive final-semicolon enum-body
     whole-embedded-lisp whole-syntax-hole syntax-hole-after-close-paren
     decorator)) (decorator (leading "@" "$") (retag-first space))
   (indent-directive (retag comment))
   (dedent (append-stored-closers) (require remaining-level)
    (fail indent)) (eof close-all)) (external-details "lib/tokenizer.x:_Layout")
  (inserted-width 0) (reclassified-width preserve))
 (pass lexical-diagnostics
  (external "src/compiler/parsing/tokens.x:Compiler.tokenize"))
 (pass conditionals (external "src/preprocess.x:Compiler.scan_conditionals")
  (after scan) (hide inactive-branches) (retain conditional-structure))
 (pass keyword-retag
  (ordered
   (retag in ident
    (unless
     (and (previous-kind-in compiler-operand-end)
      (next-kind-in compiler-operand-start))))
   (retag match ident
    (unless (next-after-optional-parenthesized-group-kind-in case "{"))))))
```

keywords

```
(keywords (keyword "associated" associated)
 (keyword "auto" auto) (keyword "break" break)
 (keyword "case" case) (keyword "catch" catch)
 (keyword "char" char) (keyword "const" const)
 (keyword "continue" continue) (keyword "default" default)
```

```
 (keyword "defer" defer) (keyword "delegate" delegate)
 (keyword "do" do) (keyword "double" double)
 (keyword "else" else) (keyword "enum" enum)
 (keyword "extern" extern) (keyword "finally" finally)
 (keyword "float" float) (keyword "for" for)
 (keyword "goto" goto) (keyword "if" if) (keyword "import" import)
 (keyword "in" in) (keyword "inline" inline)
 (keyword "int" int) (keyword "long" long)
 (keyword "match" match) (keyword "protocol" protocol)
 (keyword "raise" raise) (keyword "register" register)
 (keyword "restrict" restrict) (keyword "return" return)
 (keyword "short" short) (keyword "signed" signed)
 (keyword "sizeof" sizeof) (keyword "static" static)
 (keyword "struct" struct) (keyword "switch" switch)
 (keyword "threaded" threaded) (keyword "try" try)
 (keyword "typedef" typedef) (keyword "union" union)
 (keyword "unsigned" unsigned) (keyword "void" void)
 (keyword "volatile" volatile) (keyword "while" while)
 (keyword "__inline" inline) (keyword "__inline__" inline)
 (keyword "__restrict" restrict) (keyword "__restrict__" restrict)
 (keyword "thread_local" threaded) (keyword "_Thread_local" threaded))
```

operators

```
(operators (selection longest-prefix) (operator "...")
 (operator ">>=") (operator "<<=") (operator "===")
 (operator "!==") (operator "->") (operator "++")
 (operator "--") (operator "<<") (operator ">>")
 (operator "<=") (operator ">=") (operator "==")
 (operator "!=") (operator "&&") (operator "||")
 (operator "+=") (operator "-=") (operator "*=")
 (operator "/=") (operator "%=") (operator "&=")
 (operator "^=") (operator "|=") (operator "-")
 (operator ",") (operator ";") (operator ":")
 (operator "!") (operator "?") (operator ".")
 (operator "(") (operator ")") (operator "[")
 (operator "]") (operator "{") (operator "}")
 (operator "*") (operator "/") (operator "&")
 (operator "%") (operator "^") (operator "+")
 (operator "<") (operator "=") (operator ">")
 (operator "|") (operator "~") (operator "@"))
```

failures

```
(failures
 (scanner-result (positive matched-byte-count)
  (zero no-match) (negative failure))
 (unsupported-start (scan_keyword -1) (scan_c_operator -1))
 (empty-input (scan_white_space -1)) (status-aware ok incomplete malformed)
 (statusless-failure malformed) (record-first-status)
 (append (token error (width 0)) (token eof (width 0)))
 (callback-result 0) (dispatch-may-continue true)
 (semantic-stop first-eof) (unclosed-mode-at-eof parser-responsibility)
 (layout-status indent) (prefix-preconditions external-error)
 (numeric-signed-invalid-prefix may-return-zero)
 (legacy-number-type-helper skip-minus-only))
```

sources

```
(sources (source "lib/scan.x") (source "lib/tokenizer.x")
 (source "src/compiler/parsing/tokens.x")
 (source "src/literals/literals.x") (source "docs/src/reference/syntax/lexical.md"))
```

boundaries

```
(boundaries
 (external percent-string-decoding "src/literals/literals.x:_parse_text_segment"
  (remove continuations) (normalize cr-and-crlf-to-lf)
```

```
 (decode escapes)) (external symbol-representability "src/literals/literals.x")
(external host-preprocessing host-preprocessor)
(external native-c-acceptance host-compiler))
```

Command-line interface

After building the repository, `./x2c` links to the development compiler; `./bin/x2c` is the bootstrap compiler until `make build-install` links it to a copy of the development compiler. Every invocation begins with a command: `translate`, `build`, `run`, `new`, `script`, `env`, `install`, `remove`, `list`, or `help`.

The generated help is the short option reference:

```
./x2c --help
./x2c help translate
./x2c build --help
./x2c run --help
./x2c script --help
```

Translate to C

`translate` writes one matching `.c`, `.h`, `.xi`, and default `.d` file for each explicit `.x` input. The `.xi` file is the unit's interface, which later translations read in place of its source when it is current:

```
mkdir -p /tmp/x2c-generated
./x2c translate --out-dir /tmp/x2c-generated \
  src/main.x src/parse/parse.x
```

Without `--out-dir`, output goes to the current directory. An explicitly selected directory must already exist. x2c validates all inputs, output stems, and output paths before translating the first unit. Inputs must be regular `.x` files; two inputs with the same basename would collide in the flat output directory and are rejected.

`--out-dir` is only the generated-C directory. Native artifacts use `build --output` or `run --output`.

Several inputs translate at once, in worker processes that inherit the collected declarations the parent already read. `-j`, or `--jobs`, limits the number of workers:

```
./x2c translate -j 8 --out-dir /tmp/x2c-generated src/*.x
```

Each unit is translated by exactly one worker and writes only its own files, so the generated `.c` and `.h` are byte-identical to a serial run. A `.d` file may list more prerequisites than the serial run produces, and never fewer. A worker that parses a header records what that parse reads, where a serial run replays the header from its process cache. The default limit is the host's online processor count, or one job under Make, as for `build` below; `-j 1` translates serially in one process. The inspection and dump modes stay serial; they write one ordered stream to standard output.

A serial translation stops at the first unit that fails. In a parallel translation, a worker stops at the first failing unit of its share of the inputs while the other workers continue. Diagnostics appear in input order, because the command holds each worker's standard error until the worker ends; one unit's diagnostics are never split by another's. A parallel `build` translates each unit in its own worker, so it reports every failing unit. Stopping the command with `SIGTERM` or `SIGINT` also stops its workers.

Compile-time code runs once per translated unit. When several units import the same file, each worker runs that file's compile-time forms, including any side effects, for its own unit. A serial run may share one import between the units it translates.

The depfile options control that `.d` output. They belong to `translate`:

```
--no-deps               Do not write x2c dependency files
--dep-file <file>       Override the depfile path (one input only)
--dep-target <target>   Override the depfile target (one input only)
--no-phony-deps         Omit phony rules for included files
```

Symbol collection options apply to `translate`, `build`, and `run`:

```
--no-cpp                Skip symbol collection and preprocessing
--live-symbols          Collect symbols through the host preprocessor
--cpp-symbols           Use CPP collection for this translation
```

Source mapping

`--source-map` adds source locations to generated C for `translate`, `build`, and `run`. It is off by default. Combine it with native debug information to set breakpoints, step through statements, and read backtraces in the original `.x` files:

```
./x2c build --source-map -g -O0 --save-temps --output /tmp/example example.x
lldb /tmp/example
```

With source mapping enabled, C `__FILE__` and `__LINE__` refer to the original x2c source. Macro expansions use their invocation locations; generated cleanup uses its owning source construct. Compiler scaffolding without a source origin retains a generated-file location. `-g` alone keeps generated-C locations, and `--source-map` alone does not add native debug information. Changing the option invalidates reused translation output. Optimized builds may combine or remove statements, so source mapping does not guarantee exact stepping or recovery of optimized-away values.

On macOS, mapped executable builds with effective native debug information also produce `<output>.dSYM` before removing temporary objects. Move that companion directory with the executable to retain native debug information. Symbol assembly adds time only to these requested debug links. Its failure fails the build and retains intermediates for diagnosis. A later `-Xcc -g0` overrides an earlier `-g`, including one selected by a project profile. Linux keeps its existing debug output; `--build-dir` and `--save-temps` remain available for retaining intermediate files on either platform.

Inspection modes print an intermediate result and stop translation:

```
./x2c translate --dump-tokens source.x
```

- `--dump-tokens` prints source tokens and stops.
- `--dump-ast` prints the parsed, bound AST after macro expansion and stops.
- `--dump-transforms` prints the transformed AST and stops.
- Both AST dumps number bindings from 1 in order of first appearance, so a dump does not depend on how many bindings collection issued.
- `--dump-definitions` prints each definition's location and documentation and stops.
- `--dump-code` prints unformatted generated code and stops.
- `--dump-symbols` prints the source symbol table and stops.
- `--dump-cpp` prints host-preprocessed text and stops.
- `--dump-cpp-text` is an alias for `--dump-cpp`.
- `--dump-cpp-tokens` prints host-preprocessed tokens and stops.
- `--dump-cpp-symbols` prints the CPP symbol table and stops.
- `--dump-cache` prints the compiler cache and stops.
- `--dump-conformance` prints protocol conformance and stops.

`--dump-definitions` prints what the unit defines after translation has selected, expanded, and bound it, one Lisp form per line. Documentation generators and repository tools read it instead of scanning source text. Each unit starts with `(unit PATH)`, so one command can inspect several files. A file that opens with a block comment, after any directives, then prints `(module TEXT)` with the comment as written. Each function, including static functions, foreign aliases, and functions a Unit macro or `class` produced, prints a `function` row, and each `typedef` or `class` prints a `type` row:

```
(function (name "Point_add") (display "Point.add") (line 11)
 (source "point.x" 11)
 (span 193 242) (static 0) (origin source) (doc " Adds. ")
 (text "int Point.add(Point p, int k)")
 (type ((func (("Point") (int))) int)) (params ("p" "k")))
(type (name "Point") (kind struct) (line 6) (span 68 113) (private 0)
 (source "point.x" 6)
 (doc " A point. ") (text "typedef struct Point { int x; int y; } Point"))
```

- `name` is the C name and `display` the source spelling, such as `Point.add`.
- `line` is the one-based line of a function's declarator or a type's statement. A decorated function reports its declarator, and a function a macro produced reports the invocation. This line is in the expanded unit.

- `source` holds the authored path and one-based line, resolving module members and preprocessor line markers. It is empty for a definition without an authored source location. Documentation uses this location; `line` and `span` retain their positions in the expanded unit.
- `span` holds the byte offsets that start and end the top-level form that produced the definition, including decorators and a `meta` keyword. It is empty for a definition the compiler made itself, such as a protocol adapter, whose `line` is 1.
- `origin` is `source` for a function written in the file, `macro` for one a Unit macro, `class`, or the compiler produced, and `alias` for a foreign alias, whose `line` is that of its alias application.
- `doc` is the body of the `/**` comment that ends on the line before the form or on its first line, without its delimiters; it is empty when there is none. In a Unit macro's template, `$name` in that comment reads as the hole's argument as the invocation wrote it, so `$type` documents `$scalar(char, ...)` as `char`.
- `text` is a written function's declarator, or a type's statement without its `;`, with comments removed and each run of whitespace written as one space. It is empty for a function the file does not spell.
- `type` and `params` give a function's canonical type and parameter names, as its unit interface records them.
- `kind` is `alias`, `struct`, `union`, `enum`, `callback`, or `class`, and `private` is 1 for a type declared with `static`.

Compile-time-only `meta` functions are not part of the translated unit and are not listed.

An inspection that does not write generated files does not need `--out-dir`. Only one inspection mode runs; the last option given wins.

Build native artifacts

`build` accepts explicit `.x`, `.c`, `.o`, and `.a` operands. It translates x2c source, compiles generated and native C, then links an executable. One input may include another through a directory, as in `#include "lib/inner.x"`, when both are operands:

```
./x2c build --output /tmp/foreach examples/foreach.x
/tmp/foreach
```

`translate`, `build`, and `run` default to the host's online processor count for translation and native compilation. Detection failure falls back to one job. Under Make (`MAKELEVEL` greater than zero), the automatic default is one job so the outer `make` controls parallelism. An explicit `-j N` or `--jobs N` overrides either default. The online count does not account for Linux CPU affinity, container quotas, or memory limits; use `-j` to choose a smaller limit in those environments.

Parallel builds translate each stale unit in its own worker and generated directory. Units with the same basename remain separate, and each unit records its own prerequisites for subsequent build reuse. Parallel translation preserves native header search and link order.

The driver selects the runtime that matches the compiler and supplies that runtime's platform libraries. It also derives one object and dependency path per C source. `--kind static-library` uses the selected archiver:

```
./x2c build --kind static-library \
  --output /tmp/libwidget.a src/widget.x src/helper.c
```

`--kind meta-module` builds a native module, whose functions compile-time code can call. The module leaves the x2c runtime unresolved and uses the loading compiler's copy, which is the whole runtime, and it records that compiler's content hash. `--native-module <file>` loads a module for `translate`, `build`, `run`, and `repl`, and repeats:

```
./x2c build --kind meta-module --output helpers.so src/helpers.x
./x2c run --native-module helpers.so src/helpers.x src/main.x
```

Only the compiler that built a module loads it; any other compiler reports that the module was built by another compiler before running any of its code. The compiler loads a module only on request and keeps it loaded until it exits, and a translation binds only the modules its request names. When two of them define one name, the first supplies it and the declaration reports a warning. Modules work on macOS, Linux and WSL.

`--extension <dir>` links a package's compile-time part into the executable being built, which must be the compiler, and repeats. The package's `src/*.x` and `src/*.c` become inputs, and a generated unit registers its `meta`

functions under the package's name. The resulting compiler imports the package without loading its module. From a built compiler checkout, include every compiler unit and both header search roots:

```
python3 tools/sources.py |
  xargs ./x2c build --extension packages/tally \
    --x-include-dir src --c-include-dir builds/0/src --output /tmp/x2c
```

Native modules explains what a module contains.

`-c`, or `--compile-only`, stops after compilation. One input may name its object with `--output`. Multiple compile-only inputs require `--build-dir`, so every object has an unambiguous path.

Without `--build-dir`, a direct build uses a private temporary directory and removes intermediates after success. `--save-temps` retains intermediates; `--save-temps=<dir>` places them in a chosen directory. With a named build directory, later builds can reuse unchanged results.

`--compile-commands <file>` writes a native compilation database for C tools. It records the actual compiler argument arrays, working directory, source, and object output for every compilation, including reused objects. Manifest builds include the selected target and its dependency targets. The entries refer to generated C for x2c inputs, so the option also retains intermediates.

```
./x2c build --build-dir .x2c-build \
  --compile-commands .x2c-build/compile_commands.json examples/foreach.x
```

The destination's parent directory must exist when the build finishes. A successful build replaces the database atomically; a failed build preserves the previous file. `run` writes it before starting the program, and `-###` writes no database. This file describes native C compilation, not x2c syntax for an editor's C parser.

VS Code diagnostics, definitions, and hover

Install the VSIX built from `etc/vsc-extension` with **Extensions: Install from VSIX**. The extension uses the installed compiler directly; no separate worker build is required. Set an absolute compiler path when it is not on `PATH`:

```
"x2c.semantic.compilerPath": "/path/to/x2c/bin/x2c"
```

Selection order is an explicit `x2c.semantic.compilerPath`, an explicit legacy `x2c.semantic.workerPath`, executable `./x2c` in the trusted workspace, then `x2c` on `PATH`. An explicit selection never silently switches to another compiler. A missing executable produces one setup message in the **x2c** Output channel, and discovery retries when you use the editor again. The extension does not build or install the compiler automatically.

The default configuration discovers `x2c.toml` and its default target, or analyzes the current file directly when there is no project. Select a target or profile with ordinary compiler arguments:

```
"x2c.semantic.arguments": ["build", "--target", "app", "--profile", "debug"]
```

The selected target must include the open source as an input. For a header or macro file outside that input list, open its owning source or configure a `translate` command with the needed include and package options. Unsaved included files still affect analysis of their owning source.

Diagnostics, definition navigation, and type hover include unsaved source text. Each request runs in a fresh `x2c-editor` process with snapshots of open files; editing cancels obsolete results without writing buffers to disk. Native CPP symbol modes cannot consume unsaved source snapshots and report that limitation. Macro-generated syntax may lack a physical definition or identifier location.

Semantic execution requires a trusted local workspace because compilation can run macros. Untrusted and virtual workspaces retain syntax highlighting. Set `"x2c.semantic.enabled": false` to disable semantic execution. Completion, rename, workspace indexing, and general LSP support are separate features. The shipped `x2c-editor` command serves these providers through `x2c editor`. Its argument and response formats are not a public compiler API.

Shared libraries are unsupported. Supporting them requires platform-specific position-independent code, visibility, runtime linkage, library naming, and initialization rules.

Build and run

`run` builds an executable and launches it after the build succeeds:

```
./x2c run examples/foreach.x -- first -second input.x
```

Use `-q` to suppress build progress and receipts while preserving program output. Use `--build-dir <dir>` to retain generated files and objects for inspection or reuse. Otherwise temporary intermediates are removed after the program finishes.

`--` ends build options. Every later value is passed as one program argument, including values that begin with `-` or end in `.x`. x2c returns the program's exit status. `-###` prints the build and run actions without creating or launching anything. It reads no input file either, so it also prints the actions for an artifact a planned target would produce.

The program inherits standard input, output, and error, so interactive prompts and terminal applications work as they do when launched directly. Shell pipes and redirections also apply to the program; its output is not held until exit.

Run a script

`script` runs one source file and builds it only when needed. The file name ends in `.x`, or may be any name when the first line is a shebang:

```
./x2c script tools/report.x -- first --second
```

The first run builds an executable into the per-user cache. Later runs start that executable directly, in a few milliseconds, until one of these changes: the script, a file it includes or imports, a header its generated C includes, a package archive it links, the compiler, the runtime archive, the C compiler, the script's options, or the `CPATH`, `C_INCLUDE_PATH`, `LIBRARY_PATH`, and `SDKROOT` environment variables. The cache also records the directories the build searched: include directories, `-L` directories, the C compiler's header and library search paths, and the directory of each file the build read. Adding or removing a file in one of them rebuilds the script. This covers a header that now shadows an included one, a `__has_include` whose result changed, and a `-l` library that now resolves elsewhere. `--rebuild` builds regardless.

Options come before the script file, and every word after it is passed to the program unchanged, including `--` and words that begin with `-` or `@`. A response file given before the script supplies options. `script` accepts the include options `-I`, `--x-include-dir`, `--c-include-dir`, `--c-system-dir`, and `--package-dir`; the C compiler and linker options of `run`; and `-j`, `--source-map`, `--rebuild`, `--clean`, `-v`, `-###`, `--plain`, `--color`, `--debug`, `--max-errors`, and `--diagnostics-file`. `x2c help script` lists them. It does not read a project manifest.

A successful build prints nothing, so the program's output is all that appears; build diagnostics still print, and a failed build exits with its status. `-v` shows the build actions and the final run action. The program replaces the `x2c` process, so it receives signals and terminal input directly and its exit status is the command's. Its `argv[0]` is the script's absolute path. A `--source-map -g` build on macOS keeps its `run.dSYM` beside the cached executable.

A script file can start with a shebang line and run directly:

```
#!/usr/bin/env -S x2c script
```

That line also makes the file a script unit, which may put its statements at file scope instead of defining `main`.

A script can split its code into local modules with ordinary includes, such as `#include "lib/report.x"`. `x2c script` translates, compiles, and links every local `.x` file the script includes, directly or through another module, and a change to any of them rebuilds the script. Runtime and package modules come from their archives instead.

`env -S` splits the line into words. Where `x2c` has a fixed location, the interpreter path works without it: `#!/usr/local/bin/x2c script`.

Each script builds under `scripts/` in the cache root: `X2C_CACHE_DIR` when set, otherwise `$XDG_CACHE_HOME/x2c`, otherwise `~/.cache/x2c`. `x2c env cache_dir` prints the root. Concurrent runs of one script share its build and wait for each other. `x2c script --clean <file>` removes that script's entry without running it, and every build removes the entries of scripts that no longer exist. Removing the cache is always safe.

Selecting inputs

Shell wildcards work because the shell expands them into explicit operands:

```
./x2c build --output /tmp/tool src/*.x
```

x2c does not expand wildcard operand text or recurse through directory operands. Naming a directory does not select the sources under it.

Response files hold long explicit command lines:

```
build
--output
build/tool
-I
include
src/main.x
src/parse/parse.x
```

Invoke one as `./x2c @build.rsp`. Whitespace separates arguments; single and double quotes preserve whitespace; backslash quotes the next character. A first-non-whitespace `#` begins a comment. Response files may include other response files, but cycles are rejected. `@@name` passes the literal operand `@name`.

Start a project

`new` invokes the shipped `x2c-new` command to create a project that builds and runs as written:

```
x2c new hello
cd hello
x2c run -q            # Hello, world!
x2c run -q -- Gary    # Hello, Gary!
```

It writes `x2c.toml`, `src/main.x`, and `.gitignore`, which ignores the default build directory `.x2c-build/`. The manifest has one executable target named after the directory:

```
[target.hello]
sources = ["src/*.x"]
```

The directory may be missing or empty; `x2c new .` fills the current empty directory. `new` exits with an error for any other existing path and never overwrites a file. The target name is the directory's last component and may contain letters, digits, _, and `-`. `-q` suppresses the `x2c: created <dir>` line.

Project manifests

When `build` or `run` has no explicit input operands, x2c searches the current directory and its parents for the nearest `x2c.toml`. `--manifest-path` selects one exact file and disables discovery. Explicit input operands do not use a nearby manifest.

```
[project]
name = "demo"
default-target = "demo"
build-dir = "build/x2c"

[target.core]
kind = "static-library"
sources = ["src/core/*.x"]
exclude = ["src/core/experimental.x"]
include-dirs = ["include"]

[target.demo]
sources = ["src/main.x", "src/commands/**/*.x"]
dependencies = ["core"]
libraries = ["m"]

[target.demo.profile.release]
optimization = "O2"
debug = false
```

The manifest lists the sources of each target and how the targets depend on each other. Relative paths start at the manifest directory. Its `*`, `?`, and bracket patterns stay within a path component; `**` recurses. As in a shell, a name that begins with a dot matches only where the pattern has a literal dot. Matches are deduplicated and bytewise sorted. Unmatched patterns, unknown targets, and dependency cycles are errors before any action runs.

A target may also set defines, C flags, library directories, libraries, link flags, package directories, and an output. `kind = "meta-module"` builds a native module, and a target's `native-modules` field lists the module targets its translation loads. Those build first, like `dependencies`, and a changed module retranslates the targets that load it. Command-line target, profile, kind, output, build directory, and tool options override the corresponding defaults. `--target <name>` builds the named manifest target and `--profile <name>` applies the named manifest build profile; both work with `build` and `run`. The selected target must define the named profile; a dependency target takes it when it defines one and builds with its own settings otherwise. `--manifest-path`, `--target`, and `--profile` describe a manifest build, so each conflicts with explicit input operands, and `-c` needs operands rather than a manifest.

Pinned packages

A `[dependencies]` section pins packages by exact version. Each key is a package name in the index and each value is the version in that index row:

```
[dependencies]
pcre2 = "10.48"
```

This is separate from a target's `dependencies` field, which lists other targets in the same manifest.

`build` and `run` resolve the section once the manifest and the selected target are known good, and before planning. A pinned package missing from the x2c home at that version is installed through the index, the same way `x2c install <name>` does, and `--index` selects another index. The resolution is then written to `x2c.lock` beside the manifest, one `name version kind platform url sha256` row per package. Keep that file with the manifest so a later build reproduces the same packages. A manifest with no `[dependencies]` section pins nothing, so a lockfile left beside it is removed.

A build whose lockfile already covers every pinned package, at the pinned version and installed in the home, reads no index and makes no network request. A pin the lockfile covers but the home does not hold is installed from the url and sha256 that lockfile recorded, not from the index, so the archive is the one the lockfile pinned however the index has changed. Only a pin the lockfile does not cover reaches the index. The lockfile is then rewritten to the manifest's pins. A version the index cannot supply stops the build with an error that includes both versions. A dry run installs nothing and writes no lockfile, and the editor adapter never installs.

Include and package directories

`-I <dir>` is the shared include option. Repeated paths keep command order, and the first matching path wins in x2c quote-include collection, any requested x2c host preprocessing, and generated C compilation.

The exceptions are:

```
--x-include-dir <dir>  x2c collection and x2c preprocessing only
--c-include-dir <dir>  C compilation only, ordinary include semantics
--c-system-dir <dir>   C compilation only, system-header semantics
```

`--package-dir <dir>` adds a directory of x2c packages. An `import` statement searches those directories for a package that is not beside the source. The option applies to `translate`, `build`, and `run`, and repeats like `-I`. After the explicit and manifest directories, every command also searches `<home>/packages` when that directory exists.

Explicit `--package-dir` directories, or a target's `package-dirs` in `x2c.toml`, also select ownership when compiling package sources. Sources under `<dir>/<name>/src/` compile with the `<name>__` public-name prefix. Automatic home search resolves imports; source compilation uses explicit package directories. Compile providers for direct `#include` as ordinary sources with flat names, and consume package-compiled providers through `import`.

Packages

`install`, `remove`, and `list` dispatch to the shipped package command. The driver expands response files once. The package command handles their options and help with the legacy CLI parser. The equivalent `x2c package install`, `x2c package remove`, and `x2c package list` forms pass arguments literally, like other external commands.

`install`, `remove`, and `list` manage `<home>/packages`; see install packages. `install` takes one operand: a local directory, a local `.tar.gz`, a URL with `--sha256 <hex>`, or a name resolved through the index (`--index <url-or-path>` overrides the default). `--force` accepts a bundle built by another x2c version. `remove` takes one installed name; `list` prints `name version kind` lines on standard output, and the install and removal receipts go to standard error with the other action output. Refusals, including a package whose sources or install marker are damaged, exit with status 2 and leave the installed set as it was. One install or removal in a home runs at a time; the lock is released once the packages are in place, so a `run` that installed a dependency does not hold it while the program runs.

A project manifest can pin packages instead of installing them by hand; see pinned packages.

Environment

`x2c env` prints the resolved home, executable, include directory, runtime archive, prelude interface, package roots, C compiler, archiver, and script cache root as `name = value` lines. It also reports `libexec`, the directory for external commands, and `identity`, the compiler executable's identity. The prelude is the runtime `x2c.xi` interface the compiler replays for the runtime declarations. Only an interface the same compiler wrote qualifies; an empty value means the compiler reads `lib/x2c.x` from source once per process. `x2c env <name>` prints one value; `--package-dir`, `--cc`, and `--ar` show their effect on the report.

```
x2c env
x2c env runtime_lib
```

The home is `X2C_HOME` when set. Otherwise the compiler walks up from its executable, then from the current directory, to the nearest directory holding `include/` and `etc/compiler-sdk.xlisp`. Either way the home is reported and compared as an absolute path with symbolic links resolved, so a relative or symlinked `X2C_HOME` selects the same stage, prelude, and runtime archive as its real path. A source checkout and an installed prefix are both homes. A stage compiler under `<home>/builds/<n>/` links that stage's runtime archive. Any other compiler links `<home>/lib/libx2c.a` when it exists, and otherwise `<home>/builds/0/libx2c.a`, the stage 0 runtime whose headers a source checkout's `include/x2c` names.

There is no lowercase `-i`. `-D` and `-U` reach requested x2c preprocessing and C compilation. Optimization, debug, and `-Xcc` are compile-only; `-L`, `-l`, `--rpath`, `-Wl,`, `-Xlinker`, and `-framework` are link-only. `-pthread` reaches both the compile and the link. Unknown options are rejected. `--rpath <dir>` records `<dir>` in the program as a place to find shared libraries when it runs; a package bundle may include it. `-framework <name>` links a native framework on macOS.

The C compiler selection order is `--cc`, `X2C_CC`, `CC`, the installed toolchain record, then `cc`. The archiver follows `--ar`, `X2C_AR`, `AR`, the installed record, then `ar`. Project `meta` code runs inside the compiler's host, so it is built by `--meta-cc`, `X2C_META_CC`, `META_CC`, the installed record's `META_CC`, then `cc`, never by the target `--cc`; `translate` accepts `--meta-cc` too. `X2C_META_TIMEOUT` limits one `$` call in the project meta helper to that many seconds, 60 by default; `0` removes the limit. A call past the limit stops the helper and is reported at the call. Omitted optimization, debug, define, and undefine options preserve host defaults. `CFLAGS` and `LDFLAGS` are not shell-split or implicitly consumed. For every C compilation it runs, x2c selects signed plain `char`. The runtime requires an eight-bit signed `char`, even on hosts whose C compiler defaults to unsigned plain `char`.

Commands

The built-in commands are listed by `x2c help`. External commands are separate programs named `x2c-<name>` in the directory printed by `x2c env libexec`. For the stage 0 checkout compiler this is `builds/0/libexec`. A

development install in `bin/` and an installed prefix use `<home>/libexec/x2c`. The driver does not search `PATH` for them. `x2c help` lists their summaries from that directory's `commands.txt`.

The driver passes arguments after an external command name directly to its program, including arguments beginning with `@`. The external command owns its options and help, so `x2c help <name>` and `x2c <name> --help` run the program's `--help`. Built-in commands retain response-file expansion.

`x2c lint` reports an unreadable input's path and system error on standard error and exits `1`, while checking the other readable inputs. This also applies when checking formatting or showing formatting differences. The lint command treats `-` as an ordinary file name in these modes.

Compiler diagnostics

Translation reports each diagnostic on standard error as `file:line:column: code: message`, followed by the source line, a caret, and any notes. A unit stops after 20 errors. `--max-errors <count>` changes that bound for `translate`, `build`, `run`, and `script`, and `--max-errors 0` removes it. When the bound is greater than one, a unit that reaches it ends with `limit: too many errors, stopping`. With `--max-errors 1`, the unit's output ends after its first error. The diagnostics section of the language reference describes which errors are reported together.

Warnings

Ordinary translation reports warnings without stopping the build. Warnings do not count against `--max-errors`. The internal self-host stage option `--fatal-warnings` makes warnings fail those builds; it is not a public per-warning suppression or escalation control. A warning reports code that may be deliberate; leave the code unchanged when it is.

Eight codes are reported as warnings:

- `region` says a value allocated inside a region can still be reached after the region ends, and names how it leaves. Its note gives the line that opened the region.
- `after-free` says a local was read after `Scope.free`, `Scope.realloc`, or `Array.list_free` consumed it.
- `bad-free` says `Scope.free` or `Scope.realloc` was given a literal, a local's address, or other storage no Scope allocator returned.
- `literal` says a bare word inside a macro is a `Symbol` where a value was probably meant, and shows the unquoted spelling that inserts the value.
- `macro` carries what a macro reported through `x2c.diagnostic.warn`, at the invocation, with the notes the macro supplied.
- `conversion` says a cast or explicit converter call spells a conversion its destination already performs.
- `shadow` says a protocol binder shadows a visible type name.
- `native` says a native `meta` prototype names a function that both the compiler and a native module define, or that more than one native module defines.

The Region Model explains what the first three cover and what they do not detect. Scopes and Lifetime shows the patterns that avoid them.

`--diagnostics-file <file>` writes the same diagnostics to `<file>` as JSON Lines instead of standard error:

```
./x2c build --diagnostics-file /tmp/app.jsonl --output /tmp/app app.x
```

x2c creates or truncates the file when the command starts, so a command with no diagnostics leaves it empty. Each line is one JSON object:

```
{"code":"type","message":"operator 'is' requires Var on the
left","severity":"error","file":"app.x","line":9,"column":16,"length":2,"position":120,"notes":
["operand type: (int)"]}
```

- `code` is the diagnostic category, such as `parse`, `type`, `macro`, `protocol`, `xform`, `region`, `warning`, or `limit`.
- `severity` is `error`, `warning`, or `note`. It records how the report was submitted, so each entry in a category with both severities, such as `region`, `literal`, or `macro`, has its own severity. The `limit` notice is a `note`.
- `file` is relative to the working directory for a source inside it and absolute otherwise. A pseudo-source such as `<stdin>` keeps its name.

- `line` and `column` are one-based, `length` is the token width in bytes, and `position` is its zero-based byte offset. All five location fields are `null` for a diagnostic without a location.
- `notes` holds the note line that standard error shows, the text notes joined with spaces, or nothing when the diagnostic has no notes.

Compile-time Lisp and macros may print to standard output and standard error, so neither stream contains JSON. Parallel translation workers share the file; each diagnostic is one appended write, so lines never interleave. A diagnostic is written when it is reported, and the file is complete when the command exits, including after a failed unit. Command-line, host preprocessor, C compiler, and linker failures are not compiler diagnostics and remain on standard error; the exit status reports the failure.

Progress and receipts

Commands report completed translation, C compilation, archive, and link phases on standard error. A final build receipt identifies the artifact, elapsed time, generated C and header size, host compiler and job count, intermediate directory, and output size. Manifest builds name each target. Warm builds mark reused translation, compilation, and static archives as up to date. Executables always relink so changed libraries and native linker inputs take effect.

On a capable terminal, work lasting at least 125 ms may also use one transient progress line. x2c does not enter raw mode, switch screens, or read terminal input, and Make recipes show it too. Redirected standard error also uses stable newline-delimited receipts and no automatic color.

`--plain` selects stable receipts without terminal rendering or color. `-q` and `--quiet` suppress successful progress and receipts without suppressing diagnostics or a run program's output. `--color=auto|always|never` controls color; `NO_COLOR` disables automatic color. `--debug` enables compiler debug logging; `translate`, `build`, and `run` accept it.

`--verbose` prints each command as it runs. `-###` prints the same commands without running them. Use them first when checking runtime selection, include order, or which phase a flag reached. Neither prints completion receipts into that output, and the inspection modes keep progress out of their stdout data.

Dependencies and external builds

Translation depfiles contain the primary source, transitive quote-includes, and the runtime, macro, Lisp, and embedded-text files that unit read. The native driver adds `-MMD -MP -MF -MT` per object and rejects conflicting dependency flags passed through `-Xcc`.

Persistent direct and manifest builds use x2c depfiles for translation reuse. Before reusing a native object, the selected C compiler preprocesses its source with the current native flags and environment. Changed header search results and `__has_include` conditions therefore rebuild the object; unchanged native input reuses it. C depfiles remain available to external build tools. Missing artifacts and missing, corrupt, or old private state are cache misses rather than project errors. Manifest comments and whitespace do not invalidate reuse; changed effective settings still rebuild the affected actions.

External Make builds are supported. They can call `translate --out-dir`, include its `.d` files, and set their own C compiler and linker flags. The A project can run its own native build instead of using the driver.

For example, from the repository directory you can translate an example and compile its C output yourself:

```
mkdir -p /tmp/x2c-example
./x2c translate --out-dir /tmp/x2c-example examples/foreach.x
cc -iquote include/x2c /tmp/x2c-example/foreach.c \
  builds/0/libx2c.a -lm -o /tmp/x2c-example/foreach
/tmp/x2c-example/foreach
```

Help, version, and status

`--help` and `--version` write to standard output and return `0`. Source, host-tool, and link failures return `1`. Invalid commands, options, response files, manifests, and preflight state return `2`. After a successful `run`, the program's

status is returned. Diagnostics and action output use standard error; requested dumps and program output use standard output.

Release Notes

Each section summarizes what changed for a user of the compiler, runtime, commands, and packages since the previous release. The installation chapter carries the upgrade steps.

0.15.1 (2026-10-17)

A patch release of 0.15.0. Existing installations upgrade in place; no packages need rebuilding.

- An uncaught error prints one line naming the error, the raising function and location, and its details, then aborts with the same status. The full logger event remains available with `X2C_DEBUG` or a `<log>` policy. See Handlers and policy.
- `$auto` on a local that is already managed is a compile-time error instead of a crash at scope exit.
- The REPL continues a list literal across lines, and code it loads boxes classes, expands its own meta functions, and calls protocol defaults as an ordinary build does.
- The book gains this release notes page, and the repository gains a security policy, a pull request template, and a bug report template.

0.15.0 (2026-10-17)

Releases ship for macOS arm64 and x86_64 and Linux aarch64 and x86_64, with the book as a PDF beside the archives. Read Upgrading from 0.14.0 before replacing an existing prefix.

Language

- `meta` functions run ordinary x2c during translation. A macro body calls one with `$helper(...)`, passes constants or captured syntax, and inserts the code it returns. `meta.x` supplies the `Code` and `Type` operations for inspecting and building code. See Meta Functions and Generating Code.
- Macro values: a macro can select, apply, and collect cases as values, and typed quotations with expression holes build code with its type.
- `Stmt` macros accept an expression body, and a macro call may begin with a dotted name.
- Module blocks split one translation unit across a directory. The new `module` storage class is visible in every member file; `static` stays private to its file.
- Indentation syntax: a `.xp` file, or `#pragma indent` in a `.x` file, groups statements by indentation without braces or semicolons. See Indentation Syntax.
- `raise` is a statement head, `catch` accepts patterns as codes, and a variable may supply a `raise` code.
- Optional reference parameters, GNU statement expressions, `threaded` thread-local storage, and the contextual `in` membership operator.
- `class` types gain `alloc` and a refusing `init`, and `init` takes constructor arguments.
- Twelve language features moved out of the compiler kernel into components under `src/components/`; their behavior is unchanged.

Library

- `lib/regex.x` adds regular expressions over the bytes of a String, and every `x2c script` can use them.
- `lib/diff.x` computes line differences between two texts.
- The runtime's canonical pool operations moved from `List` to `Pool`.
- `Var.truth` replaces `Var.truthy`; `List.reduce`, `Array.reduce`, and `Iter.reduce` are removed in favor of `Iter.foldl`.
- Compile-time Lisp gains tail calls and a bounded call depth.

Commands and packages

- `x2c new` creates a starter project that builds and runs.
- `x2c repl` is a shipped command with line editing, grammar-aware and semantic completion, and named inspection of session values. It evaluates a supported subset of the language. See Experimental REPL.

- Package management, the editor adapter, and project creation are external commands under `libexec/x2c/`. `x2c lint` and `x2c graph` are shipped as experimental.
- Packages record native arguments in `<name>.native.rsp` instead of `<name>.link`. Rebuild source packages and replace native bundles.
- Automatic differentiation moved from the runtime to the optional `autodiff` package; `import "autodiff"` supplies its types and macros.
- The SQLite package renames `Statement` to `Prepared` and `Var.statement` to `Var.prepared`.
- Every package manages its handles with `$auto`, so connections, loops, images, documents, and terminals release at scope exit.
- The C* verification package is removed.

Documentation

- The book adds the x2c Syntax Map for C readers, the syntax specifications with a source grammar and lexical rules, The Region Model, and chapters for meta functions, generating code, the REPL, and indentation.
- The language reference was corrected against the implementation and its repeated rules removed.
- The public site at x2c-lang.dev carries the book under `/docs/`.

Fixed defects

- A `return` inside a `defer` or `finally` body leaves its exception frame before transferring, so a later `raise` in the caller is caught.
- Macro bindings retained across package builds, nested empty unit macros, quotation members and splices, and template members in anonymous aggregates expand correctly.
- Parameter redeclaration inside callable bodies and the author-binding `extern` regression are fixed.
- Constructed raw string segments, nested string literals inside list literals, and negative percent literals in every radix are read as written.
- Pointer arithmetic over pointer typedefs and casts to template typedefs type correctly.
- Project meta payloads, include handling, the failure cache, and host flags are repaired for package builds.

Libraries and Packages

Standard Library

The overview introduces library operations by task. The module reference documents their signatures, behavior, and ownership rules in detail, including shipped optional modules.

For specialized applications built on x2c, see Advanced Topics.

Standard Library by Task

Every `.x` translation unit receives the x2c runtime prelude. Each section below names the type to use for one task. The generated module reference lists every signature and its behavior, grouped by API tier.

The implicit prelude is the standard library. Optional x2c system modules are shipped, built, and tested with x2c, and need an explicit include. Runtime internals are implementation details. Third-party libraries are under `packages/` and use `import`.

Hold a value whose type varies

Use Var when you do not know the type of a value until the program runs. Use the C type everywhere else.

```
Var value = 42;
value = "forty-two";

if (value is String)
  printf("%s\n", value);
```

Dynamic arithmetic and truthiness are in Var operations. Checked conversion is in Var conversion. `void` means missing or exhausted, and collections and iterators never carry it as a payload.

Choose a collection

Need	Type	Important contract
persistent heterogeneous sequence	List	canonical and immutable
growing indexed sequence	Array	mutable object identity
growing keyed lookup	Map	mutable object identity
canonical byte text	String	interned, NUL-terminated
streaming traversal	Iter	caller-owned iterator storage

`List` and `String` values are canonical. `Array` and `Map` are distinct allocated objects even when empty.

Build and transform text

String does interpolation, search, split, replace, partition, case conversion, padding, escaping, slicing, joining, and checked numeric parsing. Its operations work on bytes; they do not decode Unicode.

Use Buffer for incremental text:

```
Buffer output = $auto(Buffer.new(0));
output.write("count=");
output.printf("%d", 4);
String text = output;
printf("%s\n", text);
```

Use Symbol for compact immediate tags and Atom for exact names that may be longer.

Read or write a file

File wraps C streams. Iterate a text file by line:

```
foreach(String line, input)
  printf("%s", line);
```

Use `File.readline_into` and `File.read_into` for raw bytes with separate data, EOF, and error status. Use `File.write_all` and `File.copy_to` when a partial write must be finished or reported.

Transform a sequence

Every collection supports iteration. Use `foreach` to walk one:

```
List values = %(1 2 3);
foreach(int value, values)
  printf("%d\n", value);
```

Use Iter to process values as they are requested. Use a collection's `map`, `filter`, and fold operations to compute the result immediately. `Iter.try_next` separates a value from exhaustion and never yields `void` as data.

Match structured Lists

The language `match` statement and Match use the same `List` patterns:

```
List event = %(ready 42);
match (event) {
  case %(ready ?value):
    printf("%ld\n", value);
}
```

Use the `match` statement for control flow and `List.try_match` or the other runtime matching functions when the patterns are themselves data.

Own allocation lifetime

Scope groups runtime allocations:

```
$scope() {
  String text = "temporary";
  Array values = [text, 1, 2, 3];
  printf("%s\n", values.repr());
}
```

Use Block for fixed-width growable storage and `Buffer` for text construction. External resources such as `Files` still close explicitly or through `defer`.

Each `Scope.release` must match a `Scope.retain` on the same active slot. See scopes and lifetime.

Bound temporary work

Context combines a `Scope` lifetime with independent `Error`, cleanup, and `Match` state. An isolated `Context` also gives temporary `String` and `List` values private pool branches:

```
Context work = Context.open_isolated_named("request");
List temporary = %(answer 42);
List result = work.export(temporary);
work.close();

printf("%s\n", result.repr());
```

Closing reclaims everything that was not exported. Close `Contexts` in last-opened, first-closed order. See contexts and threads.

Run work on native threads

Thread runs a fixed-signature callback in an isolated `Context`. It copies the input structure, but pointers inside that structure still point into shared process memory. A worker can read parent `Strings` and `Lists` directly. Use Mutex when a shared value may be written:

```
typedef struct CounterWork {
  Mutex mutex;
  int *counter;
  String label;
} CounterWork;

static Var increment(const void *input, size_t input_size) {
  if (input_size != sizeof(CounterWork)) raise %(bad-arg);
  const CounterWork *work = input;
```

```
  $lock(work.mutex) (*work.counter)++;
  return work.label;
}

int main(void) {
  int counter = 0;
  Mutex mutex = Mutex.new();
  CounterWork work = { mutex, &counter, "parent label" };
  Thread first = Thread.start(increment, &work, sizeof(work));
  Thread second = Thread.start(increment, &work, sizeof(work));
  String a = first.join();
  String b = second.join();
  printf("%s, %s: %d\n", a, b, counter);
  first.free();
  second.free();
  mutex.free();
  return 0;
}
```

Every started `Thread` needs one successful `Thread.join`, then `Thread.free`. Contexts and threads gives the sharing and result rules.

Handle absence and failure

These operations report absence separately from data:

- `Map.try_get` and `Map.try_del` report whether the key was present, separately from its value.
- `Iter.try_next` separates a value from exhaustion.
- `String.try_long` and `String.try_double` separate zero from parse failure.
- raw `File` readers separate data, EOF, and error.

An operation that fails raises its cause through `Error`, and a filtered `catch` or the cause's policy handles it. Absence, exhaustion, and parse results come back as return values.

Use Error transfer and cleanup

Language `raise`, `try`, filtered `catch`, `finally`, and `defer` are backed by `Error` and the cleanup-frame runtime. Application code uses those statements and leaves the runtime frame functions alone. See Errors and Cleanup.

Bind native functions

Func binds an already-linked native function with a checked signature. Construction, binding, conversion, and application failures raise their cause through `Error`; none of those causes return to the call. A fixed nonvariadic function or function-pointer value with a supported signature converts implicitly where a `Func` is expected. Direct conversions reuse a file-static binding that lives for the program, so an operation that takes ownership of a `Func`, such as `Lisp.bind`, leaves it in place. Pointer conversions snapshot the pointer in a binding held by the current `Scope`. Call a C function directly when its name is known at compile time.

Log structured events

Logger does level filtering and writes to ordered stderr, file, memory, and custom sinks. A category labels an event; it never suppresses one. The global `log_*` functions are the simplest interface; use a `Logger` object to configure logging and manage its lifetime.

Embed Lisp

Lisp provides a reader, evaluator, lexical closures, macros, global bindings, native `Func` integration, host-side value application, and file/string evaluation:

```
static String greet(String name) {
  return %"hello $name";
}

Lisp lisp = Lisp.new();
defer lisp.destroy();
```

```
$lisp.bind(lisp, "greet", greet);
String result = lisp.eval(%(greet "Ada"));
```

`Lisp.new()` creates an isolated session and loads `etc/init.xlisp`. That file supplies `defun`, short-circuit control, higher-order `List` operations, numeric comparisons, basic `String` operations, and the pattern matching described in the language reference. `Lisp.kernel()` creates the evaluator primitives alone, for hosts that build their own environment. `etc/lisp-values.xlisp` binds the core `List`, `Array`, `Map`, `String`, `Var`, and `Symbol` operations; the compiler loads it into every compile-time session, and an embedding host loads it by name. `etc/lisp-io.xlisp` stays optional and must be loaded by name. It adds `read-file`, `write-file`, `sort`, `fib`, and `subst`.

Use `$lisp.binding(group, "name")` on several direct functions and `$lisp.install(lisp, group)` when a unit has a reusable set. Functions must appear before the install call, and names in one group must be unique. Installing the same group into two sessions creates two sets of `Func` values, and each session releases its own in `Lisp.destroy()`.

A group compiles into the unit that declares it. A package can therefore publish its bindings behind one public installer that a consumer calls after importing it. See Wrapping a C Library.

`$lisp.bind` and grouped bindings accept the parameter and result types `Func` supports, including untagged pointers and structs passed as `<p48>` addresses. Native functions are trusted host code. A package-specific opaque object needs its own `Var` adoption.

Load a Lisp program with `Lisp.eval_file`, then evaluate runtime `List` forms with `Lisp.eval`. `$` inside `%(...)` inserts live x2c values; it is unrelated to compile-time `$(...)` Lisp. Within a `List` literal, `'`, `` ` ``, `,`, and `,@` write `quote`, `quasiquote`, `unquote`, and `unquote-splicing` just as they do in Lisp source. A flat `List` result destructures through the same typed conversions as any other value:

```
List outcome = lisp.eval(%(decide (quote ((role $role)))));
String decision;
int limit;
(decision, limit) = outcome;
```

Look up an exact operation

The module reference is generated from runtime source and compiler-verified signatures.

Module Reference

Reference pages list the functions and types in each runtime module. Start with the standard library overview to choose a type or find a function for a particular task.

The x2c C API collects the stable `x2c_*` entry points used by generated code, native callers, and compiler setup.

Modules

Module	Description
`lib/args.x`	parse program arguments against a declarative spec.
`lib/array.x`	dynamic contiguous arrays of `Var` elements.
`lib/atom.x`	canonical exact names.
`lib/block.x`	checked dynamic storage for fixed-width elements.
`lib/buffer.x`	growable text buffer with indentation support.
`lib/common.x`	the `Var` encoding every runtime module shares.
`lib/context.x`	bounded runtime state and value export.
`lib/diff.x`	line differences between two texts.
`lib/digest.x`	SHA-256 digests of Strings and streams.
`lib/dispatch.x`	typed descriptors for runtime `Var` behavior.
`lib/error/error.x`	handler stack and accumulated errors.
`lib/exception.x`	transfer frames for x2c `Error` unwinding and cleanup.

Module	Description
lib/file.x	`File` I/O operations and stream handling.
lib/func.x	generic native function binding.
lib/iter.x	single-pass pull iterators.
lib/json.x	JSON text to and from ordinary x2c values.
lib/lib.x	disjoint sets with union-find.
lib/lisp/lisp.x	the Lisp runtime: reader, session, and evaluator.
lib/list-selectors.x	optional compound `List` selectors.
lib/list/list.x	linked list with `Var` elements.
lib/logger.x	owned structured event delivery.
lib/macrovalue/macrovalue.x	macros as values that build and recognize code.
lib/map.x	hash table mapping `Var` keys to `Var` values.
lib/match-cache.x	caches of prepared Match plans.
lib/match-plan.x	lowering Match patterns to prepared plans.
lib/match/match.x	pattern matching and transformation utilities for lists.
lib/meta.x	the compiler surface a `meta` function calls.
lib/mutex.x	shared mutable-state coordination.
lib/path.x	filesystem locations and the operations on them.
lib/pool/pool.x	nested interning pools with region-backed object storage.
lib/process.x	run commands and pipelines without a shell.
lib/regex.x	regular expressions over the bytes of a String.
lib/rewrite.x	Pattern registration for compiler rewrites.
lib/scope/scope.x	memory allocation scope management.
lib/scripting.x	the modules every script unit includes.
lib/split.x	`String` field splitting and repeatable typed cursors.
lib/string-classify.x	byte classification for canonical strings.
lib/string-escape.x	escaped spelling of canonical strings.
lib/string-format.x	checked formatting of `Var` values into a `String`.
lib/string-number.x	numeric parsing from canonical byte strings.
lib/string/string.x	canonical byte strings and core text operations.
lib/symbol.x	immediate encoded names.
lib/symbolset.x	immutable ordered sets of compact `Symbol`s.
lib/system-macros.x	convenience macros for text, timing, and processes.
lib/thread.x	`Context`-backed native worker threads.
lib/type.x	semantic type values and their structural operations.
lib/typed-array.x	packed typed `Array`s generated from shared methods.
lib/typed-list.x	typed cons chains generated from typed methods.
lib/typed-map.x	typed `Map`s generated from shared storage.
lib/var/var.x	variant type for dynamic typing.
lib/varconvert.x	`Var` numeric conversion policy.
lib/varops.x	boxed `Var` operators, updates, and truthiness.

Optional x2c modules

These modules ship with x2c but are not loaded by the standard prelude. Include one explicitly to use its declarations.

- `lib/args.x` - Argument parsing against a declarative spec; explicitly include `args.x` to use it.
- `lib/diff.x` - Line differences between two texts; explicitly include `diff.x` to use them.
- `lib/digest.x` - SHA-256 digests of `String`s and streams; explicitly include `digest.x` to use them.
- `lib/json.x` - JSON text to and from ordinary `Map`, `Array`, `String`, and number values; explicitly include `json.x` to use it.
- `lib/list-selectors.x` - Compound selectors beyond caar, cadr, cddr, and caddr; explicitly include `list-selectors.x` to use them.
- `lib/meta.x` - Syntax builders and compiler operations for `meta` functions. Pure builders also run at runtime; operations that query the compiler remain compile-time only. Every unit reaches it through the prelude's `varops.x`; include `meta.x` explicitly where `meta` functions are written.
- `lib/path.x` - Filesystem operations on path `String`s; explicitly include `path.x` to use them.
- `lib/process.x` - Commands, pipelines, and background jobs without a shell; explicitly include `process.x` to use them.
- `lib/regex.x` - Regular expressions over the bytes of a `String`; explicitly include `regex.x` to use them.
- `lib/rewrite.x` - Register macro-pattern translators with the $rewrite decorator; include `rewrite.x`.
- `lib/scripting.x` - The `args.x`, `diff.x`, `digest.x`, `path.x`, `process.x`, and `regex.x` modules that every script unit includes; include `scripting.x` to use all six.
- `lib/system-macros.x` - Compile-time convenience macros; explicitly include system-macros.x to use them.
- `lib/type.x` - Structural operations on captured Type values, available at runtime and compile time; include `type.x` to use them.
- `lib/typed-array.x` - Packed typed storage with a raw native bracket; explicitly include `typed-array.x` to use it.
- `lib/typed-list.x` - Typed views over canonical `List` cells; explicitly include `typed-list.x` to use them.
- `lib/typed-map.x` - Typed maps over native numeric or canonical `String` fields; explicitly include `typed-map.x` to use them.

Runtime-internal

These modules implement the runtime and are not public APIs.

- `lib/array-generics.x` - Shared declaration templates for typed array operations.
- `lib/cleanup.x` - Cleanup adoption templates used by runtime owners.
- `lib/clibc.x` - C library prototypes for integer and string conversions, and string comparison, marked `meta` so compile-time code calls the native functions; it includes the C headers it declares from.
- `lib/cmath.x` - C99 `<math.h>` prototypes, declared so a `Var` argument unboxes at a math call; the unit still includes `<math.h>`.
- `lib/datum.x` - Values spelled as Lisp reader text, for interface files and the project meta helper's messages.
- `lib/error-macros.x` - Structured Error causes shared by runtime modules.
- `lib/error-private.x` - Error transfer templates used by the exception machinery.
- `lib/error_init.x` - Type-owned initialization shim that preserves `Error`'s pre-initialization boundary.
- `lib/func-errors.x` - Structured callable errors shared by runtime modules.
- `lib/integer-ops.x` - Integer conversion and arithmetic templates shared by runtime owners.
- `lib/lisp-init.x` - The standard Lisp algorithms `etc/init.xlisp` binds by name into every `Lisp` session.
- `lib/lisp-targets.x` - The native operations a `Lisp` session can bind, built outside the implicit prelude so the optional modules that supply them stay out of it.
- `lib/machine.x` - `Match` wordcode, builder, and execution-state definitions.
- `lib/map-generics.x` - Shared declaration templates for typed map operations.
- `lib/match-machine.x` - Private `Match` wordcode decoder.
- `lib/native-scalar-types.x` - The exact C scalar ledger and its compile-time projections.
- `lib/private-keywords.x` - Keyword aliases used by runtime implementation code.
- `lib/protocols.x` - The built-in `Cleanup`, `Block`, `Iter`, and `Var` protocol declarations used to generate runtime dispatch adapters.
- `lib/scan.x` - Allocation-free character scanners used by the tokenizer.
- `lib/static-init.x` - Compiler-owned first-use static storage and initialization guards.

- `lib/thread-state.x` - Native pthread-key storage for per-thread runtime state.
- `lib/tokenizer.x` - Token stream used by the compiler and the Lisp reader.
- `lib/var-adapters.x` - Pointer converter templates shared by runtime owners.
- `lib/var-ledger.x` - The `Var` tag tables and decoder table projected from `lib/var-tags.x`, in the one runtime unit that includes the ledger.
- `lib/var-tags.x` - The Var tag ledger shared by runtime and compiler projections.

x2c C API

The `x2c_*` prefix is reserved for stable C entry points used by generated code, native callers, or process-wide compiler setup. Typed operations use methods such as `Atom.promote` and `Lisp.eval`; their generated C spellings are not part of this prefix.

Function	Source	Signature
x2c_cleanup_leave	lib/exception.x	void x2c_cleanup_leave(X2CCleanup *record)
x2c_cleanup_push	lib/exception.x	void x2c_cleanup_push(X2CCleanup *record)
x2c_descriptor_registration_frozen	lib/dispatch.x	int x2c_descriptor_registration_frozen(void)
x2c_descriptor_thread_start_begin	lib/dispatch.x	void x2c_descriptor_thread_start_begin(void)
x2c_descriptor_thread_start_end	lib/dispatch.x	void x2c_descriptor_thread_start_end(int success)
x2c_error_catch_capture	lib/error/error.x	Var x2c_error_catch_capture(ErrorHandler handle, int index)
x2c_error_catch_close	lib/error/error.x	void x2c_error_catch_close(ErrorHandler handle)
x2c_error_catch_detach	lib/error/error.x	void x2c_error_catch_detach(ErrorHandler handle)
x2c_error_catch_push	lib/error/error.x	ErrorHandler x2c_error_catch_push(void *target, unsigned arm_count, ...)
x2c_error_catch_selected	lib/error/error.x	int x2c_error_catch_selected(ErrorHandler handle)
x2c_error_catch_site_pending	lib/error/error.x	int x2c_error_catch_site_pending(ErrorCatchSite *site)
x2c_error_catch_site_push	lib/error/error.x	ErrorHandler x2c_error_catch_site_push(void *target, ErrorCatchSite *site, Var *patterns)
x2c_error_raise	lib/error/error.x	void x2c_error_raise(Symbol code, List detail)
x2c_error_raise_n	lib/error/error.x	void x2c_error_raise_n(const X2CErrorSite *site, Symbol code, unsigned pair_count, ...)
x2c_exception_claim	lib/exception.x	int x2c_exception_claim(ExceptionFrame *frame)
x2c_exception_is_error_target	lib/exception.x	int x2c_exception_is_error_target(ExceptionFrame *frame)
x2c_exception_landed	lib/exception.x	void x2c_exception_landed(ExceptionFrame *frame)

Function	Source	Signature
x2c_exception_leave	lib/exception.x	void x2c_exception_leave(ExceptionFrame *frame)
x2c_exception_mark_handled	lib/exception.x	void x2c_exception_mark_handled(ExceptionFrame *frame)
x2c_exception_push	lib/exception.x	void x2c_exception_push(ExceptionFrame *e)
x2c_exception_unwinding	lib/exception.x	int x2c_exception_unwinding(void)
x2c_func_declared_reference_argument	lib/func.x	void *x2c_func_declared_reference_argument(Func fn, const FuncArg *argv, unsigned i, List declared_target, List want)
x2c_func_pointer_argument	lib/func.x	void *x2c_func_pointer_argument(Func fn, const FuncArg *argv, unsigned i)
x2c_func_record_result	lib/func.x	Var x2c_func_record_result(const void *bytes, size_t size)
x2c_func_reference_argument	lib/func.x	void *x2c_func_reference_argument(Func fn, const FuncArg *argv, unsigned i, List want)
x2c_func_reference_type	lib/func.x	List x2c_func_reference_type(Func fn, unsigned argc, unsigned index)
x2c_func_shared	lib/func.x	Func x2c_func_shared(FuncAdapter adapter, List signature)
x2c_func_unrepresentable_argument	lib/func.x	FuncArg x2c_func_unrepresentable_argument(Func fn, unsigned i, List source)
x2c_func_value_argument	lib/func.x	Var x2c_func_value_argument(Func fn, const FuncArg *argv, unsigned i, Symbol want)
x2c_get_executable	src/utils.x	String x2c_get_executable(void)
x2c_get_root	src/utils.x	String x2c_get_root(void)
x2c_hash_bytes	lib/common.x	inline unsigned x2c_hash_bytes(unsigned long seed, const void *data, size_t width)
x2c_hash_word	lib/common.x	inline unsigned x2c_hash_word(unsigned long word)
x2c_initialize	lib/common.x	void x2c_initialize(void)
x2c_initialize_command_environment	src/utils.x	void x2c_initialize_command_environment(const char *argv0, String embedded_identity)
x2c_initialize_environment	src/utils.x	void x2c_initialize_environment(const char *argv0)
x2c_initialize_protocols	lib/common.x	void x2c_initialize_protocols(void)
x2c_match_initialize	lib/match/match.x	void x2c_match_initialize(void)
x2c_match_pattern_retainable	lib/match/match.x	int x2c_match_pattern_retainable(Var pattern)
x2c_match_site_match	lib/match/match.x	List x2c_match_site_match(MatchCaptureSite *site, List input, Var pat)
x2c_match_site_match_replace	lib/match/match.x	List x2c_match_site_match_replace(MatchCaptureSite *site, List input, Var pat, Var template)

Function	Source	Signature
x2c_match_site_prepare	lib/match/match.x	MatchPlan x2c_match_site_prepare(MatchCaptureSite *site, Var pattern)
x2c_match_site_search	lib/match/match.x	List x2c_match_site_search(MatchCaptureSite *site, List input, Var pat)
x2c_match_site_search_replace	lib/match/match.x	List x2c_match_site_search_replace(MatchCaptureSite *site, List input, Var pat, Var template)
x2c_match_site_try_capture	lib/match/match.x	int x2c_match_site_try_capture(MatchCaptureSite *site, List input, Var pattern, MatchCaptureBuffer *captures)
x2c_match_site_try_match	lib/match/match.x	int x2c_match_site_try_match(MatchCaptureSite *site, List input, Var pat, List *out_bindings)
x2c_match_site_try_match_replace	lib/match/match.x	int x2c_match_site_try_match_replace(MatchCaptureSite *site, List input, Var pat, Var template, Var *out)
x2c_match_site_try_search	lib/match/match.x	int x2c_match_site_try_search(MatchCaptureSite *site, List input, Var pat, Var *out_match, List *out_bindings)
x2c_match_thread_release	lib/match-cache.x	void x2c_match_thread_release(void)
x2c_match_try_capture	lib/match/match.x	int x2c_match_try_capture(List input, Var pattern, MatchCaptureBuffer *captures)
x2c_mix64	lib/common.x	inline unsigned long x2c_mix64(unsigned long word)
x2c_normalize_index	lib/common.x	int x2c_normalize_index(int index, int length)
x2c_normalize_slice	lib/common.x	int x2c_normalize_slice(int *start, int *stop, int step, int length)
x2c_register_builtin_descriptor	lib/dispatch.x	int x2c_register_builtin_descriptor(Symbol tag, VarMethods methods)
x2c_register_descriptor	lib/dispatch.x	void x2c_register_descriptor(String name, VarMethods methods)
x2c_register_tagged_descriptor	lib/dispatch.x	void x2c_register_tagged_descriptor(Symbol tag, String name, VarMethods methods)
x2c_register_type	lib/dispatch.x	void x2c_register_type(String name)
x2c_scope_thread_release	lib/scope/scope.x	void x2c_scope_thread_release(void)
x2c_set_root	src/utils.x	void x2c_set_root(String root)
x2c_slice_bound	lib/common.x	int x2c_slice_bound(int bound)
x2c_try_register_descriptor	lib/dispatch.x	int x2c_try_register_descriptor(String name, VarMethods methods)
x2c_try_register_tagged_descriptor	lib/dispatch.x	int x2c_try_register_tagged_descriptor(Symbol tag, String name, VarMethods methods)
x2c_var_custom_descriptor	lib/var/var.x	VarDescriptor *x2c_var_custom_descriptor(Var value)
x2c_var_declare	lib/var/var.x	VarDescriptor *x2c_var_declare(Symbol tag)
x2c_var_descriptor_index	lib/var/var.x	int x2c_var_descriptor_index(Var value)

Function	Source	Signature
x2c_var_equal	lib/varops.x	int x2c_var_equal(Var a, Var b)
x2c_var_identical	lib/varops.x	int x2c_var_identical(Var a, Var b)
x2c_var_order	lib/varops.x	double x2c_var_order(Var a, Var b)
x2c_var_postfix_volatile	lib/varops.x	Var x2c_var_postfix_volatile(volatile Var &?lhs, Symbol op)
x2c_var_tag_descriptor_index	lib/var/var.x	int x2c_var_tag_descriptor_index(Symbol tag)
x2c_var_update_f32	lib/varops.x	float x2c_var_update_f32(volatile float *lhs, Symbol op, Var rhs)
x2c_var_update_f64	lib/varops.x	double x2c_var_update_f64(volatile double *lhs, Symbol op, Var rhs)
x2c_var_update_i16	lib/varops.x	short x2c_var_update_i16(volatile short *lhs, Symbol op, Var rhs)
x2c_var_update_i32	lib/varops.x	int x2c_var_update_i32(volatile int *lhs, Symbol op, Var rhs)
x2c_var_update_i8	lib/varops.x	char x2c_var_update_i8(volatile char *lhs, Symbol op, Var rhs)
x2c_var_update_long	lib/varops.x	long x2c_var_update_long(volatile long *lhs, Symbol op, Var rhs)
x2c_var_update_long_double	lib/varops.x	long double x2c_var_update_long_double(volatile long double *lhs, Symbol op, Var rhs)
x2c_var_update_long_long	lib/varops.x	long long x2c_var_update_long_long(volatile long long *lhs, Symbol op, Var rhs)
x2c_var_update_schar	lib/varops.x	signed char x2c_var_update_schar(volatile signed char *lhs, Symbol op, Var rhs)
x2c_var_update_u16	lib/varops.x	ushort x2c_var_update_u16(volatile ushort *lhs, Symbol op, Var rhs)
x2c_var_update_u32	lib/varops.x	uint x2c_var_update_u32(volatile uint *lhs, Symbol op, Var rhs)
x2c_var_update_u8	lib/varops.x	uchar x2c_var_update_u8(volatile uchar *lhs, Symbol op, Var rhs)
x2c_var_update_ulong	lib/varops.x	ulong x2c_var_update_ulong(volatile ulong *lhs, Symbol op, Var rhs)
x2c_var_update_ulong_long	lib/varops.x	unsigned long long x2c_var_update_ulong_long(volatile unsigned long long *lhs, Symbol op, Var rhs)
x2c_var_update_volatile	lib/varops.x	Var x2c_var_update_volatile(volatile Var &?lhs, Symbol op, Var rhs)

Total: 85 functions.

lib/args.x

Parse program arguments against a declarative spec.

Primary API

Function	Summary
Args.from_argv	Returns the program arguments that follow `argv[0]` as `String`s.

Function	Summary
Args.parse	Parses args against spec and returns a Map from each row's name to its value.
Args.usage	Returns usage text for spec as Args.parse reads it: a synopsis for program, then each option, then each operand that has help, in spec order.

Args

Args.from_argv

```
List Args.from_argv(int argc, char **argv)
```

Returns the program arguments that follow argv[0] as Strings.

Source: lib/args.x:337

Args.parse

```
Map Args.parse(List args, List spec)
```

Parses args against spec and returns a Map from each row's name to its value.

Each row of spec is a List. A row that begins with dashed words is an option spelled by each of them, such as -p --prefix, and named by its first long spelling without the dashes, or else by its short one. A row that begins with any other word is an operand of that name. The rest of a row may hold (value placeholder), which makes an option take a value; (default value); (help "text") for Args.usage; required; and repeated.

A long option takes its value as --name value or --name=value, and a short option as -n value or -nvalue; short flags may share one word. Operands may appear between options, and -- makes every later word an operand. Operand rows take operands in spec order, and a repeated operand takes all that remain.

A flag's value is the number of times it appeared. A repeated row's value is a List of its values. Any other option or operand holds its last String. A row that was not given holds its default, or else zero for a flag, an empty List for a repeated row, and a NULL String otherwise, so every name is present and can be tested for truth.

```
List spec = %(
  (-v --verbose)
  (-o --output (value file) (default "a.out"))
  (-I (value dir) repeated)
  (inputs repeated required));
Map options = Args.parse(%(-vI src -Ilib main.x), spec);
```

Raises: <bad-arg> with reason and the offending option or operand for an unknown option, a missing or unexpected value, an unexpected operand, or a required row that was not given; and with reason and the offending spec entry for a property or word it cannot read.

Source: lib/args.x:83

Args.usage

```
String Args.usage(String program, List spec)
```

Returns usage text for spec as Args.parse reads it: a synopsis for program, then each option, then each operand that has help, in spec order. Help text starts at column 30, as in x2c help.

Source: lib/args.x:293

Public types

Type	Kind	Summary
Args	alias	The receiverless owner of Args.parse and the other argument operations.

Args

```
typedef Symbol Args
```

The receiverless owner of `Args.parse` and the other argument operations.

Source: `lib/args.x:18`

Design notes

Args owns the reading of a command line against a spec: an ordinary List with one row per option or operand. The result is a Map from each row's name to its value, words stay Strings, and bad input raises `<bad-arg>` so the caller decides whether to print `Args.usage` and which status to return.

Tests and examples

`make verify` (`unittest/test-args.x`).

`lib/array.x`

Dynamic contiguous arrays of `Var` elements.

Primary API

Function	Summary
`Array.concat`	Returns a fresh array holding `a` followed by `b`.
`Array.contains`	Reports whether a native element equals `value`.
`Array.copy`	Returns a fresh packed copy, or NULL for a null array.
`Array.count`	Returns how many native elements equal `value`.
`Array.find`	Returns the first index whose native element equals `value`, or −1.
`Array.foldl`	Folds `fn` over `array` from `seed`, front to back.
`Array.free`	Releases the `Array` and its backing storage, invalidating every alias.
`Array.getindex`	Reads `index` from a non-null array, counting negatives from the end.
`Array.getslice`	Returns a fresh packed array for normalized `[start:end:step]`.
`Array.indexof`	Returns the same first-match index as `Array.find`.
`Array.insert`	Inserts `value` at normalized `index` and returns it.
`Array.iter`	Initializes `dest` to lazily yield boxed elements in index order.
`Array.join`	Joins the elements of `array` into one `String` separated by `separator`.
`Array.map`	Returns a new `Array` holding `func` applied to each element of `array`.
`Array.map2`	Maps `func` over corresponding elements up to the shorter `Array`.
`Array.new`	Returns a fresh empty packed array owned by the current `Scope`.
`Array.postfixindex`	Applies postfix `Array` element increment or decrement.
`Array.push`	Appends `value` and returns it.
`Array.remove`	Removes and returns the element at normalized `index`.
`Array.resize`	Resizes `arr`, truncating or appending `Null` elements as needed.
`Array.reverse`	Reverses `array` in place and returns that same array.
`Array.setindex`	Stores and returns `value`, counting negatives from the end.
`Array.setslice`	Replaces the region `array[start:end]` and returns `array`.
`Array.shift`	Removes and returns the first element, shifting the remainder left.
`Array.sort`	Sorts `array` in place in ascending order and returns that same `Array`.
`Array.sort_by`	Stably sorts `array` by keys produced once per value, returning the Array.
`Array.sort_with`	Stably sorts `array` with a borrowed synchronous comparator and returns it.
`Array.splice`	Replaces `remove_count` values at normalized `index` and returns them.
`Array.take_last`	Removes and returns the last element.
`Array.try_next`	Writes the next native element, advances `cursor`, and returns one.
`Array.try_take_last`	Removes the last element into `out` and returns one when present.

Function	Summary
`Array.unshift`	Inserts `value` at the front and returns it, shifting elements right.
`Array.updateindex`	Updates one boxed element while preserving its Var tag.
`Array.write_str`	Appends the packed-Array display text to `out`.
`Block.array`	Returns the same object as an `Array` view; mutations remain shared.
`Iter.array`	Drains `iter` into a fresh `Array`.

`Array`

Array.concat

```
Array Array.concat(Array a, Array b)
```

Returns a fresh array holding `a` followed by `b`. Neither input is modified. A null `a` returns NULL, while a null `b` produces an independent copy of `a`.

Raises: `<size-limit>` or `<alloc-fail>` when the result cannot be represented or allocated.

Source: `lib/array.x:72`

Array.contains

```
int Array.contains(Array array, Var value)
```

Reports whether a native element equals `value`. The scan is linear. Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/array.x:72`

Array.copy

```
Array Array.copy(Array array)
```

Returns a fresh packed copy, or NULL for a null array. The result has independent mutable storage containing the same stored values.

Raises: `<size-limit>` or `<alloc-fail>` when the copy cannot be represented or allocated.

Source: `lib/array.x:72`

Array.count

```
int Array.count(Array array, Var value)
```

Returns how many native elements equal `value`. A null array returns zero; the scan is linear.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/array.x:72`

Array.find

```
int Array.find(Array array, Var value)
```

Returns the first index whose native element equals `value`, or −1. A null array returns −1; the scan is linear and preserves source order.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/array.x:72`

Array.foldl

```
Var Array.foldl(Array array, Var seed, Func fn)
```

Folds `fn` over `array` from `seed`, front to back. A `void` seed means "no seed": the first element becomes the accumulator and the fold starts at the second, so folding an empty `Array` that way returns `void`. Elements are passed as values. Input with nothing left to fold does not invoke or check `fn`. The callback receives the accumulator then the next element and is not retained. It must not structurally mutate `array` during the walk. Any cause from `Func.apply` or `fn` propagates without changing `array`.

Source: `lib/array.x:227`

Array.free

```
void Array.free(Array a0)
```

Releases the `Array` and its backing storage, invalidating every alias. The `Block` protocol generates the definition.

Source: `lib/array.x:119`

Array.getindex

```
Var Array.getindex(Array array, int index)
```

Reads `index` from a non-null array, counting negatives from the end. Returns `void` when the normalized index is outside the array.

Raises: `<size-limit>` outside the integer index domain.

Source: `lib/array.x:91`

Array.getslice

```
Array Array.getslice(Array array, int start, int end, int step)
```

Returns a fresh packed array for normalized `[start:end:step]`. Negative bounds count from the end and a negative step walks backward.

Raises: `<bad-arg>` for a null array or zero step, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/array.x:72`

Array.indexof

```
int Array.indexof(Array array, Var value)
```

Returns the same first-match index as `Array.find`.

Source: `lib/array.x:72`

Array.insert

```
Var Array.insert(Array array, int index, Var value)
```

Inserts `value` at normalized `index` and returns it. The accepted range includes the position after the last element; negative indices are normalized against that wider range. Boxed `Array` returns `void` for a null array or out-of-range index without mutation.

Raises: `<bad-arg>` for a null typed array or out-of-range typed-array index, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/array.x:72`

Array.iter

```
Iter Array.iter(Array array, Iter dest)
```

Initializes `dest` to lazily yield boxed elements in index order. Returns `dest`, whose storage is caller-owned. The iterator borrows `array` and reads its current storage on each step, so both must remain live and mutation can change the remaining traversal. A null `dest` returns NULL. A null array, or one longer than the integer index

domain, is exhausted. A boxed wide element is `Scope`-owned; retaining one after a pull requires its owning `Scope` to remain live.

Raises: any cause from boxing an element.

Source: `lib/array.x:240`

Array.join

```
meta native String Array.join(Array array, String separator)
```

Joins the elements of `array` into one `String` separated by `separator`. Elements are converted with their `str` form, so a `String` element contributes its bytes without quotes and a number contributes its digits. A null `separator` joins with nothing between elements, and an empty array yields the empty `String`.

The receiver here is the `Array`. In `String.join` the separator is the receiver and the elements arrive as a `List`. Raises: `<alloc-fail>` or `<size-limit>` while rendering or canonicalizing the result, or a cause from an element's `write_str`.

Source: `lib/array.x:439`

Array.map

```
Array Array.map(Array array, Func func)
```

Returns a new `Array` holding `func` applied to each element of `array`. `array` itself is untouched, and elements are passed as values. An empty input returns a fresh empty `Array` without invoking or checking `func`. The callback is invoked front to back and is not retained. It must not mutate `array` while the walk is in progress; `Func` rejects a `void` result.

Raises: whatever `Func.apply` or `func` raises, or an allocation cause while constructing the result. The partial result is freed.

Source: `lib/array.x:196`

Array.map2

```
Array Array.map2(Array a, Array b, Func func)
```

Maps `func` over corresponding elements up to the shorter `Array`. The callback receives left then right, runs front to back, and is not retained. An empty input does not invoke or check it; neither input may be structurally mutated during the walk.

Raises: whatever `Func.apply` or `func` raises, or an allocation cause while constructing the result. The partial result is freed.

Source: `lib/array.x:210`

Array.new

```
Array Array.new(void)
```

Returns a fresh empty packed array owned by the current `Scope`.

Raises: `<alloc-fail>` when storage cannot be allocated.

Source: `lib/array.x:72`

Array.postfixindex

```
Var Array.postfixindex(Array array, int index, Symbol op)
```

Applies postfix `Array` element increment or decrement. The returned value is the original element. The slot remains unchanged if the index or operation is invalid.

Raises: `<bad-arg>` for a null `Array` or invalid index, `<size-limit>` when its length cannot be indexed, or any cause from `Var.postfix`. The element is unchanged on failure.

Source: `lib/array.x:177`

Array.push

```
Var Array.push(Array array, Var value)
```

Appends `value` and returns it. Boxed `Array` returns `void` for a null array.

Raises: `<bad-arg>` for a null typed array, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/array.x:72`

Array.remove

```
Var Array.remove(Array array, int index)
```

Removes and returns the element at normalized `index`. Negative indices count from the end; following elements shift left. Boxed `Array` returns `void` for a null array or out-of-range index.

Raises: `<bad-arg>` for a null typed array or out-of-range typed-array index, or `<size-limit>` when the length exceeds the integer index domain in either family.

Source: `lib/array.x:72`

Array.resize

```
meta native void Array.resize(Array arr, size_t size)
```

Resizes `arr`, truncating or appending `Null` elements as needed.

Raises: `<size-limit>` when `size` exceeds the `Array` index domain, plus any cause from `Block` growth. Allocation and size failure do not return here. A growth failure leaves the length and existing elements unchanged.

Source: `lib/array.x:126`

Array.reverse

```
Array Array.reverse(Array array)
```

Reverses `array` in place and returns that same array. A null or one-element array is returned unchanged.

Source: `lib/array.x:72`

Array.setindex

```
Var Array.setindex(Array array, int index, Var value)
```

Stores and returns `value`, counting negatives from the end. Requires a non-null array. Returns `void` for an out-of-range index.

Raises: `<size-limit>` outside the index domain or `<void-op>` when storing `void` at a valid index.

Source: `lib/array.x:103`

Array.setslice

```
Array Array.setslice(Array array, int start, int end, Array values)
```

Replaces the region `array[start:end]` and returns `array`. Bounds normalize as slice bounds, so a negative `end` is a stop. Reversed bounds are swapped; null `values` deletes the region. Passing `array` itself as `values` is supported and copies before mutation.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` while copying self-aliased values or growing storage for the replacement. These failures leave `array` unchanged.

Source: `lib/array.x:72`

Array.shift

```
Var Array.shift(Array array)
```

Removes and returns the first element, shifting the remainder left. This operation is O(n). Boxed `Array` returns `void` for a null or empty array.

Raises: `<bad-arg>` for a null or empty typed array, or `<size-limit>` when the length exceeds the integer index domain in either family.

Source: `lib/array.x:72`

Array.sort

```
meta native Self Array.sort(Self array)
```

Sorts `array` in place in ascending order and returns that same `Array`. Call `Array.copy` first to preserve the original order. Ordering is `Var.compare`, which compares numbers numerically and `Strings`, `Symbols`, and `Atoms` by content. The sort is `qsort`, so it is not stable, and a null or one-element `array` is returned unchanged.

```
Array sorted = numbers.sort();
printf("%s %d\n", numbers.repr(), sorted == numbers);
```

Raises: causes from element comparison. The `Array` may already be partially reordered when a catch receives the cause.

Source: `lib/array.x:261`

Array.sort_by

```
Self Array.sort_by(Self array, Func key)
```

Stably sorts `array` by keys produced once per value, returning the Array. The borrowed callback runs front to back and must not mutate this Array. Keys use `Var.compare`; equal keys preserve the original element order. A null or empty Array invokes no callback. Sorting another Array inside the callback is valid. Raises: allocation, key comparison, `Func.apply`, or callback causes. Failure leaves the original Array unchanged; callback side effects are not undone.

Source: `lib/array.x:325`

Array.sort_with

```
Self Array.sort_with(Self array, Func compare)
```

Stably sorts `array` with a borrowed synchronous comparator and returns it. The callback receives two values; its result converts to `int`, with a negative, zero, or positive result ordering the first before, equal to, or after the second. It must give a consistent ordering and must not mutate this Array. Sorting a different Array inside the callback is valid. Null and fewer than two elements return unchanged without a callback.

Raises: allocation, `Func.apply`, conversion, or callback causes. Failure leaves the original Array unchanged; temporary storage is released. Callback side effects are not undone.

Source: `lib/array.x:280`

Array.splice

```
Array Array.splice(Array array, int index, int remove_count, Array values)
```

Replaces `remove_count` values at normalized `index` and returns them. The result is a fresh packed array. A nonpositive count removes nothing; null `values` performs deletion, and self-splicing is supported.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/array.x:72`

Array.take_last

```
Var Array.take_last(Array array)
```

Removes and returns the last element. Boxed `Array` returns `void` for a null or empty array.

Raises: `<bad-arg>` for a null or empty typed array.

Source: `lib/array.x:72`

Array.try_next

```
int Array.try_next(Array array, int &?cursor, Var &?out)
```

Writes the next native element, advances `cursor`, and returns one. Initialize the caller-owned cursor to zero. A null argument, negative cursor, exhaustion, or an array outside the integer index domain returns zero without changing `cursor` or `out`. Structural mutation invalidates the cursor.

Source: `lib/array.x:72`

Array.try_take_last

```
int Array.try_take_last(Array array, Var &?out)
```

Removes the last element into `out` and returns one when present. A null or empty array, or null `out`, returns zero without writing or changing the array.

Source: `lib/array.x:72`

Array.unshift

```
Var Array.unshift(Array array, Var value)
```

Inserts `value` at the front and returns it, shifting elements right. This operation is O(n). Boxed `Array` returns `void` for a null array.

Raises: `<bad-arg>` for a null typed array, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/array.x:72`

Array.updateindex

```
Var Array.updateindex(Array array, int index, Symbol op, Var rhs)
```

Updates one boxed element while preserving its Var tag.

Source: `lib/array.x:157`

Array.write_str

```
Buffer Array.write_str(Array array, Buffer out)
```

Appends the packed-Array display text to `out`.

Source: `lib/array.x:239`

Block

Block.array

```
inline Array Block.array(Block x)
```

Returns the same object as an `Array` view; mutations remain shared.

Source: `lib/array.x:114`

Iter

Iter.array

```
meta native Array Iter.array(Iter iter)
```

Drains `iter` into a fresh `Array`.

Source: `lib/array.x:452`

Advanced and interop API

Function	Summary
`Array.cleanup`	Ends the owned lifetime when a managed local leaves its block.
`Array.compare`	Compares packed Arrays lexicographically with the element family's ordering.
`Array.equal`	Reports elementwise native equality.
`Array.heap_pop`	Removes and returns the smallest element of `heap`, or `void` when it is empty.
`Array.heap_push`	Adds `val` to `heap`, an `Array` maintained as a binary min-heap.
`Array.heapify`	Rearranges `heap` in place so that it satisfies the min-heap invariant.
`Array.remslice`	Removes the region `array[start:end]` and returns a fresh packed array.
`Array.repr`	Returns the readable packed-Array representation.
`Array.str`	Returns the packed-Array display String.
`Array.update_n`	Appends exactly `element_count` variadic values to `array`.
`Array.write_repr`	Appends the readable packed-Array representation to `out`.

`Array`

Array.cleanup

```
meta native void Array.cleanup(Array value)
```

Ends the owned lifetime when a managed local leaves its block.

Source: `lib/array.x:460`

Array.compare

```
int Array.compare(Array a, Array b)
```

Compares packed Arrays lexicographically with the element family's ordering. Identical handles compare equal and NULL sorts first.

Source: `lib/array.x:239`

Array.equal

```
int Array.equal(Array a, Array b)
```

Reports elementwise native equality. Two null arrays compare equal; one null array or different lengths compare unequal.

Source: `lib/array.x:72`

Array.heap_pop

```
meta native Var Array.heap_pop(Array heap)
```

Removes and returns the smallest element of `heap`, or `void` when it is empty. The remaining elements are re-heaped in O(log n), so repeated calls yield ascending order and draining a heap is a sort. An `Array` that never satisfied the heap invariant gives a meaningless answer instead of an error. Call `Array.heapify` first if it was not built with `Array.heap_push`.

Raises: any cause reported by element comparison while restoring the heap. The heap may already have removed its root when a catch receives the error.

Source: lib/array.x:405

Array.heap_push

```
meta native void Array.heap_push(Array heap, Var val)
```

Adds val to heap, an Array maintained as a binary min-heap. The heap operations arrange an Array as a priority queue in place, with no second data structure and no extra allocation. The elements stay in the Array with the smallest at index 0. Ordering is Var.compare, the same rule Array.sort uses.

heap_push and Array.heap_pop maintain the invariant, but the positional operations do not. After a plain Array.push, an assignment through Array.setindex, or any slicing, call Array.heapify before popping again.

```
heap.heap_push(30);
heap.heap_push(10);
heap.heap_push(20);
printf("%s %s\n", heap.heap_pop().repr(), heap.heap_pop().repr());
```

Raises: <void-op> when val is void, or <size-limit> when the heap cannot grow within the Array index domain, <alloc-fail> when storage cannot grow, or a cause from element comparison. A value or earlier swap may remain when comparison fails; pre-insertion failures leave the heap unchanged.

Source: lib/array.x:390

Array.heapify

```
meta native void Array.heapify(Array heap)
```

Rearranges heap in place so that it satisfies the min-heap invariant. Use this before the first Array.heap_pop on an Array that was built by Array.push, read in from somewhere else, or disturbed by a positional operation. It is O(n), cheaper than pushing the same elements one at a time.

Raises: <size-limit> when heap exceeds the INT_MAX index limit that Array.getindex describes, or a cause from element comparison. Comparison failure may leave a partially rearranged Array.

Source: lib/array.x:424

Array.remslice

```
Array Array.remslice(Array array, int start, int end)
```

Removes the region array[start:end] and returns a fresh packed array. Bounds normalize as slice bounds, so a negative end is a stop. Reversed bounds are swapped, and the removed values retain their order.

Raises: <bad-arg> for a null array, <size-limit> outside the integer index domain, or <alloc-fail> on allocation failure.

Source: lib/array.x:72

Array.repr

```
String Array.repr(Array array)
```

Returns the readable packed-Array representation.

Source: lib/array.x:239

Array.str

```
String Array.str(Array array)
```

Returns the packed-Array display String.

Source: lib/array.x:239

Array.update_n

`Self Array.update_n(Self array, unsigned element_count, ...)`

Appends exactly `element_count` variadic values to `array`. Values appended before a later `<void-op>`, `<size-limit>`, or `<alloc-fail>` remain in `array`. The caller must supply that many `Var` arguments.

Source: `lib/array.x:145`

Array.write_repr

`Buffer Array.write_repr(Array array, Buffer out)`

Appends the readable packed-Array representation to `out`.

Source: `lib/array.x:239`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`Array.block`	Returns the same object as a `Block` view; no copy or transfer occurs.

`Array`

Array.block

`inline Block Array.block(Array x)`

Returns the same object as a `Block` view; no copy or transfer occurs.

Source: `lib/array.x:112`

Public types

Type	Kind	Summary
`Array`	alias	Holds a mutable identity-bearing sequence of non-`void` `Var` elements.

Array

`typedef Block Array`

Holds a mutable identity-bearing sequence of non-`void` `Var` elements. `Array` is the same object as its `Block` view; that object and its backing storage belong to the `Scope` active at construction. Assignment aliases it; `Array.copy` creates a new container, but still borrows any objects referenced by its elements. `Array.free` invalidates all aliases.

Source: `lib/array.x:36`

Design notes

`Array` is mutable, identity-bearing contiguous `Var` storage over `Block`. Every constructor returns an allocated object, including for an empty `Array`; assignment aliases that object, while `Array.copy` makes a shallow independent container. `Scope` owns its allocation unless `Array.free` shortens the lifetime. `Array` data may contain raw `Null` but never `void`.

The declared `Block` protocol supplies `Array`'s generated `truth`, `pop`, `free`, and `truncate` members. `Array.pop` discards the final element; `Array.take_last` removes and returns it.

Indexing and slicing normalize negative positions here. Positional mutation preserves the object identity, and structural mutation invalidates outstanding iterators.

Tests and examples

`make verify` (`unittest/test-array.x`) and `make examples`.

lib/atom.x

Canonical exact names.

Primary API

Function	Summary
Atom.write_str	Appends the exact spelling of atom to out.
Var.is_atom	Reports whether value is an exact-spelling Atom.

Atom

Atom.write_str

```
Buffer Atom.write_str(Atom atom, Buffer out)
```

Appends the exact spelling of atom to out. Neither representation allocates a String. An <lsym> Atom already owns its bytes, and a compact <symbol> decodes straight into out. An invalid Atom appends nothing.

Raises: <size-limit> or <alloc-fail> when out cannot grow.

Source: lib/atom.x:54

Var

Var.is_atom

```
meta native int Var.is_atom(Var value)
```

Reports whether value is an exact-spelling Atom.

Source: lib/atom.x:92

Advanced and interop API

Function	Summary
Atom.bare_spelling	Reports whether a spelling can round-trip as a bare Lisp Atom.
Atom.first	Returns the first byte of atom's exact spelling, or NUL when invalid.
Atom.initialize	Initializes the process-wide Atom runtime owner.
Atom.intern	Returns the canonical Atom with the supplied exact spelling.
Atom.promote	Promotes a long Atom spelling into its parent String pool.
Atom.str	Returns the exact spelling of atom as a String.
Atom.write_repr	Writes the escaped readable spelling of atom to out.

Atom

Atom.bare_spelling

```
int Atom.bare_spelling(String spelling)
```

Reports whether a spelling can round-trip as a bare Lisp Atom.

Source: lib/atom.x:37

Atom.first

```
char Atom.first(Atom atom)
```

Returns the first byte of atom's exact spelling, or NUL when invalid.

Source: lib/atom.x:108

Atom.initialize

```
void Atom.initialize(void)
```

Initializes the process-wide `Atom` runtime owner. Repeated calls after successful registration have no effect.

Raises: `<bad-state>` after descriptor registration is frozen, `<alloc-fail>` while lower-casing the descriptor name, or `<init-fail>` when registration returns zero.

Source: `lib/atom.x:166`

Atom.intern

```
Atom Atom.intern(String spelling)
```

Returns the canonical `Atom` with the supplied exact spelling. Repeated interning of equal bytes returns a bit-identical `Atom`. A spelling that round-trips through either compact encoding becomes an immediate `Symbol`; every other spelling borrows its canonical `String` under the module lifetime rule above. Use `Atom.promote` before a long `Atom` escapes its innermost owning pool.

Raises: `<bad-arg>` when `spelling` is null or empty, `<alloc-fail>` when its canonical `String` cannot be allocated, or `<bad-enc>` if the long-`Atom` pointer cannot be boxed.

Source: `lib/atom.x:219`

Atom.promote

```
Self Atom.promote(Self atom)
```

Promotes a long `Atom` spelling into its parent `String` pool. Returns `atom` unchanged. Compact `Atoms` and long spellings not owned by the innermost pool are unchanged. A promoted spelling keeps the same pointer.

Raises: `<alloc-fail>` when promotion metadata cannot be allocated.

Source: `lib/atom.x:26`

Atom.str

```
String Atom.str(Atom atom)
```

Returns the exact spelling of `atom` as a `String`. A compact `Atom` is decoded and canonicalized through the active pool chain. A long `Atom` returns its borrowed canonical spelling under the module lifetime rule above. An invalid `Atom` returns NULL.

Raises: `<alloc-fail>` while canonicalizing a compact spelling.

Source: `lib/atom.x:101`

Atom.write_repr

```
Buffer Atom.write_repr(Atom value, Buffer out)
```

Writes the escaped readable spelling of `atom` to `out`. The escaping makes the result readable as the same exact `Atom`. An invalid `Atom` appends nothing.

Raises: `<alloc-fail>` while decoding a compact `Atom`, or `<size-limit>` or `<alloc-fail>` when `out` cannot grow.

Source: `lib/atom.x:67`

Design notes

An `Atom` preserves its spelling exactly. Compact spellings use immediate `Symbols` when `Symbol` encoding round-trips every byte; all other spellings use the private `<lsym>` representation with a direct pointer to the canonical `String`. `String` canonicalization makes equal long `Atoms` bit-identical. Compact `Atoms` have value lifetime. A long `Atom` borrows its canonical spelling, which may belong to any pool in the active chain and lives until its owning pool is released.

Tests and examples

`make verify` (`unittest/test-atom.x`).

lib/block.x

Checked dynamic storage for fixed-width elements.

Advanced and interop API

Function	Summary
Block.append	Appends count elements copied from source to block.
Block.append_fill	Appends repeated fixed-width elements to block.
Block.capacity	Returns how many elements b can hold without growing.
Block.cleanup	Ends the owned lifetime when a managed local leaves its block.
Block.clear	Removes every element from block without releasing capacity.
Block.free	Releases the Block and its backing storage, invalidating every alias.
Block.len	Returns the number of elements stored in b.
Block.new	Allocates an empty Block for elements of width bytes.
Block.pop	Removes the final element of block when present.
Block.push	Appends one copied element; storage and failures follow Block.append.
Block.reserve	Ensures block can hold at least minimum elements.
Block.truncate	Shortens block to at most length elements.
Block.truth	Returns nonzero when block contains at least one element.
Block.try_pop	Removes the final element of b, copying it to out when present.
Bytes.append	Appends count elements to bytes and returns its current base pointer.
Bytes.append_fill	Appends repeated elements to bytes and returns its current base pointer.
Bytes.block	Returns the stable Block that owns the live bytes base pointer.
Bytes.cleanup	Ends the owned lifetime when a managed local leaves its block.
Bytes.new	Allocates empty byte storage for elements of width bytes.
Bytes.push	Appends one element and returns the possibly relocated Bytes base.
Bytes.reserve	Ensures bytes can hold at least minimum elements and returns its base.
Bytes.try_pop	Removes the final element through bytes as Block.try_pop does.

Block

Block.append

```
void Block.append(Block b, const void *source, size_t count)
```

Appends count elements copied from source to block. The source may be NULL for zero-filled elements; otherwise it must name count readable elements. When growth is required, an initialized range inside this Block's reserved storage keeps its offset across relocation. Success may invalidate saved Bytes and element pointers.

Raises: <bad-arg> for a null Block, or for an internal source crossing reserved storage when growth is required; <size-limit> when the new extent cannot be represented; or <alloc-fail> when growth fails. These failures leave the Block unchanged.

Source: lib/block.x:86

Block.append_fill

```
void Block.append_fill(Block b, const void *element, size_t count)
```

Appends repeated fixed-width elements to block. A zero count is a no-op and accepts a null element. Otherwise element must point to one complete element. element may point at one complete logical element in block; its offset is preserved if growth relocates storage. Success may invalidate saved Bytes and element pointers.

Raises: <bad-arg> for a null Block, or for a null or invalid self-source element on a nonzero append; <size-limit> when the new extent cannot be represented; or <alloc-fail> when growth fails. These failures leave the Block unchanged.

Source: lib/block.x:152

Block.capacity

```
inline size_t Block.capacity(Block b)
```

Returns how many elements b can hold without growing.

Source: lib/block.x:244

Block.cleanup

```
void Block.cleanup(Block value)
```

Ends the owned lifetime when a managed local leaves its block.

Source: lib/block.x:37

Block.clear

```
inline void Block.clear(Block block)
```

Removes every element from block without releasing capacity.

Source: lib/block.x:257

Block.free

```
void Block.free(Block b)
```

Releases the Block and its backing storage, invalidating every alias.

Source: lib/block.x:306

Block.len

```
inline size_t Block.len(Block b)
```

Returns the number of elements stored in b.

Source: lib/block.x:241

Block.new

```
Block Block.new(size_t width)
```

Allocates an empty Block for elements of width bytes.

Raises: <bad-arg> when width is zero, <size-limit> when the initial allocation size cannot be represented, or <alloc-fail> when allocation fails.

Source: lib/block.x:288

Block.pop

```
inline void Block.pop(Block block)
```

Removes the final element of block when present.

Source: lib/block.x:277

Block.push

```
inline void Block.push(Block block, const void *source)
```

Appends one copied element; storage and failures follow `Block.append`.

Source: `lib/block.x:188`

Block.reserve

```
void Block.reserve(Block block, size_t minimum)
```

Ensures `block` can hold at least `minimum` elements. The `Block` handle and contents remain stable, but growth may replace `block.bytes` and invalidate saved `Bytes` or element pointers.

Raises: `<bad-arg>` when `block` is null, `<size-limit>` when the requested capacity cannot be represented, or `<alloc-fail>` when growth fails. These failures leave the `Block` unchanged.

Source: `lib/block.x:205`

Block.truncate

```
inline void Block.truncate(Block block, size_t length)
```

Shortens `block` to at most `length` elements.

Source: `lib/block.x:252`

Block.truth

```
int Block.truth(Block block)
```

Returns nonzero when `block` contains at least one element.

Source: `lib/block.x:247`

Block.try_pop

```
inline int Block.try_pop(Block b, void *out)
```

Removes the final element of `b`, copying it to `out` when present. Returns zero for a null or empty `Block` and leaves `out` unchanged. A null `out` still removes a present element.

Source: `lib/block.x:265`

Bytes

Bytes.append

```
inline Self Bytes.append(Self bytes, const void *source, size_t count)
```

Appends `count` elements to `bytes` and returns its current base pointer. Callers must use the return because growth may relocate storage. Raises: the same causes as `Block.append`; failure leaves the original view live.

Source: `lib/block.x:135`

Bytes.append_fill

```
inline Self Bytes.append_fill(Self bytes, const void *element, size_t count)
```

Appends repeated elements to `bytes` and returns its current base pointer. Callers must use the return because growth may relocate storage. Raises: the same causes as `Block.append_fill`; failure leaves the original view live.

Source: `lib/block.x:181`

Bytes.block

```
inline Block Bytes.block(Bytes bytes)
```

Returns the stable `Block` that owns the live `bytes` base pointer. An interior or stale pointer is invalid; NULL returns NULL.

Source: `lib/block.x:68`

Bytes.cleanup

```
void Bytes.cleanup(Bytes value)
```

Ends the owned lifetime when a managed local leaves its block.

Source: `lib/block.x:38`

Bytes.new

```
Bytes Bytes.new(size_t width)
```

Allocates empty byte storage for elements of `width` bytes. The result is the base view of a `Scope`-owned `Block`. Methods that may grow it return the current `Bytes` pointer, which callers must keep.

Raises: the same causes as `Block.new`.

Source: `lib/block.x:303`

Bytes.push

```
inline Self Bytes.push(Self bytes, const void *source)
```

Appends one element and returns the possibly relocated `Bytes` base.

Source: `lib/block.x:193`

Bytes.reserve

```
inline Self Bytes.reserve(Self bytes, size_t minimum)
```

Ensures `bytes` can hold at least `minimum` elements and returns its base. Growth may relocate storage, so callers must use the returned `Bytes` and discard earlier views. Raises: the same causes as `Block.reserve`; failure leaves the original view live.

Source: `lib/block.x:234`

Bytes.try_pop

```
inline int Bytes.try_pop(Bytes bytes, void *out)
```

Removes the final element through `bytes` as `Block.try_pop` does.

Source: `lib/block.x:274`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`Block.move_to`	Moves `block` and its backing allocation into `scope`.

`Block`

Block.move_to

```
void Block.move_to(Block block, Scope *scope)
```

Moves `block` and its backing allocation into `scope`. This preserves the `Block` and `Bytes` pointers and contents while changing which `Scope` ends their lifetime. `Context` export calls this method instead of assuming a `Block` is one allocation. For a nonnull `Block`, raises `<bad-arg>` for a null destination slot, or `<alloc-fail>` when an empty slot cannot acquire a `Scope`. Failure leaves ownership unchanged.

Source: `lib/block.x:320`

Public types

Type	Kind	Summary
Block	struct	Holds resizable contiguous storage for fixed-width elements.

Block

`typedef struct Block { Bytes bytes, size_t width, length, cap; } *Block`

Holds resizable contiguous storage for fixed-width elements. The object and backing allocation belong to the `Scope` active at construction unless `Block.move_to` transfers them. The `Block` handle remains stable, but `bytes` is a borrowed base pointer that growth may replace. Valid `Blocks` have nonzero `width` and `length <= cap`; `Block.free` invalidates the handle and every `Bytes` view.

Source: `lib/block.x:33`

Design notes

`Block` maintains the length and capacity invariants for contiguous fixed-width storage. `Bytes` exposes the same allocation through its data pointer and a hidden back-pointer. Capacity growth raises where the failure is detected, leaves storage unchanged, and does not return to the call.

A self-source may name any initialized range inside reserved capacity. Growth rebases that range after reallocating the owning block.

`Bytes` receives the `Block` protocol's generated members through the `Bytes.block` storage view.

`lib/buffer.x`

Growable text buffer with indentation support.

Primary API

Function	Summary
Buffer.clear	Empties `buf` and its indentation stack without releasing capacity.
Buffer.getindex	Returns `buf.get(index)`, so `buf[index]` reads a byte.
Buffer.len	Returns the number of bytes currently stored in `buf`.
Buffer.new	Allocates an empty `Buffer` whose `pad` method writes `padding` spaces.
Buffer.pop	Pops one indentation depth from `buf`'s stack when present.
Buffer.push	Pushes the current column as a later indentation depth.
Buffer.truth	Returns nonzero when `b` contains at least one byte.
Buffer.try_get	Writes the byte at normalized `index` to `out` when it exists.

`Buffer`

Buffer.clear

`meta native Self Buffer.clear(Self buf)`

Empties `buf` and its indentation stack without releasing capacity.

Source: `lib/buffer.x:327`

Buffer.getindex

`int Buffer.getindex(Buffer buf, int index)`

Returns `buf.get(index)`, so `buf[index]` reads a byte.

Source: `lib/buffer.x:255`

Buffer.len

```
size_t Buffer.len(Buffer buf)
```

Returns the number of bytes currently stored in `buf`.

Source: `lib/buffer.x:258`

Buffer.new

```
meta native Buffer Buffer.new(size_t padding)
```

Allocates an empty `Buffer` whose `pad` method writes `padding` spaces.

Raises: `<alloc-fail>` or `<size-limit>` while allocating its fixed-width backing `Blocks`.

Source: `lib/buffer.x:304`

Buffer.pop

```
meta native Self Buffer.pop(Self buf)
```

Pops one indentation depth from `buf`'s stack when present.

Source: `lib/buffer.x:213`

Buffer.push

```
meta native Self Buffer.push(Self buf)
```

Pushes the current column as a later indentation depth.

Raises: `<size-limit>` or `<alloc-fail>` if the stack cannot grow. Failure leaves the stack unchanged.

Source: `lib/buffer.x:207`

Buffer.truth

```
int Buffer.truth(Buffer b)
```

Returns nonzero when `b` contains at least one byte.

Source: `lib/buffer.x:296`

Buffer.try_get

```
int Buffer.try_get(Buffer buf, ptrdiff_t index, char &?out)
```

Writes the byte at normalized `index` to `out` when it exists. Negative indexes count from the end. Returns zero for a null `Buffer`, null output, or missing byte and leaves `out` unchanged.

Source: `lib/buffer.x:231`

Advanced and interop API

Function	Summary
`Buffer.cleanup`	Ends the owned lifetime when a managed local leaves its block.
`Buffer.free`	Releases `buf` and both backing `Blocks`, invalidating every alias.
`Buffer.indent`	Appends spaces through `buf`'s current indentation depth.
`Buffer.newline`	Appends a newline to `buf`.
`Buffer.newline_indent`	Appends a newline followed by current indentation.
`Buffer.pad`	Appends the configured number of padding spaces.
`Buffer.printf`	Appends formatted text to `buf`.
`Buffer.repr`	Returns the readable representation of `Buffer`.
`Buffer.reserve`	Reserves room for at least `minimum` output bytes in `buf`.
`Buffer.str`	Returns a canonical copy of `buf`'s text without consuming the `Buffer`.

Function	Summary
Buffer.str_free	Returns buf.str() and frees buf.
Buffer.tabstop	Returns the indentation depth on top of the stack, or 0 when empty.
Buffer.unwrite	Removes the final count bytes from buf.
Buffer.write	Appends NUL-terminated text to buf.
Buffer.write_char	Appends the non-NUL byte value to buf.
Buffer.write_len	Appends length bytes from text to buf.
Buffer.write_repeat	Appends count copies of the non-NUL byte value to buf.

Buffer

Buffer.cleanup

```
void Buffer.cleanup(Buffer value)
```

Ends the owned lifetime when a managed local leaves its block.

Source: lib/buffer.x:30

Buffer.free

```
void Buffer.free(Buffer buf)
```

Releases buf and both backing Blocks, invalidating every alias.

Source: lib/buffer.x:348

Buffer.indent

```
meta native Self Buffer.indent(Self buf)
```

Appends spaces through buf's current indentation depth.

Source: lib/buffer.x:197

Buffer.newline

```
meta native Self Buffer.newline(Self buf)
```

Appends a newline to buf.

Source: lib/buffer.x:194

Buffer.newline_indent

```
meta native Self Buffer.newline_indent(Self buf)
```

Appends a newline followed by current indentation.

Source: lib/buffer.x:201

Buffer.pad

```
meta native Self Buffer.pad(Self buf)
```

Appends the configured number of padding spaces.

Source: lib/buffer.x:191

Buffer.printf

```
Self Buffer.printf(Self buf, const char *format, ...)
```

Appends formatted text to buf.

Raises: <format> when formatting fails, <alloc-fail> when staging storage cannot be allocated, or a cause from Buffer.write_len. Existing text is preserved and this call appends nothing on failure.

Source: lib/buffer.x:81

Buffer.repr

```
String Buffer.repr(Buffer buf)
```

Returns the readable representation of Buffer.

Raises: the same causes as Buffer.str or String rendering.

Source: lib/buffer.x:289

Buffer.reserve

```
meta native Self Buffer.reserve(Self buf, size_t minimum)
```

Reserves room for at least minimum output bytes in buf. Growth may invalidate a borrowed content.bytes pointer but does not change the Buffer or content Block identity.

Raises: <size-limit> or <alloc-fail> when the requested capacity cannot be provided.

Source: lib/buffer.x:321

Buffer.str

```
String Buffer.str(Buffer buf)
```

Returns a canonical copy of buf's text without consuming the Buffer. A null or empty Buffer returns the null String. A nonempty result remains live until its owning String pool is released.

Raises: <size-limit> when the text exceeds String's representation, or <alloc-fail> while canonicalizing it.

Source: lib/buffer.x:266

Buffer.str_free

```
String Buffer.str_free(Buffer buf)
```

Returns buf.str() and frees buf. Buffer.str performs the conversion, so the result is a canonical String copied out of the Buffer. Its storage is not adopted. buf is released on success and when the conversion transfers an Error.

Raises: the same causes as Buffer.str.

Source: lib/buffer.x:281

Buffer.tabstop

```
size_t Buffer.tabstop(Buffer buf)
```

Returns the indentation depth on top of the stack, or 0 when empty.

Source: lib/buffer.x:219

Buffer.unwrite

```
meta native Self Buffer.unwrite(Self buf, size_t count)
```

Removes the final count bytes from buf.

Source: lib/buffer.x:129

Buffer.write

```
meta native Self Buffer.write(Self buf, const char *text)
```

Appends NUL-terminated text to buf.

Raises: the same causes as `Buffer.write_len`.

Source: `lib/buffer.x:71`

Buffer.write_char

```
meta native Self Buffer.write_char(Self buf, char value)
```

Appends the non-NUL byte `value` to `buf`.

Raises: `<bad-arg>` when `value` is NUL, or `<size-limit>` or `<alloc-fail>` when the `Buffer` cannot grow. These failures leave text and line state unchanged.

Source: `lib/buffer.x:104`

Buffer.write_len

```
meta native Self Buffer.write_len(Self buf, const char *text, size_t length)
```

Appends `length` bytes from `text` to `buf`. `text` may be a live range inside `buf.content`, including across growth. A zero length accepts a null source. Raises: `<bad-arg>` for a null nonempty source or embedded NUL, or `<size-limit>` or `<alloc-fail>` when the `Buffer` cannot grow. Argument, allocation, and size failures leave the text and line state unchanged.

Source: `lib/buffer.x:54`

Buffer.write_repeat

```
meta native Self Buffer.write_repeat(Self buf, char value, size_t count)
```

Appends `count` copies of the non-NUL byte `value` to `buf`. A zero count accepts any value. Raises: `<bad-arg>` when a nonzero write uses NUL, or `<size-limit>` or `<alloc-fail>` when the `Buffer` cannot grow. These failures leave text and line state unchanged.

Source: `lib/buffer.x:120`

Convenience API

Function	Summary
`Buffer.get`	Returns the byte at `index`, or NUL when `index` is out of range.

`Buffer`

Buffer.get

```
char Buffer.get(Buffer buf, ptrdiff_t index)
```

Returns the byte at `index`, or NUL when `index` is out of range. Prefer `Buffer.try_get` to distinguish an out-of-range index from a NUL byte.

Source: `lib/buffer.x:249`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`Buffer.move_to`	Moves `buf` and both owned `Block`s into `scope`, keeping their identities.

`Buffer`

Buffer.move_to

```
void Buffer.move_to(Buffer buf, Scope *scope)
```

Moves `buf` and both owned `Blocks` into `scope`, keeping their identities. Their lifetime then ends with the destination `Scope` unless freed earlier. For a nonnull `Buffer`, raises `<bad-arg>` for a null destination slot, or `<alloc-fail>` when an empty slot cannot acquire a `Scope`. Failure leaves ownership unchanged.

Source: `lib/buffer.x:340`

Public types

Type	Kind	Summary
`Buffer`	struct	Holds mutable text plus an indentation stack in two `Blocks`.

Buffer

```
typedef struct Buffer { Block content, indents, size_t padding, pos, _indent; } *Buffer
```

Holds mutable text plus an indentation stack in two `Blocks`. The `Buffer` and both `Blocks` belong to the `Scope` active at construction unless `Buffer.move_to` transfers them. `content.bytes` is a borrowed, non-NUL-terminated view that any growing write may invalidate. `pos` and `_indent` describe the current final line; `Buffer.free` invalidates the `Buffer` and both backing `Blocks`.

Source: `lib/buffer.x:26`

Design notes

`Buffer` is a text-only builder over `Block` storage. It tracks the byte position and leading indentation of the current line, supports nested tabstops, and provides bulk character writes for formatting hot paths. Embedded NUL is rejected because `Buffer` materializes canonical `Strings`.

Tests and examples

`make verify` (`unittest/test-buffer.x`).

`lib/common.x`

The `Var` encoding every runtime module shares.

Advanced and interop API

Function	Summary
`x2c_initialize`	Initializes the x2c runtime once for the current process.
`x2c_initialize_protocols`	Runs compiler-generated runtime protocol registration.
`x2c_normalize_index`	Normalizes one element index against `length`.
`x2c_normalize_slice`	Normalizes slice bounds and returns the resulting element count.
`x2c_slice_bound`	Returns an explicit slice `bound` that `x2c_normalize_slice` cannot read as omitted.
`Array.var`	Boxes an `Array` value as `Var`.
`Block.var`	Boxes a `Block` value as `Var`.
`Buffer.var`	Boxes a `Buffer` value as `Var`.
`Bytes.var`	Boxes a `Bytes` value as `Var`.
`File.var`	Boxes a `File` value as `Var`.
`Iter.truth`	Returns nonzero when `iter` is not null.
`Iter.var`	Boxes an `Iter` value as `Var`.
`List.truth`	Returns nonzero when `list` is not `nil`, the null empty List.
`List.var`	Boxes a `List` value as `Var`.
`Map.var`	Boxes a `Map` value as `Var`.
`String.truth`	Returns nonzero when `string` contains at least one byte.
`String.var`	Boxes a `String` value as `Var`.
`Symbol.var`	Boxes a `Symbol` value as `Var`.

Function	Summary
Var.array_payload	Extracts the Array payload of value, or NULL for another tag.
Var.block_payload	Extracts the Block payload of value, or NULL for another tag.
Var.box_f32	Boxes a native float as an immediate <f32> Var.
Var.box_f64	Boxes a native double as an immediate <f64> Var.
Var.box_i16	Boxes a native short as an immediate <i16> Var.
Var.box_i32_bits	Boxes an unsigned 32-bit pattern as an immediate <i32> Var.
Var.box_i8	Boxes a native char as an immediate <i8> Var.
Var.box_u16	Boxes a native ushort as an immediate <u16> Var.
Var.box_u32	Boxes a native uint as an immediate <u32> Var.
Var.box_u8	Boxes a native uchar as an immediate <u8> Var.
Var.buffer_payload	Extracts the Buffer payload of value, or NULL for another tag.
Var.bytes_payload	Extracts the Bytes payload of value, or NULL for another tag.
Var.char	Returns x as a native char under the Var.convert rules.
Var.decode_f32	Decodes an immediate <f32> Var.
Var.decode_f64	Decodes an immediate <f64> Var.
Var.double	Returns x as a native double under the Var.convert rules.
Var.file_payload	Extracts the File payload of value, or NULL for another tag.
Var.float	Returns x as a native float under the Var.convert rules.
Var.int	Returns x as a native int under the Var.convert rules.
Var.is_row	Reports whether value occupies the encoding row at top/bottom.
Var.is_wide	Reports whether v uses a scope-owned wide numeric box.
Var.iter_payload	Extracts the Iter payload of value, or NULL for another tag.
Var.list_payload	Extracts the List payload of value, or NULL for another tag.
Var.long	Returns x as a native long under the Var.convert rules.
Var.long_double	Returns x as a native long double under the Var.convert rules.
Var.long_long	Returns x as a native long long under the Var.convert rules.
Var.map_payload	Extracts the Map payload of value, or NULL for another tag.
Var.payload32	Returns the low 32 payload bits of an immediate Var.
Var.short	Returns x as a native short under the Var.convert rules.
Var.string_payload	Extracts the String payload of value, or NULL for another tag.
Var.symbol	Extracts the symbol payload after the caller establishes the matching Var kind.
Var.uchar	Returns x as a native uchar under the Var.convert rules.
Var.uint	Returns x as a native uint under the Var.convert rules.
Var.ulong	Returns x as a native ulong under the Var.convert rules.
Var.ulong_long	Returns x as a native unsigned long long under the Var.convert rules.
Var.unsigned	Returns x as a native unsigned under the Var.convert rules.
Var.ushort	Returns x as a native ushort under the Var.convert rules.
char.repr	Returns the readable representation of char.
char.str	Returns the display String of char.
char.var	Boxes a native char value as Var.
double.repr	Returns the readable representation of double.
double.str	Returns the display String of double.
double.var	Boxes a native double value as Var.

Function	Summary
float.repr	Returns the readable representation of float.
float.str	Returns the display String of float.
float.var	Boxes a native float value as Var.
int.repr	Returns the readable representation of int.
int.str	Returns the display String of int.
int.var	Boxes a native int value as Var.
long.repr	Returns the readable representation of long.
long.str	Returns the display String of long.
long.var	Boxes a long value as Var.
short.repr	Returns the readable representation of short.
short.str	Returns the display String of short.
short.var	Boxes a native short value as Var.
uchar.repr	Returns the readable representation of uchar.
uchar.str	Returns the display String of uchar.
uchar.var	Boxes a native uchar value as Var.
uint.repr	Returns the readable representation of uint.
uint.str	Returns the display String of uint.
uint.var	Boxes a native uint value as Var.
ulong.var	Boxes a ulong value as Var.
unsigned.repr	Returns the readable representation of unsigned.
unsigned.str	Returns the display String of unsigned.
unsigned.var	Boxes a native unsigned value as Var.
ushort.repr	Returns the readable representation of ushort.
ushort.str	Returns the display String of ushort.
ushort.var	Boxes a native ushort value as Var.

Functions

x2c_initialize

```
void x2c_initialize(void)
```

Initializes the x2c runtime once for the current process.

Source: lib/common.x:845

x2c_initialize_protocols

```
void x2c_initialize_protocols(void)
```

Runs compiler-generated runtime protocol registration.

Source: lib/common.x:841

x2c_normalize_index

```
int x2c_normalize_index(int index, int length)
```

Normalizes one element index against length.

Source: lib/common.x:871

x2c_normalize_slice

```
int x2c_normalize_slice(int *start, int *stop, int step, int length)
```

Normalizes slice bounds and returns the resulting element count.

Source: lib/common.x:896

x2c_slice_bound

```
int x2c_slice_bound(int bound)
```

Returns an explicit slice `bound` that `x2c_normalize_slice` cannot read as omitted. INT_MIN becomes INT_MIN + 1, which every collection shorter than INT_MAX clamps the same way.

Source: `lib/common.x:882`

Array

Array.var

```
inline Var Array.var(Array x)
```

Boxes an `Array` value as `Var`.

Source: `lib/common.x:551`

Block

Block.var

```
inline Var Block.var(Block x)
```

Boxes a `Block` value as `Var`.

Source: `lib/common.x:553`

Buffer

Buffer.var

```
inline Var Buffer.var(Buffer x)
```

Boxes a `Buffer` value as `Var`.

Source: `lib/common.x:555`

Bytes

Bytes.var

```
inline Var Bytes.var(Bytes x)
```

Boxes a `Bytes` value as `Var`.

Source: `lib/common.x:557`

File

File.var

```
inline Var File.var(File x)
```

Boxes a `File` value as `Var`.

Source: `lib/common.x:562`

Iter

Iter.truth

```
inline int Iter.truth(Iter iter)
```

Returns nonzero when `iter` is not null.

Source: `lib/common.x:425`

Iter.var

```
inline Var Iter.var(Iter x)
```

Boxes an `Iter` value as `Var`.

Source: `lib/common.x:572`

`List`

List.truth

```
inline int List.truth(List list)
```

Returns nonzero when `list` is not `nil`, the null empty List.

Source: `lib/common.x:428`

List.var

```
inline Var List.var(List x)
```

Boxes a `List` value as `Var`.

Source: `lib/common.x:559`

`Map`

Map.var

```
inline Var Map.var(Map x)
```

Boxes a `Map` value as `Var`.

Source: `lib/common.x:564`

`String`

String.truth

```
inline int String.truth(String string)
```

Returns nonzero when `string` contains at least one byte. The canonical empty String is null, with no allocated storage.

Source: `lib/common.x:432`

String.var

```
inline Var String.var(String x)
```

Boxes a `String` value as `Var`.

Source: `lib/common.x:566`

`Symbol`

Symbol.var

```
inline Var Symbol.var(Symbol x)
```

Boxes a `Symbol` value as `Var`.

Source: `lib/common.x:569`

`Var`

Var.array_payload

```
Array Var.array_payload(Var value)
```

Extracts the `Array` payload of `value`, or NULL for another tag.
Source: `lib/common.x:665`

Var.block_payload
`Block Var.block_payload(Var value)`
Extracts the `Block` payload of `value`, or NULL for another tag.
Source: `lib/common.x:666`

Var.box_f32
`inline Var Var.box_f32(float value)`
Boxes a native `float` as an immediate `<f32>` `Var`.
Source: `lib/common.x:533`

Var.box_f64
`inline Var Var.box_f64(double value)`
Boxes a native `double` as an immediate `<f64>` `Var`.
Source: `lib/common.x:540`

Var.box_i16
`Var Var.box_i16(short value)`
Boxes a native `short` as an immediate `<i16>` `Var`.
Source: `lib/common.x:524`

Var.box_i32_bits
`Var Var.box_i32_bits(unsigned value)`
Boxes an unsigned 32-bit pattern as an immediate `<i32>` `Var`.
Source: `lib/common.x:528`

Var.box_i8
`Var Var.box_i8(char value)`
Boxes a native `char` as an immediate `<i8>` `Var`.
Source: `lib/common.x:520`

Var.box_u16
`Var Var.box_u16(ushort value)`
Boxes a native `ushort` as an immediate `<u16>` `Var`.
Source: `lib/common.x:526`

Var.box_u32
`Var Var.box_u32(unsigned value)`
Boxes a native `uint` as an immediate `<u32>` `Var`.
Source: `lib/common.x:530`

Var.box_u8

```
Var Var.box_u8(uchar value)
```

Boxes a native `uchar` as an immediate `<u8>` `Var`.

Source: `lib/common.x:522`

Var.buffer_payload

```
Buffer Var.buffer_payload(Var value)
```

Extracts the `Buffer` payload of `value`, or NULL for another tag.

Source: `lib/common.x:667`

Var.bytes_payload

```
Bytes Var.bytes_payload(Var value)
```

Extracts the `Bytes` payload of `value`, or NULL for another tag.

Source: `lib/common.x:668`

Var.char

```
meta native char Var.char(Var x)
```

Returns `x` as a native `char` under the `Var.convert` rules.

Raises: `Var.convert`'s causes when the source is nonnumeric, `void`, out of floating range, or invalidly encoded.

Source: `lib/common.x:695`

Var.decode_f32

```
inline float Var.decode_f32(Var value)
```

Decodes an immediate `<f32>` `Var`.

Source: `lib/common.x:492`

Var.decode_f64

```
inline double Var.decode_f64(Var value)
```

Decodes an immediate `<f64>` `Var`.

Source: `lib/common.x:500`

Var.double

```
meta native double Var.double(Var x)
```

Returns `x` as a native `double` under the `Var.convert` rules.

Raises: `Var.convert`'s causes when the source is nonnumeric, `void`, or invalidly encoded.

Source: `lib/common.x:824`

Var.file_payload

```
File Var.file_payload(Var value)
```

Extracts the `File` payload of `value`, or NULL for another tag.

Source: `lib/common.x:669`

Var.float

```
meta native float Var.float(Var x)
```

Returns `x` as a native `float` under the `Var.convert` rules.

Raises: `Var.convert`'s causes when the source is nonnumeric, `void`, or invalidly encoded.

Source: `lib/common.x:815`

Var.int

```
int Var.int(Var x)
```

Returns `x` as a native `int` under the `Var.convert` rules.

Raises: `Var.convert`'s causes when the source is nonnumeric, `void`, out of floating range, or invalidly encoded.

Source: `lib/common.x:731`

Var.is_row

```
inline int Var.is_row( Var value, unsigned top, unsigned long mask, unsigned long bottom)
```

Reports whether `value` occupies the encoding row at `top`/`bottom`. `mask` selects the bits the decoder discriminates on within `top`'s group, so a caller that already knows the row tests it with two compares instead of a decode. `var-tags.x` projects the rows that qualify; a row whose decoded form carries a validity clause is not one of them and must ask `Var.is`.

Source: `lib/common.x:624`

Var.is_wide

```
inline int Var.is_wide(Var v)
```

Reports whether `v` uses a scope-owned wide numeric box.

Source: `lib/common.x:483`

Var.iter_payload

```
Iter Var.iter_payload(Var value)
```

Extracts the `Iter` payload of `value`, or NULL for another tag.

Source: `lib/common.x:670`

Var.list_payload

```
List Var.list_payload(Var value)
```

Extracts the `List` payload of `value`, or NULL for another tag.

Source: `lib/common.x:671`

Var.long

```
meta native long Var.long(Var x)
```

Returns `x` as a native `long` under the `Var.convert` rules.

Raises: `Var.convert`'s causes when the source is nonnumeric, `void`, out of floating range, or invalidly encoded.

Source: `lib/common.x:758`

Var.long_double

```
meta native long double Var.long_double(Var x)
```

Returns `x` as a native `long double` under the `Var.convert` rules.

Raises: `Var.convert`'s causes when the source is nonnumeric, `void`, or invalidly encoded.

Source: `lib/common.x:805`

Var.long_long

```
meta native long long Var.long_long(Var x)
```

Returns `x` as a native `long long` under the `Var.convert` rules.

Raises: `Var.convert`'s causes when the source is nonnumeric, `void`, out of floating range, or invalidly encoded.

Source: `lib/common.x:781`

Var.map_payload

```
Map Var.map_payload(Var value)
```

Extracts the `Map` payload of `value`, or NULL for another tag.

Source: `lib/common.x:672`

Var.payload32

```
inline unsigned Var.payload32(Var value)
```

Returns the low 32 payload bits of an immediate `Var`.

Source: `lib/common.x:489`

Var.short

```
meta native short Var.short(Var x)
```

Returns `x` as a native `short` under the `Var.convert` rules.

Raises: `Var.convert`'s causes when the source is nonnumeric, `void`, out of floating range, or invalidly encoded.

Source: `lib/common.x:713`

Var.string_payload

```
String Var.string_payload(Var value)
```

Extracts the `String` payload of `value`, or NULL for another tag.

Source: `lib/common.x:664`

Var.symbol

```
inline Symbol Var.symbol(Var x)
```

Extracts the `symbol` payload after the caller establishes the matching `Var` kind.

Source: `lib/common.x:681`

Var.uchar

```
meta native uchar Var.uchar(Var x)
```

Returns `x` as a native `uchar` under the `Var.convert` rules.

Raises: `Var.convert`'s causes when the source is nonnumeric, `void`, out of floating range, or invalidly encoded.

Source: `lib/common.x:704`

Var.uint

```
meta native uint Var.uint(Var x)
```

Returns `x` as a native `uint` under the `Var.convert` rules.

Raises: `Var.convert`'s causes when the source is nonnumeric, `void`, out of floating range, or invalidly encoded.

Source: `lib/common.x:740`

Var.ulong

`meta native ulong Var.ulong(Var x)`

Returns `x` as a native `ulong` under the `Var.convert` rules.

Raises: `Var.convert`'s causes when the source is nonnumeric, `void`, out of floating range, or invalidly encoded.

Source: `lib/common.x:770`

Var.ulong_long

`meta native unsigned long long Var.ulong_long(Var x)`

Returns `x` as a native `unsigned long long` under the `Var.convert` rules.

Raises: `Var.convert`'s causes when the source is nonnumeric, `void`, out of floating range, or invalidly encoded.

Source: `lib/common.x:793`

Var.unsigned

`meta native unsigned Var.unsigned(Var x)`

Returns `x` as a native `unsigned` under the `Var.convert` rules.

Raises: `Var.convert`'s causes when the source is nonnumeric, `void`, out of floating range, or invalidly encoded.

Source: `lib/common.x:749`

Var.ushort

`meta native ushort Var.ushort(Var x)`

Returns `x` as a native `ushort` under the `Var.convert` rules.

Raises: `Var.convert`'s causes when the source is nonnumeric, `void`, out of floating range, or invalidly encoded.

Source: `lib/common.x:722`

char

char.repr

`String char.repr(char x)`

Returns the readable representation of `char`.

Source: `lib/common.x:593`

char.str

`String char.str(char x)`

Returns the display `String` of `char`.

Source: `lib/common.x:593`

char.var

`Var char.var(char x)`

Boxes a native `char` value as `Var`.

Source: `lib/common.x:593`

double

double.repr

`String double.repr(double x)`

Returns the readable representation of `double`.

Source: `lib/common.x:602`

double.str

`String double.str(double x)`

Returns the display `String` of `double`.

Source: `lib/common.x:602`

double.var

`Var double.var(double x)`

Boxes a native `double` value as `Var`.

Source: `lib/common.x:602`

float

float.repr

`String float.repr(float x)`

Returns the readable representation of `float`.

Source: `lib/common.x:601`

float.str

`String float.str(float x)`

Returns the display `String` of `float`.

Source: `lib/common.x:601`

float.var

`Var float.var(float x)`

Boxes a native `float` value as `Var`.

Source: `lib/common.x:601`

int

int.repr

`String int.repr(int x)`

Returns the readable representation of `int`.

Source: `lib/common.x:597`

int.str

`String int.str(int x)`

Returns the display `String` of `int`.

Source: `lib/common.x:597`

int.var

`Var int.var(int x)`

Boxes a native `int` value as `Var`.

Source: `lib/common.x:597`

long

long.repr

```
inline String long.repr(long l)
```

Returns the readable representation of `long`.

Source: `lib/common.x:613`

long.str

```
inline String long.str(long l)
```

Returns the display `String` of `long`.

Source: `lib/common.x:611`

long.var

```
inline Var long.var(long x)
```

Boxes a long value as `Var`.

Source: `lib/common.x:607`

short

short.repr

```
String short.repr(short x)
```

Returns the readable representation of `short`.

Source: `lib/common.x:595`

short.str

```
String short.str(short x)
```

Returns the display `String` of `short`.

Source: `lib/common.x:595`

short.var

```
Var short.var(short x)
```

Boxes a native `short` value as `Var`.

Source: `lib/common.x:595`

uchar

uchar.repr

```
String uchar.repr(uchar x)
```

Returns the readable representation of `uchar`.

Source: `lib/common.x:594`

uchar.str

```
String uchar.str(uchar x)
```

Returns the display `String` of `uchar`.

Source: `lib/common.x:594`

uchar.var

`Var uchar.var(uchar x)`

Boxes a native `uchar` value as `Var`.

Source: `lib/common.x:594`

`uint`

uint.repr

`String uint.repr(uint x)`

Returns the readable representation of `uint`.

Source: `lib/common.x:598`

uint.str

`String uint.str(uint x)`

Returns the display `String` of `uint`.

Source: `lib/common.x:598`

uint.var

`Var uint.var(uint x)`

Boxes a native `uint` value as `Var`.

Source: `lib/common.x:598`

`ulong`

ulong.var

`inline Var ulong.var(ulong x)`

Boxes a `ulong` value as `Var`.

Source: `lib/common.x:609`

`unsigned`

unsigned.repr

`String unsigned.repr(unsigned x)`

Returns the readable representation of `unsigned`.

Source: `lib/common.x:599`

unsigned.str

`String unsigned.str(unsigned x)`

Returns the display `String` of `unsigned`.

Source: `lib/common.x:599`

unsigned.var

`Var unsigned.var(unsigned x)`

Boxes a native `unsigned` value as `Var`.

Source: `lib/common.x:599`

ushort

ushort.repr

`String ushort.repr(ushort x)`

Returns the readable representation of `ushort`.

Source: `lib/common.x:596`

ushort.str

`String ushort.str(ushort x)`

Returns the display `String` of `ushort`.

Source: `lib/common.x:596`

ushort.var

`Var ushort.var(ushort x)`

Boxes a native `ushort` value as `Var`.

Source: `lib/common.x:596`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`x2c_hash_bytes`	Hashes an arbitrary byte block through the shared 64-bit mixer.
`x2c_hash_word`	Narrows a mixed word to the nonzero 32-bit hash the tables expect.
`x2c_mix64`	Avalanches one 64-bit word.

Functions

x2c_hash_bytes

`inline unsigned x2c_hash_bytes( unsigned long seed, const void *data, size_t width)`

Hashes an arbitrary byte block through the shared 64-bit mixer. `seed` distinguishes callers that hash the same bytes under different types; the byte width distinguishes a final partial word from zero padding.

Source: `lib/common.x:462`

x2c_hash_word

`inline unsigned x2c_hash_word(unsigned long word)`

Narrows a mixed word to the nonzero 32-bit hash the tables expect. Zero is reserved to mark an empty bucket, so it maps to all ones.

Source: `lib/common.x:452`

x2c_mix64

`inline unsigned long x2c_mix64(unsigned long word)`

Avalanches one 64-bit word. This is MurmurHash3's fmix64 finalizer, the shared mixer behind every fixed-width hash in the runtime. It is six instructions, and its low bits are as well distributed as its high ones, which is what `Map` masks. Use `x2c_hash_bytes` for a block that spans several words.

Source: `lib/common.x:440`

Public types

Type	Kind	Summary
`Array`	alias	`Scope`-owned mutable `Block` view whose elements are `Var`s.

Type	Kind	Summary
Atom	alias	Exact-name Var using a Symbol or a pooled canonical long spelling.
Block	struct	Opaque handle to Scope-owned mutable fixed-width storage.
Buffer	struct	Opaque handle to a Scope-owned mutable text builder.
Bytes	alias	Data pointer into Block-owned raw element storage.
Context	struct	Bounded Scope, Error, Match, and optional canonical-pool state.
File	alias	Native stdio stream handle whose opener determines close ownership.
Func	struct	Scope-owned generic native-call adapter and optional bound context.
Iter	struct	Mutable traversal state whose sources must outlive its iteration.
List	struct	Immutable canonical cons chain; NULL is nil.
Map	struct	Scope-owned mutable table mapping Var keys to Var values.
Mutex	struct	Scope-owned opaque mutex that must outlive every accessing thread.
Pool	struct	Nested canonicalization table with Scope-backed object storage.
RenderPath	struct	Caller-owned frame in this thread's active recursive rendering path.
Scope	struct	Region owner for individually tracked runtime allocations.
String	alias	Immutable canonical NUL-terminated bytes; NULL is the empty String.
Symbol	alias	Immediate compact name encoding with no allocation ownership.
SymbolSet	alias	Borrowed immutable compiler-generated set of compact Symbols.
Thread	struct	Caller-owned handle to a Context-backed worker.
Var	union	Eight-byte tagged runtime value for immediate scalars and encoded handles.
VarBinaryFn	callback	Applies one descriptor-owned binary operation and returns its result.
VarCompareFn	callback	Returns negative, zero, or positive for two Vars of one descriptor.
VarContainsFn	callback	Reports whether receiver contains needle.
VarDescriptor	struct	Process-global dispatch row for one Var tag, never freed.
VarEqualFn	callback	Reports value equality for two Vars with the same descriptor.
VarExportContextFn	callback	Exports a custom value from source into its destination Context.
VarGetIndexFn	callback	Reads one dynamically indexed value.
VarHashFn	callback	Computes one Var hash; Var.hash normalizes a zero result.
VarIterIntoFn	callback	Initializes traversal in caller-provided Iter storage and returns it.
VarMethods	struct	Sparse process-global callback row for one Var descriptor.
VarPostfixIndexFn	callback	Applies an indexed postfix operation and returns the prior value.
VarSetIndexFn	callback	Stores an indexed value and returns the assignment result.
VarStrFn	callback	Renders one Var as a display or readable String.
VarTruthFn	callback	Reports whether a Var is true.
VarUnaryFn	callback	Applies one descriptor-owned unary operation and returns its result.
VarUpdateIndexFn	callback	Applies an indexed compound operation and returns the stored result.
VarWriteFn	callback	Appends one Var rendering to a caller-owned Buffer and returns it.
X2CErrorSite	struct	Borrowed source location supplied during one compiler-generated raise.
X2CStatic	struct	Zero-initialized compiler storage for one dynamic local static.
uchar	alias	Unsigned native character type.
uint	alias	Unsigned native int type.
ulong	alias	Unsigned native long type.
ushort	alias	Unsigned native short type.

Array

`typedef Block Array`

`Scope`-owned mutable `Block` view whose elements are `Var`s.

Source: `lib/common.x:80`

Atom

`typedef Var Atom`

Exact-name `Var` using a `Symbol` or a pooled canonical long spelling.

Source: `lib/common.x:78`

Block

`typedef struct Block *Block`

Opaque handle to `Scope`-owned mutable fixed-width storage.

Source: `lib/common.x:70`

Buffer

`typedef struct Buffer *Buffer`

Opaque handle to a `Scope`-owned mutable text builder.

Source: `lib/common.x:72`

Bytes

`typedef void *Bytes`

Data pointer into `Block`-owned raw element storage.

Source: `lib/common.x:68`

Context

`typedef struct Context *Context`

Bounded `Scope`, `Error`, `Match`, and optional canonical-pool state.

Source: `lib/common.x:92`

File

`typedef FILE *File`

Native stdio stream handle whose opener determines close ownership.

Source: `lib/common.x:106`

Func

`typedef struct Func *Func`

`Scope`-owned generic native-call adapter and optional bound context.

Source: `lib/common.x:100`

Iter

`typedef struct Iter *Iter`

Mutable traversal state whose sources must outlive its iteration.

Source: `lib/common.x:108`

List

`typedef struct List *List`

Immutable canonical cons chain; NULL is `nil`. Its lifetime follows its owning `List` pool, which may be an ancestor.

Source: `lib/common.x:84`

Map

`typedef struct Map *Map`

`Scope`-owned mutable table mapping `Var` keys to `Var` values.

Source: `lib/common.x:88`

Mutex

`typedef struct Mutex *Mutex`

`Scope`-owned opaque mutex that must outlive every accessing thread.

Source: `lib/common.x:94`

Pool

`typedef struct Pool *Pool`

Nested canonicalization table with `Scope`-backed object storage.

Source: `lib/common.x:90`

RenderPath

`typedef struct RenderPath { const void *identity; struct RenderPath *previous; } RenderPath`

Caller-owned frame in this thread's active recursive rendering path.

Source: `lib/common.x:124`

Scope

`typedef struct Scope *Scope`

Region owner for individually tracked runtime allocations.

Source: `lib/common.x:86`

String

`typedef char *String`

Immutable canonical NUL-terminated bytes; NULL is the empty `String`. Its lifetime follows its owning `String` pool, which may be an ancestor.

Source: `lib/common.x:104`

Symbol

`typedef unsigned long Symbol`

Immediate compact name encoding with no allocation ownership.

Source: `lib/common.x:74`

SymbolSet

```
typedef unsigned char *SymbolSet
```

Borrowed immutable compiler-generated set of compact `Symbols`.

Source: `lib/common.x:76`

Thread

```
typedef struct Thread *Thread
```

Caller-owned handle to a `Context`-backed worker. Join the worker before freeing the handle.

Source: `lib/common.x:98`

Var

```
typedef union Var { unsigned long u64; double f64, void *p64; } Var
```

Eight-byte tagged runtime value for immediate scalars and encoded handles. Pointer-bearing values follow the ownership rules of their concrete type; raw `Null` is all zero bits and `void` is all one bits.

Source: `lib/common.x:27`

VarBinaryFn

```
typedef Var (*VarBinaryFn)(Var lhs, Var rhs)
```

Applies one descriptor-owned binary operation and returns its result.

Source: `lib/common.x:151`

VarCompareFn

```
typedef int (*VarCompareFn)(Var, Var)
```

Returns negative, zero, or positive for two `Var`s of one descriptor.

Source: `lib/common.x:145`

VarContainsFn

```
typedef int (*VarContainsFn)(Var receiver, Var needle)
```

Reports whether `receiver` contains `needle`.

Source: `lib/common.x:149`

VarDescriptor

```
typedef struct VarDescriptor { Symbol tag, int row, value_dispatch; VarMethods methods; String
name; } VarDescriptor
```

Process-global dispatch row for one `Var` tag, never freed. It borrows native function pointers and the canonical `name`: callback code must remain loaded, and the name's owning pool must outlive every later use of the row. A custom descriptor's `row` is −1 until its first box assigns one.

Source: `lib/common.x:193`

VarEqualFn

```
typedef int (*VarEqualFn)(Var, Var)
```

Reports value equality for two `Var`s with the same descriptor.

Source: `lib/common.x:139`

VarExportContextFn

```
typedef Var (*VarExportContextFn)(Var value, Context source)
```

Exports a custom value from `source` into its destination `Context`. Implementations move owned storage and recursively export nested values.

Source: `lib/common.x:165`

VarGetIndexFn

```
typedef Var (*VarGetIndexFn)(Var receiver, Var key)
```

Reads one dynamically indexed value.

Source: `lib/common.x:155`

VarHashFn

```
typedef unsigned (*VarHashFn)(Var)
```

Computes one `Var` hash; `Var.hash` normalizes a zero result.

Source: `lib/common.x:137`

VarIterIntoFn

```
typedef Iter (*VarIterIntoFn)(Var, Iter)
```

Initializes traversal in caller-provided `Iter` storage and returns it. The storage and any retained receiver state must outlive traversal.

Source: `lib/common.x:143`

VarMethods

```
typedef struct VarMethods { VarStrFn str, repr, VarHashFn hash, VarEqualFn equal, VarCompareFn compare; VarTruthFn truth, VarIterIntoFn iter, VarWriteFn write_str, write_repr; VarContainsFn contains, VarBinaryFn add, sub, mul, div, mod, matmul; VarUnaryFn neg; VarGetIndexFn getindex, VarSetIndexFn setindex, VarUpdateIndexFn updateindex; VarPostfixIndexFn postfixindex, VarExportContextFn export_context; } VarMethods
```

Sparse process-global callback row for one `Var` descriptor. Registration stores each non-NULL function pointer without ownership and leaves existing slots unchanged for NULL fields. Callback code must remain loaded until its pointer is replaced or the process ends. Callbacks run synchronously when their operation dispatches; registration invokes none and retains no callback context.

Source: `lib/common.x:178`

VarPostfixIndexFn

```
typedef Var (*VarPostfixIndexFn)(Var receiver, Var key, Symbol op)
```

Applies an indexed postfix operation and returns the prior value.

Source: `lib/common.x:161`

VarSetIndexFn

```
typedef Var (*VarSetIndexFn)(Var receiver, Var key, Var value)
```

Stores an indexed value and returns the assignment result.

Source: `lib/common.x:157`

VarStrFn

```
typedef String (*VarStrFn)(Var)
```

Renders one `Var` as a display or readable `String`.
Source: `lib/common.x:133`

VarTruthFn

```
typedef int (*VarTruthFn)(Var)
```

Reports whether a `Var` is true.
Source: `lib/common.x:147`

VarUnaryFn

```
typedef Var (*VarUnaryFn)(Var value)
```

Applies one descriptor-owned unary operation and returns its result.
Source: `lib/common.x:153`

VarUpdateIndexFn

```
typedef Var (*VarUpdateIndexFn)(Var receiver, Var key, Symbol op, Var rhs)
```

Applies an indexed compound operation and returns the stored result.
Source: `lib/common.x:159`

VarWriteFn

```
typedef Buffer (*VarWriteFn)(Var, Buffer)
```

Appends one `Var` rendering to a caller-owned `Buffer` and returns it.
Source: `lib/common.x:135`

X2CErrorSite

```
typedef struct X2CErrorSite { const char *file, *function, int line, runtime; } X2CErrorSite
```

Borrowed source location supplied during one compiler-generated raise. The pointed-to file and function spellings must outlive that call. `runtime` is set for a site compiled inside the runtime library.
Source: `lib/common.x:208`

X2CStatic

```
typedef struct X2CStatic { int ready; void *payload, *owner; struct X2CStatic *next; } X2CStatic
```

Zero-initialized compiler storage for one dynamic local static. The runtime owns its fields. `payload` becomes available to the winning initializer before publication; an acquire returning zero observes the committed value. Failed attempts retain their address until teardown. Payload storage retains no referenced runtime objects.
Source: `lib/common.x:243`

uchar

```
typedef unsigned char uchar
```

Unsigned native character type.
Source: `lib/common.x:113`

uint

```
typedef unsigned int uint
```

Unsigned native int type.
Source: `lib/common.x:117`

ulong

```
typedef unsigned long ulong
```

Unsigned native long type.

Source: `lib/common.x:119`

ushort

```
typedef unsigned short ushort
```

Unsigned native short type.

Source: `lib/common.x:115`

Design notes

Every runtime module includes this one, so it owns what all of them share: the eight-byte `Var` encoding and its immediate boxing, and the declarations through which modules name each other's types and operations without an include cycle. Pointer-shaped `Var` extractors are ABI crossings whose callers establish the source kind; scalar readers convert other numeric tags through `Var.convert`.

lib/context.x

Bounded runtime state and value export.

Primary API

Function	Summary
`Context.cleanup`	Ends the owned lifetime when a managed local leaves its block.
`Context.close`	Closes the current `Context`, reclaims unexported state, and restores parent.
`Context.current`	Returns the `Context` currently active on this thread, or NULL.
`Context.export`	Exports `value` into the parent of the current `Context`.
`Context.open`	Opens and makes current a `Context` using the active canonical-value pool.
`Context.open_isolated`	Opens and makes current a `Context` with a private canonical-value pool.
`Context.open_isolated_named`	Opens and makes current a named `Context` with a private canonical pool.
`Context.open_named`	Opens and makes current a named `Context` that inherits immutable values.

Context

Context.cleanup

```
meta native void Context.cleanup(Context value)
```

Ends the owned lifetime when a managed local leaves its block.

Source: `lib/context.x:355`

Context.close

```
meta native void Context.close(Context c)
```

Closes the current `Context`, reclaims unexported state, and restores parent. The `Context` must be current; all pointers into its remaining `Scope` or private canonical pool become invalid.

Raises: `<bad-state>` for a null or noncurrent `Context`, or while its `Match` cache has an active lease. The failure leaves the `Context` active.

Source: `lib/context.x:331`

Context.current

```
meta native Context Context.current(void)
```

Returns the `Context` currently active on this thread, or NULL.

Source: `lib/context.x:321`

Context.export

```
meta native Var Context.export(Context context, Var value)
```

Exports `value` into the parent of the current `Context`. Values may belong to a retained region of that `Context`; export them while live, before releasing their `Scope` or closing the `Context`. Built-in movable families preserve identity. A custom exact-descriptor exporter defines its returned value, including identity. Private canonical immutable built-ins may change, so callers must use the returned value; inherited or borrowed built-ins are unchanged. `void` exports as `void` because it owns no storage.

Built-in `Array` and `Map` export restores the container's source ownership if recursive export fails, but nested exports already completed are not rolled back.

Raises: `<bad-state>` unless `context` is current, `<bad-types>` for an unsupported value without a registered exporter, or a cause from nested allocation, hashing, equality, or custom export.

Source: `lib/context.x:72`

Context.open

```
meta native Context Context.open(void)
```

Opens and makes current a `Context` using the active canonical-value pool. Close it before its parent; its `Scope` owns subsequent mutable allocations.

Raises: `<alloc-fail>` or `<size-limit>` while constructing nested state. Failed construction restores the parent's `Scope`, canonical pool, `Error`, `Match`, and current `Context` state.

Source: `lib/context.x:258`

Context.open_isolated

```
Context Context.open_isolated(void)
```

Opens and makes current a `Context` with a private canonical-value pool. Export surviving immutable values before closing the `Context`.

Raises: `<alloc-fail>` or `<size-limit>` while constructing nested state. Failed construction restores the parent's `Scope`, canonical pool, `Error`, `Match`, and current `Context` state.

Source: `lib/context.x:275`

Context.open_isolated_named

```
Context Context.open_isolated_named(const char *name)
```

Opens and makes current a named `Context` with a private canonical pool. The diagnostic name is copied; export survivors before closing the `Context`.

Raises: `<alloc-fail>` or `<size-limit>` while constructing nested state. Failed construction restores the parent's `Scope`, canonical pool, `Error`, `Match`, and current `Context` state.

Source: `lib/context.x:284`

Context.open_named

```
Context Context.open_named(const char *name)
```

Opens and makes current a named `Context` that inherits immutable values. The diagnostic name is copied, and the `Context` must close before its parent.

Raises: `<alloc-fail>` or `<size-limit>` while constructing nested state. Failed construction restores the parent's `Scope`, canonical pool, `Error`, `Match`, and current `Context` state.

Source: `lib/context.x:267`

Advanced and interop API

Function	Summary
`Context.move_allocation`	Moves one custom-exporter-owned allocation to the destination `Context`.
`Context.owns`	Reports whether `allocation` belongs to `context`'s `Scope` chain.

`Context`

Context.move_allocation

```
void Context.move_allocation(Context c, void *allocation)
```

Moves one custom-exporter-owned allocation to the destination `Context`. A null allocation or one outside `c`'s `Scope` chain is unchanged. Application code exports its value with `Context.export` instead.

Source: `lib/context.x:245`

Context.owns

```
int Context.owns(Context context, void *allocation)
```

Reports whether `allocation` belongs to `context`'s `Scope` chain. The nonnull pointer must come from a `Scope` allocator. Registered custom exporters call this before moving their own storage.

Source: `lib/context.x:225`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`Context.export_destination`	Returns the borrowed `Scope` slot that receives exports from `c`.
`Context.export_nested`	Recursively exports one nested value through an active or sealed `Context`.
`Context.export_scope`	Exports a sealed worker result `Scope` into the current `Scope` and pool.
`Context.initialize`	Registers `Context` cleanup before workers can start.

`Context`

Context.export_destination

```
Scope *Context.export_destination(Context c)
```

Returns the borrowed `Scope` slot that receives exports from `c`. The slot remains valid only while the `Context`'s destination state lives; a null `Context` returns NULL.

Source: `lib/context.x:239`

Context.export_nested

```
Var Context.export_nested(Context c, Var value)
```

Recursively exports one nested value through an active or sealed `Context`. Registered custom exporters call this; other callers export their complete result with `Context.export`. Built-in movable families preserve identity. A custom exact-descriptor exporter defines its returned value, including identity. For built-ins, private canonical pointers may change, so callers must use the result; inherited or borrowed values are unchanged.

Built-in `Array` and `Map` export restores the container's source ownership if recursive export fails, but nested exports already completed are not rolled back.

Raises: `<bad-types>` for an unsupported value without a registered exporter, or a cause from nested allocation, hashing, equality, or custom export.

Source: `lib/context.x:219`

Context.export_scope

```
Var Context.export_scope(Scope source_scope, Pool pool, Var value)
```

Exports a sealed worker result `Scope` into the current `Scope` and pool. Built-in movable families preserve identity. A custom exact-descriptor exporter defines its returned value, including identity. Private canonical built-ins may change, so callers must use the returned value; inherited or borrowed built-ins are unchanged. The source `Scope` and pool remain caller-owned through the call.

Built-in `Array` and `Map` export restores the container's source ownership if recursive export fails, but nested exports already completed are not rolled back.

Raises: `<bad-types>` for an unsupported value without a registered exporter, or a cause from nested allocation, hashing, equality, or custom export.

Source: `lib/context.x:93`

Context.initialize

```
void Context.initialize(void)
```

Registers `Context` cleanup before workers can start.

Source: `lib/context.x:358`

Design notes

Context owns one bounded unit of runtime work: a `Scope` lifetime with its `Error` and `Match` state and, when isolated, a nested canonical-value pool. Export carries a value out before close reclaims the rest: it moves built-in storage the source `Scope` owns and recanonicalizes immutable values only when the source's private pool owns them.

Tests and examples

`make verify` (`unittest/test-context.x`).

lib/diff.x

Line differences between two texts.

Primary API

Function	Summary
`Diff.lines`	Returns the line edits that turn `old` into `new`: a `List` of `(same line)`, `(delete line)`, and `(insert line)` forms in order, with each line's ending removed.
`Diff.unified`	Returns the unified difference between `old` and `new`, as `diff -u` prints it with `old_name` and `new_name` in the header and three lines of context, or NULL when the texts are equal line for line.

Diff

Diff.lines

```
meta native List Diff.lines(String old, String new)
```

Returns the line edits that turn `old` into `new`: a `List` of `(same line)`, `(delete line)`, and `(insert line)` forms in order, with each line's ending removed. Two equal texts give only `same` forms. Past 2,000 edits the differing middle is one run of deletions followed by one run of insertions.

Source: `lib/diff.x:34`

Diff.unified

```
meta native String Diff.unified( String old, String new, String old_name, String new_name)
```

Returns the unified difference between `old` and `new`, as `diff -u` prints it with `old_name` and `new_name` in the header and three lines of context, or NULL when the texts are equal line for line.

Source: `lib/diff.x:179`

Public types

Type	Kind	Summary
`Diff`	alias	The receiverless owner of the difference operations.

Diff

```
typedef Symbol Diff
```

The receiverless owner of the difference operations.

Source: `lib/diff.x:16`

Design notes

Diff owns line differences: a shortest edit script between two texts, found with the Myers algorithm after trimming the lines both share at each end, and its unified spelling. Each search step keeps only its frontier, so the cost is the square of the edit distance; past `_LIMIT` edits the middle becomes one deletion run and one insertion run.

Tests and examples

`make verify` (`unittest/test-diff.x`).

lib/digest.x

SHA-256 digests of Strings and streams.

Primary API

Function	Summary
`File.sha256`	Returns the SHA-256 digest of the bytes from the current position of `file` to its end, leaving the stream at end of file and open.
`String.sha256`	Returns the SHA-256 digest of the bytes of `text`; NULL is empty text.

File

File.sha256

```
String File.sha256(File file)
```

Returns the SHA-256 digest of the bytes from the current position of `file` to its end, leaving the stream at end of file and open.

Raises: `<io-fail>` on a stream read error.

Source: `lib/digest.x:118`

String

String.sha256

```
String String.sha256(String text)
```

Returns the SHA-256 digest of the bytes of `text`; NULL is empty text.

Source: `lib/digest.x:108`

Design notes

A digest is the 64-character lowercase hexadecimal `String` that `shasum -a 256` prints. A stream digest reads raw bytes, so a file that holds NUL bytes, which a `String` cannot, hashes like any other.

Tests and examples

`make verify` (`unittest/test-digest.x`).

`lib/dispatch.x`

Typed descriptors for runtime `Var` behavior.

Primary API

Function	Summary
`Var.contains`	Tests dynamic membership through the receiver's registered protocol row.
`Var.enumerate`	Returns an iterator over what a two-name `foreach` destructures.
`Var.fallback_compare`	Compares without consulting a runtime descriptor.
`Var.fallback_equal`	Applies non-dispatch equality to `a` and `b`.
`Var.fallback_hash`	Returns the non-dispatch runtime hash of `Var`.
`Var.fallback_iter`	Returns a non-dispatch iterator over `Var`.
`Var.fallback_repr`	Returns the non-dispatch readable representation of `Var`.
`Var.fallback_str`	Returns the non-dispatch display `String` of `Var`.
`Var.fallback_write_repr`	Appends the non-dispatch representation of `Var` to a `Buffer`.
`Var.fallback_write_str`	Appends the non-dispatch display text of `Var` to a `Buffer`.
`Var.getindex`	Reads a dynamic indexed value through the receiver's protocol row.
`Var.iter`	Returns an iterator over `Var`.
`Var.postfixindex`	Applies a dynamic postfix update at `key` and returns its prior value.
`Var.setindex`	Stores and returns a dynamic indexed value through its protocol row.
`Var.updateindex`	Applies the registered dynamic compound update at `key` and returns its result.
`Var.write_str`	Appends the display text of `Var` to a `Buffer`.

`Var`

Var.contains

```
meta native int Var.contains(Var value, Var needle)
```

Tests dynamic membership through the receiver's registered protocol row.

Raises: `<bad-enc>`, `<void-op>`, or `<no-member>` when the dynamic receiver cannot perform membership.

Source: `lib/dispatch.x:591`

Var.enumerate

```
meta native Iter Var.enumerate(Var x, Iter dest)
```

Returns an iterator over what a two-name `foreach` destructures. A boxed `Map` yields `(key value)` `Lists`, as `Map.enumerate` does, and any other value yields what `Var.iter` yields.

Raises: `<void-op>` for `void`. A null `dest` returns NULL without raising.

Source: `lib/dispatch.x:570`

Var.fallback_compare

```
int Var.fallback_compare(Var a, Var b)
```

Compares without consulting a runtime descriptor.

Raises: `<void-op>` when either operand is `void`.

Source: `lib/dispatch.x:442`

Var.fallback_equal

`int Var.fallback_equal(Var a, Var b)`

Applies non-dispatch equality to `a` and `b`.

Source: `lib/dispatch.x:401`

Var.fallback_hash

`unsigned Var.fallback_hash(Var v)`

Returns the non-dispatch runtime hash of `Var`.

Raises: `<void-op>` for `void`.

Source: `lib/dispatch.x:362`

Var.fallback_iter

`Iter Var.fallback_iter(Var x, Iter dest)`

Returns a non-dispatch iterator over `Var`.

Raises: `<void-op>` for `void`. A null `dest` returns NULL without raising.

Source: `lib/dispatch.x:579`

Var.fallback_repr

`String Var.fallback_repr(Var v)`

Returns the non-dispatch readable representation of `Var`.

Source: `lib/dispatch.x:188`

Var.fallback_str

`String Var.fallback_str(Var v)`

Returns the non-dispatch display `String` of `Var`.

Source: `lib/dispatch.x:91`

Var.fallback_write_repr

`Buffer Var.fallback_write_repr(Var v, Buffer out)`

Appends the non-dispatch representation of `Var` to a `Buffer`.

Source: `lib/dispatch.x:217`

Var.fallback_write_str

`Buffer Var.fallback_write_str(Var v, Buffer out)`

Appends the non-dispatch display text of `Var` to a `Buffer`. A primitive, pointer, or `void` renders straight into `out` instead of through an intermediate `String`.

Source: `lib/dispatch.x:149`

Var.getindex

`meta native Var Var.getindex(Var value, Var key)`

Reads a dynamic indexed value through the receiver's protocol row.

Raises: `<bad-enc>`, `<void-op>`, or `<no-member>` when the dynamic receiver cannot be indexed.

Source: `lib/dispatch.x:606`

Var.iter

```
meta native Iter Var.iter(Var x, Iter dest)
```

Returns an iterator over `Var`.

Raises: `<void-op>` for `void`. A null `dest` returns NULL without raising.

Source: `lib/dispatch.x:555`

Var.postfixindex

```
meta native Var Var.postfixindex(Var value, Var key, Symbol op)
```

Applies a dynamic postfix update at `key` and returns its prior value.

Raises: `<bad-enc>`, `<void-op>`, `<no-member>`, or a cause from the receiver's indexed update.

Source: `lib/dispatch.x:655`

Var.setindex

```
meta native Var Var.setindex(Var value, Var key, Var replacement)
```

Stores and returns a dynamic indexed value through its protocol row.

Raises: `<bad-enc>`, `<void-op>`, `<no-member>`, or a cause from the receiver's indexed assignment.

Source: `lib/dispatch.x:621`

Var.updateindex

```
meta native Var Var.updateindex(Var value, Var key, Symbol op, Var rhs)
```

Applies the registered dynamic compound update at `key` and returns its result. Mutation and failure behavior belong to that callback; this dispatch adds no thread or failure atomicity guarantee.

Raises: `<bad-enc>`, `<void-op>`, `<no-member>`, or a cause from the receiver's indexed update.

Source: `lib/dispatch.x:639`

Var.write_str

```
Buffer Var.write_str(Var v, Buffer out)
```

Appends the display text of `Var` to a `Buffer`. A descriptor that registers `write_str` streams straight into `out`. One that registers only `str` writes its `String` through, which materializes the text but keeps existing custom descriptors working. Neither fallback re-enters this function, so a descriptor providing neither cannot recurse.

Source: `lib/dispatch.x:131`

Advanced and interop API

Function	Summary
x2c_register_builtin_descriptor	Merges callbacks into one built-in `Var` descriptor.
x2c_register_descriptor	Attempts to reserve a custom `Var` tag and merge its descriptor callbacks.
x2c_register_tagged_descriptor	Registers an explicitly tagged type or raises `<bad-state>` for a null tag or name.
x2c_register_type	Attempts to make lowercase `name` available as a process-global `Var` tag.
x2c_try_register_descriptor	Declares a custom `Var` tag and merges its descriptor callbacks.
x2c_try_register_tagged_descriptor	Declares custom `tag` under full type `name` and merges descriptor callbacks.
RenderPath.enter	Enters an object's recursive rendering, or returns zero for a cycle.
RenderPath.leave	Restores the path after the most recent successful `enter` on this thread.
Var.as_string	Returns what a `String` destination reads from `Var`: its `String` payload, the display `String` of any other value, or NULL for `void` and for `Null`, the null pointer.
Var.compare	Compares `a` and `b` by runtime value group and registered ordering.

Function	Summary
Var.dispatch_truth	Tries the registered truth callback for value.
Var.equal	Applies the registered equality operation for a and b.
Var.hash	Returns the runtime hash of Var.
Var.pointer_string	Formats the fallback display String for a pointer-bearing Var.
Var.repr	Returns the readable representation of Var.
Var.same	Reports whether a and b have identical Var bits.
Var.str	Returns the display String of Var.
Var.try_dispatch_binary	Tries one registered binary callback for lhs.
Var.try_dispatch_unary	Tries the registered <neg> callback for value.
Var.write_pointer_repr	Writes the fallback readable form of a pointer-bearing Var.
Var.write_repr	Appends the readable representation of Var to a Buffer.

Functions

x2c_register_builtin_descriptor

```
int x2c_register_builtin_descriptor(Symbol tag, VarMethods methods)
```

Merges callbacks into one built-in Var descriptor. Returns zero for a tag without a reserved descriptor and one otherwise. Non-NULL method fields replace their process-global slots; NULL fields preserve installed callbacks, and no callback runs during registration. Function pointers are borrowed, so their code must remain loaded through later dispatch or replacement.

Raises: <bad-state> after descriptor registration is frozen.

Source: lib/dispatch.x:782

x2c_register_descriptor

```
void x2c_register_descriptor(String name, VarMethods methods)
```

Attempts to reserve a custom Var tag and merge its descriptor callbacks. This no-result form discards the status from x2c_try_register_descriptor; all registration and freeze behavior is otherwise identical. The name and function pointers are borrowed under the same process-wide lifetime requirements.

Raises: <bad-state> after descriptor registration is frozen, or <alloc-fail> while checking lowercase spelling. Distinct full names that encode one Symbol through _/- folding or truncation abort.

Source: lib/dispatch.x:829

x2c_register_tagged_descriptor

```
void x2c_register_tagged_descriptor( Symbol tag, String name, VarMethods methods)
```

Registers an explicitly tagged type or raises <bad-state> for a null tag or name. Tag/name collisions follow the existing fatal descriptor diagnostic; registration still freezes at worker startup.

Source: lib/dispatch.x:860

x2c_register_type

```
void x2c_register_type(String name)
```

Attempts to make lowercase name available as a process-global Var tag. This no-result form silently ignores invalid names. A custom tag spends a Var row only when a value is first boxed. An active built-in name selects its existing row. The canonical name is borrowed for process-wide collision diagnostics, so its owning pool must outlive later descriptor use.

Raises: <bad-state> after descriptor registration is frozen, or <alloc-fail> while checking lowercase spelling. Distinct full names that encode one Symbol through _/- folding or truncation abort.

Source: lib/dispatch.x:765

x2c_try_register_descriptor

```
int x2c_try_register_descriptor(String name, VarMethods methods)
```

Declares a custom `Var` tag and merges its descriptor callbacks. Returns zero for a null or non-lowercase name, and one otherwise. An already active built-in name selects its existing row. Non-NULL fields replace process-global slots; NULL fields preserve installed callbacks. The canonical name and function pointers are borrowed process-wide: the name's owning pool and callback code must outlive later descriptor use. Registration invokes no callback.

Raises: `<bad-state>` after descriptor registration is frozen, or `<alloc-fail>` while checking lowercase spelling. Distinct full names that encode one `Symbol` through _/- folding or truncation abort.

Source: `lib/dispatch.x:807`

x2c_try_register_tagged_descriptor

```
int x2c_try_register_tagged_descriptor( Symbol tag, String name, VarMethods methods)
```

Declares custom `tag` under full type `name` and merges descriptor callbacks. Unlike `x2c_try_register_descriptor`, the tag need not be the restricted-Symbol encoding of the name. A built-in tag is rejected so an explicit custom type cannot replace built-in behavior. The name and callbacks have the same process-wide lifetime as ordinary descriptors.

Returns zero for a null tag or name, and one otherwise. A built-in tag or distinct names for one tag abort. Explicit names retain case; ordinary inferred-tag registration still requires lowercase names. Raises: `<bad-state>` after registration freezes.

Source: `lib/dispatch.x:844`

RenderPath

RenderPath.enter

```
int RenderPath.enter(RenderPath *path, const void *identity)
```

Enters an object's recursive rendering, or returns zero for a cycle. Keep `path` alive and defer `path.leave()` after a successful entry. Identities are compared only along this thread's active path, so repeated references outside that path render independently. No allocation occurs.

Source: `lib/dispatch.x:318`

RenderPath.leave

```
void RenderPath.leave(RenderPath *path)
```

Restores the path after the most recent successful `enter` on this thread. Ordinary defer unwinding also restores it when a child renderer raises.

Source: `lib/dispatch.x:330`

Var

Var.as_string

```
String Var.as_string(Var v)
```

Returns what a `String` destination reads from `Var`: its `String` payload, the display `String` of any other value, or NULL for `void` and for `Null`, the null pointer.

Source: `lib/dispatch.x:84`

Var.compare

```
meta native int Var.compare(Var a, Var b)
```

Compares `a` and `b` by runtime value group and registered ordering. The order is total, so sorting uses it: numbers order by value, NaN after every other number, and equal values of different types by type. The relational operators compare two numbers as C does instead.

Raises: `<void-op>` when either operand is `void`.

Source: `lib/dispatch.x:425`

Var.dispatch_truth

```
int Var.dispatch_truth(Var value, int &?handled)
```

Tries the registered truth callback for `value`. A null `handled`, missing descriptor, or missing callback returns zero. Otherwise `handled` is set to one and the synchronous callback result is returned; an available `handled` is cleared before lookup.

Source: `lib/dispatch.x:688`

Var.equal

```
int Var.equal(Var a, Var b)
```

Applies the registered equality operation for `a` and `b`. Two numbers are equal only with the same tag and the same bits, so this agrees with `Var.hash`; Map and Pool keys compare this way. The == operator compares two numbers as C does instead.

Source: `lib/dispatch.x:378`

Var.hash

```
unsigned Var.hash(Var v)
```

Returns the runtime hash of `Var`.

Raises: `<void-op>` for `void`.

Source: `lib/dispatch.x:339`

Var.pointer_string

```
String Var.pointer_string(Var v)
```

Formats the fallback display `String` for a pointer-bearing `Var`.

Source: `lib/dispatch.x:118`

Var.repr

```
String Var.repr(Var v)
```

Returns the readable representation of `Var`.

Source: `lib/dispatch.x:180`

Var.same

```
meta native int Var.same(Var a, Var b)
```

Reports whether `a` and `b` have identical `Var` bits. Two boxes of one wide number differ here, although === holds for them.

Source: `lib/dispatch.x:411`

Var.str

```
String Var.str(Var v)
```

Returns the display `String` of `Var`.

Source: `lib/dispatch.x:74`

Var.try_dispatch_binary

```
int Var.try_dispatch_binary(Var lhs, Symbol member, Var rhs, Var &?result)
```

Tries one registered binary callback for `lhs`. Only `<add>`, `<sub>`, `<mul>`, `<div>`, `<mod>`, and `<matmul>` select callbacks. Returns one and writes the synchronous callback result when available; otherwise returns zero and leaves `result` unchanged. A null `result` returns zero.

Source: `lib/dispatch.x:704`

Var.try_dispatch_unary

```
int Var.try_dispatch_unary(Var value, Symbol member, Var &?result)
```

Tries the registered `<neg>` callback for `value`. Returns one and writes the synchronous callback result when available; otherwise returns zero and leaves `result` unchanged. A null `result` returns zero.

Source: `lib/dispatch.x:731`

Var.write_pointer_repr

```
Buffer Var.write_pointer_repr(Var v, Buffer out)
```

Writes the fallback readable form of a pointer-bearing `Var`.

Source: `lib/dispatch.x:301`

Var.write_repr

```
Buffer Var.write_repr(Var v, Buffer out)
```

Appends the readable representation of `Var` to a `Buffer`.

Source: `lib/dispatch.x:204`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
x2c_descriptor_registration_frozen	Reports under the descriptor lock whether registration is frozen.
x2c_descriptor_thread_start_begin	Locks descriptor registration while a native worker starts.
x2c_descriptor_thread_start_end	Ends a native worker start and unlocks descriptor registration.
Var.try_export_context	Calls the registered `Context` exporter for `value` when one exists.

Functions

x2c_descriptor_registration_frozen

```
int x2c_descriptor_registration_frozen(void)
```

Reports under the descriptor lock whether registration is frozen.

Source: `lib/dispatch.x:976`

x2c_descriptor_thread_start_begin

```
void x2c_descriptor_thread_start_begin(void)
```

Locks descriptor registration while a native worker starts. The caller must pair this on the same thread with `x2c_descriptor_thread_start_end`.

Source: `lib/dispatch.x:962`

x2c_descriptor_thread_start_end

```
void x2c_descriptor_thread_start_end(int success)
```

Ends a native worker start and unlocks descriptor registration. A nonzero `success` permanently freezes subsequent registration; zero leaves it open for another attempt.

Source: `lib/dispatch.x:970`

`Var`

Var.try_export_context

```
int Var.try_export_context(Var value, Context source, Var &?out)
```

Calls the registered `Context` exporter for `value` when one exists. Returns nonzero when the descriptor registers an exporter, and writes its result to `out`. `Context` handles built-in value families directly.

Source: `lib/dispatch.x:743`

Design notes

A registered type supplies a descriptor of function pointers. Boxed `Var` operations find the descriptor by tag and call its registered functions; an operation without one uses built-in behavior or raises `<no-member>`. Ordinary method lookup remains static.

Equality, identity, and rendering can inspect `void`. Operations that need an ordinary value, including hashing, ordering, truthiness, and iteration, reject it as an invariant violation.

Tests and examples

`make verify` (`unittest/test-var.x`).

`lib/error/error.x`

Handler stack and accumulated errors.

Advanced and interop API

Function	Summary
`x2c_error_catch_capture`	Returns one borrowed capture from a selected catch arm.
`x2c_error_catch_close`	Closes and invalidates a transferring-catch handle.
`x2c_error_catch_detach`	Unregisters the top transferring catch before its arm runs.
`x2c_error_catch_push`	Registers one transferring catch from patterns supplied per call.
`x2c_error_catch_selected`	Returns the selected zero-based catch arm, or −1 before selection or for a null handle.
`x2c_error_catch_site_pending`	Reports whether one catch site still needs its patterns at registration.
`x2c_error_catch_site_push`	Registers one compiler-generated transferring catch through its site.
`x2c_error_raise`	Raises one compiler-generated error from a prepared detail `List`.
`x2c_error_raise_n`	Raises one compiler-generated error from native key-value arguments.
`Error.bound`	Returns the maximum number of errors that may remain accumulated.
`Error.bound_set`	Sets the accumulated-error bound when `bound` is positive.
`Error.count`	Returns the number of errors currently accumulated.
`Error.depth`	Returns the current nested error-dispatch depth.
`Error.handler_depth`	Returns the current thread's number of registered `Error` handlers.
`Error.handler_head`	Returns the current thread's borrowed top `Error`-handler pointer.
`Error.initialize_raw`	Initializes the current thread's `Error` runtime without lifecycle insertion.
`Error.mark`	Captures the current error-stack position.
`Error.note_rendered`	Marks the current nested error dispatch as already rendered by `Logger`.
`Error.policy_adopt`	Adopts a policy capture on the calling thread and releases it.
`Error.policy_capture`	Captures the calling thread's `Error` policy for another thread to adopt.
`Error.policy_get`	Returns the default disposition for `code`.
`Error.policy_release`	Releases a policy capture that no thread adopted.
`Error.policy_set`	Sets the default disposition for `code`.

Function	Summary
`Error.pop`	Closes the observing handler on top of the handler stack.
`Error.push`	Pushes an observing handler and returns its removal handle.
`Error.raise`	Raises one cause with optional structured detail.
`Error.ready`	Reports whether the rich `Error` runtime can currently accept raises.
`Error.restore`	Discards handler and record growth after one cleanup callback.
`Error.restore_landing`	Restores `Error` state after an exception frame lands.
`Error.shutdown_raw`	Releases the current thread's `Error` storage without lifecycle insertion.
`Error.since`	Returns the accumulated errors at and after `mark`, oldest first.
`Error.snapshot`	Copies one admissible error value into the caller's ordinary owners.
`Error.trim`	Trims `Error` state while an exception frame leaves.
`Error.unwind_head`	Returns the handler head retained for the current `Error` transfer.

Functions

x2c_error_catch_capture

```
Var x2c_error_catch_capture(ErrorHandler handle, int index)
```

Returns one borrowed capture from a selected catch arm. An invalid index, a null or unselected handle, or an unbound alternative returns `void`. The value remains valid until the handle is closed; use `Error.snapshot` to keep it longer.

Source: `lib/error/handlers.x:402`

x2c_error_catch_close

```
void x2c_error_catch_close(ErrorHandler handle)
```

Closes and invalidates a transferring-catch handle. The handle releases its plans, captures, and retained error records. A still-registered handle also leaves the handler stack and truncates records above its registration watermark; a detached one leaves the chain of running arms instead. Handles must close in stack order on whichever chain holds them; violating that order reaches the raw error floor. A null handle does nothing.

Source: `lib/error/handlers.x:439`

x2c_error_catch_detach

```
void x2c_error_catch_detach(ErrorHandler handle)
```

Unregisters the top transferring catch before its arm runs. The handler stack then holds only registrations that can still be selected, so the arm may raise outward without matching itself and may close handlers the `try` was nested inside. The handle moves to the thread's chain of running arms, which owns the retained `Error` and captures the arm still reads until `x2c_error_catch_close`; `Error` shutdown walks that chain, so an `exit()` from inside an arm still reclaims them. Repeated detach and a null handle do nothing. Detaching out of stack order reaches the raw error floor.

Source: `lib/error/handlers.x:420`

x2c_error_catch_push

```
ErrorHandler x2c_error_catch_push(void *target, unsigned arm_count, ...)
```

Registers one transferring catch from patterns supplied per call. A hand-written caller that has no static site uses this form; `arm_count` `List` pattern `Vars` follow in source order, and one plan per arm is prepared for this registration alone. Results and failures follow `x2c_error_catch_site_push`.

Source: `lib/error/handlers.x:363`

x2c_error_catch_selected

```
int x2c_error_catch_selected(ErrorHandler handle)
```

Returns the selected zero-based catch arm, or −1 before selection or for a null handle.

Source: `lib/error/handlers.x:394`

x2c_error_catch_site_pending

```
int x2c_error_catch_site_pending(ErrorCatchSite *site)
```

Reports whether one catch site still needs its patterns at registration. A bound static site answers 0, so its caller can skip constructing them.

Source: `lib/error/handlers.x:386`

x2c_error_catch_site_push

```
ErrorHandler x2c_error_catch_site_push( void *target, ErrorCatchSite *site, Var *patterns)
```

Registers one compiler-generated transferring catch through its site. `target` names the `ExceptionFrame` that the caller pushes immediately afterward and `site` is the static site of this `try`, whose `patterns` are read in source order. A pending site reads `patterns`; a bound static site ignores them, so a caller may pass anything once `x2c_error_catch_site_pending` answers 0. The first registration of a pending site promotes its patterns, which are literals in compiler output, to process lifetime and binds them there when it can. A transient registration borrows its patterns, and they and the target frame must outlive the registration. A null target or site, a zero arm count, or an unavailable `Error` runtime reaches the raw error floor.

Raises: `<alloc-fail>` when registration or fence-detail storage cannot be allocated, or `<size-limit>` when an arm's pattern crosses a `Match` lowering fence. A fenced arm can never be selected. The registration is reclaimed and the error reaches the enclosing handler; the caller's own frame is not yet pushed, so it never sees its own failure.

Source: `lib/error/handlers.x:254`

x2c_error_raise

```
void x2c_error_raise(Symbol code, List detail)
```

Raises one compiler-generated error from a prepared detail `List`. The runtime copies admissible detail before synchronous handler dispatch. Resumable causes may return after handling or policy; shared non-returning causes may transfer to a filtered catch but never return here. Invalid detail or unavailable, reentrant, or failed `Error` machinery reaches the raw error floor.

Source: `lib/error/dispatch.x:41`

x2c_error_raise_n

```
void x2c_error_raise_n( const X2CErrorSite *site, Symbol code, unsigned pair_count, ...)
```

Raises one compiler-generated error from native key-value arguments. `pair_count` controls the following alternating `Var` keys and values; `site` may be NULL. The runtime copies admissible values and preserves pair order. Resumable causes may return after handling or policy; shared non-returning causes may transfer to a filtered catch but never return here. Invalid detail or unavailable, reentrant, or failed `Error` machinery reaches the raw error floor.

Source: `lib/error/dispatch.x:53`

Error

Error.bound

```
int Error.bound(void)
```

Returns the maximum number of errors that may remain accumulated.

Source: `lib/error/policy.x:59`

Error.bound_set

```
void Error.bound_set(int bound)
```

Sets the accumulated-error bound when `bound` is positive. A zero or negative value leaves the current bound unchanged.

Source: `lib/error/policy.x:67`

Error.count

```
int Error.count(void)
```

Returns the number of errors currently accumulated. Returns zero before `Error` initialization and after shutdown.

Source: `lib/error/records.x:202`

Error.depth

```
int Error.depth(void)
```

Returns the current nested error-dispatch depth. This is the nesting depth of error dispatch. `Error.count` returns the number of accumulated errors.

Source: `lib/error/dispatch.x:101`

Error.handler_depth

```
int Error.handler_depth(void)
```

Returns the current thread's number of registered `Error` handlers.

Source: `lib/error/handlers.x:219`

Error.handler_head

```
void *Error.handler_head(void)
```

Returns the current thread's borrowed top `Error`-handler pointer. Exception frames use this opaque value as a restore watermark; it remains valid only while its registration remains live.

Source: `lib/error/handlers.x:229`

Error.initialize_raw

```
void Error.initialize_raw(void)
```

Initializes the current thread's `Error` runtime without lifecycle insertion. Repeated calls after successful initialization and calls after shutdown do nothing. Initialization owns a private `Scope`, record stack, and policy `Map`; failure before the `Error` runtime becomes ready reaches the raw error floor.

Source: `lib/error/runtime.x:47`

Error.mark

```
int Error.mark(void)
```

Captures the current error-stack position. Pass the result to `Error.since` to inspect only later errors.

Source: `lib/error/records.x:207`

Error.note_rendered

```
void Error.note_rendered(void)
```

Marks the current nested error dispatch as already rendered by `Logger`. This suppresses only `Error`'s fallback report for a `<log>` policy. Calling outside dispatch has no effect.

Source: `lib/error/dispatch.x:207`

Error.policy_adopt

```
void Error.policy_adopt(void *capture)
```

Adopts a policy capture on the calling thread and releases it. Each captured code takes its captured disposition; codes the capture does not name keep the disposition this thread already has. A NULL capture, or one adopted

while `Error` is unavailable, is released without effect. Failure to update `Error`-owned storage reaches the non-reentrant error floor.

Source: `lib/error/policy.x:133`

Error.policy_capture

```
void *Error.policy_capture(void)
```

Captures the calling thread's `Error` policy for another thread to adopt. Policy is per-thread state, so a worker starts with only the shared non-returning causes locked to `<abort>`; a capture carries the starting thread's dispositions across. `Error.policy_adopt` consumes the capture and `Error.policy_release` discards one that no thread adopted. An unavailable `Error` runtime captures nothing and answers NULL.

Raises: `<alloc-fail>` when the capture cannot be allocated.

Source: `lib/error/policy.x:89`

Error.policy_get

```
Symbol Error.policy_get(Symbol code)
```

Returns the default disposition for `code`. Unknown codes and an unavailable `Error` runtime default to `<abort>`.

Source: `lib/error/policy.x:46`

Error.policy_release

```
void Error.policy_release(void *capture)
```

Releases a policy capture that no thread adopted. A NULL capture is accepted and does nothing.

Source: `lib/error/policy.x:153`

Error.policy_set

```
void Error.policy_set(Symbol code, Symbol disposition)
```

Sets the default disposition for `code`. Supported policy values are `<abort>`, `<collect>`, `<log>`, and `<ignore>`. Another value raises `<bad-arg>` and leaves the previous policy unchanged. Shared non-returning causes accept only `<abort>`; another disposition raises `<bad-arg>` and leaves their policy unchanged. The call is a no-op while `Error` is unavailable; failure to update `Error`-owned storage reaches the non-reentrant error floor.

Source: `lib/error/policy.x:25`

Error.pop

```
void Error.pop(ErrorHandler handle)
```

Closes the observing handler on top of the handler stack. Closing truncates and reclaims every error above the handler's registration watermark, then unregisters it. `Errors` below the watermark remain. Handles must be popped in stack order. An out-of-order pop reaches the non-reentrant error floor; a null handle does nothing.

Source: `lib/error/handlers.x:159`

Error.push

```
ErrorHandler Error.push(ErrorHandlerFn fn, Var data)
```

Pushes an observing handler and returns its removal handle. The handler sees a borrowed view of errors raised after this registration; the view is valid only during the callback. Use `Error.snapshot` for any value that must escape. Returning `<handled>` consumes that slice, `<declined>` leaves it for outer handlers, and `<fatal>` reaches the error floor. For a shared non-returning cause, `<handled>` also reaches the floor. A null callback or unavailable runtime returns NULL without registering. Raises `<alloc-fail>` if registration storage cannot be allocated. `data` is retained by value without copying its referent, so any referenced storage must outlive the registration.

Source: `lib/error/handlers.x:137`

Error.raise

```
Symbol Error.raise(Symbol code, List detail)
```

Raises one cause with optional structured detail. This functional entry records and dispatches like the `raise` statement but has no source-location record. A handled resumable error returns `<handled>`; a collected or policy-consumed resumable error returns `<declined>`. Shared non-returning causes may transfer to a matching filtered catch but never return from this call. Detail is recursively restricted to null/`nil`, numeric values, enums, `Symbols`/`Atoms`, `Strings`, and `Lists` of those values. An invalid dynamic detail, unhandled abort-policy error, unavailable runtime, or reentrant failure reaches the non-reentrant error floor.

Prefer the `raise` statement in source so generated location detail is retained.

Source: `lib/error/dispatch.x:32`

Error.ready

```
int Error.ready(void)
```

Reports whether the rich `Error` runtime can currently accept raises. This is per-thread state and is false before initialization and after shutdown.

Source: `lib/error/runtime.x:115`

Error.restore

```
void Error.restore(int handler_depth, int stack_height)
```

Discards handler and record growth after one cleanup callback. Registrations and records created by that callback are reclaimed toward the saved heights before the interrupted unwind continues. State the callback itself removed is not reconstructed.

Source: `lib/error/handlers.x:503`

Error.restore_landing

```
void Error.restore_landing(void *saved_head, int saved_depth)
```

Restores `Error` state after an exception frame lands. `saved_head` is a prior handler watermark and `saved_depth` is the dispatch depth captured when the frame was pushed. The saved head is restored only when the current handler head is still a suffix of its chain; dispatch bookkeeping is cleared.

Source: `lib/error/handlers.x:478`

Error.shutdown_raw

```
void Error.shutdown_raw(void)
```

Releases the current thread's `Error` storage without lifecycle insertion. The call reclaims all registrations, records, policies, and private storage and makes `Error.ready` false. Repeated calls do nothing; later raises reach the raw error floor.

Source: `lib/error/runtime.x:68`

Error.since

```
List Error.since(int mark)
```

Returns the accumulated errors at and after `mark`, oldest first. Each entry has `code`, `detail`, and `location` fields; a location inside the runtime library also holds `(runtime 1)`. An invalid mark or an unavailable `Error` runtime returns `nil`. The snapshot enters the caller's outermost `Scope` and canonical pools and remains live until those owners are released. Failure to materialize it reaches the non-reentrant error floor.

Source: `lib/error/records.x:217`

Error.snapshot

```
Var Error.snapshot(Var value)
```

Copies one admissible error value into the caller's ordinary owners. Handler slices and filtered-catch bindings are borrowed. Snapshot a value that must outlive its callback or selected arm; `Strings` and `Lists` enter the caller's outermost canonical pools and wide scalar boxes enter its outermost `Scope`, so nested caller brackets may be released safely. Invalid or identity-bearing values and failures while copying reach the raw error floor; they never re-enter handler dispatch.

Source: `lib/error/records.x:271`

Error.trim

```
void Error.trim(void *saved_head, int stack_height)
```

Trims `Error` state while an exception frame leaves. Handlers newer than `saved_head` are reclaimed. Records at and above `stack_height` are also reclaimed; pass the current height on normal frame exit to preserve collected errors.

Source: `lib/error/handlers.x:491`

Error.unwind_head

```
void *Error.unwind_head(void)
```

Returns the handler head retained for the current `Error` transfer. During handler dispatch this is the head saved by the outermost dispatch, even though the running callbacks are temporarily hidden from nested raises. Exception frames store the opaque result as their landing watermark.

Source: `lib/error/handlers.x:467`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`Error.context_close`	Closes one `Context Error` overlay.
`Error.context_open`	Opens the `Error` state owned by one `Context`.
`Error.since_in`	Copies errors at and after `mark` into explicit owners, oldest first.
`Error.snapshot_in`	Copies one admissible error value into explicit runtime owners.

Error

Error.context_close

```
void Error.context_close(void *token, int preserve_records)
```

Closes one `Context Error` overlay. An exception unwinding out of the `Context` keeps its records for the outer handler; an ordinary close discards records accumulated inside it. In both cases handlers pushed inside the `Context` are reclaimed. Tokens must close in nesting order; an out-of-order close reaches the raw error floor. A null token does nothing and a closed token is invalid.

Source: `lib/error/policy.x:180`

Error.context_open

```
void *Error.context_open(void)
```

Opens the `Error` state owned by one `Context`. Policy and bound changes become local overlays; handlers and accumulated records are restored by `Error.context_close`. The returned opaque token is allocated in the current `Scope`, which must remain live through the matching close. An unavailable `Error` runtime returns NULL.

Raises: `<alloc-fail>` when the overlay cannot be allocated.

Source: `lib/error/policy.x:162`

Error.since_in

```
List Error.since_in(int mark, Scope *values, Pool pool)
```

Copies errors at and after `mark` into explicit owners, oldest first. `Strings` and `Lists` are canonicalized through `pool`'s chain and remain live until their actual owning pool is released. Wide scalar boxes enter `*values`, whose possibly updated `Scope` head is written back, and remain live until that `Scope` is destroyed. An unavailable runtime or negative mark returns `nil`. `Null` owners or failures while copying reach the raw floor.

Source: `lib/error/records.x:246`

Error.snapshot_in

```
Var Error.snapshot_in(Var value, Scope *values, Pool pool)
```

Copies one admissible error value into explicit runtime owners. `Strings` and `Lists` are canonicalized through `pool`'s chain and remain live until their actual owning pool is released. Wide scalar boxes enter `*values`, whose possibly updated `Scope` head is written back, and remain live until that `Scope` is destroyed. `Null` owners, invalid or identity-bearing values, and failures while copying reach the raw floor.

Source: `lib/error/records.x:286`

Public types

Type	Kind	Summary
`Error`	struct	Names the structured-error runtime and its static operations.
`ErrorCatchSite`	struct	Holds the process-lifetime plans of one compiler-generated filtered catch.
`ErrorHandler`	struct	Identifies one `Error`-owned handler registration.
`ErrorHandlerFn`	callback	Handles a borrowed oldest-first `List` of errors raised since registration.

Error

```
typedef struct Error *Error
```

Names the structured-error runtime and its static operations. Programs do not construct `Error` values; automatic runtime initialization owns the per-thread handler, record, policy, and dispatch state.

Source: `lib/error/error.x:26`

ErrorCatchSite

```
typedef struct ErrorCatchSite { MatchCaptureSite *arms; int default_arm, arm_count, state, fenced_arm; } ErrorCatchSite
```

Holds the process-lifetime plans of one compiler-generated filtered catch. `arms` is a zero-initialized static array of `arm_count` `Match` sites and `default_arm` is the first unpatterned arm, or −1. `state` and `fenced_arm` belong to `Error`; a site must be static storage that the first registration binds to its patterns.

Source: `lib/error/error.x:50`

ErrorHandler

```
typedef struct ErrorHandler *ErrorHandler
```

Identifies one `Error`-owned handler registration. An observing handle remains valid until `Error.pop`; a transferring-catch handle remains valid until `x2c_error_catch_close`. Enclosing frame or `Context` cleanup and `Error` shutdown may reclaim a registration first.

Source: `lib/error/error.x:33`

ErrorHandlerFn

```
typedef Symbol (*ErrorHandlerFn)(List errors, Var data)
```

Handles a borrowed oldest-first `List` of errors raised since registration. `Error` invokes callbacks synchronously from innermost registration outward. Return `<handled>`, `<declined>`, or `<fatal>`; `<unwind>` is reserved for compiler-generated catches. Any other result behaves as `<declined>` and continues outward or to policy. The `List` and its contents expire on return.

Source: `lib/error/error.x:42`

Design notes

Raising an error records it and calls registered handlers, innermost first. Each handler sees the errors raised since it was registered and decides how to respond. The caller sets the policy for errors no handler accepts.

Each accumulated record owns an independent `Scope` and one canonical `List` and `String` pool. A handler watermark bounds those regions, so closing the handler reclaims its complete slice without touching application pools. Raising while the error path is itself failing reaches the error floor, which allocates nothing.

lib/exception.x

Transfer frames for x2c `Error` unwinding and cleanup.

Advanced and interop API

Function	Summary
x2c_cleanup_leave	Removes and runs the current compiler-generated cleanup record.
x2c_cleanup_push	Pushes one compiler-generated cleanup record.
x2c_exception_claim	Claims a frame's cleanup for the calling exit path, once per frame.
x2c_exception_is_error_target	Reports whether `frame` is carrying an `Error` transfer targeted to itself.
x2c_exception_landed	Restores `Error` handler and dispatch state after a frame landing.
x2c_exception_leave	Removes an active exception frame and continues any pending `Error` transfer.
x2c_exception_mark_handled	Marks a selected exception-frame `Error` transfer as handled.
x2c_exception_push	Initializes and pushes one compiler-generated exception frame.
ExceptionFrame.unwind	Transfers an `Error` toward the selected active exception frame.

Functions

x2c_cleanup_leave

```
void x2c_cleanup_leave(X2CCleanup *record)
```

Removes and runs the current compiler-generated cleanup record. Records must leave in last-in, first-out order. The record is unlinked before its callback runs, so a callback that transfers cannot run it again. A null or out-of-order record exits through the raw exception fatal path.

Source: `lib/exception.x:83`

x2c_cleanup_push

```
void x2c_cleanup_push(X2CCleanup *record)
```

Pushes one compiler-generated cleanup record. `record` and its `fn` must be nonnull, and the record and borrowed `env` must remain live until leave or `Error` transfer. Invalid registration exits through the raw exception fatal path.

Source: `lib/exception.x:71`

x2c_exception_claim

```
int x2c_exception_claim(ExceptionFrame *frame)
```

Claims a frame's cleanup for the calling exit path, once per frame. Compiler-generated code tests this on every path that reaches a finalizer and runs the finalizer only when it reports the claim. Claiming also retires the frame's landing: the frame has already landed, so a `raise` from the finalizer transfers to the enclosing frame instead of re-entering this landing and running the finalizer again; that transfer abandons this frame and replaces any `Error` it was already carrying. A null or already claimed frame reports zero.

Source: `lib/exception.x:180`

x2c_exception_is_error_target

```
int x2c_exception_is_error_target(ExceptionFrame *frame)
```

Reports whether `frame` is carrying an `Error` transfer targeted to itself. A null, inactive, handled, or intervening frame returns false.

Source: `lib/exception.x:168`

x2c_exception_landed

```
void x2c_exception_landed(ExceptionFrame *frame)
```

Restores `Error` handler and dispatch state after a frame landing. Compiler-generated code calls this only on the nonzero `sigsetjmp` path. A null frame does nothing.

Source: `lib/exception.x:150`

x2c_exception_leave

```
void x2c_exception_leave(ExceptionFrame *frame)
```

Removes an active exception frame and continues any pending `Error` transfer. The frame must be left in nesting order after its cleanup records have been removed. Normal leave preserves accumulated errors but reclaims handlers registered inside the frame. An intervening unwind instead restores the frame's error-stack watermark and transfers to the next outer frame. A null or already left frame does nothing; cleanup imbalance exits through the raw fatal path.

Source: `lib/exception.x:203`

x2c_exception_mark_handled

```
void x2c_exception_mark_handled(ExceptionFrame *frame)
```

Marks a selected exception-frame `Error` transfer as handled. This prevents `x2c_exception_leave` from continuing the transfer outward. A null frame does nothing.

Source: `lib/exception.x:190`

x2c_exception_push

```
void x2c_exception_push(ExceptionFrame *e)
```

Initializes and pushes one compiler-generated exception frame. The frame records the current cleanup, handler, dispatch-depth, and error stack watermarks and must remain live until `x2c_exception_leave`. A null frame does nothing.

Source: `lib/exception.x:96`

`ExceptionFrame`

ExceptionFrame.unwind

```
void ExceptionFrame.unwind(void *target_ptr)
```

Transfers an `Error` toward the selected active exception frame. `target_ptr` must identify a landable frame in the current thread. The call selects the innermost frame that is not already running its own cleanup, abandons any frame it skips, marks the selected frame, drains newer cleanup records in last-in, first-out order, records the post-cleanup handler head, and jumps to that frame's landing. It never returns normally. An invalid target exits through the raw exception fatal path. The transfer does not restore the process signal mask.

Source: `lib/exception.x:128`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`x2c_exception_unwinding`	Reports whether any active frame is carrying an `Error` transfer.

Functions

x2c_exception_unwinding

```
int x2c_exception_unwinding(void)
```

Reports whether any active frame is carrying an `Error` transfer.

Source: `lib/exception.x:158`

Public types

Type	Kind	Summary
`ExceptionFrame`	struct	Holds one caller-owned non-local `Error` transfer frame.
`X2CCleanup`	struct	Holds one caller-owned cleanup registration.
`X2CCleanupFn`	callback	Performs one compiler-generated cleanup using its borrowed environment.

ExceptionFrame

```
typedef struct ExceptionFrame { struct ExceptionFrame *prev; X2CCleanup *cleanup_watermark; sigjmp_buf env, volatile Symbol state; struct ExceptionFrame *volatile unwind_target, void *error_handler_head; void *volatile error_landing_head, int error_dispatch_depth; int error_stack_height, volatile int cleanup_active; } ExceptionFrame
```

Holds one caller-owned non-local `Error` transfer frame. Compiler-generated code keeps the frame on the C stack, pushes it before establishing its `sigsetjmp` landing, and leaves it in nesting order. Its fields capture cleanup and `Error` restore points and are runtime-managed while the frame is active.

Source: `lib/exception.x:39`

X2CCleanup

```
typedef struct X2CCleanup { struct X2CCleanup *prev, X2CCleanupFn fn, void *env; } X2CCleanup
```

Holds one caller-owned cleanup registration. The record and its callback environment storage must remain live from `x2c_cleanup_push` through `x2c_cleanup_leave` or `Error` transfer. The runtime owns the record's links while registered. Cleanup records nest and run in last-in, first-out order.

Source: `lib/exception.x:29`

X2CCleanupFn

```
typedef void (*X2CCleanupFn)(void *env)
```

Performs one compiler-generated cleanup using its borrowed environment. Exception invokes the callback once after unlinking its record, on normal leave or `Error` transfer. The environment must remain live until then.

Source: `lib/exception.x:21`

Design notes

The emitter allocates `ExceptionFrame` objects on the C stack for filtered catches and finally clauses. Callable cleanup records cover `defer`-only regions without adding a setjmp landing. `Error` unwind drains those records before landing each intervening exception frame.

`lib/file.x`

`File` I/O operations and stream handling.

Primary API

Function	Summary
`File.cleanup`	Ends the owned lifetime when a managed local leaves its block.
`File.copy_to`	Copies the remaining bytes from `source` to `output`.
`File.getw`	Reads and returns one native `int`, or EOF when a full word is unavailable.
`File.iter`	Returns an iterator over `File`.
`File.open`	Opens `path` with the requested stdio mode.
`File.path_error`	Raises `<not-found>` when `error` is `ENOENT` and `<io-fail>` otherwise, with `operation`, `path`, and `errno` details.
`File.read_into`	Reads the remaining stream bytes into caller-owned storage.
`File.readline_into`	Reads one raw line into caller-owned byte storage.
`File.write_all`	Writes every requested byte unless the stream reports failure.
`String.open`	Opens the filesystem path named by `s`.

`File`

File.cleanup

```
void File.cleanup(File value)
```

Ends the owned lifetime when a managed local leaves its block.

Source: `lib/file.x:67`

File.copy_to

```
int File.copy_to(File source, File output, size_t &?copied)
```

Copies the remaining bytes from `source` to `output`. A null stream returns zero without raising; every other outcome returns nonzero or transfers. When `copied` is nonnull it receives the number of bytes written, including the partial count a catch observes after a failure.

Raises: `<io-fail>` on a source read or destination write failure.

Source: `lib/file.x:313`

File.getw

```
inline int File.getw(File file)
```

Reads and returns one native `int`, or EOF when a full word is unavailable. A stored value equal to EOF is indistinguishable from the sentinel without inspecting the stream indicators.

Source: `lib/file.x:406`

File.iter

```
Iter File.iter(File file, Iter dest)
```

Returns an iterator over `File`. The caller supplies `dest`; each pull yields one `String` under the canonical pool-chain lifetime described above, including its newline when present. The iterator borrows `file`, which must remain open through every pull. Clean EOF exhausts the iterator and releases its line storage. Abandoning it before exhaustion leaves that `Scope`-owned `Block` until its `Scope` is cleaned up. A null `dest` returns NULL; a null `file` initializes `dest` as exhausted.

Raises: `<alloc-fail>` while initializing. Pulling may raise the same causes as `File.readline`; none return to the pull, and a transfer releases the iterator's line storage.

Source: `lib/file.x:354`

File.open

```
File File.open(const char *path, const char *mode)
```

Opens `path` with the requested stdio mode. The caller owns a successful stream and must close it. A missing path raises `<not-found>`; another host failure raises `<io-fail>`; and a null path or mode raises `<bad-arg>`. Host failures carry the path, operation, and captured errno.

Source: `lib/file.x:528`

File.path_error

```
void File.path_error(Var operation, String path, int error)
```

Raises `<not-found>` when `error` is `ENOENT` and `<io-fail>` otherwise, with `operation`, `path`, and `errno` details.

Source: `lib/file.x:496`

File.read_into

```
FileReadStatus File.read_into(File f, Block dest)
```

Reads the remaining stream bytes into caller-owned storage. Valid inputs clear `dest` first. Returns `<data>` when bytes were read or `<eof>` at clean EOF. `Null` inputs or a `Block` width other than one report `<error>` without raising and leave a nonnull `dest` unchanged; a read failure transfers instead, leaving its partial bytes in `dest`.

Raises: `<io-fail>` on a stream read error, or the cause reported by `Block.append` when the destination cannot grow.

Source: `lib/file.x:273`

File.readline_into

```
FileReadStatus File.readline_into(File f, Block dest)
```

Reads one raw line into caller-owned byte storage. Valid inputs clear `dest`, then include the newline when one is read. Returns `<data>` when bytes were read or `<eof>` at clean EOF. `Null` inputs or a `Block` width other than one report `<error>` without raising and leave a nonnull `dest` unchanged; a read failure transfers instead, leaving its partial bytes in `dest`.

Raises: `<io-fail>` on a stream read error, or `<size-limit>` or `<alloc-fail>` when the destination cannot grow.

Source: `lib/file.x:243`

File.write_all

```
int File.write_all(File file, const void *ptr, size_t size)
```

Writes every requested byte unless the stream reports failure. A null stream or a null pointer with nonzero size returns zero without raising; every other outcome returns nonzero or transfers. A transfer may leave a prefix already written.

Raises: `<io-fail>` on a short or failed write, which does not return here.

Source: `lib/file.x:300`

String

String.open

```
File String.open(String s, const char *mode)
```

Opens the filesystem path named by `s`. The caller owns a successful stream and must close it. A missing path raises `<not-found>`; another host failure raises `<io-fail>`; and a null path or mode raises `<bad-arg>`. Host failures carry the path, operation, and captured errno.

Source: `lib/file.x:520`

Advanced and interop API

Function	Summary
`File.equal`	Reports whether `x` and `y` are the same native stream handle.
`File.fdopen`	Wraps an open file descriptor in a `File` stream.
`File.gets`	Reads a native line into `str` and returns `str`, or NULL at EOF or error.
`File.hash`	Returns a handle-identity hash consistent with `File.equal`.
`File.popen`	Opens a process pipe with the requested mode.
`File.printf`	Formats values into `file`.
`File.putc`	Writes one byte and returns it as an unsigned char, or EOF on failure.
`File.puts`	Writes a NUL-terminated C string.
`File.putw`	Writes one native `int` and returns `w`, or EOF on a short write.
`File.read`	Reads up to `nitems` elements and returns the number read.
`File.reopen`	Reuses `file` for a newly opened path and mode.
`File.repr`	Returns a readable handle and descriptor representation without reading.
`File.scanf`	Scans values from `file`, returning the assignment count or EOF.
`File.setbuffer`	Installs caller-supplied buffering, or disables buffering for a null `buf`.
`File.setlinebuf`	Requests line buffering for `file` and returns zero.
`File.stat`	Writes metadata for `file` into `buf`.
`File.str`	Returns the display `String` for `file`.
`File.string`	Reads the remaining stream into one canonical `String`.
`File.string_close`	Reads the remaining text, then closes `file` on return or transfer.
`File.ungetc`	Pushes one byte back and returns it, or EOF when it cannot be pushed.
`File.write`	Writes up to `nitems` elements and returns the number written.
`File.write_repr`	Appends the readable pointer representation of `file` to `out`.

`File`

File.equal

```
int File.equal(File x, File y)
```

Reports whether `x` and `y` are the same native stream handle.

Source: `lib/file.x:469`

File.fdopen

```
File File.fdopen(int fildes, const char *mode)
```

Wraps an open file descriptor in a `File` stream. On success the returned stream owns `fildes`, which must be closed through the stream. On failure the caller still owns the descriptor. A host failure raises `<io-fail>` with the operation and captured errno; a null mode raises `<bad-arg>`.

Source: `lib/file.x:537`

File.gets

```
inline char *File.gets(File f, char *str, int size)
```

Reads a native line into `str` and returns `str`, or NULL at EOF or error. At most `size - 1` bytes are stored followed by NUL, and a newline is kept when it fits.

Source: `lib/file.x:388`

File.hash

```
unsigned File.hash(File file)
```

Returns a handle-identity hash consistent with `File.equal`.

Source: `lib/file.x:466`

File.popen

```
File File.popen(const char *cmd, const char *mode)
```

Opens a process pipe with the requested mode. The caller owns a successful stream and must finish it with `File.pclose` to close the pipe and collect the child status. A host failure raises `<not-found>` for `ENOENT` or `<io-fail>` otherwise, with the command as `path`, operation, and captured errno; a null command or mode raises `<bad-arg>`.

Source: `lib/file.x:552`

File.printf

```
int File.printf(File file, const char *format, ...)
```

Formats values into `file`. Returns the character count or a negative value on failure.

Source: `lib/file.x:446`

File.putc

```
inline int File.putc(File file, int c)
```

Writes one byte and returns it as an unsigned char, or EOF on failure.

Source: `lib/file.x:391`

File.puts

```
inline int File.puts(File file, const char *s)
```

Writes a NUL-terminated C string. Returns a nonnegative value on success or EOF on failure.

Source: `lib/file.x:396`

File.putw

```
inline int File.putw(File file, int w)
```

Writes one native `int` and returns `w`, or EOF on a short write.

Source: `lib/file.x:399`

File.read

```
inline size_t File.read(File file, void *ptr, size_t size, size_t nitems)
```

Reads up to `nitems` elements and returns the number read.

Source: `lib/file.x:423`

File.reopen

```
Self File.reopen(Self file, const char *path, const char *mode)
```

Reuses `file` for a newly opened path and mode. This consumes the original stream even when the native reopen fails; after a transfer the caller must not close or reuse the old handle. A missing path raises `<not-found>`; another host failure raises `<io-fail>`; and a null stream, path, or mode raises `<bad-arg>`. Host failures carry the path, operation, and captured errno.

Source: `lib/file.x:567`

File.repr

```
String File.repr(File file)
```

Returns a readable handle and descriptor representation without reading.

Source: `lib/file.x:472`

File.scanf

```
int File.scanf(File file, const char *format, ...)
```

Scans values from `file`, returning the assignment count or EOF.

Source: `lib/file.x:455`

File.setbuffer

```
inline void File.setbuffer(File file, char *buf, int size)
```

Installs caller-supplied buffering, or disables buffering for a null `buf`. A nonnull buffer is borrowed until the stream closes or buffering changes.

Source: `lib/file.x:434`

File.setlinebuf

```
inline int File.setlinebuf(File file)
```

Requests line buffering for `file` and returns zero. The underlying `setvbuf` result is intentionally not exposed.

Source: `lib/file.x:414`

File.stat

```
inline int File.stat(File file, struct stat *buf)
```

Writes metadata for `file` into `buf`. Returns zero on success or −1 on a native error.

Source: `lib/file.x:441`

File.str

```
String File.str(File file)
```

Returns the display `String` for `file`. A nonnull file follows `File.string`. It advances through EOF, returns a canonical result under the pool-chain lifetime described above or NULL when empty, and transfers the same causes. A null handle returns its pointer representation without reading.

Source: `lib/file.x:483`

File.string

```
String File.string(File file)
```

Reads the remaining stream into one canonical `String`. Starts at the current position, advances through EOF, and returns NULL when no bytes remain.

Raises: `<io-fail>` on a stream read error, `<bad-arg>` when returned bytes contain an embedded NUL, `<size-limit>` when the `String` cannot be represented, or `<alloc-fail>` while constructing the result.

Source: `lib/file.x:93`

File.string_close

```
String File.string_close(File file)
```

Reads the remaining text, then closes `file` on return or transfer. The close result is discarded, so a close failure is not reported. The stream and all of its aliases are invalid afterward.

Raises: the same causes as `File.string`.

Source: `lib/file.x:147`

File.ungetc

```
inline int File.ungetc(File file, int c)
```

Pushes one byte back and returns it, or EOF when it cannot be pushed.

Source: `lib/file.x:420`

File.write

```
inline size_t File.write( File file, const void *ptr, size_t size, size_t nitems)
```

Writes up to `nitems` elements and returns the number written.

Source: `lib/file.x:427`

File.write_repr

```
Buffer File.write_repr(File file, Buffer out)
```

Appends the readable pointer representation of `file` to `out`.

Source: `lib/file.x:486`

Convenience API

Function	Summary
`File.readblock`	Reads up to `size` bytes into a canonical `String`.
`File.readline`	Reads one raw line into a canonical `String`.

`File`

File.readblock

```
String File.readblock(File file, long size)
```

Reads up to `size` bytes into a canonical `String`. Reading starts at the current stream position and returns NULL when no bytes are read. Prefer `File.read_into` for raw bytes and explicit status.

Raises: `<bad-arg>` for a negative `size`, `<size-limit>` when the requested `String` cannot be represented, `<bad-arg>` when the bytes contain an embedded NUL, `<io-fail>` on a stream read error, or `<alloc-fail>` while constructing the result.

Source: `lib/file.x:130`

File.readline

```
String File.readline(File file)
```

Reads one raw line into a canonical `String`. Includes the newline when present. Clean EOF and `<error>` both map to NULL. Prefer `File.readline_into` or `File.iter` when raw status and bytes should remain separate.

Raises: `<io-fail>` on a stream read error, `<bad-arg>` when returned bytes are not valid `String` text, `<size-limit>` when the line cannot be represented, or `<alloc-fail>` while constructing the result.

Source: `lib/file.x:115`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`File.initialize`	Publishes the process's borrowed standard streams as `File` globals.

`File`

File.initialize

```
void File.initialize(void)
```

Publishes the process's borrowed standard streams as `File` globals. The globals do not take ownership or arrange cleanup of the native streams.

Source: `lib/file.x:586`

Public types

Type	Kind	Summary
`File`	alias	Names a native stdio stream handle.
`FileReadStatus`	alias	Reports `<data>` when a raw `File` read produced bytes, `<eof>` at clean EOF, or `<error>` for invalid arguments.

File

```
typedef FILE *File
```

Names a native stdio stream handle. A successful open returns an owned stream. `Stdin`, `Stdout`, and `Stderr` are borrowed. An alias does not retain a stream, and closing one invalidates every alias.

Source: `lib/file.x:30`

FileReadStatus

```
typedef Symbol FileReadStatus
```

Reports `<data>` when a raw `File` read produced bytes, `<eof>` at clean EOF, or `<error>` for invalid arguments. Host read failures transfer `<io-fail>` instead of returning `<error>`.

Source: `lib/file.x:65`

Design notes

A `File` is a native stdio stream whose handle identity serves hashing, equality, and ordering. Raw reads and copies report status separately from the bytes they move. Other results, stream position, buffering, and error indicators follow stdio unless an operation documents otherwise. The standard streams are borrowed from the process.

`String` adapters canonicalize through the active pool chain. A returned identity may already belong to an ancestor and lives until its actual owning pool is released; NULL is the empty `String`.

Tests and examples

`make verify` (`unittest/test-file.x`) and `make examples` (`docs-word-count`).

`lib/func.x`

Generic native function binding.

Advanced and interop API

Function	Summary
x2c_func_pointer_argument	Returns the address value argument i carries.
x2c_func_record_result	Boxes a record result as a <p48> to a copy of its size bytes in the active Scope.
x2c_func_reference_argument	Checks a reference argument whose stored pointee matches want exactly.
x2c_func_value_argument	Checks and converts value argument i, reporting its position.
Func.apply	Invokes a bound native function synchronously and returns its boxed result.
Func.apply_value	Applies fn to one argument passed by value.
Func.apply_values	Applies fn to two arguments passed by value, left then right.
Func.context	Returns borrowed read-only access to a Func's copied context.
Func.new	Builds a fixed-arity native-function binding in the current Scope.
Func.new_context	Builds a fixed-arity binding with copied context in the current Scope.
Func.new_rest	Builds a native-function binding that takes any number of value arguments.
Func.var	Boxes function without copying or retaining the Func.
FuncArg.reference	Constructs a Func argument borrowing a typed lvalue address.
FuncArg.value	Constructs a Func argument by copying one Var value.
Var.func	Returns the borrowed native callable carried by v.

Functions

x2c_func_pointer_argument

```
void *x2c_func_pointer_argument(Func fn, const FuncArg *argv, unsigned i)
```

Returns the address value argument i carries. Generated adapters use it for a pointer parameter with no Var tag of its own, such as const char * or struct timespec *, and for a record passed by value, whose bytes the adapter copies. Any pointer, reference, or String argument is accepted, as C converts it to the parameter's pointer type.

Raises: <bad-types> for any other argument.

Source: lib/func.x:249

x2c_func_record_result

```
Var x2c_func_record_result(const void *bytes, size_t size)
```

Boxes a record result as a <p48> to a copy of its size bytes in the active Scope.

Source: lib/func.x:271

x2c_func_reference_argument

```
void *x2c_func_reference_argument( Func fn, const FuncArg *argv, unsigned i, List want)
```

Checks a reference argument whose stored pointee matches want exactly.

Raises: <bad-types> on the same mismatches as the shared checker.

Source: lib/func.x:278

x2c_func_value_argument

```
Var x2c_func_value_argument( Func fn, const FuncArg *argv, unsigned i, Symbol want)
```

Checks and converts value argument i, reporting its position. Generated adapters call this once per value parameter before unboxing, so a reference carrier or wrong tag is refused before reaching native code. argv must address the prepared argument array and i must be in bounds; compiler-generated adapters establish both facts.

Raises: <void-op> for a void argument, <bad-types> when an object or Symbol argument does not carry want or the carrier holds a reference, and <alloc-fail>, <bad-enc>, <bad-target>, <conv-range>, or <no-convert> from a numeric conversion. The result has tag want. A detail names the argument's tag, since any tag may arrive

here and an identity-bearing detail value terminates at the error floor before it reaches the handler that would report it.

Source: `lib/func.x:215`

`Func`

Func.apply

```
Var Func.apply(Func f, unsigned argc, const FuncArg *argv)
```

Invokes a bound native function synchronously and returns its boxed result. `argv` is borrowed only for the call. Value bits are passed by value; reference arguments alias their caller-owned targets and may be mutated. A rest binding first interns one `List` containing the value arguments in their original order; rest arguments must not be `void`, since `Lists` exclude `void` from their element domain.

Raises: `<bad-arg>` for a null binding or argument storage, `<bad-arity>` for the wrong argument count, `<void-op>`, `<bad-types>`, `<bad-enc>`, `<bad-target>`, `<no-convert>`, or `<conv-range>` while converting an argument, `<alloc-fail>` or `<size-limit>` while packing rest arguments, or any cause raised by the adapter or native target. A `void` adapter result remains no-value. Other results have the ownership of the value the adapter returned.

Source: `lib/func.x:106`

Func.apply_value

```
inline Var Func.apply_value(Func fn, Var value)
```

Applies `fn` to one argument passed by value. `value` travels as `FuncArg.value` carries it, so a callback with a reference parameter raises `<bad-types>`. Returns, owns, and raises as `Func.apply` does.

Source: `lib/func.x:149`

Func.apply_values

```
inline Var Func.apply_values(Func fn, Var left, Var right)
```

Applies `fn` to two arguments passed by value, `left` then `right`. Each travels as `Func.apply_value` carries its argument. Returns, owns, and raises as `Func.apply` does.

Source: `lib/func.x:158`

Func.context

```
const void *Func.context(Func function)
```

Returns borrowed read-only access to a `Func`'s copied context. The pointer remains valid only for the `Func`'s `Scope` lifetime and is NULL when the binding has no context.

Raises: `<bad-arg>` for a null binding. It does not return on failure.

Source: `lib/func.x:443`

Func.new

```
Func Func.new(FuncAdapter adapter, List signature)
```

Builds a fixed-arity native-function binding in the current `Scope`. A direct function expression is compiler-adapted; an expression already typed `FuncAdapter` is stored as supplied. The canonical `signature` and adapter code are borrowed for the `Func` lifetime. Only the signature's outer shape is checked here, so a hand-written adapter must agree with its parameter count and types.

Raises: `<bad-sig>` for a null adapter or a malformed signature, and `<alloc-fail>` when binding storage cannot be allocated.

Source: `lib/func.x:351`

Func.new_context

```
Func Func.new_context( FuncAdapter adapter, List signature, const void *context, size_t context_size)
```

Builds a fixed-arity binding with copied context in the current `Scope`. The max_align_t-aligned bytes share the `Func` allocation. The byte copy is shallow: changing the source bytes does not change the binding, while any pointers inside them keep their original targets and lifetimes. A zero size may use NULL storage. The canonical `signature` and adapter code are borrowed for the `Func` lifetime.

Raises: `<bad-arg>` when a nonzero size has no source, `<size-limit>` when the allocation size overflows, `<bad-sig>` for a null adapter or malformed signature, or `<alloc-fail>` when storage cannot be allocated. None return.

Source: `lib/func.x:378`

Func.new_rest

```
Func Func.new_rest(FuncAdapter adapter, List signature)
```

Builds a native-function binding that takes any number of value arguments. The signature declares one `List` parameter, and the adapter receives every argument in order through that `List`. `Func.apply` enforces no arity and conses the arguments itself. Use this for a variadic operation; a fixed one belongs in `Func.new`, which is faster and reports a wrong count. The canonical `signature` and adapter code are borrowed for the `Func` lifetime. The result belongs to the current `Scope`.

Raises: `<bad-sig>` for a null adapter, a malformed signature, or a signature whose parameters are anything but one `List`, and `<alloc-fail>` when binding storage cannot be allocated.

Source: `lib/func.x:365`

Func.var

```
Var Func.var(Func function)
```

Boxes `function` without copying or retaining the `Func`. The returned `Var` carries the same pointer and shares its `Scope` lifetime.

Source: `lib/func.x:462`

FuncArg

FuncArg.reference

```
inline FuncArg FuncArg.reference(const void *reference, List type)
```

Constructs a `Func` argument borrowing a typed lvalue address. The address and canonical `type` must remain valid through `Func.apply`; an optional reference may use a null address. This constructor performs no validation. Compiler-generated adapters use the checked reference reader before calling native code.

Source: `lib/func.x:141`

FuncArg.value

```
inline FuncArg FuncArg.value(Var value)
```

Constructs a `Func` argument by copying one `Var` value. Pointer-bearing payload storage is not copied or retained and must outlive the call that consumes the argument.

Source: `lib/func.x:133`

Var

Var.func

```
Func Var.func(Var v)
```

Returns the borrowed native callable carried by `v`.

Raises: `<bad-types>` when the value is not a `Func`.

Source: `lib/func.x:466`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`x2c_func_declared_reference_argument`	Checks a generated adapter's declared pointee against the stored signature and its resolved `want` against the source address before casting.
`x2c_func_reference_type`	Returns the borrowed pointee type for reference parameter `index`.
`x2c_func_shared`	Builds the shared handle for a direct function or noncapturing lambda.
`x2c_func_unrepresentable_argument`	Rejects a value argument whose source type has no `Var` representation.
`Func.move`	Transfers `function`'s storage to the `Scope` held by `slot`.
`Func.signature`	Returns a native binding's borrowed canonical signature.

Functions

x2c_func_declared_reference_argument

```
void *x2c_func_declared_reference_argument( Func fn, const FuncArg *argv, unsigned i, List declared_target, List want)
```

Checks a generated adapter's declared pointee against the stored signature and its resolved `want` against the source address before casting.

Raises: `<bad-types>` on either mismatch.

Source: `lib/func.x:285`

x2c_func_reference_type

```
List x2c_func_reference_type(Func fn, unsigned argc, unsigned index)
```

Returns the borrowed pointee type for reference parameter `index`. A value or rest parameter returns NULL. Generated direct calls query it before evaluating the source argument, so an output-only lvalue is not read.

Raises: `<bad-arg>` for a null `Func` or an index outside `argc`, or `<bad-arity>` when `argc` disagrees with a fixed signature.

Source: `lib/func.x:179`

x2c_func_shared

```
Func x2c_func_shared(FuncAdapter adapter, List signature)
```

Builds the shared handle for a direct function or noncapturing lambda. The compiler stores the result in a file-static `Func` that every conversion of that function reuses, so the binding lives for the program and `Func.move` never transfers it.

Raises: the causes of `Func.new`.

Source: `lib/func.x:389`

x2c_func_unrepresentable_argument

```
FuncArg x2c_func_unrepresentable_argument(Func fn, unsigned i, List source)
```

Rejects a value argument whose source type has no `Var` representation. Generated calls use this branch instead of compiling an impossible conversion. Raises: `<bad-types>`.

Source: `lib/func.x:264`

Func

Func.move

```
void Func.move(Func function, Scope *slot)
```

Transfers `function`'s storage to the `Scope` held by `slot`. The shared handle of a direct function or noncapturing lambda lives for the program and stays where it is, so a caller may hand any `Func` to an operation that takes ownership. A NULL `function` does nothing.

Raises: the causes of `Scope.move`, which leave ownership unchanged.

Source: `lib/func.x:454`

Func.signature

```
List Func.signature(Func function)
```

Returns a native binding's borrowed canonical signature. The result has the `((func (PARAMETERS...)) RESULT...)` shape supplied to the constructor and remains valid for the binding's lifetime.

Source: `lib/func.x:433`

Public types

Type	Kind	Summary
`FuncAdapter`	callback	Uniform shape of a native `Func` adapter.
`FuncArg`	struct	Argument carrier borrowed for one synchronous `Func.apply` call.
`FuncArgData`	union	Storage for either the value or reference member selected by `FuncArg`.

FuncAdapter

```
typedef Var (*FuncAdapter)(Func fn, const FuncArg *argv)
```

Uniform shape of a native `Func` adapter. A `Func` stores the callback pointer without retaining it and calls it synchronously with borrowed `fn` and `argv`; the callback code must outlive the `Func`. Compiler-generated adapters use the checked value and reference readers, call their native target, and box its result. A returned `void` remains no-value. Adapters receive no count. Fixed bindings check arity first, and rest bindings receive one packed `List` argument.

Source: `lib/func.x:48`

FuncArg

```
typedef struct FuncArg { FuncArgData data; List reference_type; } FuncArg
```

Argument carrier borrowed for one synchronous `Func.apply` call. A null `reference_type` selects the copied `Var` bits; otherwise `reference` and the canonical source type are borrowed and must remain valid through the call. A reference target may be mutated by the native function.

Source: `lib/func.x:35`

FuncArgData

```
typedef union FuncArgData { Var value; const void *reference; } FuncArgData
```

Storage for either the value or reference member selected by `FuncArg`. Pointer-bearing `Var` storage and referenced objects belong to the caller.

Source: `lib/func.x:24`

Design notes

`Func` owns the binding of one synchronous native call: a canonical signature, an adapter, and optional context bytes copied into the same `Scope` allocation. Arguments reach native code only through the checked value and reference readers that generated adapters call.

lib/iter.x

Single-pass pull iterators.

Primary API

Function	Summary
range	Returns an inclusive integer range over caller-supplied iter storage.
Iter.all	Reports whether every remaining element satisfies pred.
Iter.any	Reports whether any remaining element satisfies pred.
Iter.chain	Returns a lazy iterator over first followed by second.
Iter.count	Returns the number of remaining elements, consuming iter.
Iter.enumerate	Returns a lazy iterator over (index value) Lists starting at start.
Iter.filter	Returns a lazy iterator over elements accepted by func's Var truthiness.
Iter.find	Returns the first element accepted by pred's Var truthiness, else void.
Iter.foldl	Folds fn over iter from seed, left to right, and returns the final accumulator.
Iter.head	Returns a lazy iterator over at most count leading source values.
Iter.iter	Returns iter unchanged as its own iterator.
Iter.map	Returns a lazy iterator over func applied to each element of iter.
Iter.map2	Returns a lazy iterator over fn(left, right), requiring fn.
Iter.max	Returns the largest remaining element, or void when there is none.
Iter.min	Returns the smallest remaining element, or void when there is none.
Iter.new	Returns zeroed iterator storage owned by the active Scope.
Iter.product	Returns the product of the remaining elements, consuming iter.
Iter.repeat	Returns a lazy iterator that yields value at most count times.
Iter.scan	Returns a lazy iterator over every new accumulator of fn.
Iter.sum	Returns the sum of the remaining elements, consuming iter.
Iter.try_next	Advances iter, writing the next element through out.
Iter.unique	Returns a lazy iterator that yields the first occurrence of each value.
Iter.zip	Returns a lazy iterator over canonical (left right) Lists.
Iter.zip_with	Returns a lazy iterator over fn(left, right), applied pairwise.

Functions

range

```
meta native Iter range(int start, int end, int step, Iter iter)
```

Returns an inclusive integer range over caller-supplied iter storage. Both endpoints belong to the range when the step direction reaches them: range(1, 4, 1, &storage) yields 1, 2, 3, 4, and range(3, 1, -1, &storage) counts down 3, 2, 1. A step that would cross end stops short of it, so range(0, 9, 2, &storage) yields 0, 2, 4, 6, 8. A range pointed against its step is empty. A range needs no state struct and no cleanup.

Raises: <bad-arg> when step is zero. A null iter returns NULL without raising.

Source: lib/iter.x:227

Iter

Iter.all

```
int Iter.all(Iter iter, Func pred)
```

Reports whether every remaining element satisfies pred. Vacuously true for an empty iterator, decided before pred is consulted. Otherwise it stops at the first element the predicate rejects and answers 0, leaving the rest unconsumed. Elements are passed as values and the result uses ordinary Var truthiness.

Raises: whatever the source, Func.apply, or pred raises. A null pred answers 1 for an empty iterator and 0 for any other.

Source: `lib/iter.x:773`

Iter.any

```
int Iter.any(Iter iter, Func pred)
```

Reports whether any remaining element satisfies `pred`. Stops at the first element the predicate accepts, so the iterator is left positioned after it and the rest is never pulled. An empty iterator answers 0. Elements are passed as values and the result uses ordinary `Var` truthiness.

Raises: whatever the source, `Func.apply`, or `pred` raises. A null `pred` answers 0.

Source: `lib/iter.x:759`

Iter.chain

```
meta native Iter Iter.chain(Iter first, Iter second, Iter dest)
```

Returns a lazy iterator over `first` followed by `second`. Pulls do not reach `second` until `first` is exhausted. Both sources and `dest` are borrowed and must outlive traversal. A null source contributes no elements, and a null `dest` returns NULL. Pulling may raise any cause raised by either source.

Source: `lib/iter.x:528`

Iter.count

```
meta native int Iter.count(Iter iter)
```

Returns the number of remaining elements, consuming `iter`. Counting drains the iterator, and an `Iter` cannot be rewound, so if you need the elements as well, collect them with `Iter.list` or `Iter.array` and ask the collection for its length. The remaining element count must fit in `int`.

Raises: `<void-op>` for a source callback that yields `void`, plus any cause raised by that callback.

Source: `lib/iter.x:804`

Iter.enumerate

```
meta native Iter Iter.enumerate(Iter iter, int start, Iter dest)
```

Returns a lazy iterator over `(index value)` `Lists` starting at `start`. The first pulled value is paired with exactly `start`, then the native `int` index increases by one for each source value. The source and `dest` are borrowed and must outlive traversal; `start` plus the number of successfully pulled values must remain within the `int` range.

Raises: `<alloc-fail>` or `<size-limit>` while interning a pair, plus any cause raised by the source. A null `dest` returns NULL.

Source: `lib/iter.x:505`

Iter.filter

```
meta native Iter Iter.filter(Iter iter, Func func, Iter dest)
```

Returns a lazy iterator over elements accepted by `func`'s `Var` truthiness. The predicate fits in `dest`, so that storage is all you declare. Rejected elements are consumed without being yielded, so one request for an element can pull many from the source.

Elements are passed as values. Both `iter` and its storage, `dest`, and any dynamic or captured `func` must remain valid while the result is used.

A null `dest` returns NULL. A null `func` raises `<bad-arg>` on the first invocation, after the source element has been consumed. An empty source does not invoke or check `func`.

Raises: whatever the source, `Func.apply`, or `func` raises while pulling.

Source: `lib/iter.x:347`

Iter.find

```
Var Iter.find(Iter iter, Func pred)
```

Returns the first element accepted by `pred`'s `Var` truthiness, else `void`. Stops as soon as it finds one, so the iterator can be pulled further for the elements after the match. `void` means "no element matched", which is unambiguous because no iterator may yield `void`. Elements are passed as values. Raises: whatever the source, `Func.apply`, or `pred` raises. A null `pred` returns `void`.

Source: `lib/iter.x:790`

Iter.foldl

```
Var Iter.foldl(Iter iter, Var seed, Func fn)
```

Folds `fn` over `iter` from `seed`, left to right, and returns the final accumulator. Consumes the whole iterator. A `void seed` means "use the first element as the seed", so folding an empty iterator from `void` returns `void`; any other `seed` is returned unchanged when there is nothing to fold. A null `fn` raises `<bad-arg>` on the first combining invocation. Inputs pulled before a failure stay consumed. The accumulator and elements are passed as values. Empty input, or one element with a `void` seed, does not invoke or check `fn`.

Raises: whatever the source, `Func.apply`, or `fn` raises.

Source: `lib/iter.x:736`

Iter.head

```
meta native Iter Iter.head(Iter iter, int count, Iter dest)
```

Returns a lazy iterator over at most `count` leading source values. Construction consumes nothing. Pulling stops after `count` values or source exhaustion, whichever comes first, and leaves any later source values unconsumed. A nonpositive count yields nothing. The source and caller-owned `dest` must outlive traversal; a null `dest` returns NULL. Pulling may raise any cause raised by the source.

Source: `lib/iter.x:549`

Iter.iter

```
Iter Iter.iter(Iter x, Iter dest)
```

Returns `iter` unchanged as its own iterator. `dest` is ignored; ownership and remaining traversal state are unchanged.

Source: `lib/iter.x:204`

Iter.map

```
meta native Iter Iter.map(Iter iter, Func func, Iter dest)
```

Returns a lazy iterator over `func` applied to each element of `iter`. `dest` is caller-owned `struct Iter` storage. It and the source iterator's storage, plus any dynamic or captured `func`, must remain valid while the result is used. Elements are passed as values. Nothing runs until an element is pulled, and each pull travels back up the chain for exactly one element per stage, so taking two elements from a map over a range calls `func` twice. A null `func` raises `<bad-arg>` on the first invocation, after the source element has been consumed.

```
struct Iter source_storage, map_storage;
Iter squares = range(1, 4, 1, &source_storage).map(_square,
                                                   &map_storage);
printf("%s\n", squares.list().str());
```

A null `dest` returns NULL. An empty source does not invoke or check `func`. Raises: whatever the source, `Func.apply`, or `func` raises while pulling, including `<void-op>` when `func` returns `void` and the iterator rejects it as an element.

Source: `lib/iter.x:320`

Iter.map2

```
meta native Iter Iter.map2(Iter left, Iter right, Func fn, Iter dest)
```

Returns a lazy iterator over `fn(left, right)`, requiring `fn`. The strict form of `Iter.zip_with`. Behavior is identical, except that a null `fn` returns NULL instead of yielding pairs. Use it when a missing callback should fail at construction instead of changing the element type.

The sources, destination, callback lifetime, value passing, and pull-time failures are those of `Iter.zip_with`. A null `fn` or `dest` returns NULL.

Source: `lib/iter.x:491`

Iter.max

```
meta native Var Iter.max(Iter iter)
```

Returns the largest remaining element, or `void` when there is none. Consumes the iterator, comparing with `Var.compare`, which is total across types. Equal values keep the earliest, so the result is the first of any tie.

Raises: `<void-op>` for a source callback that yields `void`, plus any cause from the source or `Var.compare`.

Source: `lib/iter.x:844`

Iter.min

```
meta native Var Iter.min(Iter iter)
```

Returns the smallest remaining element, or `void` when there is none. Consumes the iterator, comparing with `Var.compare`, which is total across types. Equal values keep the earliest, so the result is the first of any tie.

Raises: `<void-op>` for a source callback that yields `void`, plus any cause from the source or `Var.compare`.

Source: `lib/iter.x:858`

Iter.new

```
meta native Iter Iter.new(void)
```

Returns zeroed iterator storage owned by the active `Scope`. Use this when an iterator itself must be returned or stored as a value. Producers such as `Map.keys` initialize the result through their existing destination argument. The iterator still borrows its source and callback, which must outlive every pull. The active `Scope` must remain live for as long as the returned handle is used.

Caller-supplied `struct Iter` storage remains preferable for a local traversal because it allocates nothing.

Raises: `<alloc-fail>` when storage cannot be allocated.

Source: `lib/iter.x:152`

Iter.product

```
meta native Var Iter.product(Iter iter)
```

Returns the product of the remaining elements, consuming `iter`. Starts from the integer 1 and multiplies with ordinary `Var` arithmetic, so an empty iterator produces 1 and numeric types promote as they would in an expression. Integer results wrap to `Var.binary`'s promoted type width.

Raises: any cause from the source or `Var.binary` while multiplying an element into the running product.

Source: `lib/iter.x:831`

Iter.repeat

```
meta native Iter Iter.repeat(Var value, int count, Iter dest)
```

Returns a lazy iterator that yields `value` at most `count` times. A nonpositive count yields nothing. `dest` is caller-owned, and any storage referenced by `value` must outlive traversal. A null `dest` returns NULL. Pulling a repeated `void` raises `<void-op>`.

Source: `lib/iter.x:568`

Iter.scan

```
meta native Iter Iter.scan(Iter iter, Var seed, Func fn, Iter dest)
```

Returns a lazy iterator over every new accumulator of `fn`. Each step computes `fn(accumulator, element)`, keeps the result as the new accumulator, and yields it. `seed` is the initial accumulator and is never yielded, so a scan produces exactly as many elements as its source. `fn` is required. The accumulator and each element are passed as values.

```
struct Iter source_storage, scan_storage;
Iter running = range(1, 4, 1, &source_storage).scan(0, _add,
                                                    &scan_storage);
foreach (int total, running) printf("%d\n", total);
```

That prints 1, 3, 6, and 10; the seed 0 never appears.

The source and its storage, `dest`, and any dynamic or captured `fn` must remain valid while the result is used. Constructing the iterator does not invoke `fn`. Pulling may raise whatever `Func.apply`, the source, or `fn` raises, including `<void-op>` when `fn` returns `void` and the iterator rejects it as an element. A null `fn` or `dest` returns NULL.

Source: `lib/iter.x:395`

Iter.sum

```
meta native Var Iter.sum(Iter iter)
```

Returns the sum of the remaining elements, consuming `iter`. Starts from the integer 0 and adds with ordinary `Var` arithmetic, so numeric element types promote as they would in an expression and an empty iterator sums to 0. Use `Iter.accumulate` for the running totals.

Raises: any cause from the source or `Var.binary` while adding an element to the running total.

Source: `lib/iter.x:818`

Iter.try_next

```
int Iter.try_next(Iter iter, Var &?out)
```

Advances `iter`, writing the next element through `out`. Returns nonzero when it produced an element and zero once the iterator is exhausted, and writes `out` only in the nonzero case. Prefer this form. Status travels separately from the payload, so no value is reserved to mean "finished".

An `Iter` is single-pass. There is no rewind, so build a fresh iterator when you need a second traversal.

```
struct Iter storage;
Iter counts = range(3, 1, -1, &storage);
Var value;
while (counts.try_next(value)) printf("%d\n", value);
```

Raises: `<void-op>` when a source callback claims success with a `void` element, plus any cause raised by that callback. A null `iter` or `out` reads as exhausted without raising.

Source: `lib/iter.x:176`

Iter.unique

```
meta native Iter Iter.unique(Iter iter, Iter dest)
```

Returns a lazy iterator that yields the first occurrence of each value. Equality and hashing follow `Map`, so source order decides which equal value survives. Construction allocates a `Scope`-owned seen table; pulls may grow it. The source, `dest`, and owning `Scope` must outlive traversal.

Raises: `<alloc-fail>`, `<size-limit>`, `<invariant>`, or a cause from the source, hashing, or equality while constructing or pulling. A null `dest` returns NULL without allocating.

Source: `lib/iter.x:591`

Iter.zip

```
meta native Iter Iter.zip(Iter left, Iter right, Iter dest)
```

Returns a lazy iterator over canonical `(left right)` `Lists`. Destructure each pair with `Var (a, b) = pair;`. Pairing ends as soon as either source does. Each pull advances the left side first: if the right side is exhausted, that unmatched left value is consumed; if the left side is exhausted, the right side is not pulled. Both sources and `dest` must outlive traversal. Each yielded pair follows the lifetime of the `List` pool owning its canonical match.

Raises: `<alloc-fail>` or `<size-limit>` while interning a pair, plus any cause raised by either source. A null `dest` returns NULL.

Source: `lib/iter.x:448`

Iter.zip_with

```
meta native Iter Iter.zip_with(Iter left, Iter right, Func fn, Iter dest)
```

Returns a lazy iterator over `fn(left, right)`, applied pairwise. Like `Iter.zip`, but each pair is combined by `fn` instead of being built into a `List`, and the result likewise ends with the shorter side. `fn` is optional. A null `fn` yields two-element pair `Lists` as `Iter.zip` does. `Iter.map2` is the same operation with the callback required.

Both sources and their storage, `dest`, and any dynamic or captured `fn` must remain valid while the result is used. Values are passed, not aliases into either source. A null `dest` returns NULL. If either source is empty, pulling does not invoke or check `fn`.

Raises: whatever either source, `Func.apply`, or `fn` raises while pulling. With a null `fn`, pair interning may raise `<alloc-fail>` or `<size-limit>`.

Source: `lib/iter.x:465`

Advanced and interop API

Function	Summary
`Iter.init`	Initializes caller-supplied iterator storage and returns it.
`Iter.unzip`	Returns an iterator over the two column iterators of paired elements.

`Iter`

Iter.init

```
Self Iter.init(Self iter, Var obj, IterNextFn next, Var state)
```

Initializes caller-supplied iterator storage and returns it. `iter` is caller-owned storage, normally a `struct Iter` local. This call writes `obj`, the `next` callback, and the initial `state`, clears the auxiliary callback slot, and returns `iter`. It allocates nothing; the iterator lives as long as the supplied storage.

A callback fills `*out` and returns 1, or returns 0 to report exhaustion. It must never report success with `void`, which the element domain excludes. `Iter.try_next` raises `<void-op>` for that violation. A null `next` yields an already-exhausted iterator. Inside a callback, read the stored fields with `->`; `iter.next(...)` is receiver syntax and calls the method instead of the field.

```
struct Iter storage;
Iter evens = Iter.init(&storage, 6, _evens_next, 0);
Var value;
while (evens.try_next(value)) printf("%d\n", value);
```

Storage must outlive every iterator derived from it, because a pipeline stage keeps a pointer to its source. Never return an `Iter` built over locals, since the iterator would outlive the struct it names; return a collected `List` or `Array`, or take the storage as a parameter. One `struct Iter` is one live iterator, so reusing a single storage variable for two stages of a pipeline silently overwrites the first.

A null `iter` returns NULL.

Source: `lib/iter.x:132`

Iter.unzip

```
Iter Iter.unzip(Iter iter, UnzipShared *shared, Iter dest)
```

Returns an iterator over the two column iterators of paired elements. Every element of `iter` must be a two-element `List`. The result yields two elements, the left column and then the right one, and is exhausted after that. Each column arrives as a `Var` holding an iterator embedded in `shared`. Passing nonnull storage to `.iter(&storage)` performs the `Var` conversion, but `Iter.iter` ignores that storage and returns the embedded column; `shared` remains its owner.

The columns are independent, and only the lag between them is buffered. Draining one column holds every element the other has not reached yet, so interleaving the two costs little and front-loading one costs memory proportional to the source. Each column retains fewer consumed slots than the larger of 256 and its unread backlog. Compaction leaves its capacity unchanged. `shared` owns those buffers and must outlive both columns.

```
List pairs = %((1 10) (2 20));
struct Iter source_storage, columns_storage, left_storage, right_storage;
UnzipShared shared;
Iter columns = pairs.iter(&source_storage).unzip(&shared,
                                                 &columns_storage);
Iter lefts = columns.next().iter(&left_storage);
Iter rights = columns.next().iter(&right_storage);
printf("%s %s\n", lefts.list().str(), rights.list().str());
```

Raises: `<bad-types>` when an element is not a `List`, `<bad-arg>` when it is not a two-element `List`, or `<alloc-fail>` / `<size-limit>` while creating or growing the column buffers. A source may also raise while a column pulls. A null `shared` or `dest` returns NULL without raising or allocating.

Source: `lib/iter.x:647`

Convenience API

Function	Summary
`Iter.accumulate`	Returns a lazy iterator over the running numeric sum of `iter`.
`Iter.next`	Returns the next element, or `void` once `iter` is exhausted.

`Iter`

Iter.accumulate

```
meta native Iter Iter.accumulate(Iter iter, Var initial, Iter dest)
```

Returns a lazy iterator over the running numeric sum of `iter`. The numeric special case of `Iter.scan`. Each element is added through `Var.binary` with the same promotion and failure rules as `Iter.sum`, and the resulting `Var` is yielded without narrowing. A `void initial` starts the total at integer zero; any other `initial` seeds it, and the seed itself is never yielded. Prefer `Iter.scan` for a different operation and `Iter.sum` when only the final total matters.

Raises: any cause from the source or `Var.binary` while adding an element to the running total. A null `dest` returns NULL without raising.

Source: `lib/iter.x:423`

Iter.next

```
meta native Var Iter.next(Iter iter)
```

Returns the next element, or `void` once `iter` is exhausted. An adapter over `Iter.try_next`, unambiguous because no iterator may yield `void` as an element. Prefer `Iter.try_next` in new code; it reports exhaustion separately from the element.

Raises: `<void-op>` when the source callback claims success with `void`, plus any cause raised by that callback.

Source: `lib/iter.x:196`

Public types

Type	Kind	Summary
`Iter`	struct	Caller-owned handle to single-pass iterator state.
`IterNextFn`	callback	Advances an `Iter`, reporting success separately from the element.
`UnzipColumn`	struct	Internal column selector embedded in `UnzipShared`.
`UnzipShared`	struct	Caller-owned buffering shared by the two iterators from `Iter.unzip`.

Iter

```
typedef struct Iter *Iter
```

Caller-owned handle to single-pass iterator state. A valid handle points to storage initialized by `Iter.init` or a collection adapter. Its storage, borrowed sources, and stored callbacks must outlive traversal. The handle has no destructor; auxiliary `Map` or `Array` storage follows the lifetime of its owning `Scope`.

Source: `lib/iter.x:23`

IterNextFn

```
typedef int (*IterNextFn)(Iter iter, Var *out)
```

Advances an `Iter`, reporting success separately from the element. The iterator borrows the callback and invokes it synchronously. A nonzero result must write one non-`void` value through `out`; zero reports exhaustion and must leave `out` unchanged.

Source: `lib/iter.x:30`

UnzipColumn

```
typedef struct { UnzipShared *shared, int column; } UnzipColumn
```

Internal column selector embedded in `UnzipShared`. Callers reserve it as part of that shared record and do not initialize or use it independently.

Source: `lib/iter.x:54`

UnzipShared

```
typedef struct UnzipShared UnzipShared
```

Caller-owned buffering shared by the two iterators from `Iter.unzip`. Its source and this record must outlive both columns. `Buffer` allocations belong to the `Scope` that owns the `Arrays` created at initialization; that `Scope` must remain live through all column pulls and provides cleanup.

Source: `lib/iter.x:48`

Design notes

`Iter` advances one element at a time and reports exhaustion separately. Lazy operations borrow their source iterators and callbacks. The caller supplies iterator storage, which must outlive traversal. Iterators cannot yield `void`.

Tests and examples

`make verify` (`unittest/test-iter.x`) and `make examples` (`iter-examples`).

`lib/json.x`

JSON text to and from ordinary x2c values.

Primary API

Function	Summary
`Json.bool`	Returns the JSON boolean for the truth of `value`.
`Json.boolean`	Returns 1 for JSON `true` and 0 for JSON `false`.
`Json.is_bool`	Reports whether `value` is a JSON boolean.
`Json.parse`	Returns the x2c value of the JSON text `source`.
`Json.read_file`	Returns the x2c value of the JSON file at `path`, as `Json.parse` does.
`Json.write_file`	Replaces the file at `path` with `value` as compact JSON text.
`JsonBool.repr`	Returns `true` or `false`.
`JsonBool.str`	Returns `true` or `false`.
`JsonBool.truth`	Returns nonzero for `true`.
`JsonBool.var`	Boxes a JSON boolean.
`Var.json`	Returns `value` as compact JSON text.
`Var.jsonbool`	Unboxes a JSON boolean from a `Var` produced by `JsonBool.var`.
`Var.pretty_json`	Returns `value` as JSON text indented two spaces per level.

`Json`

Json.bool

```
Var Json.bool(int value)
```

Returns the JSON boolean for the truth of `value`.

Source: `lib/json.x:156`

Json.boolean

```
int Json.boolean(Var value)
```

Returns 1 for JSON `true` and 0 for JSON `false`.

Raises: `<bad-types>` when `value` is not a JSON boolean.

Source: `lib/json.x:164`

Json.is_bool

```
int Json.is_bool(Var value)
```

Reports whether `value` is a JSON boolean.

Source: `lib/json.x:159`

Json.parse

```
meta native Var Json.parse(String source)
```

Returns the x2c value of the JSON text `source`. Objects become `Maps`, arrays `Arrays`, and strings `Strings`. A number without a fraction or exponent is an `int` `Var` when it fits, then a `long`, then an `unsigned long`; any other number is a `double`. JSON null is the all-zero `Var`, and true and false are `JsonBools`. A repeated object name keeps its last value.

Raises: `<malformed>` with `reason`, `offset`, `line`, and `column` details when `source` is not one JSON value surrounded only by whitespace, nests arrays and objects more than 512 deep, or contains a number too large for a `double`, an unpaired surrogate escape, or `\u0000`.

Source: `lib/json.x:203`

Json.read_file

```
Var Json.read_file(Path path)
```

Returns the x2c value of the JSON file at `path`, as `Json.parse` does.

Raises: the causes of `Path.read_text`, or `<malformed>` as `Json.parse` does, with a `path` detail added.

Source: `lib/json.x:209`

Json.write_file

```
void Json.write_file(Var value, Path path)
```

Replaces the file at `path` with `value` as compact JSON text.

Raises: the causes of `Var.json` and `Path.write_text`.

Source: `lib/json.x:482`

JsonBool

JsonBool.repr

```
String JsonBool.repr(JsonBool value)
```

Returns `true` or `false`.

Source: `lib/json.x:150`

JsonBool.str

```
String JsonBool.str(JsonBool value)
```

Returns `true` or `false`.

Source: `lib/json.x:147`

JsonBool.truth

```
int JsonBool.truth(JsonBool value)
```

Returns nonzero for `true`.

Source: `lib/json.x:153`

JsonBool.var

```
Var JsonBool.var(JsonBool value)
```

Boxes a JSON boolean.

Source: `lib/json.x:141`

Var

Var.json

```
String Var.json(Var value)
```

Returns `value` as compact JSON text. `Map` names are written in byte order and must be `Strings` or `Symbols` with distinct spellings; a `Symbol` value is written as a string. A `List` is written as an array. A `double` is written with the fewest digits that read back to the same value. Each maximal ill-formed UTF-8 subsequence in a string is written as U+FFFD, as Python and JavaScript decoders replace it.

Raises: `<bad-types>` for a value or name JSON cannot hold, `<bad-arg>` for a `String` and a `Symbol` name with the same spelling, `<conv-range>` for NaN or an infinity, or `<size-limit>` for nesting deeper than 512 levels.

Source: `lib/json.x:472`

Var.jsonbool

```
meta native JsonBool Var.jsonbool(Var value)
```

Unboxes a JSON boolean from a `Var` produced by `JsonBool.var`.

Source: `lib/json.x:144`

Var.pretty_json

```
String Var.pretty_json(Var value)
```

Returns `value` as JSON text indented two spaces per level.

Raises: the causes of `Var.json`.

Source: `lib/json.x:477`

Public types

Type	Kind	Summary
`Json`	alias	The receiverless owner of `Json.parse` and the other JSON operations.
`JsonBool`	struct	A JSON `true` or `false`, kept distinct from the numbers 1 and 0.

Json

```
typedef Symbol Json
```

The receiverless owner of `Json.parse` and the other JSON operations.

Source: `lib/json.x:20`

JsonBool

```
typedef struct JsonBool *JsonBool
```

A JSON `true` or `false`, kept distinct from the numbers 1 and 0. The two values are process-lifetime singletons made by `Json.bool`.

Source: `lib/json.x:25`

Design notes

Json owns the crossing between JSON text and ordinary x2c values: an object is a Map with String keys, an array an Array, null the all-zero Var, and true or false a `JsonBool`. A Map keeps neither member order nor duplicate names, so a repeated name keeps its last value and output writes names in byte order; `packages/yyjson` serves a program that needs either.

Tests and examples

`make verify` (`unittest/test-json.x`).

lib/lib.x

Disjoint sets with union-find.

Advanced and interop API

Function	Summary
`DisjointSet.drop`	Releases the set's arrays when `DisjointSet.free` releases the set.
`DisjointSet.find`	Returns the representative of `x` and compresses its traversed path.
`DisjointSet.init`	Creates a union-find over elements `0` through `n - 1`.
`DisjointSet.num_components`	Returns the live set's current number of disjoint components.
`DisjointSet.sizes`	Returns canonical `(size representative)` rows for the current roots.
`DisjointSet.union`	Merges the components containing `a` and `b` by size.

`DisjointSet`

DisjointSet.drop

```
void DisjointSet.drop(DisjointSet set)
```

Releases the set's arrays when `DisjointSet.free` releases the set.

Source: `lib/lib.x:43`

DisjointSet.find

```
int DisjointSet.find(DisjointSet set, int x)
```

Returns the representative of `x` and compresses its traversed path. `set` must be live and `x` must be between zero and `set.length - 1`.

Source: `lib/lib.x:51`

DisjointSet.init

```
void DisjointSet.init(DisjointSet set, int n)
```

Creates a union-find over elements `0` through `n - 1`. `n` must be nonnegative. The result belongs to the active `Scope` and starts with each element in its own component.

Raises: `<size-limit>` or `<alloc-fail>` while allocating the structure or its arrays.

Source: `lib/lib.x:31`

DisjointSet.num_components

```
int DisjointSet.num_components(DisjointSet set)
```

Returns the live set's current number of disjoint components.

Source: `lib/lib.x:93`

DisjointSet.sizes

```
List DisjointSet.sizes(DisjointSet set)
```

Returns canonical `(size representative)` rows for the current roots. Rows are ordered by `Var.compare`, ascending first by size and then by representative. Returned `Lists` live through their owning `List` pool, which may be an ancestor of the active pool.

Raises: `<size-limit>` or `<alloc-fail>` while collecting or sorting rows.

Source: `lib/lib.x:85`

DisjointSet.union

```
void DisjointSet.union(DisjointSet set, int a, int b)
```

Merges the components containing `a` and `b` by size. `set` must be live and both elements must be in range. The larger component's root wins unless the sizes tie, when the root of `a` wins. Merging an existing component is a no-op.

Source: `lib/lib.x:67`

Public types

Type	Kind	Summary
`DisjointSet`	class	Owns a mutable union-find forest over integer elements.

DisjointSet

`class DisjointSet struct { int *parent, *size, length, ncmpnts; } *`

Owns a mutable union-find forest over integer elements.

`DisjointSet.new` creates the only valid state. The structure and its two arrays are `Scope`-owned; `DisjointSet.free` releases them early and invalidates every alias.

Source: `lib/lib.x:19`

Design notes

This module is aggregated into the x2c.x runtime and implicitly available through the prelude. DisjointSet owns a mutable union-find structure behind a pointer typedef; callers release it with DisjointSet.free.

`lib/lisp/lisp.x`

The Lisp runtime: reader, session, and evaluator.

Primary API

Function	Summary
`lisp_car`	Returns the first element of `value`, or `void` when it is `nil`.
`lisp_cdr`	Returns the tail of `value`, or `nil` when it is `nil`.
`lisp_match_replace`	Returns the instantiated `template` when `input` matches `pat`.
`lisp_string_lstrip`	Trims leading bytes, accepting the same charset values as `strip`.
`lisp_string_rstrip`	Trims trailing bytes, accepting the same charset values as `strip`.
`lisp_string_strip`	Trims the bytes in `chars`, using whitespace for an empty String.
`lisp_truth`	Returns Lisp true for every value except nil and `void`.
`native_scalar_access`	Returns the access record for an exact C scalar type such as `(unsigned long)`, or NULL for any other type.
`Lisp.adopt`	Makes `lisp` read `parent`'s definitions for names it does not bind.
`Lisp.apply`	Applies `callable` to already evaluated `values`.
`Lisp.bind`	Installs `function` as `name` and transfers its storage to `lisp`.
`Lisp.call_budget`	Sets how many calls one evaluation of `lisp` may make before it is stopped.
`Lisp.eval`	Evaluates one Lisp form in `lisp`.
`Lisp.eval_file`	Reads and evaluates every form from `source`.
`Lisp.eval_string`	Reads and evaluates every form in `source`.
`Lisp.freeze`	Marks `lisp` complete, so nothing produced later may reach it.
`Lisp.kernel`	Creates an isolated embedded `Lisp` session with only evaluator primitives.
`Lisp.new`	Creates an isolated session with the standard Lisp environment loaded.
`Lisp.set_interrupted`	Sets whether evaluation in `lisp` is interrupted.
`Lisp.try_get`	Writes the global binding for `name` to `out` when present.

Functions

lisp_car

`Var lisp_car(Var value)`

Returns the first element of `value`, or `void` when it is `nil`. A nonlist operand raises `<bad-types>`.

Source: `lib/lisp/primitives.x:26`

lisp_cdr

`Var lisp_cdr(Var value)`

Returns the tail of `value`, or `nil` when it is `nil`. A nonlist operand raises `<bad-types>`.

Source: `lib/lisp/primitives.x:35`

lisp_match_replace

```
Var lisp_match_replace(List input, Var pat, Var template)
```

Returns the instantiated `template` when `input` matches `pat`. `List.match_replace` returns a `List`, so a template that is a bare binder loses a scalar result. Lisp sees the replacement itself. A miss, malformed pattern, cache pressure, or machine error returns `input` unchanged.

Source: `lib/lisp/primitives.x:246`

lisp_string_lstrip

```
String lisp_string_lstrip(String string, Var chars)
```

Trims leading bytes, accepting the same charset values as `strip`.

Source: `lib/lisp/primitives.x:234`

lisp_string_rstrip

```
String lisp_string_rstrip(String string, Var chars)
```

Trims trailing bytes, accepting the same charset values as `strip`.

Source: `lib/lisp/primitives.x:238`

lisp_string_strip

```
String lisp_string_strip(String string, Var chars)
```

Trims the bytes in `chars`, using whitespace for an empty String.

Source: `lib/lisp/primitives.x:230`

lisp_truth

```
Var lisp_truth(Var value)
```

Returns Lisp true for every value except nil and `void`.

Source: `lib/lisp/primitives.x:14`

native_scalar_access

```
NativeScalarAccess native_scalar_access(List exact_type)
```

Returns the access record for an exact C scalar type such as `(unsigned long)`, or NULL for any other type.

Source: `lib/lisp/lisp.x:51`

Lisp

Lisp.adopt

```
void Lisp.adopt(Lisp lisp, Lisp parent)
```

Makes `lisp` read `parent`'s definitions for names it does not bind.

Lisp definitions cannot replace names supplied by an ancestor. `Lisp.set_global` may shadow them in the child. Child writes never reach the parent or another child. The child also takes the parent's special forms in place of its own, because the evaluator recognizes a special form by the identity of the `Func` a name resolves to, and the child resolves reserved names in the parent.

The caller keeps `parent` alive for as long as any child names it, and freezes it with `Lisp.freeze` before the first child runs: a child's values belong to a narrower `Context` than the parent's, so nothing a child produces may become reachable from the parent.

Source: `lib/lisp/sessions.x:165`

Lisp.apply

```
Var Lisp.apply(Lisp lisp, Var callable, List values)
```

Applies `callable` to already evaluated `values`. Macros and evaluator special forms other than the built-in `apply` are not procedures and are rejected. The built-in accepts a callable and a `List` and recursively applies those already evaluated values. The Lisp session owns `Scope` allocations made by the call; returned canonical or caller-

supplied values keep their existing owners. The caller retains responsibility for `callable`, `values`, and their referents.

Raises: `<bad-arg>` for a null session, `<not-call>` for a non-procedure or evaluator-only callable, `<bad-arity>` or `<bad-types>` at the call boundary, or a cause raised by the called procedure.

Source: `lib/lisp/evaluate.x:770`

Lisp.bind

```
void Lisp.bind(Lisp lisp, String name, Func function)
```

Installs `function` as `name` and transfers its storage to `lisp`. Invalid arguments raise `<bad-arg>` without transferring ownership. Once validation succeeds, the `Func` moves before the global insertion. The caller relinquishes ownership even if name canonicalization or `Map` growth then raises `<alloc-fail>`, `<size-limit>`, or `<bad-enc>`, but may borrow the pointer while the session lives. A direct function or noncapturing lambda converts to a shared handle that lives for the program; the session only borrows it. Values inside the `Func`, including its signature graph, retain their existing owners.

Source: `lib/lisp/sessions.x:226`

Lisp.call_budget

```
void Lisp.call_budget(Lisp lisp, long budget)
```

Sets how many calls one evaluation of `lisp` may make before it is stopped. `lisp` must be a live session and `budget` must be positive. The default is `LISP_CALL_STEP_MAX`, which is large enough that only a computation that does not end reaches it; a test sets a small one to reach it quickly.

The budget belongs to the public entry. `Lisp.eval`, `Lisp.apply`, and `Lisp.eval_string` each open one, and a call that runs it out does not renew it, so one entry reports a runaway once however many calls follow.

Source: `lib/lisp/evaluate.x:677`

Lisp.eval

```
Var Lisp.eval(Lisp lisp, Var expression)
```

Evaluates one Lisp form in `lisp`. Evaluation is synchronous and may retain the expression or values it reaches in session globals, Lambdas, or captures as described by the module ownership rule. Effects completed before a later failure are not rolled back. Raises: `<bad-arg>` for a null session, or any evaluator, imported operation, or called-procedure cause.

Source: `lib/lisp/evaluate.x:753`

Lisp.eval_file

```
Var Lisp.eval_file(Lisp l, File source)
```

Reads and evaluates every form from `source`. Reading starts at the current stream position, consumes through EOF, and leaves the borrowed stream open. Forms run in order, so effects from forms completed before a later read or evaluation failure are not rolled back. A null `Lisp` session is rejected by `Lisp.eval_string` only after the stream has been consumed.

Raises: `<bad-arg>` for a null stream or embedded NUL, `<io-fail>`, `<size-limit>`, or `<alloc-fail>` while reading, or any cause from `Lisp.eval_string`.

Source: `lib/lisp/evaluate.x:811`

Lisp.eval_string

```
Var Lisp.eval_string(Lisp lisp, String source)
```

Reads and evaluates every form in `source`. Forms run in source order and the return value is the last result, or Lisp `nil` for a null, empty, or comment-only source. Globals and other effects completed before a later reader

or evaluator failure remain installed. Lambdas retain local values that may become reads after a global callable changes, including names in current data. These captures belong to the session; borrowed referents keep their original lifetimes.

Raises: <bad-arg> for a null session, <incomplete> or <malformed> while reading, or any cause from Lisp.eval.

Source: lib/lisp/evaluate.x:786

Lisp.freeze

```
void Lisp.freeze(Lisp lisp)
```

Marks lisp complete, so nothing produced later may reach it.

A frozen session rejects def and Lisp.set_global: a value produced while a narrower Context is current would leave the session holding values that die with that Context.

Source: lib/lisp/sessions.x:179

Lisp.kernel

```
Lisp Lisp.kernel(void)
```

Creates an isolated embedded Lisp session with only evaluator primitives. The caller owns a successful session and must pass it to Lisp.destroy. Returns NULL if a long shared name cannot be owned by the active pool chain. Failure of native once initialization writes a diagnostic and aborts the process.

Raises: <alloc-fail> or <size-limit> while creating session storage, or <bad-enc> while interning shared or special-form names.

Source: lib/lisp/sessions.x:96

Lisp.new

```
Lisp Lisp.new(void)
```

Creates an isolated session with the standard Lisp environment loaded. The caller owns the result and must pass it to Lisp.destroy. If standard-source evaluation transfers, no handle is returned and the constructed session remains allocated. Raises any cause from Lisp.kernel or Lisp.eval_string.

Source: lib/lisp/sessions.x:133

Lisp.set_interrupted

```
void Lisp.set_interrupted(Lisp lisp, int interrupted)
```

Sets whether evaluation in lisp is interrupted. While interrupted is nonzero, every interpreted call raises <interrupt>, including calls made after a catch. The store is async-signal-safe, so a signal handler may interrupt a running evaluation; the owner clears the flag before the next one. lisp must be a live session.

Source: lib/lisp/evaluate.x:686

Lisp.try_get

```
int Lisp.try_get(Lisp lisp, String name, Var &?out)
```

Writes the global binding for name to out when present. Returns 1 only after writing the borrowed value. A null session, name, or output, or an absent name returns 0 and leaves out unchanged. Raises <alloc-fail> or <bad-enc> when a nonempty lookup name cannot be canonicalized.

Source: lib/lisp/sessions.x:189

Advanced and interop API

Function	Summary
Lisp.cleanup	Ends the owned lifetime when a managed local leaves its block.

Function	Summary
Lisp.destroy	Releases a Lisp session and invalidates all session-owned state.
Lisp.read	Reads one Lisp form and returns <value> or <eof>.
Lisp.set_global	Binds name to value in the embedded Lisp global environment.

Lisp

Lisp.cleanup

```
void Lisp.cleanup(Lisp value)
```

Ends the owned lifetime when a managed local leaves its block.

Source: lib/lisp/lisp.x:64

Lisp.destroy

```
void Lisp.destroy(Lisp lisp)
```

Releases a Lisp session and invalidates all session-owned state. This includes its global and reserved Maps, Lambdas, and transferred Funcs. Borrowed values are not released. A null session does nothing; no evaluation may remain active. Destroying its still-active Scope raises <bad-state>.

Source: lib/lisp/sessions.x:145

Lisp.read

```
Symbol Lisp.read(Lisp lisp, String source, unsigned &?cursor, Var &?out)
```

Reads one Lisp form and returns <value> or <eof>. A nonnull out receives the form only for <value>; it is otherwise unchanged. New result storage uses the caller's active Scope and canonical pools, not the temporary token Scope, and remains valid until those owners are released. On success cursor advances past the form, at EOF it becomes the source length, and on a reader error it identifies the failing form's first token. A null source or cursor returns <eof> without raising. A nonnull cursor must initially hold a byte offset no greater than the source length. A successful session construction must first initialize the shared reader names; afterward lisp is not consulted and may be null.

Raises: <incomplete> for a truncated form, <malformed> for invalid reader syntax, or <alloc-fail>, <size-limit>, or <bad-enc> while tokenizing, constructing, interning, or boxing the form.

Source: lib/lisp/reader.x:173

Lisp.set_global

```
void Lisp.set_global(Lisp lisp, String name, Var value)
```

Binds name to value in the embedded Lisp global environment. The session stores the value in a raw slot, so void remains distinct from Lisp nil. The binding does not take ownership of value referents; their canonical graphs or other owners must outlive the binding or session. A host binding may shadow an inherited name in this session; it does not change the ancestor. Lisp def still rejects inherited names, including a name the host has shadowed locally. Binding an x2c. name protects that namespace from later Lisp def forms, but direct calls to this function may replace such a binding.

Raises: <bad-arg> for a null session or name, or <alloc-fail>, <size-limit>, or <bad-enc> while canonicalizing or storing the binding.

Source: lib/lisp/sessions.x:205

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
lisp_add	Concatenates when either operand is String, otherwise adds dynamically.
lisp_atom	Returns Lisp true unless value is a nonempty List.
lisp_compare	Compares Lisp numbers by value and returns a boxed negative, zero, or positive.
lisp_divide	Divides the first value by each later one; one value reciprocates it.
lisp_eq	Returns Lisp true when a and b are equal by Var.equal.
lisp_eq_chain	Reports whether every neighbouring pair of numbers compares equal.
lisp_ge_chain	Reports whether two or more numbers never increase.
lisp_gt_chain	Reports whether two or more numbers strictly decrease.
lisp_le_chain	Reports whether two or more numbers never decrease.
lisp_list	Returns Lisp true when value is List-typed, including nil.
lisp_lt_chain	Reports whether two or more numbers strictly increase.
lisp_minus	Subtracts each later value from the first; one value negates it.
lisp_number	Returns Lisp true when value has an integer or floating kind.
lisp_pair	Returns Lisp true when value is a nonempty List.
lisp_plus	Adds or concatenates every value left to right; no values gives 0.
lisp_procedure	Returns Lisp true when value is a native function or Lambda.
lisp_read_file	Reads path completely, closes it, and returns its boxed String contents.
lisp_repr	Boxes the readable representation of value.
lisp_source_function	Marks the actual Lambda installed for one lowered source function.
lisp_str	Boxes the display String of value.
lisp_string	Returns Lisp true when value is a String.
lisp_string_append	Returns the boxed concatenation of left and right.
lisp_string_downcase	Returns a boxed lower-case copy of string.
lisp_substring	Returns the boxed unit-step slice string[start:stop].
lisp_symbol	Returns Lisp true when value has Symbol kind.
lisp_times	Multiplies every value; no values gives 1.
lisp_type	Returns the runtime tag of value.
lisp_write_file	Replaces path with text and reports Lisp success.
Lisp.active	Borrows the session executing the current native Lisp callback.
Lisp.automatic_storage	Borrows the current lowered source activation's automatic storage.
Lisp.result_storage	Borrows the caller's storage for a lowered record result.
Lisp.storage	Borrows the storage owner for callback state retained by this session.

Functions

lisp_add

```
Var lisp_add(Var a, Var b)
```

Concatenates when either operand is String, otherwise adds dynamically.

Source: lib/lisp/primitives.x:114

lisp_atom

```
Var lisp_atom(Var value)
```

Returns Lisp true unless value is a nonempty List.

Source: lib/lisp/primitives.x:17

lisp_compare

```
Var lisp_compare(Var a, Var b)
```

Compares Lisp numbers by value and returns a boxed negative, zero, or positive. Integer and floating encodings of one number compare equal. A nonnumeric operand raises `<bad-types>`.

Source: `lib/lisp/primitives.x:105`

lisp_divide

```
Var lisp_divide(List values)
```

Divides the first value by each later one; one value reciprocates it. An empty input raises `<bad-arity>`.

Source: `lib/lisp/primitives.x:151`

lisp_eq

```
Var lisp_eq(Var a, Var b)
```

Returns Lisp true when `a` and `b` are equal by `Var.equal`.

Source: `lib/lisp/primitives.x:42`

lisp_eq_chain

```
Var lisp_eq_chain(List values)
```

Reports whether every neighbouring pair of numbers compares equal. Fewer than two values raise `<bad-arity>`; a nonnumber raises `<bad-types>`.

Source: `lib/lisp/primitives.x:181`

lisp_ge_chain

```
Var lisp_ge_chain(List values)
```

Reports whether two or more numbers never increase. Fewer than two values raise `<bad-arity>`; a nonnumber raises `<bad-types>`.

Source: `lib/lisp/primitives.x:205`

lisp_gt_chain

```
Var lisp_gt_chain(List values)
```

Reports whether two or more numbers strictly decrease. Fewer than two values raise `<bad-arity>`; a nonnumber raises `<bad-types>`.

Source: `lib/lisp/primitives.x:199`

lisp_le_chain

```
Var lisp_le_chain(List values)
```

Reports whether two or more numbers never decrease. Fewer than two values raise `<bad-arity>`; a nonnumber raises `<bad-types>`.

Source: `lib/lisp/primitives.x:193`

lisp_list

```
Var lisp_list(Var value)
```

Returns Lisp true when `value` is `List`-typed, including `nil`.

Source: `lib/lisp/primitives.x:48`

lisp_lt_chain

```
Var lisp_lt_chain(List values)
```

Reports whether two or more numbers strictly increase. Fewer than two values raise `<bad-arity>`; a nonnumber raises `<bad-types>`.

Source: `lib/lisp/primitives.x:187`

lisp_minus

```
Var lisp_minus(List values)
```

Subtracts each later value from the first; one value negates it. An empty input raises `<bad-arity>`.
Source: `lib/lisp/primitives.x:130`

lisp_number
`Var lisp_number(Var value)`
Returns Lisp true when `value` has an integer or floating kind.
Source: `lib/lisp/primitives.x:56`

lisp_pair
`Var lisp_pair(Var value)`
Returns Lisp true when `value` is a nonempty `List`.
Source: `lib/lisp/primitives.x:45`

lisp_plus
`Var lisp_plus(List values)`
Adds or concatenates every value left to right; no values gives 0.
Source: `lib/lisp/primitives.x:120`

lisp_procedure
`Var lisp_procedure(Var value)`
Returns Lisp true when `value` is a native function or Lambda.
Source: `lib/lisp/primitives.x:65`

lisp_read_file
`Var lisp_read_file(String path)`
Reads `path` completely, closes it, and returns its boxed `String` contents. Raises the open, read, size, or allocation cause reported by `File`. An opened stream is still closed on transfer.
Source: `lib/lisp/primitives.x:256`

lisp_repr
`Var lisp_repr(Var value)`
Boxes the readable representation of `value`.
Source: `lib/lisp/primitives.x:213`

lisp_source_function
`Var lisp_source_function(Var callable)`
Marks the actual Lambda installed for one lowered source function.
Source: `lib/lisp/evaluate.x:827`

lisp_str
`Var lisp_str(Var value)`
Boxes the display `String` of `value`.
Source: `lib/lisp/primitives.x:210`

lisp_string
`Var lisp_string(Var value)`
Returns Lisp true when `value` is a `String`.
Source: `lib/lisp/primitives.x:59`

lisp_string_append
`Var lisp_string_append(String left, String right)`
Returns the boxed concatenation of `left` and `right`.
Source: `lib/lisp/primitives.x:216`

lisp_string_downcase

`Var lisp_string_downcase(String string)`

Returns a boxed lower-case copy of `string`.

Source: `lib/lisp/primitives.x:223`

lisp_substring

`Var lisp_substring(String string, int start, int stop)`

Returns the boxed unit-step slice `string[start:stop]`.

Source: `lib/lisp/primitives.x:219`

lisp_symbol

`Var lisp_symbol(Var value)`

Returns Lisp true when `value` has `Symbol` kind.

Source: `lib/lisp/primitives.x:62`

lisp_times

`Var lisp_times(List values)`

Multiplies every value; no values gives 1.

Source: `lib/lisp/primitives.x:142`

lisp_type

`Symbol lisp_type(Var value)`

Returns the runtime tag of `value`.

Source: `lib/lisp/primitives.x:68`

lisp_write_file

`Var lisp_write_file(String path, String text)`

Replaces `path` with `text` and reports Lisp success. The file is opened with truncation and always closed. A write or close failure returns `nil` after any accepted bytes; this operation is not atomic. An open failure transfers its `File` cause. `Null` text writes an empty file.

Source: `lib/lisp/primitives.x:264`

Lisp

Lisp.active

`Lisp Lisp.active(void)`

Borrows the session executing the current native Lisp callback.

Source: `lib/lisp/sessions.x:235`

Lisp.automatic_storage

`Scope *Lisp.automatic_storage(Lisp lisp)`

Borrows the current lowered source activation's automatic storage.

Source: `lib/lisp/sessions.x:239`

Lisp.result_storage

`Scope *Lisp.result_storage(Lisp lisp)`

Borrows the caller's storage for a lowered record result.

Source: `lib/lisp/sessions.x:242`

Lisp.storage

```
Scope *Lisp.storage(Lisp lisp)
```

Borrows the storage owner for callback state retained by this session.

Source: `lib/lisp/sessions.x:237`

Public types

Type	Kind	Summary
`Lisp`	struct	Represents one isolated embedded `Lisp` session.
`NativeScalarAccess`	struct	Describes one exact C scalar type in native bytes: its Var tag, size, alignment, load, and store.
`NativeScalarLoad`	callback	Reads one exact C scalar from `bytes`; a wide result is boxed in `owner`.
`NativeScalarStore`	callback	Writes `value`, converted to one exact C scalar type, to `bytes`.

Lisp

```
typedef struct Lisp *Lisp
```

Represents one isolated embedded `Lisp` session. Create it with `Lisp.new` or `Lisp.kernel` and end it with `Lisp.destroy`. The module header describes its owned and borrowed state. A session is mutable and requires caller serialization.

Source: `lib/lisp/lisp.x:59`

NativeScalarAccess

```
typedef struct NativeScalarAccess { Symbol tag; size_t size, alignment; NativeScalarLoad load; NativeScalarStore store; } *NativeScalarAccess
```

Describes one exact C scalar type in native bytes: its Var tag, size, alignment, load, and store. Compile-time code that keeps C objects in bytes, and the compiler's layouts, select a row with `native_scalar_access`.

Source: `lib/lisp/lisp.x:37`

NativeScalarLoad

```
typedef Var (*NativeScalarLoad)(const void *bytes, Scope *owner)
```

Reads one exact C scalar from `bytes`; a wide result is boxed in `owner`.

Source: `lib/lisp/lisp.x:30`

NativeScalarStore

```
typedef void (*NativeScalarStore)(void *bytes, Var value)
```

Writes `value`, converted to one exact C scalar type, to `bytes`.

Source: `lib/lisp/lisp.x:32`

Design notes

Lisp owns embedded compile-time Lisp sessions: the reader, the evaluator, and each session's global environment. Lambdas and natives receive their arguments evaluated left to right, macros and special forms receive them raw, nil and void are false, and a Lambda captures by value the free locals its body reads. Names are canonical Atoms, and `bind` reaches native operations through the table that lisp-targets.x builds.

The session Scope owns the Lisp record, its Maps, Lambdas, and bound Func storage, which hold Vars by value without cloning their referents. Those referents keep their own pool or caller owners and must outlive every session entry that refers to them. A session is not synchronized: its caller serializes evaluation and destroys it only after every call has returned.

Tests and examples

`make verify` (`unittest/test-lisp.x`) and `make examples`.

lib/list-selectors.x

Optional compound `List` selectors.

Primary API

Function	Summary
`List.caaaar`	Returns the element that `caaaar` selects from `value`.
`List.caaadr`	Returns the element that `caaadr` selects from `value`.
`List.caaar`	Returns the element that `caaar` selects from `value`.
`List.caadar`	Returns the element that `caadar` selects from `value`.
`List.caaddr`	Returns the element that `caaddr` selects from `value`.
`List.caadr`	Returns the element that `caadr` selects from `value`.
`List.cadaar`	Returns the element that `cadaar` selects from `value`.
`List.cadadr`	Returns the element that `cadadr` selects from `value`.
`List.cadar`	Returns the element that `cadar` selects from `value`.
`List.caddar`	Returns the element that `caddar` selects from `value`.
`List.cadddr`	Returns the element that `cadddr` selects from `value`.
`List.cdaaar`	Returns the tail that `cdaaar` selects from `value`.
`List.cdaadr`	Returns the tail that `cdaadr` selects from `value`.
`List.cdaar`	Returns the tail that `cdaar` selects from `value`.
`List.cdadar`	Returns the tail that `cdadar` selects from `value`.
`List.cdaddr`	Returns the tail that `cdaddr` selects from `value`.
`List.cdadr`	Returns the tail that `cdadr` selects from `value`.
`List.cdar`	Returns the tail that `cdar` selects from `value`.
`List.cddaar`	Returns the tail that `cddaar` selects from `value`.
`List.cddadr`	Returns the tail that `cddadr` selects from `value`.
`List.cddar`	Returns the tail that `cddar` selects from `value`.
`List.cdddar`	Returns the tail that `cdddar` selects from `value`.
`List.cddddr`	Returns the tail that `cddddr` selects from `value`.
`List.cdddr`	Returns the tail that `cdddr` selects from `value`.
`Var.caaaar`	Returns the element that `caaaar` selects from `value` as a `List`.
`Var.caaadr`	Returns the element that `caaadr` selects from `value` as a `List`.
`Var.caaar`	Returns the element that `caaar` selects from `value` as a `List`.
`Var.caadar`	Returns the element that `caadar` selects from `value` as a `List`.
`Var.caaddr`	Returns the element that `caaddr` selects from `value` as a `List`.
`Var.caadr`	Returns the element that `caadr` selects from `value` as a `List`.
`Var.cadaar`	Returns the element that `cadaar` selects from `value` as a `List`.
`Var.cadadr`	Returns the element that `cadadr` selects from `value` as a `List`.
`Var.cadar`	Returns the element that `cadar` selects from `value` as a `List`.
`Var.caddar`	Returns the element that `caddar` selects from `value` as a `List`.
`Var.cadddr`	Returns the element that `cadddr` selects from `value` as a `List`.
`Var.cdaaar`	Returns the tail that `cdaaar` selects from `value` as a `List`.
`Var.cdaadr`	Returns the tail that `cdaadr` selects from `value` as a `List`.
`Var.cdaar`	Returns the tail that `cdaar` selects from `value` as a `List`.
`Var.cdadar`	Returns the tail that `cdadar` selects from `value` as a `List`.
`Var.cdaddr`	Returns the tail that `cdaddr` selects from `value` as a `List`.

Function	Summary
Var.cdadr	Returns the tail that cdadr selects from value as a List.
Var.cdar	Returns the tail that cdar selects from value as a List.
Var.cddaar	Returns the tail that cddaar selects from value as a List.
Var.cddadr	Returns the tail that cddadr selects from value as a List.
Var.cddar	Returns the tail that cddar selects from value as a List.
Var.cdddar	Returns the tail that cdddar selects from value as a List.
Var.cddddr	Returns the tail that cddddr selects from value as a List.
Var.cdddr	Returns the tail that cdddr selects from value as a List.

List

List.caaaar

```
Var List.caaaar(List value)
```

Returns the element that caaaar selects from value.

Source: lib/list-selectors.x:81

List.caaadr

```
Var List.caaadr(List value)
```

Returns the element that caaadr selects from value.

Source: lib/list-selectors.x:81

List.caaar

```
Var List.caaar(List value)
```

Returns the element that caaar selects from value.

Source: lib/list-selectors.x:81

List.caadar

```
Var List.caadar(List value)
```

Returns the element that caadar selects from value.

Source: lib/list-selectors.x:81

List.caaddr

```
Var List.caaddr(List value)
```

Returns the element that caaddr selects from value.

Source: lib/list-selectors.x:81

List.caadr

```
Var List.caadr(List value)
```

Returns the element that caadr selects from value.

Source: lib/list-selectors.x:81

List.cadaar

```
Var List.cadaar(List value)
```

Returns the element that `cadaar` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cadadr
`Var List.cadadr(List value)`
Returns the element that `cadadr` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cadar
`Var List.cadar(List value)`
Returns the element that `cadar` selects from `value`.
Source: `lib/list-selectors.x:81`

List.caddar
`Var List.caddar(List value)`
Returns the element that `caddar` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cadddr
`Var List.cadddr(List value)`
Returns the element that `cadddr` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cdaaar
`List List.cdaaar(List value)`
Returns the tail that `cdaaar` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cdaadr
`List List.cdaadr(List value)`
Returns the tail that `cdaadr` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cdaar
`List List.cdaar(List value)`
Returns the tail that `cdaar` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cdadar
`List List.cdadar(List value)`
Returns the tail that `cdadar` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cdaddr
`List List.cdaddr(List value)`
Returns the tail that `cdaddr` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cdadr
`List List.cdadr(List value)`
Returns the tail that `cdadr` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cdar
`List List.cdar(List value)`
Returns the tail that `cdar` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cddaar
`List List.cddaar(List value)`
Returns the tail that `cddaar` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cddadr
`List List.cddadr(List value)`
Returns the tail that `cddadr` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cddar
`List List.cddar(List value)`
Returns the tail that `cddar` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cdddar
`List List.cdddar(List value)`
Returns the tail that `cdddar` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cddddr
`List List.cddddr(List value)`
Returns the tail that `cddddr` selects from `value`.
Source: `lib/list-selectors.x:81`

List.cdddr
`List List.cdddr(List value)`
Returns the tail that `cdddr` selects from `value`.
Source: `lib/list-selectors.x:81`

`Var`

Var.caaaar

```
Var Var.caaaar(Var value)
```

Returns the element that `caaaar` selects from `value` as a `List`.

Source: `lib/list-selectors.x:81`

Var.caaadr

```
Var Var.caaadr(Var value)
```

Returns the element that `caaadr` selects from `value` as a `List`.

Source: `lib/list-selectors.x:81`

Var.caaar

```
Var Var.caaar(Var value)
```

Returns the element that `caaar` selects from `value` as a `List`.

Source: `lib/list-selectors.x:81`

Var.caadar

```
Var Var.caadar(Var value)
```

Returns the element that `caadar` selects from `value` as a `List`.

Source: `lib/list-selectors.x:81`

Var.caaddr

```
Var Var.caaddr(Var value)
```

Returns the element that `caaddr` selects from `value` as a `List`.

Source: `lib/list-selectors.x:81`

Var.caadr

```
Var Var.caadr(Var value)
```

Returns the element that `caadr` selects from `value` as a `List`.

Source: `lib/list-selectors.x:81`

Var.cadaar

```
Var Var.cadaar(Var value)
```

Returns the element that `cadaar` selects from `value` as a `List`.

Source: `lib/list-selectors.x:81`

Var.cadadr

```
Var Var.cadadr(Var value)
```

Returns the element that `cadadr` selects from `value` as a `List`.

Source: `lib/list-selectors.x:81`

Var.cadar

```
Var Var.cadar(Var value)
```

Returns the element that `cadar` selects from `value` as a `List`.
Source: `lib/list-selectors.x:81`

Var.caddar
`Var Var.caddar(Var value)`
Returns the element that `caddar` selects from `value` as a `List`.
Source: `lib/list-selectors.x:81`

Var.cadddr
`Var Var.cadddr(Var value)`
Returns the element that `cadddr` selects from `value` as a `List`.
Source: `lib/list-selectors.x:81`

Var.cdaaar
`List Var.cdaaar(Var value)`
Returns the tail that `cdaaar` selects from `value` as a `List`.
Source: `lib/list-selectors.x:81`

Var.cdaadr
`List Var.cdaadr(Var value)`
Returns the tail that `cdaadr` selects from `value` as a `List`.
Source: `lib/list-selectors.x:81`

Var.cdaar
`List Var.cdaar(Var value)`
Returns the tail that `cdaar` selects from `value` as a `List`.
Source: `lib/list-selectors.x:81`

Var.cdadar
`List Var.cdadar(Var value)`
Returns the tail that `cdadar` selects from `value` as a `List`.
Source: `lib/list-selectors.x:81`

Var.cdaddr
`List Var.cdaddr(Var value)`
Returns the tail that `cdaddr` selects from `value` as a `List`.
Source: `lib/list-selectors.x:81`

Var.cdadr
`List Var.cdadr(Var value)`
Returns the tail that `cdadr` selects from `value` as a `List`.
Source: `lib/list-selectors.x:81`

Var.cdar

`List Var.cdar(Var value)`

Returns the tail that `cdar` selects from `value` as a `List`.

Source: `lib/list-selectors.x:81`

Var.cddaar

`List Var.cddaar(Var value)`

Returns the tail that `cddaar` selects from `value` as a `List`.

Source: `lib/list-selectors.x:81`

Var.cddadr

`List Var.cddadr(Var value)`

Returns the tail that `cddadr` selects from `value` as a `List`.

Source: `lib/list-selectors.x:81`

Var.cddar

`List Var.cddar(Var value)`

Returns the tail that `cddar` selects from `value` as a `List`.

Source: `lib/list-selectors.x:81`

Var.cdddar

`List Var.cdddar(Var value)`

Returns the tail that `cdddar` selects from `value` as a `List`.

Source: `lib/list-selectors.x:81`

Var.cddddr

`List Var.cddddr(Var value)`

Returns the tail that `cddddr` selects from `value` as a `List`.

Source: `lib/list-selectors.x:81`

Var.cdddr

`List Var.cdddr(Var value)`

Returns the tail that `cdddr` selects from `value` as a `List`.

Source: `lib/list-selectors.x:81`

Design notes

The prelude keeps car, cdr, caar, cadr, cddr, and caddr. Include this module for the remaining Common Lisp selector spellings through depth 4. Names read from right to left: `a` applies car and `d` applies cdr. Every step is `nil`-safe, and selected tails share their canonical `List` structure.

Tests and examples

`make verify` (`unittest/test-list.x`).

`lib/list/list.x`

Linked list with `Var` elements.

Primary API

Function	Summary
car	Returns the head of x, or void when x is nil.
cdr	Returns the tail of x, or nil when x is nil.
cons	Returns the canonical cons cell for head and tail.
Array.list	Returns a new List holding the elements of arr in order.
Array.list_free	Returns arr.list() and frees arr.
Iter.list	Drains iter into a new List.
List.all	True when every element satisfies pred by ordinary Var truthiness.
List.any	True when at least one element satisfies pred by ordinary Var truthiness.
List.append	Returns the concatenation of a and b.
List.array	Returns a new Array holding the elements of lst in order.
List.concat_n	Returns the concatenation of exactly list_count List arguments.
List.contains	Reports whether lst contains key by Var equality.
List.filter	Returns the elements pred accepts by ordinary Var truthiness.
List.find	Returns the first element pred accepts by ordinary Var truthiness, or void.
List.foldl	Folds fn over the elements of lst from the left, starting at seed.
List.get	Looks up an integer index or association key in list.
List.getindex	Returns list[index], or void when out of range.
List.getslice	Returns list[start:stop:step].
List.head	Returns the first count elements of list.
List.index	Returns the first index of key, or −1 when absent.
List.iter	Initializes caller-owned dest as a forward iterator over lst.
List.last	Returns the last value in lst, or void when it is empty.
List.len	Returns the number of cells in lst in O(n) time.
List.list_n	Builds a List from exactly element_count Var arguments.
List.map	Returns a canonical List holding fn applied front to back.
List.map2	Maps fn over aligned pairs from a and b.
List.reverse	Returns a new List holding the elements of lst in reverse order.
List.sort	Returns a copy of lst ordered by Var.compare.
List.sort_by	Returns a stable sorted copy using Array.sort_by's callback contract.
List.sort_with	Returns a stable sorted copy using Array.sort_with's callback contract.
List.tail	Returns the last count elements of list.
List.try_next	Writes the next element, advances cursor, and returns one.
List.unique	Returns a copy of lst with later duplicates removed.
List.unpack_n	Writes at most destination_count elements through List pointers.
List.unpack_vars_n	Writes at most destination_count elements through Var pointers.
List.write_str	Appends the List display text to out, using each element's write_str.
List.zip_with	Combines aligned values from two Lists with fn.
Map.list	Returns the entries of map as a List of two-element (key value) Lists, in the map's iteration order.

Functions

car

```
inline Var car(List x)
```

Returns the head of x, or void when x is nil.

Source: `lib/list/traversal.x:6`

cdr

```
inline List cdr(List x)
```

Returns the tail of `x`, or `nil` when `x` is `nil`. Nil-safe like `car`. The tail is the same canonical structure the cell was built from.

Source: `lib/list/traversal.x:12`

cons

```
List cons(Var head, List tail)
```

Returns the canonical cons cell for `head` and `tail`. Repeating the call with the same head bits and canonical tail returns the same cell from the active pool chain. An ancestor hit remains owned there; a miss belongs to the active pool. The tail is shared, and `nil` is the null pointer.

Raises: `<void-op>` when `head` is `void`, or `<alloc-fail>`, `<size-limit>`, or `<invariant>` when a new canonical cell cannot be installed.

Source: `lib/list/cells.x:14`

`Array`

Array.list

```
List Array.list(Array arr)
```

Returns a new `List` holding the elements of `arr` in order. Cells are built from the end backwards through `cons`, so the result is canonical and shares whatever tail it already has in common with another `List`. `arr` is neither consumed nor freed.

Raises: `<alloc-fail>` while constructing the result.

Source: `lib/list/building.x:11`

Array.list_free

```
List Array.list_free(Array arr)
```

Returns `arr.list()` and frees `arr`. The conversion uses `Array.list`, so the result is canonical and may share existing cells, including cells owned by an ancestor pool. It does not adopt the `Array`'s storage. `arr` is released on success and when the conversion transfers an `Error`.

Raises: `<alloc-fail>` while constructing the result.

Source: `lib/list/building.x:26`

`Iter`

Iter.list

```
meta native List Iter.list(Iter iter)
```

Drains `iter` into a new `List`. The iterator is consumed to exhaustion, so this is meaningful once and never returns for an infinite source. Elements appear in iteration order.

Raises: whatever the iterator's source raises, or `<alloc-fail>` while constructing the result.

Source: `lib/list/building.x:64`

`List`

List.all

```
int List.all(List lst, Func pred)
```

True when every element satisfies `pred` by ordinary `Var` truthiness. Stops at the first rejection. Nil is true; a null `pred` is false for a nonempty `List`.

Any cause raised by `pred` or its result's truth operation propagates.

Source: `lib/list/transformations.x:77`

List.any

```
int List.any(List lst, Func pred)
```

True when at least one element satisfies `pred` by ordinary `Var` truthiness. Stops at the first accepted element. Nil and a null `pred` are false.

Any cause raised by `pred` or its result's truth operation propagates.

Source: `lib/list/transformations.x:66`

List.append

```
Self List.append(Self a, Self b)
```

Returns the concatenation of `a` and `b`. Neither input is modified. `b` becomes the shared tail of the result, so only `a`'s cells are rebuilt, O(len(a)) of them. When either side is `nil` the other side is returned as it stands.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the copied prefix.

Source: `lib/list/building.x:79`

List.array

```
Array List.array(List lst)
```

Returns a new `Array` holding the elements of `lst` in order. The `Array` is a fresh mutable container the caller owns and should free; the elements are shared, since they are only `Var`s. Convert when you need indexed access or in-place mutation.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the result. The partial result is freed.

Source: `lib/list/building.x:38`

List.concat_n

```
List List.concat_n(unsigned list_count, ...)
```

Returns the concatenation of exactly `list_count` `List` arguments. Nil is a valid argument, no sentinel is read, and the final nonempty `List` becomes the shared tail of the result.

Raises: `<size-limit>` when `list_count` exceeds the supported index range, or `<alloc-fail>` while constructing the result.

Source: `lib/list/building.x:92`

List.contains

```
int List.contains(List lst, Var key)
```

Reports whether `lst` contains `key` by `Var` equality.

Source: `lib/list/traversal.x:105`

List.filter

```
Self List.filter(Self lst, Func pred)
```

Returns the elements `pred` accepts by ordinary `Var` truthiness. Elements are passed as values, and filtering `nil` gives `nil` without invoking or checking `pred`.

Raises: whatever `Func.apply`, `pred`, or result truthiness raises, or `<alloc-fail>` or `<size-limit>` while constructing the result. A null `pred` on nonempty input raises `<bad-arg>` from `Func.apply`.

Source: `lib/list/transformations.x:27`

List.find

```
Var List.find(List lst, Func pred)
```

Returns the first element `pred` accepts by ordinary `Var` truthiness, or `void`.

Any cause raised by `pred` or its result's truth operation propagates. A null `pred` returns `void`.

Source: `lib/list/transformations.x:55`

List.foldl

```
Var List.foldl(List lst, Var seed, Func fn)
```

Folds `fn` over the elements of `lst` from the left, starting at `seed`. `fn` receives the accumulator and then the next element, and returns the next accumulator. A `void` seed means "no seed": the first element becomes the initial accumulator and the fold starts at the second, and folding `nil` that way returns `void`. Empty input and a singleton with a `void` seed do not invoke `fn`. A null `fn` raises `<bad-arg>` when combining an element with the accumulator.

Any cause raised by `fn` propagates.

Source: `lib/list/transformations.x:44`

List.get

```
Var List.get(List list, Var key)
```

Looks up an integer index or association key in `list`. Integer keys use `List.getindex`, including negative indexes; every other key uses `List.assoc`. Either absent form returns `void`, as does an integer key outside the `int` index domain, which no `List` can reach.

Source: `lib/list/traversal.x:142`

List.getindex

```
Var List.getindex(List list, int index)
```

Returns `list[index]`, or `void` when out of range. A negative index counts from the end and is found without a length pass.

Source: `lib/list/traversal.x:118`

List.getslice

```
meta native Self List.getslice(Self list, int start, int stop, int step)
```

Returns `list[start:stop:step]`. `stop` is exclusive, negative bounds count from the end, and a negative `step` walks backwards. A full forward slice preserves `list` only when its identity is canonical in the active pool chain; otherwise it rebuilds the cells there so a detached pool cannot escape through the shortcut.

Raises: `<bad-arg>` when `step` is zero, or `<alloc-fail>` or `<size-limit>` while constructing the result.

Source: `lib/list/traversal.x:228`

List.head

```
meta native Self List.head(Self list, unsigned count)
```

Returns the first `count` elements of `list`. When `count` reaches or exceeds the length, `list` itself comes back, the same pointer. `Lists` are immutable, so sharing it is safe. Otherwise fresh canonical cells are built for the prefix.

Raises: `<alloc-fail>` or `<size-limit>` while constructing that prefix.

Source: `lib/list/traversal.x:170`

List.index

```
meta native int List.index(List l, Var key)
```

Returns the first index of `key`, or −1 when absent.

Source: `lib/list/traversal.x:99`

List.iter

```
Iter List.iter(List lst, Iter dest)
```

Initializes caller-owned `dest` as a forward iterator over `lst`. The iterator borrows the immutable cells and yields their stored `Var` bits without retaining them, so the owning pool must outlive iteration. A null `dest` returns NULL; `nil` produces an exhausted iterator.

Source: `lib/list/traversal.x:68`

List.last

```
Var List.last(List lst)
```

Returns the last value in `lst`, or `void` when it is empty.

Source: `lib/list/traversal.x:91`

List.len

```
int List.len(List lst)
```

Returns the number of cells in `lst` in O(n) time.

Source: `lib/list/traversal.x:84`

List.list_n

```
List List.list_n(unsigned element_count, ...)
```

Builds a `List` from exactly `element_count` `Var` arguments. Every argument is data, so `void` raises instead of being read as a terminator. A zero count returns `nil`.

Raises: `<void-op>` when an argument is `void`, or `<alloc-fail>` or `<size-limit>` while constructing the result.

Source: `lib/list/building.x:123`

List.map

```
List List.map(List lst, Func fn)
```

Returns a canonical `List` holding `fn` applied front to back. Elements are passed as values, and mapping `nil` gives `nil` without invoking or checking `fn`. A null `fn` on nonempty input raises `<bad-arg>`, and a callback result of `void` raises `<void-op>` when the result `List` is built.

Raises: those causes, whatever `Func.apply` or `fn` raises, or `<alloc-fail>` or `<size-limit>` while constructing the result.

Source: `lib/list/transformations.x:13`

List.map2

```
List List.map2(List a, List b, Func fn)
```

Maps `fn` over aligned pairs from `a` and `b`. A null callback returns `nil` without examining either `List`; otherwise this has the length, order, ownership, and failures of `List.zip_with`.

Source: `lib/list/transformations.x:102`

List.reverse

```
Self List.reverse(Self lst)
```

Returns a new `List` holding the elements of `lst` in reverse order. Fresh cells are built through `cons`, so the result is canonical and `lst` is untouched. Reversing `nil` gives `nil`.

Raises: `<alloc-fail>` while constructing the result.

Source: `lib/list/building.x:146`

List.sort

```
Self List.sort(Self lst)
```

Returns a copy of `lst` ordered by `Var.compare`. `lst` is unchanged. There is no comparator parameter. `Var.compare` orders the element tags involved, so a mixed-kind `List` still sorts. A `List` of fewer than two cells is returned as it stands.

Raises: whatever element comparison raises, or `<alloc-fail>` while constructing the result.

Source: `lib/list/transformations.x:116`

List.sort_by

```
Self List.sort_by(Self lst, Func key)
```

Returns a stable sorted copy using `Array.sort_by`'s callback contract. The key runs once per element in input order; an empty List invokes none. The input List is unchanged. Raises: allocation and the key's ordinary causes, including failure to compare the resulting keys.

Source: `lib/list/transformations.x:139`

List.sort_with

```
Self List.sort_with(Self lst, Func compare)
```

Returns a stable sorted copy using `Array.sort_with`'s callback contract. The input List is unchanged. Fewer than two cells return unchanged without a callback. Raises: allocation and the comparator's ordinary causes.

Source: `lib/list/transformations.x:127`

List.tail

```
meta native Self List.tail(Self list, unsigned count)
```

Returns the last `count` elements of `list`. The result is an existing tail of `list`, so nothing is allocated. When `count` reaches or exceeds the length, the whole `list` comes back.

Source: `lib/list/traversal.x:184`

List.try_next

```
int List.try_next(List lst, List &?cursor, Var &?out)
```

Writes the next element, advances `cursor`, and returns one. Initialize the caller-owned cursor to `lst`. A null pointer or an exhausted cursor returns zero without changing `cursor` or `out`. The cells are immutable, so only releasing the owning pool invalidates a cursor.

`foreach (Var item, lst)` compiles to this loop.

Source: `lib/list/traversal.x:53`

List.unique

```
meta native Self List.unique(Self lst)
```

Returns a copy of `lst` with later duplicates removed. The first occurrence of each value is kept and the original order is preserved. Duplicate detection runs through `Iter.unique`, whose state is held inside a `Scope` bracket that is released before returning. A `List` of fewer than two cells is returned as it stands.

Raises: causes from `Map` hashing or equality, or `<alloc-fail>` or `<size-limit>` while constructing the result.

Source: `lib/list/transformations.x:154`

List.unpack_n

```
int List.unpack_n(List src, unsigned destination_count, ...)
```

Writes at most `destination_count` elements through `List` pointers. Returns the number written. Extra source cells are left unread, and a short source leaves remaining destinations untouched. Each source value is decoded as a `List`, so another tag writes `nil`; a null destination is skipped but still counted.

Raises: `<size-limit>` when `destination_count` exceeds `INT_MAX`. The failure occurs before any destination is written.

Source: `lib/list/traversal.x:253`

List.unpack_vars_n

```
int List.unpack_vars_n(List src, unsigned destination_count, ...)
```

Writes at most `destination_count` elements through `Var` pointers. Returns the number written. Extra source cells are left unread, and a short source leaves remaining destinations untouched. A null destination is skipped but still counted.

Raises: `<size-limit>` when `destination_count` exceeds `INT_MAX`. The failure occurs before any destination is written.

Source: `lib/list/traversal.x:268`

List.write_str

```
Buffer List.write_str(List lst, Buffer out)
```

Appends the `List` display text to `out`, using each element's `write_str`. `List.str` calls this to build its result. The display form has one space inside each parenthesis, as in `( a b )`, including when nested in another container. This method temporarily sets the destination `Buffer`'s padding to one and restores it afterward.

Source: `lib/list/rendering.x:40`

List.zip_with

```
List List.zip_with(List a, List b, Func fn)
```

Combines aligned values from two `Lists` with `fn`. A null `fn` produces two-element pair `Lists`. The shorter input determines the result length, and an empty input does not inspect the callback. A callback receives the left and right values and is invoked front to back.

Raises: whatever `Func.apply`, `fn`, or result canonicalization raises.

Source: `lib/list/transformations.x:88`

Map

Map.list

```
meta native List Map.list(Map map)
```

Returns the entries of `map` as a `List` of two-element `(key value)` `Lists`, in the map's iteration order. This is the eager form of `Map.enumerate`: the pairs arrive as ordinary `List` data instead of through a cursor. An empty `map` returns nil. The pair cells are new; the keys and values are shared.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the result.

Source: `lib/list/building.x:52`

Advanced and interop API

Function	Summary
`List.assoc`	Returns the second value of the first association whose key equals `key`.
`List.caar`	Returns `car(car(lst))`.
`List.caddr`	Returns the third element, or `void`.
`List.cadr`	Returns `car(cdr(lst))`, or `void` when there is no second element.
`List.car`	Method form of `car`: the head of `lst`, or `void` when `lst` is `nil`.
`List.cddr`	Returns the tail after two cells, or `nil`.
`List.cdr`	Method form of `cdr`: the tail of `lst`, or `nil` when `lst` is `nil`.
`List.compare`	Compares `a` and `b` lexicographically through `Var.compare`.
`List.cons`	Method form of `cons`, with the same identity, lifetime, and failures.
`List.cons_in`	Returns the canonical cell for `head` and `tail` in `pool`'s chain.
`List.equal`	Reports equality of canonical chains by exact head and tail identity.
`List.flatten`	Flattens one level of nested `List`s into a canonical result.
`List.flatten_all`	Recursively flattens every nested `List` into a canonical result.
`List.hash`	Returns the stable hash of `List`'s exact head bits and tail identity.
`List.nth_cdr`	Returns the shared tail beginning `n` cells in.
`List.promote`	Moves `lst` out of the innermost interning pool into its parent.
`List.repr`	Returns the re-readable rendering of `lst`.
`List.str`	Returns the human-readable rendering of `lst`.
`List.sublis`	Recursively substitutes non-`List` nodes in `tree` from `alist`.
`List.subseq`	Returns every `step`th element from `start` up to exclusive `stop`.
`List.write_repr`	Appends the readable representation of `List` to a `Buffer`.
`Var.caar`	Applies the `caar` selector chain to `Var`.
`Var.caddr`	Applies the `caddr` selector chain to `Var`.
`Var.cadr`	Applies the `cadr` selector chain to `Var`.
`Var.car`	Treats `var` as a `List` and returns its first element.
`Var.cddr`	Applies the `cddr` selector chain to `Var`.
`Var.cdr`	Treats `var` as a `List` and returns its tail.
`Var.cons`	Returns `cons(head, tail)`, with the same identity and failures.

`List`

List.assoc

```
Var List.assoc(List list, Var key)
```

Returns the second value of the first association whose key equals `key`. Nil entries are skipped. A missing association and a missing second value both return `void`, so the two cases look the same here.

Source: `lib/list/traversal.x:155`

List.caar

```
inline Var List.caar(List lst)
```

Returns `car(car(lst))`. Compound accessors read from right to left: `a` applies `car` and `d` applies `cdr`. Every step is `nil`-safe.

Source: `lib/list/traversal.x:22`

List.caddr

```
inline Var List.caddr(List lst)
```

Returns the third element, or `void`.

Source: `lib/list/traversal.x:28`

List.cadr

```
inline Var List.cadr(List lst)
```

Returns `car(cdr(lst))`, or `void` when there is no second element.

Source: `lib/list/traversal.x:24`

List.car

```
inline Var List.car(List lst)
```

Method form of `car`: the head of `lst`, or `void` when `lst` is `nil`.

Source: `lib/list/traversal.x:15`

List.cddr

```
inline Self List.cddr(Self lst)
```

Returns the tail after two cells, or `nil`.

Source: `lib/list/traversal.x:26`

List.cdr

```
inline Self List.cdr(Self lst)
```

Method form of `cdr`: the tail of `lst`, or `nil` when `lst` is `nil`.

Source: `lib/list/traversal.x:17`

List.compare

```
meta native int List.compare(List a, List b)
```

Compares `a` and `b` lexicographically through `Var.compare`. Element comparison causes propagate.

Source: `lib/list/cells.x:61`

List.cons

```
List List.cons(Var head, List tail)
```

Method form of `cons`, with the same identity, lifetime, and failures.

Source: `lib/list/cells.x:20`

List.cons_in

```
List List.cons_in(Pool pool, Var head, List tail)
```

Returns the canonical cell for `head` and `tail` in `pool`'s chain. An ancestor hit keeps that ancestor's ownership; a miss is owned by `pool`. A null pool or `void` head returns `nil` without allocating. Any pool-managed graph reachable through `head` or `tail` is borrowed and must remain live for at least as long as the result.

Raises: `<alloc-fail>`, `<size-limit>`, or `<invariant>` while installing a new cell.

Source: `lib/list/list.x:44`

List.equal

```
int List.equal(List a, List b)
```

Reports equality of canonical chains by exact head and tail identity. It matches `List.hash`, so the same elements decide it: content for a small number, `Symbol`, `Atom`, canonical `String`, or nested `List`, and identity for a wide number boxed in a `Scope`, an `Array`, or a `Map`. Two `Lists` built from one `long` value hold two boxes and are unequal, while `List.compare` reads the values and calls them equal. Use `List.compare` where content has to decide.

Source: `lib/list/cells.x:52`

List.flatten

```
meta native Self List.flatten(Self lst)
```

Flattens one level of nested `Lists` into a canonical result. A nested `nil` contributes no element, non-`List` values retain their identity, and `nil` returns `nil`.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the result.

Source: `lib/list/transformations.x:169`

List.flatten_all

```
meta native Self List.flatten_all(Self lst)
```

Recursively flattens every nested `List` into a canonical result. Nested `nil` contributes no element and `nil` returns `nil`.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the result.

Source: `lib/list/transformations.x:186`

List.hash

```
meta native unsigned List.hash(List lst)
```

Returns the stable hash of `List`'s exact head bits and tail identity. This is a constant-time hash of one cell, which canonical cells make sound. It reaches an element's content only when the `Var` word holds it: a small number, a `Symbol`, an `Atom`, a canonical `String`, or a nested `List`. An element the word points at hashes by identity, so a wide number boxed in a `Scope`, an `Array`, or a `Map` gives two equal-looking `Lists` two hashes. Mutating an object referenced by the head does not change this hash.

Source: `lib/list/cells.x:35`

List.nth_cdr

```
meta native Self List.nth_cdr(Self list, int n)
```

Returns the shared tail beginning `n` cells in. Returns `nil` past the end and `list` itself when `n` is nonpositive.

Source: `lib/list/traversal.x:110`

List.promote

```
meta native Self List.promote(Self lst)
```

Moves `lst` out of the innermost interning pool into its parent. Cells, nested `Lists`, interned `String` cars, and long `Atom` payloads all move together, so a promoted `List` keeps its complete identity graph across the matching `Pool.close`. Pointers never change and ancestor-owned structure is left where it is; the return value is `lst` itself.

Cells are immutable, so an ancestor-owned cell can only reference ancestor-owned cars and tails. The walk stops at the first cell the innermost pool does not own. Promoting an already-promoted `List` is therefore cheap. `Var` kinds other than `String`, `Atom`, and `List` are `Scope`-managed and outside pool jurisdiction.

When another thread's pool promotes an equal cell first, that cell stays canonical; this one still survives, but `Pool.is_permanent` answers 0 for it. A null `lst` returns itself unchanged.

Source: `lib/list/cells.x:91`

List.repr

```
String List.repr(List lst)
```

Returns the re-readable rendering of `lst`. This is the `%(...)` spelling of the same data. Elements are rendered with their own `repr`, so `String`s come back quoted and `Atom`s print as bare names, and `nil` renders as `()`. Long nested structure wraps at about 80 columns. The canonical result follows the active `String` pool chain, may already be owned by an ancestor, and remains live until that owner is released.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the result.

Source: `lib/list/rendering.x:55`

List.str

```
String List.str(List lst)
```

Returns the human-readable rendering of `lst`. Elements are rendered with their own `str`, so a `String` element appears unquoted. The writer pads inside parentheses and breaks nested structure across lines at about 80 columns. Nil renders as `()`. The canonical result follows the active `String` pool chain, may already be owned by an ancestor, and remains live until its owning pool is released. Use `List.repr` when the text has to read back in.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the result.

Source: `lib/list/rendering.x:28`

List.sublis

```
meta native List List.sublis(List alist, List tree)
```

Recursively substitutes non-`List` nodes in `tree` from `alist`. Each association is a `List` whose first value is the key and whose second value is the replacement. Unmatched leaves are shared; `List` structure is rebuilt canonically, and `nil` returns `nil`.

Raises: a cause from key equality, or `<alloc-fail>` or `<size-limit>` while constructing the result.

Source: `lib/list/transformations.x:207`

List.subseq

```
meta native Self List.subseq(Self list, int start, int stop, int step)
```

Returns every `step`th element from `start` up to exclusive `stop`. Negative bounds count from the end. `step` must be positive.

Raises: `<bad-arg>` when `step` is less than 1, or `<alloc-fail>` while constructing the result.

Source: `lib/list/traversal.x:203`

List.write_repr

```
Buffer List.write_repr(List lst, Buffer out)
```

Appends the readable representation of `List` to a `Buffer`.

Source: `lib/list/rendering.x:62`

Var

Var.caar

```
inline Var Var.caar(Var var)
```

Applies the `caar` selector chain to `Var`.

Source: `lib/list/traversal.x:35`

Var.caddr

`inline Var Var.caddr(Var var)`

Applies the `caddr` selector chain to `Var`.

Source: `lib/list/traversal.x:41`

Var.cadr

`inline Var Var.cadr(Var var)`

Applies the `cadr` selector chain to `Var`.

Source: `lib/list/traversal.x:37`

Var.car

`Var Var.car(Var var)`

Treats `var` as a `List` and returns its first element.

Source: `lib/list/traversal.x:31`

Var.cddr

`inline List Var.cddr(Var var)`

Applies the `cddr` selector chain to `Var`.

Source: `lib/list/traversal.x:39`

Var.cdr

`List Var.cdr(Var var)`

Treats `var` as a `List` and returns its tail.

Source: `lib/list/traversal.x:33`

Var.cons

`List Var.cons(Var head, List tail)`

Returns `cons(head, tail)`, with the same identity and failures.

Source: `lib/list/cells.x:22`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`List.try_own`	Proves `lst` and its canonical children safe beyond every active pool.

`List`

List.try_own

`int List.try_own(List lst)`

Proves `lst` and its canonical children safe beyond every active pool. Returns 1 when the complete value is `nil`, already permanent, or can be promoted to the outermost `List` and `String` pools. Returns 0 when an active pool does not own part of the value. A zero result may follow successful promotion of an earlier cell or child. A

thread that loses a concurrent promotion of an equal value still gets 1, though `Pool.is_permanent` answers 0 for its surviving copy.

Raises: `<alloc-fail>` when promotion metadata cannot be allocated.

Source: `lib/list/cells.x:119`

Public types

Type	Kind	Summary
`List`	struct	Names an immutable canonical cons cell, or `nil` as NULL.

List

```
typedef struct List { Var car; struct List *cdr; } *List
```

Names an immutable canonical cons cell, or `nil` as NULL. A nonnull cell is borrowed from the pool that owns its exact car and tail identity; callers neither mutate nor free it. The car is never `void`, and the tail is `nil` or another live canonical `List`.

Source: `lib/list/list.x:28`

Design notes

List owns canonical immutable cons chains. A car holds any non-void Var and a cdr is nil or another List; identity is the car's exact bits plus the canonical tail, so mutating an Array or Map stored in a car leaves the identity of its cell unchanged.

Construction searches the requested pool and its ancestors: an existing cell keeps its ancestor's lifetime, and a miss belongs to the requested pool. Releasing a nested pool invalidates its unpromoted cells, while promotion keeps whole canonical List, String, and long-Atom structure at the same addresses. `void` is a terminal sentinel and never List data.

Tests and examples

`make verify` (`unittest/test-list.x`) and `make examples` (`docs-tour`).

`lib/logger.x`

Owned structured event delivery.

Primary API

Function	Summary
`Logger.new`	Creates a `Logger` in the active `Scope`, filtering below `min_level`.

`Logger`

Logger.new

```
Logger Logger.new(Symbol min_level)
```

Creates a `Logger` in the active `Scope`, filtering below `min_level`. The `Logger` borrows the active canonical pool for any memory sinks and owns a private `Scope` for sink state. An invalid level returns NULL.

Raises: `<alloc-fail>` when the `Logger` or its private `Scope` cannot be allocated.

Source: `lib/logger.x:807`

Advanced and interop API

Function	Summary
`log_debug`	Logs borrowed `fields` globally at debug level under `category`.
`log_error`	Logs borrowed `fields` globally at error level under `category`.
`log_event`	Delivers one event synchronously through the current global `Logger`.

Function	Summary
log_fatal	Logs borrowed fields globally at fatal level under category.
log_get_global_logger	Returns the borrowed current global Logger, or NULL when none is set.
log_info	Logs borrowed fields globally at info level under category.
log_set_global_logger	Installs a borrowed global Logger and returns the previous borrowed value.
log_should_log	Reports whether the current global Logger would deliver this event.
log_trace	Logs borrowed fields globally at trace level under category.
log_warn	Logs borrowed fields globally at warn level under category.
Logger.add_file_sink	Adds a plain-text sink for borrowed file and returns its handle.
Logger.add_memory_sink	Adds a sink that prepends captured events to destination.
Logger.add_sink	Appends a callback sink and returns its Logger-owned handle.
Logger.add_stderr_sink	Adds a text sink for borrowed stderr and returns its Logger-owned handle.
Logger.clear_sinks	Removes and invalidates every sink from logger.
Logger.debug	Logs borrowed fields synchronously at debug level under category.
Logger.error	Logs borrowed fields synchronously at error level under category.
Logger.fatal	Logs borrowed fields synchronously at fatal level under category.
Logger.flush	Calls configured flushers synchronously in registration order.
Logger.free	Flushes and retires every sink, then releases logger and its storage.
Logger.info	Logs borrowed fields synchronously at info level under category.
Logger.initialize	Installs the process-wide info-level Logger and stderr sink.
Logger.level_priority	Returns the ordering priority of a built-in level, or −1 when invalid.
Logger.log	Delivers one event synchronously to eligible sinks in registration order.
Logger.min_level	Returns logger's minimum enabled level, or the null Symbol for NULL.
Logger.remove_sink	Removes and invalidates sink, returning one when it belonged to logger.
Logger.set_min_level	Sets the minimum enabled level and returns one.
Logger.should_log	Reports whether an event would reach at least one sink.
Logger.shutdown	Shuts down process-wide Logger integration.
Logger.sink_count	Returns the number of configured sinks, or zero for a NULL Logger.
Logger.trace	Logs borrowed fields synchronously at trace level under category.
Logger.warn	Logs borrowed fields synchronously at warn level under category.

Functions

log_debug

```
void log_debug(Symbol category, List fields)
```

Logs borrowed fields globally at debug level under category. Delivery and failure behavior follow log_event.

Source: lib/logger.x:792

log_error

```
void log_error(Symbol category, List fields)
```

Logs borrowed fields globally at error level under category. Delivery and failure behavior follow log_event.

Source: lib/logger.x:795

log_event

```
void log_event(Symbol level, Symbol category, List fields)
```

Delivers one event synchronously through the current global Logger. With no global Logger this is a no-op; otherwise delivery and failures are those of Logger.log.

Source: `lib/logger.x:696`

log_fatal

```
void log_fatal(Symbol category, List fields)
```

Logs borrowed `fields` globally at fatal level under `category`. Delivery and failure behavior follow `log_event`.

Source: `lib/logger.x:796`

log_get_global_logger

```
Logger log_get_global_logger(void)
```

Returns the borrowed current global `Logger`, or NULL when none is set.

Source: `lib/logger.x:689`

log_info

```
void log_info(Symbol category, List fields)
```

Logs borrowed `fields` globally at info level under `category`. Delivery and failure behavior follow `log_event`.

Source: `lib/logger.x:793`

log_set_global_logger

```
Logger log_set_global_logger(Logger logger)
```

Installs a borrowed global `Logger` and returns the previous borrowed value. Neither `Logger` is flushed, freed, or otherwise retained by this call; the installed `Logger` must remain live until it is replaced or shutdown runs.

Source: `lib/logger.x:681`

log_should_log

```
int log_should_log(Symbol level, Symbol category)
```

Reports whether the current global `Logger` would deliver this event.

Source: `lib/logger.x:197`

log_trace

```
void log_trace(Symbol category, List fields)
```

Logs borrowed `fields` globally at trace level under `category`. Delivery and failure behavior follow `log_event`.

Source: `lib/logger.x:791`

log_warn

```
void log_warn(Symbol category, List fields)
```

Logs borrowed `fields` globally at warn level under `category`. Delivery and failure behavior follow `log_event`.

Source: `lib/logger.x:794`

`Logger`

Logger.add_file_sink

```
LogSink Logger.add_file_sink(Logger logger, File file)
```

Adds a plain-text sink for borrowed `file` and returns its handle. The caller must keep `file` open until the sink is removed or the `Logger` is freed. Removing the sink flushes but never closes the `File`. A NULL argument returns NULL.

Raises: `<alloc-fail>` or `<bad-enc>` when sink state cannot be represented.

Source: `lib/logger.x:382`

Logger.add_memory_sink

```
LogSink Logger.add_memory_sink(Logger logger, List *destination)
```

Adds a sink that prepends captured events to `destination`. The destination pointer is borrowed until the sink is retired. Entry `Lists`, `Strings`, and Lisp symbols are interned through the pool captured by `Logger.new`; a

canonical hit may retain an ancestor pool's lifetime. Wide values move to the `Logger`'s owning `Scope` on retirement, while other pointer-bearing values remain borrowed. Newest events appear first. A NULL `Logger` or destination returns NULL. Registration may raise `<alloc-fail>` or `<bad-enc>` while constructing sink state. Later event delivery may raise `<alloc-fail>`, `<size-limit>`, or `<bad-enc>` while retaining values.

Source: `lib/logger.x:580`

Logger.add_sink

```
LogSink Logger.add_sink( Logger logger, LogEmitter emit, LogFlusher flush, Var data)
```

Appends a callback sink and returns its `Logger`-owned handle. A NULL `Logger` returns NULL; a quiescent `Logger` also returns NULL for a NULL emitter. The `Logger` borrows `data` and both callbacks until the sink is removed, cleared, or the `Logger` is freed, so anything they reference must remain live. Emitters run synchronously in registration order.

Raises: `<bad-state>` during active sink delivery, or `<alloc-fail>` when the sink cannot be allocated.

Source: `lib/logger.x:256`

Logger.add_stderr_sink

```
LogSink Logger.add_stderr_sink(Logger logger)
```

Adds a text sink for borrowed `stderr` and returns its `Logger`-owned handle. The sink selects color from terminal state at registration and flushes after every event. A NULL `Logger` returns NULL.

Raises: `<alloc-fail>` or `<bad-enc>` when sink state cannot be represented.

Source: `lib/logger.x:371`

Logger.clear_sinks

```
void Logger.clear_sinks(Logger logger)
```

Removes and invalidates every sink from `logger`. A NULL `Logger` does nothing.

Raises: `<bad-state>` during active sink delivery.

Source: `lib/logger.x:281`

Logger.debug

```
void Logger.debug(Logger logger, Symbol category, List fields)
```

Logs borrowed `fields` synchronously at debug level under `category`. Delivery and failure behavior follow `Logger.log`.

Source: `lib/logger.x:775`

Logger.error

```
void Logger.error(Logger logger, Symbol category, List fields)
```

Logs borrowed `fields` synchronously at error level under `category`. Delivery and failure behavior follow `Logger.log`.

Source: `lib/logger.x:778`

Logger.fatal

```
void Logger.fatal(Logger logger, Symbol category, List fields)
```

Logs borrowed `fields` synchronously at fatal level under `category`. Delivery and failure behavior follow `Logger.log`.

Source: `lib/logger.x:779`

Logger.flush

```
void Logger.flush(Logger logger)
```

Calls configured flushers synchronously in registration order. A NULL `Logger` does nothing. A cause raised by a flusher transfers immediately, so later sinks are not flushed. A flusher must not mutate the sink list or free its `Logger`.

Source: `lib/logger.x:308`

Logger.free

```
void Logger.free(Logger logger)
```

Flushes and retires every sink, then releases `logger` and its storage. Borrowed `Files` and callback data are not released. The global and default slots are cleared when they refer to this `Logger`. A NULL `Logger` does nothing.

Raises: `<bad-state>` during sink delivery, or any cause from a flusher. Either failure leaves the `Logger` and all sinks live; earlier flushers may already have run.

Source: `lib/logger.x:825`

Logger.info

```
void Logger.info(Logger logger, Symbol category, List fields)
```

Logs borrowed `fields` synchronously at info level under `category`. Delivery and failure behavior follow `Logger.log`.

Source: `lib/logger.x:776`

Logger.initialize

```
void Logger.initialize(void)
```

Installs the process-wide info-level `Logger` and stderr sink. `Error` initializes first, then this function records its handler watermark and installs an observing `Error` handler. Runtime initialization calls this once and later shutdown hooks invoke `Logger.shutdown` before `Error` shuts down.

Raises: any cause from `Error` or `Logger` initialization.

Source: `lib/logger.x:845`

Logger.level_priority

```
int Logger.level_priority(Symbol level)
```

Returns the ordering priority of a built-in level, or −1 when invalid. Priorities run from `<trace>` at zero through `<off>` at six.

Source: `lib/logger.x:223`

Logger.log

```
void Logger.log(Logger logger, Symbol level, Symbol category, List fields)
```

Delivers one event synchronously to eligible sinks in registration order. `fields` is borrowed for the call. All sinks observe one immutable event, sequence numbers increase per `Logger`, and elapsed time starts with its first delivered event. NULL, filtered, invalid, sinkless, or category-less events do nothing. Fatal delivery flushes all sinks afterward.

Raises: any cause from an emitter or fatal-event flusher; later callbacks are then skipped.

Source: `lib/logger.x:148`

Logger.min_level

```
Symbol Logger.min_level(Logger logger)
```

Returns `logger`'s minimum enabled level, or the null `Symbol` for NULL.

Source: `lib/logger.x:227`

Logger.remove_sink

```
int Logger.remove_sink(Logger logger, LogSink sink)
```

Removes and invalidates `sink`, returning one when it belonged to `logger`. When quiescent, a NULL argument, foreign sink, or already retired sink returns zero.

Raises: `<bad-state>` during active sink delivery.

Source: `lib/logger.x:268`

Logger.set_min_level

```
int Logger.set_min_level(Logger logger, Symbol level)
```

Sets the minimum enabled level and returns one. A NULL `Logger` or invalid level returns zero without changing anything.

Source: `lib/logger.x:233`

Logger.should_log

```
int Logger.should_log(Logger logger, Symbol level, Symbol category)
```

Reports whether an event would reach at least one sink. A category names an event rather than filtering it. A NULL `Logger` or category, no sinks, `<off>`, an invalid level, or a filtered level returns zero.

Source: `lib/logger.x:192`

Logger.shutdown

```
void Logger.shutdown(void)
```

Shuts down process-wide `Logger` integration. Pending root `Errors` are offered to the active `Logger` before the observing handler is removed. The global slot is then cleared; a custom active `Logger` is flushed but remains caller-owned, while the default `Logger` is flushed, retired, and freed. A repeated call after shutdown is a no-op.

Raises: `<alloc-fail>` while opening the temporary canonical pool, `<bad-state>` when default-`Logger` delivery is active, or any cause from event delivery or a sink flusher. The failure may interrupt the remaining cleanup.

Source: `lib/logger.x:866`

Logger.sink_count

```
int Logger.sink_count(Logger logger)
```

Returns the number of configured sinks, or zero for a NULL `Logger`.

Source: `lib/logger.x:295`

Logger.trace

```
void Logger.trace(Logger logger, Symbol category, List fields)
```

Logs borrowed `fields` synchronously at trace level under `category`. Delivery and failure behavior follow `Logger.log`.

Source: `lib/logger.x:774`

Logger.warn

```
void Logger.warn(Logger logger, Symbol category, List fields)
```

Logs borrowed `fields` synchronously at warn level under `category`. Delivery and failure behavior follow `Logger.log`.

Source: `lib/logger.x:777`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`Logger.error_handler`	Offers the newest `Error` to the current global `Logger`.

`Logger`

Logger.error_handler

```
Symbol Logger.error_handler(List errors, Var data)
```

Offers the newest `Error` to the current global `Logger`. A well-formed entry with `<abort>` or `<log>` policy is rendered as an `<err-report>` event; missing, malformed, `<collect>`, and `<ignore>` input produces no event. An `<abort>` entry reaching the default `Logger` with only its stderr sink is instead reported as one line, `x2c: uncaught <code> in function (file:line): key value, ...`, omitting a location inside the runtime library; setting `X2C_DEBUG` keeps the event. The borrowed input is never consumed, `data` is ignored, and the handler always returns `<declined>`, leaving transfer to `Error` or another handler.

Raises: any cause from `Logger` delivery.

Source: `lib/logger.x:713`

Public types

Type	Kind	Summary
`LogEmitter`	callback	Receives one borrowed event and the `data` supplied at registration.
`LogEvent`	struct	Describes one immutable event during synchronous sink delivery.
`LogFlusher`	callback	Flushes one sink with its borrowed registration data.
`LogSink`	struct	A borrowed handle to one `Logger`-owned sink registration.
`Logger`	struct	An ordered, reentrant sink dispatcher allocated in the active `Scope`.

LogEmitter

```
typedef void (*LogEmitter)(Logger logger, const LogEvent *event, Var data)
```

Receives one borrowed event and the `data` supplied at registration. The callback may log recursively, but it must not retain `event`, mutate the active sink list, or free `logger` during delivery.

Source: `lib/logger.x:46`

LogEvent

```
typedef struct LogEvent { unsigned long sequence; long long wall_time_us; long long elapsed_us;
Symbol level, category, List fields; } LogEvent
```

Describes one immutable event during synchronous sink delivery. The event pointer is valid only for the callback. `fields` is borrowed with its caller-provided lifetime, and every sink receives the same value.

Source: `lib/logger.x:35`

LogFlusher

```
typedef void (*LogFlusher)(Logger logger, Var data)
```

Flushes one sink with its borrowed registration data. The callback must not mutate the active sink list or free `logger`.

Source: `lib/logger.x:51`

LogSink

`typedef struct LogSink *LogSink`

A borrowed handle to one `Logger`-owned sink registration. Removal, clearing, or freeing its `Logger` invalidates the handle; callers never free it directly.

Source: `lib/logger.x:29`

Logger

`typedef struct Logger *Logger`

An ordered, reentrant sink dispatcher allocated in the active `Scope`. `Logger.free` must run before that `Scope` is released. Memory sinks also require the canonical pool captured by `Logger.new` to remain live.

Source: `lib/logger.x:23`

Design notes

Logger owns filtering, event timestamps, sink ordering, and sink lifetime. It is reentrant and serializes configuration, sequencing, and delivery. Expensive callers may query Logger.should_log before constructing canonical field `Lists`.

Events carry one wall-clock timestamp and one monotonic elapsed timestamp. Every sink for an event observes the same immutable LogEvent. Built-in text sinks render through `Buffer` and `Var.write_repr` without constructing canonical presentation `Strings`.

Tests and examples

`make verify` (`unittest/test-logger.x`).

`lib/macrovalue/macrovalue.x`

Macros as values that build and recognize code.

Primary API

Function	Summary
`Macro_apply`	Applies a macro value to code values.
`Macro_case_capture_at`	Recognizes code built by `t` for the `case` whose site is `site`, which may be NULL, and publishes the captures under `names`.
`Macro_case_enter`	Stores in `*cursor` the List a macro-valued `case` examines for `value`, beneath its source wrappers, and returns whether there is one.
`Macro_case_own`	Whether an `expr` node whose content is `content` is its own view: the content is not a List headed by an Atom, as a retained invocation's is, or by the `expr` a template shell holds.
`Macro_case_pattern`	The pattern a macro-valued `case` compiles to; the compiler lowers a call of this to `Macro_case_capture_at` over the match subject.
`Macro_case_root`	Starts a macro-valued `case` whose derived pattern generated `match` code tests in place.
`Macro_case_shape`	The pattern a macro-valued `case` compiles to when the compiler derived `shape`, the pattern the case tests, while compiling it.
`Macro_case_start`	Starts a macro-valued `case` without `Macro_case_root` when `code` is its own view.
`Macro_case_symbol`	The short Symbol `value` holds, or 0 for any other value.
`Macro_case_view`	`Macro_case_enter` for a `value` that may be wrapped or may hold no List.
`Macro_close`	Records the Macro values an anonymous macro captured where it was created, so applying it later applies the same children.

Function	Summary
Macro_pattern	Derives the Match pattern that recognizes code this macro builds, capturing each parameter under the given binder.
Macro.declared	Returns the name a typed quotation declares with a Name hole whose local holds value: the name form of an x2c_ident spelling or a String, or a binding as it is.
Macro.inserted	Returns what a typed quotation inserts for one use of a hole whose local holds value, as a rebuild inserts it.
Macro.inserted_items	Returns the items a typed quotation splices for an expression sequence hole whose local holds values: each number, String, or Symbol becomes its literal, as a spliced data List's items do.
Macro.matcher	Prepares repeated recognition without retaining subject-specific bindings.
Macro.matches	Tests complete code against a macro and its hole patterns.
Macro.number_literal	Returns the literal expression of type result that holds value, a number of the scalar type type.
Macro.number_type	Returns the C type of a number's Var family, or NULL when value is not a number.
Macro.subject	Returns the table Macro.use_subject last set, or void.
Macro.typed	Returns what a typed quotation builds when its code is one hole whose local holds value: the inserted expression with the type type.
Macro.use_subject	Sets the (SPELLING BINDING) rows for global references and the (source-spelling BINDING SPELLING) rows for renamed local bindings in a compile-time call's syntax arguments.
MacroMatcher.matches	Tests code with the same source views and identity rules as a macro case.

Functions

Macro_apply

```
List Macro_apply(Macro t, List values)
```

Applies a macro value to code values. The result is a pending invocation; inserting it into a program expands and binds it there.

Source: lib/macrovalue/construction.x:12

Macro_case_capture_at

```
int Macro_case_capture_at( MacroCaseSite *site, List code, Macro t, List names, MatchCaptureBuffer *published)
```

Recognizes code built by t for the case whose site is site, which may be NULL, and publishes the captures under names. A pattern that does not depend on the current call's subject is prepared once and kept in the site; generated match code calls this for a macro-valued case.

Source: lib/macrovalue/recognition.x:27

Macro_case_enter

```
inline int Macro_case_enter(Var value, List *cursor)
```

Stores in *cursor the List a macro-valued case examines for value, beneath its source wrappers, and returns whether there is one. A List headed by another Symbol than a wrapper or expr is its own view.

Source: lib/macrovalue/macrovalue.x:95

Macro_case_own

```
inline int Macro_case_own(Var content)
```

Whether an expr node whose content is content is its own view: the content is not a List headed by an Atom, as a retained invocation's is, or by the expr a template shell holds.

Source: lib/macrovalue/macrovalue.x:78

Macro_case_pattern

```
List Macro_case_pattern(Macro t, List names)
```

The pattern a macro-valued `case` compiles to; the compiler lowers a call of this to `Macro_case_capture_at` over the match subject.

Source: `lib/macrovalue/recognition.x:12`

Macro_case_root

```
int Macro_case_root( List code, Macro t, List names, int statement, List *cursor, MatchCaptureBuffer *published)
```

Starts a macro-valued `case` whose derived pattern generated `match` code tests in place. Returns 1 when `code` is a retained invocation of `t` and its arguments captured under `names` into `published`, 0 when `code` cannot match, and 2 after storing in `*cursor` the List the pattern tests: the view of `code` or, for a `statement` body, of the one statement a `seq` holds.

Source: `lib/macrovalue/recognition.x:45`

Macro_case_shape

```
List Macro_case_shape(Macro t, List names, List shape)
```

The pattern a macro-valued `case` compiles to when the compiler derived `shape`, the pattern the case tests, while compiling it. Generated `match` code tests `shape` in place through `Macro_case_root` when it can, and otherwise lowers a call of this as it lowers `Macro_case_pattern`.

Source: `lib/macrovalue/recognition.x:19`

Macro_case_start

```
inline int Macro_case_start(List code, Symbol head, List *cursor)
```

Starts a macro-valued `case` without `Macro_case_root` when `code` is its own view. `head`, the Symbol heading the derived pattern, is not a source wrapper or `seq`. Returns 2 after storing `code` in `*cursor` for a subject headed by `head`, 0 for an empty subject or one headed by another Symbol than a wrapper, `expr`, or `seq`, which cannot match, and -1 when `Macro_case_root` must decide.

Source: `lib/macrovalue/macrovalue.x:64`

Macro_case_symbol

```
inline Symbol Macro_case_symbol(Var value)
```

The short Symbol `value` holds, or 0 for any other value.

Source: `lib/macrovalue/macrovalue.x:86`

Macro_case_view

```
int Macro_case_view(Var value, List *cursor)
```

`Macro_case_enter` for a `value` that may be wrapped or may hold no List.

Source: `lib/macrovalue/recognition.x:63`

Macro_close

```
Macro Macro_close(Macro value, List captures)
```

Records the Macro values an anonymous macro captured where it was created, so applying it later applies the same children.

Source: `lib/macrovalue/construction.x:7`

Macro_pattern

```
List Macro_pattern(Macro t, List names)
```

Derives the Match pattern that recognizes code this macro builds, capturing each parameter under the given binder.

Source: `lib/macrovalue/patterns.x:44`

`Macro`

Macro.declared

```
Var Macro.declared(Var value)
```

Returns the name a typed quotation declares with a Name hole whose local holds `value`: the name form of an `x2c_ident` spelling or a String, or a binding as it is.

Source: `lib/macrovalue/construction.x:66`

Macro.inserted

```
Var Macro.inserted(Var value, int lifts, int expression)
```

Returns what a typed quotation inserts for one use of a hole whose local holds `value`, as a rebuild inserts it. An expression hole (`lifts`) takes a number, String, or Symbol as its literal. Where the code takes an expression (`expression`), a binding, an `x2c_ident` spelling, or a Name hole's String becomes an identifier expression; in a Name hole's member position, an `x2c_ident` spelling is its String.

Source: `lib/macrovalue/construction.x:53`

Macro.inserted_items

```
List Macro.inserted_items(List values)
```

Returns the items a typed quotation splices for an expression sequence hole whose local holds `values`: each number, String, or Symbol becomes its literal, as a spliced data List's items do.

Source: `lib/macrovalue/construction.x:74`

Macro.matcher

```
MacroMatcher Macro.matcher(Macro shape, List holes)
```

Prepares repeated recognition without retaining subject-specific bindings.

Source: `lib/macrovalue/recognition.x:72`

Macro.matches

```
int Macro.matches(Macro t, List code, List holes)
```

Tests complete code against a macro and its hole patterns. Repeated recognition can retain `Macro.matcher` so the pattern is derived once.

Source: `lib/macrovalue/recognition.x:97`

Macro.number_literal

```
List Macro.number_literal(List result, List type, Var value)
```

Returns the literal expression of type `result` that holds `value`, a number of the scalar type `type`. An `int` value is its decimal literal; another number is its exact bits cast to `type`.

Source: `lib/macrovalue/construction.x:137`

Macro.number_type

```
List Macro.number_type(Var value)
```

Returns the C type of a number's Var family, or NULL when `value` is not a number. An untyped integer is an `int` when it fits one.

Source: `lib/macrovalue/construction.x:111`

Macro.subject

```
Var Macro.subject(void)
```

Returns the table `Macro.use_subject` last set, or void.

Source: `lib/macrovalue/patterns.x:14`

Macro.typed

```
List Macro.typed(List type, Var value)
```

Returns what a typed quotation builds when its code is one hole whose local holds `value`: the inserted expression with the type `type`. A String typed `String` is a String literal.

Source: `lib/macrovalue/construction.x:79`

Macro.use_subject

```
void Macro.use_subject(Var rows)
```

Sets the `(SPELLING BINDING)` rows for global references and the `(source-spelling BINDING SPELLING)` rows for renamed local bindings in a compile-time call's syntax arguments. A macro value's free reference recognizes only the recorded global binding; with void it recognizes any binding of its spelling. The compiler sets these rows for each `meta` call and carries them through the helper.

Source: `lib/macrovalue/patterns.x:22`

MacroMatcher

MacroMatcher.matches

```
int MacroMatcher.matches(MacroMatcher &m, List code)
```

Tests code with the same source views and identity rules as a macro case.

Source: `lib/macrovalue/recognition.x:83`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`Macro.binder`	Returns the binder that captures one projection of the template hole whose binder is `binder`.

Macro

Macro.binder

```
Atom Macro.binder(Var binder, String projection, int sequence)
```

Returns the binder that captures one projection of the template hole whose binder is `binder`. Splice and construction projections are always sequences; return, declarator, and member projections never are; source, value, and expression follow the hole's `sequence`.

Source: `lib/macrovalue/patterns.x:109`

Public types

Type	Kind	Summary
MacroCaseSite	struct	Holds one macro-valued case site's prepared recognition for the process: the plan Match keeps, the slots of the macro's fixed locals, and where each binder reads its capture.
MacroFixedSlots	struct	Records fixed-local slots for distinct-identity checks and Name slots for member-spelling comparisons during recognition.
MacroMatcher	struct	Retains a macro and hole patterns for repeated recognition.
MacroPublishing	struct	Records where each of a case's binders reads its capture: the slot of its internal binder in the pattern that captured, and its own slot in the case, which need not share the parameters' order.

MacroCaseSite

```
typedef struct MacroCaseSite { MatchCaptureSite match; MacroFixedSlots policy; MacroPublishing route; int ready; } MacroCaseSite
```

Holds one macro-valued case site's prepared recognition for the process: the plan Match keeps, the slots of the macro's fixed locals, and where each binder reads its capture. The compiler emits one zero-initialized static site per case.

Source: lib/macrovalue/macrovalue.x:41

MacroFixedSlots

```
typedef struct MacroFixedSlots { int count, slots[MACHINE_BINDER_MAX]; int names, name_slots[MACHINE_BINDER_MAX]; } MacroFixedSlots
```

Records fixed-local slots for distinct-identity checks and Name slots for member-spelling comparisons during recognition.

Source: lib/macrovalue/macrovalue.x:20

MacroMatcher

```
typedef struct MacroMatcher { Macro shape; List holes, pattern; } MacroMatcher
```

Retains a macro and hole patterns for repeated recognition. A pattern independent of subject bindings is derived once; contextual references are resolved against each subject. The caller keeps the source values alive as long as this record. Match plans use the active Match cache.

Source: lib/macrovalue/macrovalue.x:108

MacroPublishing

```
typedef struct MacroPublishing { int from[MACHINE_BINDER_MAX], fallback[MACHINE_BINDER_MAX]; int to[MACHINE_BINDER_MAX]; int count, binders, complete; unsigned long definite; } MacroPublishing
```

Records where each of a case's binders reads its capture: the slot of its internal binder in the pattern that captured, and its own slot in the case, which need not share the parameters' order.

Source: lib/macrovalue/macrovalue.x:29

Design notes

A Macro is the canonical macrodef record of a definition. $name selects one and macro Kind(...) => ... creates one. Applying it returns a pending invocation the compiler expands and binds at the insertion site; naming it in a case derives a Match pattern from the same body. Generated code calls these operations by name, so the module is part of the prelude.

Tests and examples

make verify (macro-values, macro-sequence-case, and macro-slot-recognition compiler fixtures).

lib/map.x

Hash table mapping `Var` keys to `Var` values.

Primary API

Function	Summary
`Map.cleanup`	Releases this Map and both backing Blocks without freeing stored values.
`Map.contains`	Returns one when `key` is present, or zero for a null map.
`Map.copy`	Returns a fresh shallow copy in the current `Scope`.
`Map.enumerate`	Initializes `dest` to yield canonical `(key value)` `List`s in bucket order.
`Map.get_hashed`	Returns the value stored under `key`, or `void` when absent, probing with the caller's precomputed `key_hash`.
`Map.getdefault`	Returns the value for `key`, or `defval` without inserting it.
`Map.iter`	Initializes `dest` as a single-pass iterator over boxed values.
`Map.keys`	Initializes `dest` as a single-pass iterator over boxed keys.
`Map.len`	Returns the number of entries, or zero for a null map.
`Map.merge`	Copies every entry of `other` into `map` and returns the destination.
`Map.new`	Returns a fresh empty map owned by the current `Scope`.
`Map.postfixindex`	Applies postfix `++` or `--` and returns the previous value.
`Map.set`	Copies `key` and `val` into the table, replacing an existing value.
`Map.setdefault`	Returns the value for `key`, inserting `defval` when absent.
`Map.setindex`	Stores `val` under `key` and returns `val`.
`Map.truth`	Returns one when `map` contains an entry.
`Map.try_del`	Removes `key`, writes its value to `out`, and returns one when present.
`Map.try_get`	Writes the value for `key` to `out` and returns one when present.
`Map.try_next`	Writes the next bucket's key and value, advances `cursor`, and returns one.
`Map.updateindex`	Applies `op` to the value for `key` and returns the stored result.
`Map.write_str`	Appends the typed-Map display text in bucket order.

Map

Map.cleanup

```
meta native void Map.cleanup(Map value)
```

Releases this Map and both backing Blocks without freeing stored values.

Source: `lib/map.x:236`

Map.contains

```
int Map.contains(Map map, Var key)
```

Returns one when `key` is present, or zero for a null map.

Source: `lib/map.x:87`

Map.copy

```
Map Map.copy(Map map)
```

Returns a fresh shallow copy in the current `Scope`. The result has independent table storage but copies native key and value fields. A null input produces a fresh empty map.

Raises: an allocation, size, or invariant cause while copying.

Source: `lib/map.x:87`

Map.enumerate

```
Iter Map.enumerate(Map map, Iter dest)
```

Initializes `dest` to yield canonical `(key value)` `Lists` in bucket order. The caller owns `dest`, and the iterator borrows `map`; both must remain live during traversal. A null `dest` returns NULL, and a null map is exhausted. Each step boxes both fields and interns two `List` cells. Each cell belongs to the pool owning its canonical match, which may be an ancestor pool, or to the current pool when newly allocated. Structural mutation invalidates the iterator. Each MapLongDouble pull allocates its long key as a `Var` in the current `Scope`; the yielded `List` and anything retaining it must not outlive that `Scope`.

Raises: `<alloc-fail>` or `<size-limit>` while interning the `List`, and for MapLongDouble, `<alloc-fail>` or `<bad-enc>` while boxing its key. Shared causes do not return from the pull.

Source: `lib/map.x:231`

Map.get_hashed

```
meta native Var Map.get_hashed(Map map, Var key, unsigned key_hash)
```

Returns the value stored under `key`, or `void` when absent, probing with the caller's precomputed `key_hash`. `key_hash` must be `Var.hash` of `key`; another value reports the key as absent. This serves a caller that probes several `Maps` with one key, and is `Map.get` in every other respect. A null `Map` reports absence.

Raises: a cause raised by custom key equality. Hashing happens in the caller, so a `void` key raises there instead.

Source: `lib/map.x:148`

Map.getdefault

```
Var Map.getdefault(Map map, Var key, Var defval)
```

Returns the value for `key`, or `defval` without inserting it. A null map is treated as an empty one.

Source: `lib/map.x:87`

Map.iter

```
Iter Map.iter(Map map, Iter dest)
```

Initializes `dest` as a single-pass iterator over boxed values. The caller owns `dest`, and the iterator borrows `map`; both must remain live during traversal. A null `dest` returns NULL. Values follow bucket order, a null map is exhausted, and structural mutation invalidates the iterator.

Source: `lib/map.x:231`

Map.keys

```
Iter Map.keys(Map map, Iter dest)
```

Initializes `dest` as a single-pass iterator over boxed keys. The caller owns `dest`, and the iterator borrows `map`; both must remain live during traversal. A null `dest` returns NULL. Keys follow bucket order, a null map is exhausted, and structural mutation invalidates the iterator. MapIntInt and both String-keyed families box keys without allocation. Each MapLongDouble pull allocates its long key as a `Var` in the current `Scope`; the yielded `Var` and anything retaining it must not outlive that `Scope`.

Raises: MapLongDouble iteration may raise `<alloc-fail>` or `<bad-enc>` while boxing a key.

Source: `lib/map.x:231`

Map.len

```
unsigned Map.len(Map map)
```

Returns the number of entries, or zero for a null map.

Source: `lib/map.x:87`

Map.merge

```
Map Map.merge(Map map, Map other)
```

Copies every entry of `other` into `map` and returns the destination. `other` wins conflicts. A null `other` is a no-op; a null destination is replaced by a fresh map. When a supplied destination fails partway, entries already copied remain; a newly created destination is discarded.

Raises: an allocation, size, or invariant cause while inserting.

Source: `lib/map.x:87`

Map.new

```
Map Map.new(void)
```

Returns a fresh empty map owned by the current `Scope`. The initial table has two buckets and grows automatically.

Raises: `<alloc-fail>` or `<size-limit>` when storage cannot be created.

Source: `lib/map.x:87`

Map.postfixindex

```
Var Map.postfixindex(Map map, Var key, Symbol op)
```

Applies postfix `++` or `--` and returns the previous value. The key must already exist. A failing value operation leaves the entry unchanged.

Raises: `<bad-arg>` for a null map or absent key, `<void-op>` for a void key, or a cause from `Var.postfix` for an unsupported value or operation.

Source: `lib/map.x:128`

Map.set

```
void Map.set(Map map, Var key, Var val)
```

Copies `key` and `val` into the table, replacing an existing value. Insertion invalidates live cursors and iterators; replacement does not.

Raises: `<bad-arg>` for a null map, or an allocation, size, or invariant cause while inserting. Allocation and size failures leave the table unchanged.

Source: `lib/map.x:87`

Map.setdefault

```
Var Map.setdefault(Map map, Var key, Var defval)
```

Returns the value for `key`, inserting `defval` when absent. Insertion copies the native key and value fields and invalidates live cursors and iterators; an existing-key read does not mutate the table.

Raises: `<bad-arg>` for a null map, or an allocation, size, or invariant cause while inserting. Allocation and size failures leave the table unchanged.

Source: `lib/map.x:87`

Map.setindex

```
Var Map.setindex(Map map, Var key, Var val)
```

Stores `val` under `key` and returns `val`. This is the bracket-facing form of `Map.set`, with the same cursor invalidation and failure behavior.

Source: `lib/map.x:87`

Map.truth

```
int Map.truth(Map map)
```

Returns one when `map` contains an entry.

Source: `lib/map.x:87`

Map.try_del

```
int Map.try_del(Map map, Var key, Var &?out)
```

Removes `key`, writes its value to `out`, and returns one when present. A null map, null `out`, or absent key returns zero without writing or mutation. Successful removal invalidates live cursors and iterators.

Source: `lib/map.x:87`

Map.try_get

```
int Map.try_get(Map map, Var key, Var &?out)
```

Writes the value for `key` to `out` and returns one when present. A null map, null `out`, or absent key returns zero without writing.

Source: `lib/map.x:87`

Map.try_next

```
int Map.try_next(Map map, unsigned &?cursor, Var &?key, Var &?val)
```

Writes the next bucket's key and value, advances `cursor`, and returns one. Initialize the caller-owned cursor to zero. A null argument or exhaustion returns zero without changing `key` or `val`. Traversal follows bucket order, not insertion order, and structural mutation invalidates the cursor.

Source: `lib/map.x:87`

Map.updateindex

```
Var Map.updateindex(Map map, Var key, Symbol op, Var rhs)
```

Applies `op` to the value for `key` and returns the stored result. `+` inserts a numeric `rhs` when the key is absent; other updates require an existing key. A failing value operation leaves the entry unchanged.

Raises: `<bad-arg>` for a null map or missing required key, `<void-op>` for a void key or value, a cause from `Var.update`, or an allocation, size, or invariant cause while inserting.

Source: `lib/map.x:105`

Map.write_str

```
Buffer Map.write_str(Map map, Buffer out)
```

Appends the typed-Map display text in bucket order.

Source: `lib/map.x:230`

Advanced and interop API

Function	Summary
`Map.compare`	Compares typed Maps by size and then sorted native key/value contents.
`Map.equal`	Reports key/value equality independent of bucket and insertion order.
`Map.repr`	Returns the readable typed-Map representation.
`Map.str`	Returns the typed-Map display String.
`Map.update_n`	Adds exactly `pair_count` key/value pairs to `map` in argument order.

Function	Summary
Map.write_repr	Appends the readable typed-Map representation in bucket order.

Map

Map.compare

```
int Map.compare(Map a, Map b)
```

Compares typed Maps by size and then sorted native key/value contents. Identical handles compare equal and NULL sorts first. Neither Map is mutated. Raises: <alloc-fail> while creating temporary storage, or any cause from key or value comparison.

Source: lib/map.x:230

Map.equal

```
int Map.equal(Map a, Map b)
```

Reports key/value equality independent of bucket and insertion order. Two null maps compare equal; a null and a nonnull map compare unequal.

Source: lib/map.x:87

Map.repr

```
String Map.repr(Map map)
```

Returns the readable typed-Map representation.

Source: lib/map.x:230

Map.str

```
String Map.str(Map map)
```

Returns the typed-Map display String.

Source: lib/map.x:230

Map.update_n

```
Self Map.update_n(Self map, unsigned pair_count, ...)
```

Adds exactly pair_count key/value pairs to map in argument order. Arguments alternate Var keys and values. A null Map returns NULL without reading them. Each completed pair remains if a later pair fails. Inserting a new key invalidates active traversal; replacing an existing value does not.

Raises: the same causes as Map.set.

Source: lib/map.x:161

Map.write_repr

```
Buffer Map.write_repr(Map map, Buffer out)
```

Appends the readable typed-Map representation in bucket order.

Source: lib/map.x:230

Convenience API

Function	Summary
Map.del	Removes key and returns its value.
Map.get	Returns the value for key.

Function	Summary
Map.getindex	Returns the value selected by bracket indexing.

Map

Map.del

```
Var Map.del(Map map, Var key)
```

Removes key and returns its value. Successful removal invalidates live cursors and iterators. Prefer try_del when absence is an ordinary outcome. Boxed families return void for absence; native families raise <bad-arg>.

Source: lib/map.x:87

Map.get

```
Var Map.get(Map map, Var key)
```

Returns the value for key. Prefer try_get when absence is an ordinary outcome. Boxed families return void for absence; native families raise <bad-arg>.

Source: lib/map.x:87

Map.getindex

```
Var Map.getindex(Map map, Var key)
```

Returns the value selected by bracket indexing. This is the bracket-facing form of Map.get, with the same failure behavior. Prefer bracket indexing in ordinary code.

Source: lib/map.x:87

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
Map.export_to	Exports every key and value, rebuilds the table, then moves its Blocks.
Map.new_capacity	Returns a fresh empty map with exactly capacity buckets.

Map

Map.export_to

```
void Map.export_to( Map map, Context source, VarExportContextFn export_value, Scope *scope)
```

Exports every key and value, rebuilds the table, then moves its Blocks. The borrowed export_value callback runs synchronously for each stored key and value and may return replacement Vars. Rebuilding is required because an exported key may hash differently. On success the old backing Blocks are freed, the rebuilt Blocks move into scope, and the Map identity remains unchanged. Context moves that identity before this call to break cycles.

A failure while staging leaves the Map's original records, backing storage, and recorded Scope unchanged. Effects of callbacks that already completed, including nested exports, are not rolled back. Any failure while moving the rebuilt Blocks occurs after the table has replaced the original. A null Map, callback, or Scope pointer does nothing. Raises: <alloc-fail>, <size-limit>, or <invariant> while rebuilding, or any cause from export, key hashing, or moving the rebuilt Blocks.

Source: lib/map.x:189

Map.new_capacity

```
Map Map.new_capacity(unsigned capacity)
```

Returns a fresh empty map with exactly `capacity` buckets. The result and its backing arrays belong to the current `Scope`. `capacity` must be a power of two of at least two.

Raises: `<bad-arg>` for another capacity, or `<alloc-fail>` / `<size-limit>` when storage cannot be created.

Source: `lib/map.x:87`

Public types

Type	Kind	Summary
Map	struct	`Scope`-backed mutable hash table from `Var` keys to `Var` values.

Map

```
typedef struct Map { Scope scope, Bytes hashes, entries; unsigned used; unsigned capacity; unsigned mask; } *Map
```

`Scope`-backed mutable hash table from `Var` keys to `Var` values. The `Map` records the `Scope` used for its two backing `Blocks`. `Context` moves the `Map` allocation itself before recursively exporting those `Blocks`. Stored `Var` bits are shallow and retain no pointee; those values must remain valid while the `Map` can read, compare, hash, or return them. A valid empty `Map` is allocated and distinct from NULL.

Source: `lib/map.x:30`

Design notes

`Map` uses Robin Hood hashing split across two scope-owned `Blocks`. The hash `Block` holds one 32-bit hash per bucket, zero for an empty bucket; the entry `Block` holds the `Var` key and value at the same bucket index. Lookups probe the compact hash `Block` and touch an entry only when its hash matches. Capacity is always a power of two and probe distances are computed as needed.

Deletion back-shifts hashes and entries together to close holes. Structural mutation invalidates traversal state. Keys and values are stored as `Var` bits; the `Map` does not free pointer-bearing payloads or canonical values reachable through them.

Tests and examples

`make verify` (`unittest/test-map.x`) and `make examples` (`docs-word-count`).

lib/match-cache.x

Caches of prepared Match plans.

Primary API

Function	Summary
x2c_match_thread_release	Disposes this thread's default `Match` plan cache.
MatchCache.context_close	Disposes and removes the top `Context`'s default `Match` cache.
MatchCache.context_open	Opens one `Context`-local default `Match`-cache state.
MatchLease.plan	Borrows the acquired program until this active lease is released.

Functions

x2c_match_thread_release

```
void x2c_match_thread_release(void)
```

Disposes this thread's default `Match` plan cache. `x2c_thread_state_release` calls it before `Scope` releases the `Scope` that holds that cache; a thread that never matched has no cache and nothing happens. All default-cache leases and `Context` states must already be closed.

Raises: `<bad-state>` when a lease remains active.

Source: `lib/match-cache.x:520`

MatchCache

MatchCache.context_close

```
void MatchCache.context_close(void *token)
```

Disposes and removes the top `Context`'s default `Match` cache. A null token is ignored. Successful close restores the previous default; the token storage remains owned by its `Context` `Scope`.

Raises: `<bad-state>` when `token` is not the top state or its cache has an active lease. The failure leaves the state installed.

Source: `lib/match-cache.x:503`

MatchCache.context_open

```
void *MatchCache.context_open(void)
```

Opens one `Context`-local default `Match`-cache state. The returned opaque token becomes the top of a thread-local LIFO stack; its cache is created only on first use. The token is allocated in the active `Scope` and must be passed to `MatchCache.context_close` before that `Scope` ends.

Raises: `<alloc-fail>` when the state cannot be allocated.

Source: `lib/match-cache.x:489`

MatchLease

MatchLease.plan

```
MatchPlan MatchLease.plan(MatchLease *m)
```

Borrows the acquired program until this active lease is released.

Source: `lib/match-cache.x:316`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`MatchCache.acquire`	Acquires a lease for a cached prepared pattern.
`MatchCache.current`	Returns the active default `Match` cache, creating it on first use.
`MatchCache.dispose`	Destroys a `Match` cache with no active leases.
`MatchCache.flush_default`	Destroys the active `Context`-local or thread-local `Match` cache.
`MatchCache.new`	Creates a `MatchCache` retaining up to `capacity` prepared patterns.
`MatchCache.search`	Searches through `cache`, writing every match.
`MatchCache.search_replace`	Replaces every match through `cache` from the leaves upward.
`MatchCache.try_capture`	Matches through `cache` into caller-owned positional storage.
`MatchCache.try_match`	Matches through `cache`, writing bindings on success.
`MatchCache.try_match_replace`	`Match`-replaces through `cache`, writing the replacement on success.
`MatchCache.try_search`	Searches through `cache`, writing the first match and bindings.
`MatchLease.release`	Releases the prepared program held by `lease`.

MatchCache

MatchCache.acquire

```
Symbol MatchCache.acquire( MatchCache m, Var pattern, MatchLease &lease, const char *owner)
```

Acquires a lease for a cached prepared pattern. `cache` and `lease` must be nonnull, and `owner` names the operation a fence diagnostic should report. The lease is initialized on every returning path. An admitted pattern reuses or creates an LRU entry; an inadmissible pattern gets a transient plan owned by the lease. Returns the plan's status symbol, or `<pressure>` when every entry is pinned. Release the lease after any returned status; releasing the inactive pressure lease is a no-op.

Raises: `<size-limit>` when `pattern` exceeds a lowering limit. Such a pattern compiles to no program, so it is never cached and never leased. `<alloc-fail>` may also be raised while preparing or growing storage.

Source: `lib/match-cache.x:87`

MatchCache.current

```
MatchCache MatchCache.current(void)
```

Returns the active default `Match` cache, creating it on first use.

Raises: `<alloc-fail>` when the cache cannot be created.

Source: `lib/match-cache.x:456`

MatchCache.dispose

```
void MatchCache.dispose(MatchCache cache)
```

Destroys a `Match` cache with no active leases. A null cache is ignored. Disposal frees all plans and cache storage and invalidates every alias.

Raises: `<bad-state>` when a lease remains active. The failure leaves the cache intact.

Source: `lib/match-cache.x:565`

MatchCache.flush_default

```
void MatchCache.flush_default(void)
```

Destroys the active `Context`-local or thread-local `Match` cache. A missing cache is ignored; the next `Match` recreates it lazily. Static compiler capture sites are unaffected.

Raises: `<bad-state>` when a lease remains active. The failure leaves the cache installed.

Source: `lib/match-cache.x:475`

MatchCache.new

```
MatchCache MatchCache.new(int capacity)
```

Creates a `MatchCache` retaining up to `capacity` prepared patterns. The returned cache owns a named `Scope` and is not synchronized. It borrows admitted pattern identities, so dispose it before their owning canonical pools. `MatchCache.dispose` is required after every lease is released.

Raises: `<bad-arg>` when capacity is not positive, `<size-limit>` when its storage dimensions cannot be represented, and `<alloc-fail>` when cache storage cannot be allocated.

Source: `lib/match-cache.x:537`

MatchCache.search

```
int MatchCache.search( MatchCache cache, List input, Var pattern, List &out_results, const char *owner)
```

Searches through `cache`, writing every match. `out_results` must be nonnull. It receives the reverse-visitation result `List`, or `nil` when there are no matches, the pattern is malformed, cache pressure prevents execution, or the machine fails. Returns 1 exactly when that `List` is nonempty.

Raises: `<size-limit>` for an ineligible pattern, or `<alloc-fail>` while preparing or constructing results.

Source: `lib/match-cache.x:396`

MatchCache.search_replace

```
int MatchCache.search_replace( MatchCache cache, List input, Var pattern, Var template, List &out, const char *owner)
```

Replaces every match through `cache` from the leaves upward. `out` must be nonnull. A completed prepared traversal returns 1 and writes its result even when nothing matched. A malformed pattern, cache pressure, or machine error returns 0 and writes `input` unchanged.

Raises: `<size-limit>` for an ineligible pattern, or `<alloc-fail>` while preparing, traversing, or replacing.

Source: `lib/match-cache.x:427`

MatchCache.try_capture

```
int MatchCache.try_capture( MatchCache cache, List input, Var pattern, MatchCaptureBuffer &? captures, const char *owner)
```

Matches through `cache` into caller-owned positional storage. Returns 1 only after atomically committing a valid buffer. A miss, malformed pattern, invalid buffer, cache pressure, or machine error returns 0 and leaves it unchanged.

Raises: `<size-limit>` for an ineligible pattern, or `<alloc-fail>` while preparing or matching.

Source: `lib/match-cache.x:353`

MatchCache.try_match

```
int MatchCache.try_match( MatchCache cache, List input, Var pattern, List &?out_bindings, const char *owner)
```

Matches through `cache`, writing bindings on success. `out_bindings` must be nonnull. Returns 0 and leaves it unchanged for a miss, malformed pattern, cache pressure, or machine error. Successful binding shape and order follow `MatchPlan.execute`.

Raises: `<size-limit>` for an ineligible pattern, or `<alloc-fail>` while preparing, materializing, or publishing.

Source: `lib/match-cache.x:367`

MatchCache.try_match_replace

```
int MatchCache.try_match_replace( MatchCache cache, List input, Var pattern, Var template, Var &? out, const char *owner)
```

`Match`-replaces through `cache`, writing the replacement on success. `out` must be nonnull. Returns 0 and leaves it unchanged on a miss, malformed pattern, cache pressure, or machine error. The successful result may be any `Var`.

Raises: `<size-limit>` for an ineligible pattern, or `<alloc-fail>` while preparing, materializing, or replacing.

Source: `lib/match-cache.x:413`

MatchCache.try_search

```
int MatchCache.try_search( MatchCache cache, List input, Var pattern, Var &?out_match, List &? out_bindings, const char *owner)
```

Searches through `cache`, writing the first match and bindings. Both outputs must be nonnull. Returns 0 and leaves them unchanged on a miss, malformed pattern, cache pressure, or machine error. Traversal order follows `MatchPlan.try_search`.

Raises: `<size-limit>` for an ineligible pattern, or `<alloc-fail>` while preparing or constructing bindings.

Source: `lib/match-cache.x:381`

MatchLease

MatchLease.release

`void MatchLease.release(MatchLease *lease)`

Releases the prepared program held by `lease`. Releasing an inactive lease has no effect. A successful release destroys a transient plan or unpins its cached entry and makes the lease inactive.

Raises: `<bad-arg>` when lease is NULL and `<bad-state>` when its cache or entry state is inconsistent. The failure leaves the lease active.

Source: `lib/match-cache.x:292`

Public types

Type	Kind	Summary
`MatchCache`	struct	Names an explicit cache of immutable prepared Match plans.
`MatchLease`	struct	Represents one acquired use of a cached or transient Match plan.

MatchCache

`typedef struct MatchCache *MatchCache`

Names an explicit cache of immutable prepared Match plans. A cache is not synchronized. Its caller must serialize access, keep every admitted pattern value alive until disposal, and dispose it when no lease remains active.

Source: `lib/match-cache.x:21`

MatchLease

`typedef struct MatchLease { MatchCache cache; MatchPlan transient_plan; unsigned long generation; int slot, active; } MatchLease`

Represents one acquired use of a cached or transient Match plan. A cached lease pins its entry; a transient lease owns its plan. Initialize it only through `MatchCache.acquire` and call `MatchLease.release` on every exit after acquisition returns, including a pressure result.

Source: `lib/match-cache.x:28`

Design notes

A MatchCache keeps immutable prepared plans under the canonical identity of their patterns and lends them through leases. The `List.match` family runs through the active default cache, which is Context-local while a Context is open and otherwise thread-local.

Tests and examples

`make verify` (`unittest/test-match-cache.x`).

lib/match-plan.x

Lowering Match patterns to prepared plans.

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`MatchPlan.free`	Releases resources owned by `plan`.
`MatchPlan.prepare`	Compiles `pattern` into a reusable immutable `MatchPlan`.

MatchPlan

MatchPlan.free

```
void MatchPlan.free(MatchPlan plan)
```

Releases resources owned by `plan`. A null plan is ignored; the plan, layout, program, and all aliases to them are invalid afterward. Borrowed pattern constants are not released.

Source: `lib/match-plan.x:79`

MatchPlan.prepare

```
MatchPlan MatchPlan.prepare(Var pattern)
```

Compiles `pattern` into a reusable immutable `MatchPlan`. The caller owns the returned plan in the active `Scope`. Preparation records `<prepared>`, `<malformed>`, or `<ineligible>` in the plan and raises for none of them; only a prepared plan has a program. `reason` is a borrowed static category string. The plan borrows pattern constants, which must outlive it.

Raises: `<alloc-fail>` while analyzing, lowering, or freezing.

Source: `lib/match-plan.x:46`

Design notes

A MatchPlan is the prepared form of one pattern: its capture layout and a program in the wordcode of `machine.x`. A plan owns both and borrows the canonical values in its pattern. Preparation records why a pattern has no program and raises for none of them; `match.x` runs the plan.

lib/match/match.x

Pattern matching and transformation utilities for lists.

Primary API

Function	Summary
x2c_match_initialize	Registers process-wide `Match` cleanup exactly once.
x2c_match_site_match	Returns bindings through one compiler-owned site, or `nil` on a miss.
x2c_match_site_match_replace	Returns the `List` replacement through one compiler-owned site.
x2c_match_site_search	Returns every matching subtree through one compiler-owned site.
x2c_match_site_search_replace	Replaces every match through one compiler-owned site.
x2c_match_site_try_match	Matches through one compiler-owned site, writing bindings on success.
x2c_match_site_try_match_replace	`Match`-replaces through one compiler-owned site.
x2c_match_site_try_search	Searches through one compiler-owned site, writing the first match.
List.replace	Replaces named binders in `template` according to `bindings`.
List.search	Returns every matching subtree of `input` with its bindings.
List.search_replace	Replaces every matching subtree in `input` from the leaves upward.
List.try_match	Matches `input` against `pat`, writing bindings on success.
List.try_match_replace	Matches `input` and writes the instantiated `template` on success.
List.try_search	Searches `input` for `pat`, writing the first match and bindings.
MatchPlan.admits	Reports whether `input`, seen through `view` when one is given, begins as one of the keys of prepared `plan` requires.

Functions

x2c_match_initialize

```
void x2c_match_initialize(void)
```

Registers process-wide `Match` cleanup exactly once. Repeated calls have no effect. Failure of the native once primitive writes a diagnostic and aborts the process.

Raises: `<alloc-fail>` or `<size-limit>` while registering the shutdown hook.

Source: `lib/match/sites.x:275`

x2c_match_site_match

```
List x2c_match_site_match(MatchCaptureSite *site, List input, Var pat)
```

Returns bindings through one compiler-owned site, or `nil` on a miss. Results follow `List.match`.

Source: `lib/match/sites.x:201`

x2c_match_site_match_replace

```
List x2c_match_site_match_replace( MatchCaptureSite *site, List input, Var pat, Var template)
```

Returns the `List` replacement through one compiler-owned site. Results follow `List.match_replace`.

Source: `lib/match/sites.x:244`

x2c_match_site_search

```
List x2c_match_site_search(MatchCaptureSite *site, List input, Var pat)
```

Returns every matching subtree through one compiler-owned site. Results follow `List.search`.

Source: `lib/match/sites.x:222`

x2c_match_site_search_replace

```
List x2c_match_site_search_replace( MatchCaptureSite *site, List input, Var pat, Var template)
```

Replaces every match through one compiler-owned site. Results follow `List.search_replace`.

Source: `lib/match/sites.x:255`

x2c_match_site_try_match

```
int x2c_match_site_try_match( MatchCaptureSite *site, List input, Var pat, List *out_bindings)
```

Matches through one compiler-owned site, writing bindings on success. Results follow `List.try_match`.

Source: `lib/match/sites.x:190`

x2c_match_site_try_match_replace

```
int x2c_match_site_try_match_replace( MatchCaptureSite *site, List input, Var pat, Var template,
Var *out)
```

`Match`-replaces through one compiler-owned site. Results follow `List.try_match_replace`.

Source: `lib/match/sites.x:233`

x2c_match_site_try_search

```
int x2c_match_site_try_search( MatchCaptureSite *site, List input, Var pat, Var *out_match, List
*out_bindings)
```

Searches through one compiler-owned site, writing the first match. Results follow `List.try_search`.

Source: `lib/match/sites.x:210`

`List`

List.replace

```
meta native List List.replace(List template, List bindings)
```

Replaces named binders in `template` according to `bindings`. A sequence binder in list-head position splices its captured `List`; `!quote` removes itself and leaves its operand literal. Missing binders are retained. A null template returns `nil`, and null bindings return `template` unchanged. New structure follows the module pool-chain lifetime above.

Raises: `<alloc-fail>` while constructing replacement `List`s.

Source: `lib/match/templates.x:72`

List.search

```
List List.search(List input, Var pat)
```

Returns every matching subtree of `input` with its bindings. Each result begins with `(* matched)` followed by reverse-slot-order binder pairs. Traversal visits a `List`'s head, then tail, then the `List` itself; results are prepended and therefore returned in reverse visitation order. Explicit `nil` values are nodes, but a proper `List`'s terminal cdr is not. A miss, malformed pattern, or machine error returns `nil`.

Raises: `<size-limit>` for an ineligible pattern, or `<alloc-fail>` while preparing or constructing results.

Source: `lib/match/operations.x:78`

List.search_replace

```
List List.search_replace(List input, Var pat, Var template)
```

Replaces every matching subtree in `input` from the leaves upward. Children are rewritten before their reconstructed containing `List` is tested. A miss, malformed pattern, or machine error returns `input` unchanged. New structure follows the module pool-chain lifetime above.

Raises: `<size-limit>` for an ineligible pattern, or `<alloc-fail>` while preparing, traversing, or replacing.

Source: `lib/match/operations.x:102`

List.try_match

```
int List.try_match(List input, Var pat, List &?out_bindings)
```

Matches `input` against `pat`, writing bindings on success. Returns 1 on a match and writes a reverse-slot-order association `List`, or returns 0 and leaves `out_bindings` unchanged. A successful binder-free match writes `nil`. A null output pointer returns 0.

Raises: `<size-limit>` for an ineligible pattern, or `<alloc-fail>` while preparing, materializing captures, or publishing bindings.

Source: `lib/match/operations.x:32`

List.try_match_replace

```
int List.try_match_replace(List input, Var pat, Var template, Var &?out)
```

Matches `input` and writes the instantiated `template` on success. The output may be any `Var`, including typed `nil` or a scalar. A template that is one binder the match left unbound, such as the binder of an `!or` alternative another alternative satisfied, writes that binder. Returns 0 for a miss, malformed pattern, invalid output, or machine error and leaves `out` unchanged.

Raises: `<size-limit>` for an ineligible pattern, or `<alloc-fail>` while preparing, materializing, or replacing.

Source: `lib/match/operations.x:54`

List.try_search

```
int List.try_search( List input, Var pat, Var &?out_match, List &?out_bindings)
```

Searches `input` for `pat`, writing the first match and bindings. The depth-first order is head, tail, then containing `List`, with the same explicit-`nil` rule as `List.search`. Returns 1 on success; otherwise returns 0 and leaves both outputs unchanged. Either null output returns 0.

Raises: the same causes as `List.search`.

Source: `lib/match/operations.x:90`

MatchPlan

MatchPlan.admits

```
int MatchPlan.admits(MatchPlan plan, Var input, Var (*view)(Var))
```

Reports whether `input`, seen through `view` when one is given, begins as one of the keys of prepared `plan` requires. A plan without keys admits every input, and a refused input is one the plan cannot match.

Source: `lib/match/execution.x:78`

Advanced and interop API

Function	Summary
`x2c_match_pattern_retainable`	Reports whether a compiler-owned site can retain `pattern`.
`x2c_match_site_prepare`	Returns the process-lifetime plan for one compiler-owned site.
`x2c_match_site_try_capture`	Matches through one compiler-owned static capture site.
`x2c_match_try_capture`	Matches a runtime pattern into positional storage.
`List.match`	Returns bindings when `input` matches `pat`, or `nil` on a miss.
`List.match_replace`	Returns the `List` replacement when `input` matches `pat`.
`Var.is_atom_binder`	Reports whether `atom` is a valid named or anonymous `?` binder.
`Var.is_binder`	Reports whether `atom` is either valid `Match` binder form.
`Var.is_list_binder`	Reports whether `atom` is a valid named or anonymous * binder.
`Var.is_match_op`	Reports whether `atom` is a compact built-in `Match` guard operator.

Functions

x2c_match_pattern_retainable

```
int x2c_match_pattern_retainable(Var pattern)
```

Reports whether a compiler-owned site can retain `pattern`. A site borrows its pattern's values for the life of the process, so only a graph of values that outlives every call qualifies.

Source: `lib/match/sites.x:162`

x2c_match_site_prepare

```
MatchPlan x2c_match_site_prepare(MatchCaptureSite *site, Var pattern)
```

Returns the process-lifetime plan for one compiler-owned site. The first retainable pattern binds the site permanently. A pattern the site cannot retain returns NULL; an ineligible one returns its fenced plan so the caller can name the fence.

Raises: `<alloc-fail>` while publishing.

Source: `lib/match/sites.x:171`

x2c_match_site_try_capture

```
int x2c_match_site_try_capture( MatchCaptureSite *site, List input, Var pattern, MatchCaptureBuffer
*captures)
```

Matches through one compiler-owned static capture site. Its first admissible pattern permanently binds the site; direct C callers must not reuse one site for different patterns. `site` must be zero-initialized static storage and `pattern` must contain only values that remain live through `Match` shutdown. A pattern the site cannot retain takes the ordinary runtime route, with the same result, and the site records that refusal once, so later calls take that route directly. Returns 1 only after atomically committing `captures`; invalid arguments, malformed patterns, misses, and machine errors return 0 without changing it.

Raises: `<size-limit>` for an ineligible pattern, or `<alloc-fail>` while publishing or matching.

Source: `lib/match/sites.x:80`

x2c_match_try_capture

```
int x2c_match_try_capture( List input, Var pattern, MatchCaptureBuffer *captures)
```

Matches a runtime pattern into positional storage. The active default cache prepares the pattern on its first use and reuses that program afterward. Returns 1 on success and 0 on a miss, malformed pattern, invalid buffer, cache pressure, or machine error. A nonnull buffer is written atomically as described by `MatchCaptureBuffer`; NULL returns 0.

Raises: `<size-limit>` for an ineligible pattern, or `<alloc-fail>` while preparing or matching.

Source: `lib/match/operations.x:19`

List

List.match

```
List List.match(List input, Var pat)
```

Returns bindings when `input` matches `pat`, or `nil` on a miss. A binder-free success returns the nonnull `%(())` sentinel with no associations. Binding order and failures follow `List.try_match`.

Source: `lib/match/operations.x:39`

List.match_replace

```
meta native List List.match_replace(List input, Var pat, Var template)
```

Returns the `List` replacement when `input` matches `pat`. A miss returns `input` unchanged. A successful scalar replacement cannot inhabit the `List` result and returns `nil`. Matching and replacement failures follow `List.try_match_replace`.

Source: `lib/match/operations.x:63`

Var

Var.is_atom_binder

```
meta native int Var.is_atom_binder(Var atom)
```

Reports whether `atom` is a valid named or anonymous `?` binder.

Source: `lib/match/layout.x:32`

Var.is_binder

```
meta native int Var.is_binder(Var atom)
```

Reports whether `atom` is either valid `Match` binder form.

Source: `lib/match/layout.x:38`

Var.is_list_binder

```
meta native int Var.is_list_binder(Var atom)
```

Reports whether `atom` is a valid named or anonymous `*` binder.

Source: `lib/match/layout.x:35`

Var.is_match_op

```
meta native int Var.is_match_op(Var atom)
```

Reports whether `atom` is a compact built-in `Match` guard operator.

Source: `lib/match/layout.x:41`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`MatchCaptureBuffer.has`	Reports whether a committed capture slot is present.
`MatchCaptureLayout.analyze`	Analyzes one `Match` pattern into its canonical positional layout.
`MatchCaptureLayout.definite_list`	Returns definite binders in canonical positional order.
`MatchCaptureLayout.free`	Releases one canonical `Match` capture layout.
`MatchCaptureLayout.index`	Returns the canonical slot for `binder`, or −1 when it is absent.

Function	Summary
MatchCaptureLayout.possible_list	Returns possible binders in canonical positional order.
MatchPlan.borrowable	Reports whether a plan may borrow pattern by its canonical identity.
MatchPlan.execute	Executes plan against input and publishes association bindings.
MatchPlan.execute_capture	Executes a prepared plan into caller-owned positional storage.
MatchPlan.raise_ineligible	Raises the fence an ineligible plan crossed, naming owner.
MatchPlan.search	Writes all matches of plan within input to out_results.
MatchPlan.search_replace	Replaces every match of plan from the leaves upward.
MatchPlan.try_capture	Executes a prepared List match into caller-owned positional storage.
MatchPlan.try_match	Executes prepared plan against input, writing bindings on success.
MatchPlan.try_match_replace	Executes plan and writes the instantiated template on success.
MatchPlan.try_search	Searches input with plan, writing the first match and bindings.

MatchCaptureBuffer

MatchCaptureBuffer.has

```
int MatchCaptureBuffer.has(MatchCaptureBuffer *m, int index)
```

Reports whether a committed capture slot is present. A null buffer or an index outside its capacity or Match's binder limit returns false. Presence is independent of the captured Var value.

Source: lib/match/layout.x:334

MatchCaptureLayout

MatchCaptureLayout.analyze

```
MatchCaptureLayout MatchCaptureLayout.analyze(Var pattern)
```

Analyzes one Match pattern into its canonical positional layout. Distinct named binders receive slots in lexical preorder. !quote is opaque; alternatives contribute possible binders, while only binders in every alternative are definite. The caller owns the returned layout, including when status is <malformed>.

Raises: <alloc-fail> while normalizing or allocating the layout.

Source: lib/match/layout.x:130

MatchCaptureLayout.definite_list

```
List MatchCaptureLayout.definite_list(MatchCaptureLayout layout)
```

Returns definite binders in canonical positional order. A null layout or no definite binders returns nil. The canonical result follows the module pool-chain lifetime above.

Raises: <alloc-fail> while constructing the List.

Source: lib/match/layout.x:304

MatchCaptureLayout.free

```
void MatchCaptureLayout.free(MatchCaptureLayout layout)
```

Releases one canonical Match capture layout. A null layout is ignored; every alias is invalid afterward.

Source: lib/match/layout.x:295

MatchCaptureLayout.index

```
int MatchCaptureLayout.index(MatchCaptureLayout layout, Atom binder)
```

Returns the canonical slot for `binder`, or −1 when it is absent. A null layout returns −1. Comparison uses exact `Atom` identity.

Source: `lib/match/layout.x:327`

MatchCaptureLayout.possible_list

```
List MatchCaptureLayout.possible_list(MatchCaptureLayout layout)
```

Returns possible binders in canonical positional order. A null layout or no possible binders returns `nil`. The canonical result follows the module pool-chain lifetime above.

Raises: `<alloc-fail>` while constructing the `List`.

Source: `lib/match/layout.x:312`

MatchPlan

MatchPlan.borrowable

```
int MatchPlan.borrowable(Var pattern, int permanent_lists)
```

Reports whether a plan may borrow `pattern` by its canonical identity. `permanent_lists` requires every `List` in it to belong to the outermost pool, as a plan kept for the life of the process does. Without it, a `List` that is canonical now qualifies until its pool level is released.

Source: `lib/match/sites.x:49`

MatchPlan.execute

```
int MatchPlan.execute( MatchPlan plan, Var input, List &?out_bindings, MachineStats &?stats)
```

Executes `plan` against `input` and publishes association bindings. Returns 1 and writes `out_bindings` on a match, 0 on a miss, and −1 for a null or malformed plan, null output, or machine error. Failure leaves the output unchanged. A binder-free success writes `nil`; other bindings are in reverse canonical slot order. `stats`, when nonnull, receives increments and is not initialized here.

Raises: `<size-limit>` for an ineligible plan, or `<alloc-fail>` while materializing or publishing bindings.

Source: `lib/match/execution.x:158`

MatchPlan.execute_capture

```
int MatchPlan.execute_capture( MatchPlan m, Var input, MatchCaptureBuffer &?captures, MachineStats &?stats)
```

Executes a prepared plan into caller-owned positional storage. Returns 1 on a match, 0 on a miss, and −1 for a null or malformed plan, an invalid buffer, or a machine error. Only success replaces `present` and the indicated values; all other results leave the buffer unchanged. `stats`, when nonnull, receives increments and is not initialized here.

Raises: `<size-limit>` for an ineligible plan, or `<alloc-fail>` while materializing captures.

Source: `lib/match/execution.x:48`

MatchPlan.raise_ineligible

```
void MatchPlan.raise_ineligible(const char *reason, const char *owner)
```

Raises the fence an ineligible plan crossed, naming `owner`. This is the one place an ineligible pattern is reported. Such a pattern compiles to no program, so answering "no match" would be wrong and no caller could tell it from a real miss. `reason` is the plan's static category string.

Raises: `<size-limit>` with `operation` and the fence.

Source: `lib/match/execution.x:15`

MatchPlan.search

```
int MatchPlan.search(MatchPlan plan, List input, List &?out_results)
```

Writes all matches of `plan` within `input` to `out_results`. Each result has the shape documented by `List.search` and the completed `List` is in reverse visitation order. Returns 1 after a complete traversal, including when it writes `nil` for no matches; returns −1 and leaves the output unchanged for an unusable plan, null output, or machine error.

Raises: `<size-limit>` for an ineligible plan, or `<alloc-fail>` while constructing results.

Source: `lib/match/traversal.x:76`

MatchPlan.search_replace

```
int MatchPlan.search_replace( MatchPlan plan, List input, Var template, List &?out)
```

Replaces every match of `plan` from the leaves upward. Returns 1 and writes the completed `List` even when nothing matched. Returns −1 and leaves `out` unchanged for an unusable plan, null output, or machine error. Children are replaced before their containing `List` is tested.

Raises: `<size-limit>` for an ineligible plan, or `<alloc-fail>` while traversing or replacing.

Source: `lib/match/traversal.x:100`

MatchPlan.try_capture

```
int MatchPlan.try_capture( MatchPlan plan, List input, MatchCaptureBuffer &?captures)
```

Executes a prepared `List` match into caller-owned positional storage. This is `MatchPlan.execute_capture` without statistics and has the same results, atomicity, and failures.

Source: `lib/match/execution.x:61`

MatchPlan.try_match

```
int MatchPlan.try_match(MatchPlan plan, List input, List &?out_bindings)
```

Executes prepared `plan` against `input`, writing bindings on success. This is `MatchPlan.execute` without statistics and has the same status, output atomicity, ordering, and failures.

Source: `lib/match/execution.x:174`

MatchPlan.try_match_replace

```
int MatchPlan.try_match_replace( MatchPlan plan, List input, Var template, Var &?out)
```

Executes `plan` and writes the instantiated `template` on success. Returns 1 after writing any `Var` result, 0 on a miss, and −1 for an unusable plan, null output, or machine error. Non-success leaves `out` unchanged.

Raises: `<size-limit>` for an ineligible plan, or `<alloc-fail>` while materializing captures or replacing.

Source: `lib/match/templates.x:30`

MatchPlan.try_search

```
int MatchPlan.try_search( MatchPlan plan, List input, Var &?out_match, List &?out_bindings)
```

Searches `input` with `plan`, writing the first match and bindings. Traversal is depth-first head, tail, then containing `List`. Returns 1 on a match, 0 on a miss, and −1 for an unusable plan, invalid outputs, or a machine error. Unless it returns 1, both outputs remain unchanged.

Raises: `<size-limit>` for an ineligible plan, or `<alloc-fail>` while materializing or publishing bindings.

Source: `lib/match/traversal.x:45`

Public types

Type	Kind	Summary
MatchCaptureBuffer	struct	Supplies caller-owned positional storage for one Match invocation.
MatchCaptureLayout	struct	Describes the canonical binder slots and preparation status of a pattern.
MatchCaptureSite	struct	Stores the process-lifetime plan for one compiler-emitted Match site.
MatchKey	struct	Describes one way an input a plan matches can begin: its first element equals head and, when index is nonzero, its element at index equals inner, or is a List whose first element does when nested is set.
MatchPlan	struct	Holds one reusable immutable compiled Match pattern.

MatchCaptureBuffer

```
typedef struct MatchCaptureBuffer { Var *values; unsigned long present; int capacity; } MatchCaptureBuffer
```

Supplies caller-owned positional storage for one Match invocation. values has capacity elements. On success present identifies the elements Match wrote; presence is separate from a captured void value. A miss or error leaves values and present unchanged. Captured values are borrowed and keep their ordinary Var, List, and pool lifetimes.

Source: lib/match/match.x:42

MatchCaptureLayout

```
typedef struct MatchCaptureLayout { Atom *binders; Var normalized; unsigned long definite, possible; int binder_count, Symbol status, const char *reason; } *MatchCaptureLayout
```

Describes the canonical binder slots and preparation status of a pattern. binders points into the layout allocation and lists each distinct named binder in lexical preorder. definite and possible select those slots. normalized and the binder Atoms are borrowed canonical values. The caller owns the layout returned by MatchCaptureLayout.analyze and must free it before those values or its allocating Scope expire.

Source: lib/match/match.x:29

MatchCaptureSite

```
typedef struct MatchCaptureSite { MatchPlan plan; int refused; } MatchCaptureSite
```

Stores the process-lifetime plan for one compiler-emitted Match site. The object must be zero-initialized static storage. Its first admissible pattern binds it permanently; Match shutdown frees the plan and clears the site. Direct callers must not reuse a site for another pattern.

Source: lib/match/match.x:81

MatchKey

```
typedef struct MatchKey { Var head, inner; int index, nested; } MatchKey
```

Describes one way an input a plan matches can begin: its first element equals head and, when index is nonzero, its element at index equals inner, or is a List whose first element does when nested is set. Both values are borrowed pattern constants.

Source: lib/match/match.x:53

MatchPlan

```
typedef struct MatchPlan { MachineProgram program; MatchCaptureLayout layout; Symbol status, const char *reason; int key_count, key_bits; MatchKey keys[MATCH_KEY_MAX]; } *MatchPlan
```

Holds one reusable immutable compiled Match pattern. A prepared plan owns program and layout; malformed and ineligible plans have no executable program and retain their categorized status and static reason. A

prepared plan whose pattern fixes how a matching input begins lists the `keys` it may begin with; `key_bits` says that raw bits compare them. Pattern constants are borrowed, so the caller must free the plan before their canonical pools or other owners expire.

Source: `lib/match/match.x:68`

Design notes

Match owns the pattern vocabulary, capture layouts, and the execution of plans that `match-plan.x` prepares and `match-cache.x` retains. Core matching runs only prepared plans; `unittest/match-recursive.x` is the reference matcher of the differential tests. An invocation commits captures to caller-owned storage only after the whole match succeeds.

Association-List results are published from committed captures. New Lists canonicalize through the active pool chain; a result may belong to an ancestor and lives until its owning pool is released.

Tests and examples

`make verify` (`unittest/test-match.x`) and `make examples` (`match-nested`).

`lib/meta.x`

The compiler surface a `meta` function calls.

Primary API

Function	Summary
`type_base_suffix`	Returns the suffix of `type` that begins at its typedef name or base keyword, sharing `type`, or `NULL` when it has none.
`type_declaration_parts`	Returns `(base modifiers)` for reconstructing a declaration of `type`.
`type_name_error`	Returns the diagnostic for a `type` that names its type with an Atom other than a C type keyword, as `%(String)` and `%(* Point)` do, or `NULL`.

Functions

type_base_suffix

```
List type_base_suffix(List type)
```

Returns the suffix of `type` that begins at its typedef name or base keyword, sharing `type`, or `NULL` when it has none.

Source: `lib/meta.x:427`

type_declaration_parts

```
List type_declaration_parts(List type)
```

Returns `(base modifiers)` for reconstructing a declaration of `type`. Function modifiers hold parameter syntax, and modifier order retains C declarator precedence.

Source: `lib/meta.x:457`

type_name_error

```
String type_name_error(List type)
```

Returns the diagnostic for a `type` that names its type with an Atom other than a C type keyword, as `%(String)` and `%(* Point)` do, or `NULL`. A type name is a String, as in `%("String")` and `%(* "Point")`; neither a short Symbol nor a long Atom supplies that representation.

Source: `lib/meta.x:440`

Public types

Type	Kind	Summary
`Code`	alias	Captured code parsed from source or produced by a quotation.
`Macro`	alias	A macro as a value: called to build code, or used in a Match `case` to recognize code and capture its parameters.

Type	Kind	Summary
Source	alias	A meta parameter declared Source receives, at a $ call, captured syntax with the source text it came from: ((text T) (file F) (syntax S)).
Type	alias	Semantic type syntax with canonical List-pool lifetime.
TypeInfo	alias	A meta parameter declared TypeInfo receives, at a $ call, the description of its argument's type: ((name N) (kind K) (type T) (fields F) (methods M)).

Code

`typedef List Code`

Captured code parsed from source or produced by a quotation.

Source: `lib/meta.x:60`

Macro

`typedef List Macro`

A macro as a value: called to build code, or used in a Match `case` to recognize code and capture its parameters.

Source: `lib/meta.x:54`

Source

`typedef List Source`

A `meta` parameter declared `Source` receives, at a `$` call, captured syntax with the source text it came from: `((text T) (file F) (syntax S))`. `Code.source_text` and `x2c_embed_text` read it directly.

Source: `lib/meta.x:72`

Type

`typedef List Type`

Semantic type syntax with canonical List-pool lifetime.

Source: `lib/meta.x:57`

TypeInfo

`typedef List TypeInfo`

A `meta` parameter declared `TypeInfo` receives, at a `$` call, the description of its argument's type: `((name N) (kind K) (type T) (fields F) (methods M))`. Read a part with `List.assoc`.

Source: `lib/meta.x:67`

Design notes

A `meta` function runs inside the compiler, so it can ask the compiler questions and build syntax for it to bind. This module declares those operations in x2c; the macro values a `meta` function applies and recognizes are in `macrovalue/macrovalue.x`. Code builds code with quotations (`$!( ... )`, `$!{ ... }`, `$!T{ ... }`) and `%(...)` Lists, so the surface declares no constructors.

The operations on a value are methods of the value's type: `Code`, captured code, and `Type`, a semantic type. The rest are `x2c_` helpers that have no captured receiver: the identifier check, the invocation site, embedded text, ancestry and placement, diagnostics, and the definition hashes. Compile-time Lisp names a method `Owner.member`, as in `Code.binding_spelling`, and a helper with each `_` of its name as `.`, as in `x2c.invocation.line`. The semantics are specified under "Compile-time Lisp and imports" in the language reference.

Each operation is a bodyless `meta` prototype the compiler supplies. A `meta` function that reaches one, directly or through another `meta` function, is compile-time only: the compiler emits no run-time form for it and diagnoses a run-time call where it is written.

Three operations let a macro contribute code beyond its result. `x2c_enclosing` answers the initialized declarator, block item, function, or unit around the invocation, and `x2c_place` puts code after that declarator or block item, on the exits of its block, at the entry of its function, among the unit's support declarations, or in its initialization, under the expansion's transaction. `x2c_binding_fresh` returns a new binding, so placed code can declare a name that the result also refers to. `$auto` is built on them.

The library builds it as an optional module, but the prelude's `varops.x` includes it for its own `meta` rows, so every unit sees its declarations. Include it explicitly where `meta` functions are written. Shared compile-time definitions use ordinary source includes.

Tests and examples

`make verify` (`meta-comptime-only-call`, `meta-builder-compiler-only`, and `meta-literal-arguments` compiler fixtures).

`lib/mutex.x`

Shared mutable-state coordination.

Primary API

Function	Summary
`Mutex.cleanup`	Ends the owned lifetime when a managed local leaves its block.
`Mutex.free`	Destroys an unused, unlocked `Mutex` and releases its `Scope` storage.
`Mutex.lock`	Locks `mutex`, waiting until it becomes available.
`Mutex.new`	Creates an unlocked `Mutex` owned by the active `Scope`.
`Mutex.try_lock`	Attempts to lock `mutex` without waiting.
`Mutex.unlock`	Unlocks a `Mutex` held by the calling thread.

`Mutex`

Mutex.cleanup

```
void Mutex.cleanup(Mutex value)
```

Ends the owned lifetime when a managed local leaves its block.

Source: `lib/mutex.x:22`

Mutex.free

```
void Mutex.free(Mutex mutex)
```

Destroys an unused, unlocked `Mutex` and releases its `Scope` storage. Native destruction occurs first, so a native failure leaves the allocation intact. No thread may retain the handle or be waiting on it.

Raises: `<bad-state>` for NULL, or `<io-fail>` when native destruction fails.

Source: `lib/mutex.x:129`

Mutex.lock

```
void Mutex.lock(Mutex mutex)
```

Locks `mutex`, waiting until it becomes available. The `Mutex` is non-recursive; the caller must not already hold it.

Raises: `<bad-state>` for NULL, or `<io-fail>` when native locking fails.

Source: `lib/mutex.x:96`

Mutex.new

```
Mutex Mutex.new(void)
```

Creates an unlocked `Mutex` owned by the active `Scope`.

Raises: `<alloc-fail>` when storage cannot be allocated, or `<io-fail>` when native mutex initialization fails. An initialization failure releases the allocated storage.

Source: `lib/mutex.x:82`

Mutex.try_lock

```
int Mutex.try_lock(Mutex mutex)
```

Attempts to lock `mutex` without waiting. Returns one when acquired or zero when busy.

Raises: `<bad-state>` for NULL, or `<io-fail>` for another native failure.

Source: `lib/mutex.x:106`

Mutex.unlock

```
void Mutex.unlock(Mutex mutex)
```

Unlocks a `Mutex` held by the calling thread.

Raises: `<bad-state>` for NULL, or `<io-fail>` when native unlocking fails.

Source: `lib/mutex.x:117`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`Mutex.recursive_initialize`	Initializes a process-lifetime recursive lock.
`Mutex.recursive_lock`	Initializes through `once`, then locks the recursive mutex.
`Mutex.recursive_unlock`	Unlocks the recursive mutex.

`Mutex`

Mutex.recursive_initialize

```
void Mutex.recursive_initialize(pthread_mutex_t *mutex, const char *failure)
```

Initializes a process-lifetime recursive lock. On pthread failure, prints `failure` to native stderr and aborts without allocating. This boundary can run while Error or Scope is failing.

Source: `lib/mutex.x:35`

Mutex.recursive_lock

```
void Mutex.recursive_lock( pthread_mutex_t *mutex, pthread_once_t *once, void (*initialize)(void),
const char *failure)
```

Initializes through `once`, then locks the recursive mutex. On pthread failure, prints `failure` to native stderr and aborts.

Source: `lib/mutex.x:49`

Mutex.recursive_unlock

```
void Mutex.recursive_unlock(pthread_mutex_t *mutex, const char *failure)
```

Unlocks the recursive mutex. On pthread failure, prints `failure` to native stderr and aborts.

Source: `lib/mutex.x:61`

Public types

Type	Kind	Summary
Mutex	struct	A native mutex allocated in the active Scope without recursive locking.

Mutex

`typedef struct Mutex *Mutex`

A native mutex allocated in the active `Scope` without recursive locking. Its `Scope` and handle must outlive every user and waiter. Destroy it only while unlocked, and do not use the handle or its storage after that.

Source: `lib/mutex.x:20`

Design notes

`Mutex` is an opaque native mutex. Its `Scope` owner must outlive every thread that can use it, and it may be freed only while unlocked.

Tests and examples

`make verify` (`unittest/test-mutex.x`).

`lib/path.x`

Filesystem locations and the operations on them.

Primary API

Function	Summary
Path.absolute	Returns an absolute form of `path` with symbolic links resolved.
Path.basename	Returns the last component of `path`, ignoring trailing slashes.
Path.copy_file	Copies the regular file `source` to `target`, replacing `target` and giving it `source`'s permission bits.
Path.copy_tree	Copies `source` to `target`: a directory recursively, a symbolic link as a link, and a regular file with `Path.copy_file`.
Path.dirname	Returns the directory part of `path`: `.` when it has no slash and `/` for a path directly under the root.
Path.exists	Reports whether `p` names an existing file, following links.
Path.extension	Returns the extension of `path`'s last component, including its dot, or NULL when there is none.
Path.glob	Returns the existing paths that match the glob `pattern`, sorted.
Path.glob_match	Reports whether all of `path` matches the glob `pattern`.
Path.is_dir	Reports whether `p` names a directory, following links.
Path.is_executable	Reports whether this process may execute `path`, as the shell's `-x`.
Path.is_file	Reports whether `p` names a regular file, following links.
Path.join	Returns `name` joined to `base` with one separating slash.
Path.list_dir	Returns the names in the directory `path`, sorted, without `.` and `..`.
Path.make_dirs	Creates the directory `p` and any missing parents.
Path.modified_time	Returns the modification time of `path` in seconds since the epoch, with the fraction the filesystem records.
Path.move_to	Moves `source` to `target`, copying and removing when they are on different filesystems.
Path.read_text	Returns the contents of the file at `path`, or NULL when it is empty.
Path.remove_file	Removes the file or symbolic link `path` when it exists.
Path.remove_tree	Removes `path` and everything below it when it exists.
Path.size	Returns the size of the file at `path` in bytes.
Path.stem	Returns `path`'s last component without its extension.
Path.symlink_to	Creates the symbolic link `link` pointing at `target`.
Path.temp_dir	Creates a new private directory under `TMPDIR`, or `/tmp`, and returns its path.

Function	Summary
Path.walk	Returns a lazy iterator over every path below the directory root, parents before their contents and siblings sorted.
Path.write_text	Replaces the contents of the file at path with text.

Path

Path.absolute

```
Self Path.absolute(Self path)
```

Returns an absolute form of path with symbolic links resolved. Missing trailing components are appended to their resolved parent with . and .. normalized, so a path that does not exist yet still has a stable identity.

Raises: <io-fail> when a relative path needs the current directory and it cannot be read.

Source: lib/path.x:102

Path.basename

```
meta native Self Path.basename(Self path)
```

Returns the last component of path, ignoring trailing slashes.

Source: lib/path.x:65

Path.copy_file

```
void Path.copy_file(Path source, Path target)
```

Copies the regular file source to target, replacing target and giving it source's permission bits. A target that is already the same file as source is left as it is.

Raises: <not-found> or <io-fail>, including for a directory source.

Source: lib/path.x:474

Path.copy_tree

```
void Path.copy_tree(Path source, Path target)
```

Copies source to target: a directory recursively, a symbolic link as a link, and a regular file with Path.copy_file.

Raises: <not-found> or <io-fail>, with EINVAL when source is a directory and target is that directory or inside it.

Source: lib/path.x:514

Path.dirname

```
meta native Self Path.dirname(Self path)
```

Returns the directory part of path: . when it has no slash and / for a path directly under the root.

Source: lib/path.x:56

Path.exists

```
int Path.exists(Path p)
```

Reports whether p names an existing file, following links.

Source: lib/path.x:131

Path.extension

```
meta native String Path.extension(Path path)
```

Returns the extension of `path`'s last component, including its dot, or NULL when there is none. A leading or trailing dot does not start an extension, so `..` and `notes.` have none.

Source: `lib/path.x:82`

Path.glob

```
List Path.glob(Path pattern)
```

Returns the existing paths that match the glob `pattern`, sorted. The walk starts at the longest leading directory without a wildcard, spelled as the pattern spells it, and descends only as deep as the pattern can match. As in a shell, a name that begins with a dot matches only where the pattern spells the dot, and a pattern that ends with a slash matches only directories, each returned with a trailing slash. No match returns an empty `List`.

Source: `lib/path.x:377`

Path.glob_match

```
int Path.glob_match(Path pattern, Path path)
```

Reports whether all of `path` matches the glob `pattern`. A path component that begins with a dot matches only a pattern component that begins with one, and a run of slashes matches a run of slashes.

Source: `lib/path.x:354`

Path.is_dir

```
int Path.is_dir(Path p)
```

Reports whether `p` names a directory, following links.

Source: `lib/path.x:134`

Path.is_executable

```
int Path.is_executable(Path path)
```

Reports whether this process may execute `path`, as the shell's `-x`.

Source: `lib/path.x:140`

Path.is_file

```
int Path.is_file(Path p)
```

Reports whether `p` names a regular file, following links.

Source: `lib/path.x:137`

Path.join

```
meta native Self Path.join(Self base, Path name)
```

Returns `name` joined to `base` with one separating slash. An absolute `name`, or an empty `base`, is returned unchanged.

Source: `lib/path.x:47`

Path.list_dir

```
List Path.list_dir(Path path)
```

Returns the names in the directory `path`, sorted, without `.` and `..`.

Raises: `<not-found>` or `<io-fail>`.

Source: `lib/path.x:184`

Path.make_dirs

```
void Path.make_dirs(Path p)
```

Creates the directory `p` and any missing parents. An existing directory is left as it is.

Raises: `<io-fail>` when a component cannot be created or names an existing non-directory, or `<not-found>`.

Source: `lib/path.x:404`

Path.modified_time

```
double Path.modified_time(Path path)
```

Returns the modification time of `path` in seconds since the epoch, with the fraction the filesystem records.

Raises: `<not-found>` or `<io-fail>`.

Source: `lib/path.x:170`

Path.move_to

```
void Path.move_to(Path source, Path target)
```

Moves `source` to `target`, copying and removing when they are on different filesystems.

Raises: `<not-found>` or `<io-fail>`.

Source: `lib/path.x:526`

Path.read_text

```
String Path.read_text(Path path)
```

Returns the contents of the file at `path`, or NULL when it is empty.

Raises: `<not-found>`, `<io-fail>`, or `<bad-arg>` when the file contains a NUL byte.

Source: `lib/path.x:546`

Path.remove_file

```
void Path.remove_file(Path path)
```

Removes the file or symbolic link `path` when it exists.

Raises: `<io-fail>` when it exists and cannot be removed.

Source: `lib/path.x:417`

Path.remove_tree

```
void Path.remove_tree(Path path)
```

Removes `path` and everything below it when it exists. Symbolic links are removed, not followed. Removal continues past an entry that cannot be removed.

Raises: `<io-fail>` naming the first path that could not be removed.

Source: `lib/path.x:461`

Path.size

```
long Path.size(Path path)
```

Returns the size of the file at `path` in bytes.

Raises: `<not-found>` or `<io-fail>`.

Source: `lib/path.x:164`

Path.stem

```
meta native String Path.stem(Path path)
```

Returns `path`'s last component without its extension.

Source: `lib/path.x:89`

Path.symlink_to

```
void Path.symlink_to(Path link, Path target)
```

Creates the symbolic link `link` pointing at `target`.

Raises: `<io-fail>` when the link cannot be created.

Source: `lib/path.x:536`

Path.temp_dir

```
Path Path.temp_dir(void)
```

Creates a new private directory under `TMPDIR`, or `/tmp`, and returns its path. The caller removes it, usually with `Path.remove_tree`.

Raises: `<io-fail>` when the directory cannot be created.

Source: `lib/path.x:562`

Path.walk

```
Iter Path.walk(Path root, Iter dest)
```

Returns a lazy iterator over every path below the directory `root`, parents before their contents and siblings sorted. Symbolic links to directories are listed but not followed, and a directory that cannot be read is listed without its contents. Only the paths not yet visited are held; the yielded paths live in the active pool.

```
Path source = "src";
foreach (Path path, source.walk()) printf("%s\n", path);
long units = source.walk().filter(%!(p) => p.str().endswith(".x")).count();
```

Raises: `<not-found>` or `<io-fail>` when `root` cannot be listed, and `<io-fail>` from a pull when a directory vanishes during the walk.

Source: `lib/path.x:236`

Path.write_text

```
void Path.write_text(Path path, String text)
```

Replaces the contents of the file at `path` with `text`.

Raises: `<not-found>` when the directory does not exist, or `<io-fail>`.

Source: `lib/path.x:552`

Public types

Type	Kind	Summary
Path	class	A `String` that names a filesystem location.

Path

```
class Path String
```

A `String` that names a filesystem location. A literal or a `String` converts to a `Path` wherever one is expected, and a `Path` passes wherever a `String` is expected. Slicing and `+` are `String` operations.

Source: `lib/path.x:25`

Design notes

A `Path` is a `String` that names a filesystem location; its methods read and change the files it names. A failure raises `<not-found>` when a named path does not exist and `<io-fail>` for any other host failure, both with `operation`, `path`, and `errno` details. Removing a path that is already absent succeeds.

The path-part methods only examine text. `dirname` and `basename` follow POSIX, ignoring trailing slashes. Glob patterns use `*` and `?` within one path component, `[...]` character classes, `**` for any number of components, and backslash to quote the next character. As in a shell, a wildcard does not match a leading dot; spell the dot to match it.

Tests and examples

`make verify` (`unittest/test-path.x`) and `make examples` (`scripts/line-counts`).

`lib/pool/pool.x`

Nested interning pools with region-backed object storage.

Primary API

Function	Summary
`Pool.close`	Closes the innermost open bracket and makes its parent active.
`Pool.current`	Returns the borrowed canonical-value pool active on this thread.
`Pool.detach`	Removes the innermost open bracket without destroying it.
`Pool.epoch`	Returns a counter that changes whenever any pool level is destroyed.
`Pool.intern_new`	Returns the canonical value equal to `object`, installing it in `inner`.
`Pool.open`	Opens a child of the shared `String` and `List` canonical-value pool.
`Pool.open_named`	Opens and returns a named child of the pool active on this thread.
`Pool.stats`	Returns this level's canonical counts plus process-wide storage counters.

`Pool`

Pool.close

```
void Pool.close(void)
```

Closes the innermost open bracket and makes its parent active. Canonical `Strings`, `Lists`, and transient `String` buffers owned by that pool are reclaimed unless promoted; ancestor-owned values remain live.

Raises: `<bad-state>` when no bracket is open. The failure leaves the active pool unchanged.

Source: `lib/pool/threads.x:101`

Pool.current

```
Pool Pool.current(void)
```

Returns the borrowed canonical-value pool active on this thread. The process root is installed lazily on first use, so the result is never NULL. The pool is borrowed: do not release it with `Pool.release`.

Source: `lib/pool/threads.x:34`

Pool.detach

```
Pool Pool.detach(void)
```

Removes the innermost open bracket without destroying it. `Thread` keeps the detached pool sealed until join copies its survivors. The returned pool remains owned by the caller until that transfer or an explicit `Pool.release`.

Raises: `<bad-state>` when no bracket is open. The failure leaves the active pool unchanged.

Source: `lib/pool/threads.x:113`

Pool.epoch

```
unsigned long Pool.epoch(void)
```

Returns a counter that changes whenever any pool level is destroyed. A cache that borrows canonical identities reads this before trusting an entry: a released level's addresses can be reused, so identities admitted under an earlier value prove nothing. The counter only advances, so a reader needs no lock to tell that something was released.

Source: `lib/pool/threads.x:132`

Pool.intern_new

```
Var Pool.intern_new(Pool inner, Var object, void *alloc)
```

Returns the canonical value equal to `object`, installing it in `inner`. This is `Pool.intern` for a caller that has already searched the whole chain from `inner` outward and found nothing, so only the innermost level is probed. The fused table operation still decides the identity, which keeps one canonical pointer per equal value in `inner` even when another worker interns the same value first.

Raises: `<bad-arg>` when `inner` or `alloc` is NULL. Table insertion causes propagate and leave the object unregistered.

Source: `lib/pool/interning.x:131`

Pool.open

```
Pool Pool.open(void)
```

Opens a child of the shared `String` and `List` canonical-value pool. New canonical misses enter the child, while equal ancestor values retain their existing owner. Promote anything that must outlive the bracket with `String.promote` or `List.promote`: unpromoted values are discarded and their identities no longer resolve. Brackets nest. The caller must match this with one `Pool.close` or detach it for transfer.

Raises: `<alloc-fail>` while opening the pool.

See: `Pool.close`, `Pool.open_named`, `List.promote`

Source: `lib/pool/threads.x:93`

Pool.open_named

```
Pool Pool.open_named(const char *name)
```

Opens and returns a named child of the pool active on this thread. New canonical misses enter the child; an equal ancestor value keeps its existing owner and lifetime. The caller must match this with one `Pool.close` or detach it for transfer. The name appears in `Scope` diagnostics.

Raises: `<alloc-fail>` while opening the pool.

See: `Pool.open`, `Pool.close`

Source: `lib/pool/threads.x:78`

Pool.stats

```
PoolStats Pool.stats(Pool inner)
```

Returns this level's canonical counts plus process-wide storage counters. A null pool reports depth and per-level counts as zero. The counters are a snapshot; nothing in the result stays live with the pool.

Source: `lib/pool/storage.x:534`

Advanced and interop API

Function	Summary
Pool.release	Destroys one pool and returns its parent, or returns NULL for NULL input.
Pool.retain	Returns an unnamed child of inner, without making it thread-active.
Pool.retain_named	Returns a new named child of inner without making it thread-active.

Pool

Pool.release

```
Pool Pool.release(Pool inner)
```

Destroys one pool and returns its parent, or returns NULL for NULL input. Promoted slots and large allocations survive under the parent with stable addresses; other identities, the table, and the control Scope become invalid. Children must already be released, and no caller may use inner afterward. Use Pool.close instead to end a bracket opened with Pool.open.

Source: lib/pool/interning.x:234

Pool.retain

```
Pool Pool.retain(Pool inner)
```

Returns an unnamed child of inner, without making it thread-active. Ownership, failures, and the comparison with Pool.open follow Pool.retain_named.

Source: lib/pool/interning.x:225

Pool.retain_named

```
Pool Pool.retain_named(Pool inner, const char *name)
```

Returns a new named child of inner without making it thread-active. The child owns its control Scope, table, and mutex and must be released before inner. Use Pool.open_named instead to open a bracket that the canonical String and List operations allocate into.

Raises: <alloc-fail> while building the child. A transfer destroys partial child resources and leaves inner unchanged; native mutex initialization failure aborts.

Source: lib/pool/interning.x:191

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
Pool.free	Releases a losing or transient allocation from inner's pool chain.
Pool.initialize	Creates the process-wide canonical-value root and makes it active here.
Pool.insert	Installs object as its own canonical value in exactly inner.
Pool.intern	Returns the canonical value equal to object, installing it when absent.
Pool.is_permanent	Reports whether the process root owns value as its canonical identity.
Pool.lookup	Returns the first value equal to key from inner outward, or void.
Pool.malloc	Allocates object storage owned by inner.
Pool.own	Proves object safe beyond every pool in inner's chain.
Pool.owns	Reports whether this exact level stores key as its canonical identity.
Pool.promote	Publishes an identity owned by inner in its parent.
Pool.shutdown	Releases this thread's open brackets and then the process root.

Function	Summary
`Pool.thread_initialize`	Installs the existing process root in a newly created worker thread.
`Pool.thread_start`	Makes `Pool` lock from here on, for a process about to start a worker.

`Pool`

Pool.free

```
void Pool.free(Pool inner, void *alloc)
```

Releases a losing or transient allocation from `inner`'s pool chain. Region slots become immediately reusable; large allocations use `Scope.free`. A canonical allocation still present in a table must not be freed this way. A null allocation does nothing.

Raises: `<bad-arg>` when `inner` is NULL and `alloc` is not.

Source: `lib/pool/storage.x:167`

Pool.initialize

```
void Pool.initialize(void)
```

Creates the process-wide canonical-value root and makes it active here. `x2c_initialize` calls this once; a repeat call does nothing.

Raises: `<alloc-fail>` if the root cannot be constructed. Native mutex initialization failure aborts.

Source: `lib/pool/threads.x:44`

Pool.insert

```
void Pool.insert(Pool inner, Var object)
```

Installs `object` as its own canonical value in exactly `inner`. The caller must have ruled out an equal identity in this pool chain; this primitive neither searches ancestors nor takes ownership of separate object storage.

Raises: `<bad-arg>` when `inner` is NULL. Table insertion causes propagate and leave the object unregistered.

Source: `lib/pool/interning.x:153`

Pool.intern

```
Var Pool.intern(Pool inner, Var object, void *alloc)
```

Returns the canonical value equal to `object`, installing it when absent. A miss lands in `inner`. `alloc` is the object's `Pool`-owned allocation; a hit releases that losing candidate immediately. A failed insertion leaves the candidate owned by the caller so its existing cleanup boundary runs.

Ancestors are checked before the innermost fused table operation. `Pool`'s single-canonical-pointer invariant makes that order equivalent to outward shadowing while allowing the innermost table to be probed exactly once. A caller that has already searched the chain uses `Pool.intern_new`.

Raises: `<bad-arg>` when `inner` or `alloc` is NULL. Table lookup and insertion causes propagate.

Source: `lib/pool/interning.x:92`

Pool.is_permanent

```
int Pool.is_permanent(Var value)
```

Reports whether the process root owns `value` as its canonical identity. Nothing is promoted, and no open bracket can reclaim a value that answers one.

Source: `lib/pool/threads.x:124`

Pool.lookup

```
Var Pool.lookup(Pool inner, Var key)
```

Returns the first value equal to `key` from `inner` outward, or `void`. The returned identity remains owned by the level where it was found.

Source: `lib/pool/interning.x:43`

Pool.malloc

```
void *Pool.malloc(Pool inner, size_t size)
```

Allocates object storage owned by `inner`. Requests through 512 bytes use a size-class region; larger requests retain ordinary `Scope` ownership.

Raises: `<bad-arg>` when `inner` is NULL, `<size-limit>` when a large request overflows `Scope` storage, or `<alloc-fail>` when storage cannot be allocated.

Source: `lib/pool/storage.x:83`

Pool.own

```
int Pool.own(Pool inner, Var object, void *alloc)
```

Proves `object` safe beyond every pool in `inner`'s chain. It is promoted to the outermost pool when a pool in the chain owns it. `alloc` is the object's `Pool`-owned allocation. Returns one when the value is already outermost or reaches it, and zero when no pool in the chain owns it or a promotion fails. A null `inner` reports safe, since no pool can then reclaim the value. Promotion is not transactional across levels: an earlier level remains promoted if a later promotion transfers. A thread that loses a concurrent promotion of an equal value still gets one, though `Pool.is_permanent` answers zero for its surviving copy.

Raises: `<alloc-fail>` when promotion metadata cannot be allocated.

Source: `lib/pool/storage.x:390`

Pool.owns

```
int Pool.owns(Pool pool, Var key)
```

Reports whether this exact level stores `key` as its canonical identity. Ancestors are not searched, and a null pool reports zero.

Source: `lib/pool/interning.x:169`

Pool.promote

```
int Pool.promote(Pool inner, Var object, void *alloc)
```

Publishes an identity owned by `inner` in its parent. `alloc` survives `inner`'s release without changing its address. When the parent already holds an equal identity, that one stays canonical and `object` only survives. Returns zero for a missing owner, root pool, or null allocation.

Raises: `<alloc-fail>`, `<size-limit>`, or `<invariant>` while recording the promotion; that transfer may happen before storage is marked or moved.

Source: `lib/pool/storage.x:373`

Pool.shutdown

```
void Pool.shutdown(void)
```

Releases this thread's open brackets and then the process root. Call it only after every worker has stopped and no canonical value is still in use. A repeat call after the root is gone does nothing.

Source: `lib/pool/threads.x:63`

Pool.thread_initialize

`void Pool.thread_initialize(void)`

Installs the existing process root in a newly created worker thread. The root must already exist; otherwise the process aborts.

Source: `lib/pool/threads.x:53`

Pool.thread_start

`void Pool.thread_start(void)`

Makes `Pool` lock from here on, for a process about to start a worker. `Thread.start` calls this before `pthread_create`. It never clears.

Source: `lib/pool/threads.x:20`

Public types

Type	Kind	Summary
`Pool`	struct	Holds one level of canonical values and their backing storage.
`PoolStats`	struct	Reports one pool level's activity and process-wide storage counters.
`PoolTable`	struct	Maps each canonical value of one `Pool` level to itself.

Pool

`typedef struct Pool { Scope scope, PoolTable table, struct Pool *up, pthread_mutex_t mutex; unsigned child_capacity; size_t interned, promoted, void *blocks, *current[10], *promotions; } *Pool`

Holds one level of canonical values and their backing storage. Pools are released from child to parent. Their handles and unpromoted allocations become invalid at release; promoted identities retain their pointers under the parent.

Source: `lib/pool/pool.x:38`

PoolStats

`typedef struct PoolStats { int depth, size_t interned, promoted, allocation_calls, free_calls; size_t requested_bytes, block_allocations, block_reuses, slot_reuses; size_t backing_bytes, active_blocks, active_bytes, depot_blocks, depot_bytes; } PoolStats`

Reports one pool level's activity and process-wide storage counters. The snapshot owns no storage; `requested_bytes` saturates at `SIZE_MAX`.

Source: `lib/pool/pool.x:47`

PoolTable

`typedef struct PoolTable { Scope scope, Bytes hashes, entries; unsigned used, capacity, mask; } *PoolTable`

Maps each canonical value of one `Pool` level to itself. A bucket stores the value once, beside its 32-bit hash, so the table costs 12 bytes per bucket where a `Map` costs 20. It belongs to the level's `Scope`.

Source: `lib/pool/pool.x:28`

Design notes

Pool owns canonical identity for a stack of nested levels. Lookup walks outward and a miss lands in the innermost level; release returns empty blocks to one process-wide depot and moves promoted survivors to the parent at the same address. Levels have no sibling branches, so the active chain holds one canonical pointer per equal value. `String` and `List` intern into one such stack per thread.

Tests and examples

`make verify` (`unittest/test-pool.x`).

lib/process.x

Run commands and pipelines without a shell.

Primary API

Function	Summary
Env.get	Returns the value of this process's environment variable name, or NULL when it is unset.
Job.check	Returns job once its status is zero, starting it and waiting as needed.
Job.cleanup	Terminates and reaps a job that is still running: SIGTERM, then SIGKILL to any stage still running a second later.
Job.errors	Returns the captured standard error of job, starting it and waiting as needed, or NULL when standard error was not captured or was empty.
Job.kill	Sends signal to every stage of job that is still running.
Job.lines	Returns the captured standard output of job as lines without their endings.
Job.live	Makes job pass standard output through instead of capturing it, the same as options({stdout: <inherit>}), and returns it.
Job.options	Sets options on job and returns it.
Job.output	Returns the captured standard output of job, starting it and waiting as needed.
Job.pipe	Adds command after the last stage of job, reading that stage's output, and returns the job.
Job.ready	Reports whether every stage of job has exited, without blocking.
Job.run	Passes standard output through, waits for job, and raises when its status is not zero: live() followed by check().
Job.start	Starts job without waiting and returns it.
Job.status	Returns the status of job, starting it and waiting as needed: the exit status, or 128 plus a signal.
Job.wait_any	Removes and returns the first job in jobs that has finished, waiting until one does.
List.job	Returns a Job for command, a command or pipeline, without starting it.

Env

Env.get

```
String Env.get(String name)
```

Returns the value of this process's environment variable name, or NULL when it is unset. The env option sets variables for a child instead.

Source: lib/process.x:664

Job

Job.check

```
Job Job.check(Job job)
```

Returns job once its status is zero, starting it and waiting as needed.

Raises: <cmd-fail> with command and status details, plus output and errors when they were captured, or the start causes of Job.start.

Source: lib/process.x:558

Job.cleanup

```
void Job.cleanup(Job job)
```

Terminates and reaps a job that is still running: SIGTERM, then SIGKILL to any stage still running a second later.

Source: lib/process.x:634

Job.errors

```
String Job.errors(Job job)
```

Returns the captured standard error of `job`, starting it and waiting as needed, or NULL when standard error was not captured or was empty.

Raises: the start causes of `Job.start`, or `<bad-arg>` when the captured text contains a NUL byte.

Source: `lib/process.x:609`

Job.kill

```
void Job.kill(Job job, int signal)
```

Sends `signal` to every stage of `job` that is still running.

Source: `lib/process.x:626`

Job.lines

```
List Job.lines(Job job)
```

Returns the captured standard output of `job` as lines without their endings.

Raises: the causes of `Job.output`.

Source: `lib/process.x:602`

Job.live

```
Job Job.live(Job job)
```

Makes `job` pass standard output through instead of capturing it, the same as `options({stdout: <inherit>})`, and returns it.

Raises: `<bad-arg>` for a job that has started.

Source: `lib/process.x:527`

Job.options

```
Job Job.options(Job job, Map options)
```

Sets `options` on `job` and returns it. The keys are atoms: `dir` names the working directory, `env` is a `Map` of variables added to the inherited environment, and `input` is a `String` given to the first stage's standard input. `stdout` is `capture`, `inherit`, or a file path and applies to the last stage; `stderr` is `inherit`, `capture`, `stdout` to merge, or a file path and applies to every stage. A key set again replaces its earlier value.

```
String root = %(pwd).job().options({dir: "/"}).output();
```

Raises: `<bad-arg>` for an unknown key or a job that has started.

Source: `lib/process.x:487`

Job.output

```
String Job.output(Job job)
```

Returns the captured standard output of `job`, starting it and waiting as needed. A live job, or one whose output was empty, returns NULL.

Raises: the causes of `Job.check`, or `<bad-arg>` when the output contains a NUL byte.

Source: `lib/process.x:588`

Job.pipe

```
Job Job.pipe(Job job, List command)
```

Adds `command` after the last stage of `job`, reading that stage's output, and returns the job. A pipeline `command` adds each of its stages.

Raises: `<bad-arg>` for a job that has started.

Source: `lib/process.x:455`

Job.ready

```
int Job.ready(Job job)
```

Reports whether every stage of `job` has exited, without blocking. A job that has not started reports 0.

Source: `lib/process.x:617`

Job.run

```
void Job.run(Job job)
```

Passes standard output through, waits for `job`, and raises when its status is not zero: `live()` followed by `check()`.

Raises: the causes of `Job.live` and `Job.check`.

Source: `lib/process.x:579`

Job.start

```
meta native Job Job.start(Job job)
```

Starts `job` without waiting and returns it. A job that has started is returned unchanged.

Raises: `<not-found>` when a program or the `dir` option does not exist, `<io-fail>` when a pipe, fork, output file, or other start step fails, or `<bad-arg>` for an empty command.

Source: `lib/process.x:537`

Job.status

```
int Job.status(Job job)
```

Returns the status of `job`, starting it and waiting as needed: the exit status, or 128 plus a signal. A stage that never ran because the start raised reports 127. A status that is not zero is an ordinary result here.

Raises: the start causes of `Job.start`.

Source: `lib/process.x:547`

Job.wait_any

```
Job Job.wait_any(Array jobs)
```

Removes and returns the first job in `jobs` that has finished, waiting until one does. An empty `jobs` returns NULL.

Raises: `<bad-arg>` when a job in `jobs` has not started.

Source: `lib/process.x:644`

List

List.job

```
meta native Job List.job(List command)
```

Returns a `Job` for `command`, a command or pipeline, without starting it. The job captures standard output and passes standard error through. Its first result starts it, waits, and records the run. A `Job` destination calls this converter, so the call is usually left implicit.

```
Job job = %(printf "a\nb\n");
```

Source: `lib/process.x:427`

Public types

Type	Kind	Summary
Env	alias	The receiverless owner of Env.get.
Job	class	A command or pipeline and the record of its one run.
JobLaunch	struct	The launch options embedded in a Job.

Env

```
typedef Symbol Env
```

The receiverless owner of Env.get.

Source: lib/process.x:49

Job

```
class Job { List stages; JobLaunch launch; long *pids; int *statuses; int count, started, finished, status, nul_output, nul_errors; File output_file, errors_file; String output_text, errors_text; } *
```

A command or pipeline and the record of its one run. A job that is still running when its Scope ends, or when its $auto block exits, is terminated and reaped.

Source: lib/process.x:33

JobLaunch

```
typedef struct _JobLaunch { String dir, input, stdout_path, stderr_path; int has_input, capture_output, capture_errors, errors_to_output; char **environment; } JobLaunch
```

The launch options embedded in a Job. Set these through Job.options before the job starts. The record has the job's lifetime. Everything a child needs is prepared before fork, so the child only duplicates descriptors, changes directory, and calls execvp.

Source: lib/process.x:23

Design notes

Job owns running a command or pipeline without a shell. A command is an ordinary List whose elements' strs become its arguments, and a List of commands is a pipeline. A job runs once, when its first result is requested, and records that run; its status is the last failing stage's status, with 128 plus the signal for a signalled stage. The calling process's own environment is read here too, beside the env option that sets a child's.

Tests and examples

make verify (unittest/test-process.x) and make examples (scripts/line-counts).

lib/regex.x

Regular expressions over the bytes of a String.

Primary API

Function	Summary
Regex.capture_count	Returns the number of capturing groups in regex.
Regex.capture_names	Returns the names of the capturing groups of regex in capture order, with NULL for a group without a name.
Regex.compile	Compiles pattern and returns the Regex, owned by the current scope.
Regex.escape	Returns literal with every ASCII byte that is not a letter, digit, or underscore escaped, so that it matches itself inside a pattern.
Regex.find_all	Returns every non-overlapping match of r in subject, in order.
Regex.match	Returns the first match of r in subject, or NULL.

Function	Summary
`Regex.match_from`	Returns the first match of `r` in `subject` at or after the byte `offset`, or NULL.
`Regex.pattern`	Returns the text `regex` was compiled from.
`Regex.replace`	Returns `subject` with the first match of `regex` replaced.
`Regex.replace_all`	Returns `subject` with every match of `regex` replaced, expanding `replacement` as `replace` does.
`Regex.replace_fn`	Returns `subject` with every match of `regex` replaced by what `fn` returns for its `RegexMatch`, inserted as is.
`Regex.split`	Returns the text of `subject` between the matches of `regex`, keeping an empty field where two matches touch or a match sits at either end.
`RegexCapture.end`	Returns the byte offset after the capture, or −1.
`RegexCapture.index`	Returns the capture number, with 0 for the whole match.
`RegexCapture.matched`	Reports whether the capture took part in the match.
`RegexCapture.name`	Returns the capture's name, or NULL.
`RegexCapture.start`	Returns the byte offset where the capture starts, or −1.
`RegexCapture.text`	Returns the matched text, or NULL for a capture that did not take part.
`RegexMatch.capture`	Returns the capture of `found` selected by `key`: a capture number, or a name as a `Symbol` or `String`.
`RegexMatch.getindex`	Returns the text of the capture selected by `key`, or NULL when the capture does not exist or did not take part in the match.
`Var.regexcapture`	Reads a `RegexCapture` back out of a `Var`, as `foreach` does.
`Var.regexmatch`	Reads a `RegexMatch` back out of a `Var`, as `foreach` does.

`Regex`

Regex.capture_count

```
int Regex.capture_count(Regex regex)
```

Returns the number of capturing groups in `regex`.

Source: `lib/regex.x:723`

Regex.capture_names

```
List Regex.capture_names(Regex regex)
```

Returns the names of the capturing groups of `regex` in capture order, with NULL for a group without a name.

Source: `lib/regex.x:728`

Regex.compile

```
Regex Regex.compile(String pattern)
```

Compiles `pattern` and returns the `Regex`, owned by the current scope.

Raises: `<malformed>` with `reason`, the `pattern`, and the byte `offset` of the problem when the pattern does not parse.

Source: `lib/regex.x:717`

Regex.escape

```
String Regex.escape(String literal)
```

Returns `literal` with every ASCII byte that is not a letter, digit, or underscore escaped, so that it matches itself inside a pattern.

Source: `lib/regex.x:733`

Regex.find_all

```
List Regex.find_all(Regex r, String subject)
```

Returns every non-overlapping match of `r` in `subject`, in order. An empty match advances one byte. No match returns an empty `List`.

Source: `lib/regex.x:758`

Regex.match

```
RegexMatch Regex.match(Regex r, String subject)
```

Returns the first match of `r` in `subject`, or NULL.

Source: `lib/regex.x:753`

Regex.match_from

```
RegexMatch Regex.match_from(Regex r, String subject, int offset)
```

Returns the first match of `r` in `subject` at or after the byte `offset`, or NULL. An offset beyond the end of `subject` finds nothing.

Source: `lib/regex.x:749`

Regex.pattern

```
String Regex.pattern(Regex regex)
```

Returns the text `regex` was compiled from.

Source: `lib/regex.x:720`

Regex.replace

```
String Regex.replace(Regex regex, String subject, String replacement)
```

Returns `subject` with the first match of `regex` replaced. In `replacement`, `$0` through `$9` and `${name}` insert a capture and `$$` is a dollar sign.

Source: `lib/regex.x:785`

Regex.replace_all

```
String Regex.replace_all(Regex regex, String subject, String replacement)
```

Returns `subject` with every match of `regex` replaced, expanding `replacement` as `replace` does.

Source: `lib/regex.x:791`

Regex.replace_fn

```
String Regex.replace_fn(Regex regex, String subject, Func fn)
```

Returns `subject` with every match of `regex` replaced by what `fn` returns for its `RegexMatch`, inserted as is.

Raises: whatever `fn` raises.

Source: `lib/regex.x:798`

Regex.split

```
List Regex.split(Regex regex, String subject)
```

Returns the text of `subject` between the matches of `regex`, keeping an empty field where two matches touch or a match sits at either end.

Source: `lib/regex.x:769`

RegexCapture

RegexCapture.end

```
int RegexCapture.end(RegexCapture capture)
```

Returns the byte offset after the capture, or −1.

Source: `lib/regex.x:840`

RegexCapture.index

```
int RegexCapture.index(RegexCapture capture)
```

Returns the capture number, with 0 for the whole match.

Source: `lib/regex.x:825`

RegexCapture.matched

```
int RegexCapture.matched(RegexCapture capture)
```

Reports whether the capture took part in the match.

Source: `lib/regex.x:831`

RegexCapture.name

```
String RegexCapture.name(RegexCapture capture)
```

Returns the capture's name, or NULL.

Source: `lib/regex.x:828`

RegexCapture.start

```
int RegexCapture.start(RegexCapture capture)
```

Returns the byte offset where the capture starts, or −1.

Source: `lib/regex.x:837`

RegexCapture.text

```
String RegexCapture.text(RegexCapture capture)
```

Returns the matched text, or NULL for a capture that did not take part.

Source: `lib/regex.x:834`

RegexMatch

RegexMatch.capture

```
RegexCapture RegexMatch.capture(RegexMatch found, Var key)
```

Returns the capture of `found` selected by `key`: a capture number, or a name as a `Symbol` or `String`. NULL when there is no such capture.

Source: `lib/regex.x:806`

RegexMatch.getindex

```
String RegexMatch.getindex(RegexMatch found, Var key)
```

Returns the text of the capture selected by `key`, or NULL when the capture does not exist or did not take part in the match.

Source: `lib/regex.x:819`

`Var`

Var.regexcapture

`meta native RegexCapture Var.regexcapture(Var v)`

Reads a `RegexCapture` back out of a `Var`, as `foreach` does.

Source: `lib/regex.x:843`

Var.regexmatch

`meta native RegexMatch Var.regexmatch(Var value)`

Reads a `RegexMatch` back out of a `Var`, as `foreach` does.

Source: `lib/regex.x:846`

Public types

Type	Kind	Summary
`Regex`	class	A compiled pattern.
`RegexCapture`	alias	One capture of a match: index, name, matched, text, start, end.
`RegexMatch`	alias	The captures of one match, with the whole match at index 0.

Regex

`class Regex struct { String pattern; _RegexNode program; int capture_count; List capture_names; int caseless, multiline, dotall; } *`

A compiled pattern. `Regex.compile` makes one; the scope that made it frees it. A `RegexMatch` is the `List` of `RegexCaptures` of one match, indexed by capture number or name; each capture is the `List` of its index, name, whether it took part, text, start, and end.

Source: `lib/regex.x:23`

RegexCapture

`typedef List RegexCapture`

One capture of a match: index, name, matched, text, start, end.

Source: `lib/regex.x:34`

RegexMatch

`typedef List RegexMatch`

The captures of one match, with the whole match at index 0.

Source: `lib/regex.x:37`

Design notes

A Regex is a compiled pattern, owned by the scope that compiled it. A RegexMatch and a RegexCapture are immutable Lists, so their Strings and offsets stay valid after the next match. Matching backtracks over a small node tree; a repeated single byte class runs as a loop, so the common `.*` and `\w+` forms do not recurse per byte. The syntax is the byte-oriented subset in the book's "Patterns" section.

Tests and examples

`make verify` (`unittest/test-regex.x`).

`lib/rewrite.x`

Pattern registration for compiler rewrites.

lib/scope/scope.x

Memory allocation scope management.

Primary API

Function	Summary
x2c_scope_thread_release	Destroys this thread's Scope chain and its push and retain stacks.
Scope.malloc_finalized	Allocates size uninitialized bytes in the active scope with a finalizer.
Scope.malloc_finalized_in	Allocates size bytes with finalizer drop in the scope held by s.
Scope.new	Creates a detached, unnamed scope and returns it.
Scope.pop	Restores the slot that was active before the matching Scope.push.
Scope.push	Makes the scope in scope active until a matching Scope.pop.

Functions

x2c_scope_thread_release

```
void x2c_scope_thread_release(void)
```

Destroys this thread's Scope chain and its push and retain stacks. It runs last in x2c_thread_state_release and repeats harmlessly; a thread that never created a Scope has nothing to destroy.

Source: lib/scope/regions.x:300

Scope

Scope.malloc_finalized

```
void *Scope.malloc_finalized(size_t size, void (*drop)(void *))
```

Allocates size uninitialized bytes in the active scope with a finalizer. drop runs exactly once with the block's pointer when the block is reclaimed: by Scope.free, by Scope.realloc to size zero, by the release or destruction of its scope, or by thread and process shutdown. The finalizer follows the block through Scope.move and survives Scope.realloc, which passes drop the resized pointer. Blocks are reclaimed most recent first, so a finalizer sees older blocks still live.

A wrapper for a native handle allocates its record this way and releases the handle from drop; an explicit early release that clears the field leaves nothing for the finalizer to do. drop runs on the thread that reclaims the block, with the block already unlinked, so it must not free or move the block itself. It must not raise. It may allocate into other scopes, and into the dying scope only for scratch the same destruction reclaims.

Raises: <bad-arg> when drop is NULL, <size-limit> when the size would overflow the allocation header, or <alloc-fail> when the underlying allocation fails. Before Error initialization they terminate at the error floor.

Source: lib/scope/allocation.x:73

Scope.malloc_finalized_in

```
void *Scope.malloc_finalized_in(Scope *s, size_t size, void (*drop)(void *))
```

Allocates size bytes with finalizer drop in the scope held by s. The slot-targeted form of Scope.malloc_finalized, with the same lazy scope creation as Scope.malloc_in; the active scope is left alone.

Raises: <bad-arg> when s or drop is NULL, <size-limit> when the size overflows, or <alloc-fail> when allocation fails. Before Error initialization they terminate at the error floor.

Source: lib/scope/allocation.x:101

Scope.new

```
meta native Scope Scope.new(void)
```

Creates a detached, unnamed scope and returns it. A detached scope sits in no slot and is not active, so nothing is charged to it until you allocate through Scope.malloc_in and friends or make it active with Scope.push. End

it with `Scope.destroy`. Prefer `Scope.new_named` for anything long-lived; the name appears in the exit-time leak report.

Raises: `<alloc-fail>` when the scope cannot be allocated. Before `Error` initialization it terminates at the error floor.

Source: `lib/scope/regions.x:219`

Scope.pop

```
meta native void Scope.pop(void)
```

Restores the slot that was active before the matching `Scope.push`. Popping frees nothing. The popped slot keeps its scope and every allocation in it, so you can destroy that scope later or pass it elsewhere. With no pushes outstanding, the process root slot becomes active again.

Raises: `<bad-state>` when the scope stack is empty, so a mismatched push and pop raises instead of redirecting allocations. The failure leaves the active slot unchanged. Before `Error` initialization it terminates at the error floor.

Source: `lib/scope/regions.x:73`

Scope.push

```
meta native void Scope.push(Scope *scope)
```

Makes the scope in `scope` active until a matching `Scope.pop`. `scope` is the address of a caller-owned `Scope` variable, and it may hold NULL: the slot is filled lazily by the first allocation that lands in it. Pushing lets several operations share one lifetime without threading a slot through every call. Slots stack, and each push needs one pop.

```
Scope work = NULL;
Scope.push(&work);
char *buffer = Scope.malloc(16);
snprintf(buffer, 16, "in work");
Scope.pop();
puts(buffer);
Scope.destroy(work);
```

The pointer stays valid after the pop. Popping changes which slot is active; `work` still owns the allocation until it is destroyed. A pushed slot cannot be destroyed while it is on the stack.

Raises: `<bad-arg>` when `scope` is NULL, `<size-limit>` when the stack cannot grow within its representation, or `<alloc-fail>` when its storage cannot be allocated. These failures leave the active slot unchanged. Before `Error` initialization they terminate at the error floor.

Source: `lib/scope/regions.x:40`

Advanced and interop API

Function	Summary
`Scope_shutdown`	Releases resources owned by `Scope`.
`Scope.calloc`	Allocates `count` objects of `size` bytes each, zeroed, in the active scope.
`Scope.calloc_in`	Allocates `count` zeroed objects of `size` bytes in the scope in `slot`.
`Scope.cleanup`	Ends the owned lifetime when a managed local leaves its block.
`Scope.destroy`	Destroys a detached scope and frees every allocation it owns.
`Scope.free`	Frees one scope-owned allocation before its scope ends.
`Scope.malloc`	Allocates `size` uninitialized bytes in the active scope.
`Scope.malloc_in`	Allocates `size` uninitialized bytes in the scope held by `slot`.
`Scope.memdup`	Copies `size` bytes from `ptr` into a new allocation in the active scope.
`Scope.memdup_in`	Copies `size` bytes from `ptr` into the scope held by `slot`.

Function	Summary
Scope.move	Relinks one allocation onto the scope held by slot.
Scope.name	Returns the diagnostic name of scope, or NULL if it has none.
Scope.new_named	Creates a detached scope carrying a copy of name for diagnostics.
Scope.realloc	Resizes one scope-owned allocation and returns the new pointer.
Scope.release	Destroys the scope in the active slot and frees everything it owns.
Scope.retain	Opens a new scope in the active slot and makes it the current one.
Scope.stats	Returns a snapshot of the allocator's counters.
Scope.top	Returns the address of the active scope slot.

Functions

Scope_shutdown

```
void Scope_shutdown(void)
```

Releases resources owned by Scope. Scope groups managed allocations by lifetime; balanced retain/release and push/pop boundaries remain caller responsibilities.

Source: lib/scope/runtime.x:94

Scope

Scope.calloc

```
meta native void *Scope.calloc(size_t count, size_t size)
```

Allocates count objects of size bytes each, zeroed, in the active scope. The product is checked for overflow before anything is allocated, and the bytes are set to zero; in every other respect this behaves like Scope.malloc. A request that multiplies out to zero still returns a distinct pointer the scope owns, so it is not a failure signal.

Raises: <size-limit> when the object count overflows, or <alloc-fail> when allocation fails. Before Error initialization they terminate at the error floor.

Source: lib/scope/allocation.x:143

Scope.calloc_in

```
void *Scope.calloc_in(Scope *slot, size_t count, size_t size)
```

Allocates count zeroed objects of size bytes in the scope in slot. The slot-targeted form of Scope.calloc, with the same overflow check and the same lazy scope creation as Scope.malloc_in; the active scope is left alone.

Raises: <bad-arg> when slot is NULL, <size-limit> when the object count overflows, or <alloc-fail> when allocation fails. Before Error initialization they terminate at the error floor.

Source: lib/scope/allocation.x:156

Scope.cleanup

```
meta native void Scope.cleanup(Scope value)
```

Ends the owned lifetime when a managed local leaves its block.

Source: lib/scope/regions.x:277

Scope.destroy

```
meta native void Scope.destroy(Scope scope)
```

Destroys a detached scope and frees every allocation it owns. This ends a scope you hold in a variable, and is the counterpart to Scope.new and Scope.new_named. It frees the scope's allocations and discards its name; pointers

into it are dangling afterwards, and nothing diagnoses their use. Any regions still linked below it, from retains that were never released, are destroyed with it.

`Scope.destroy` refuses a scope that is still in use: the active root scope, a slot still on the pushed stack (pop it first), and a scope that a later `Scope.retain` layered another region on top of, which the runtime reports as an attached lower scope. The handle is a pointer into the destroyed scope, so destroying it again is undefined behavior.

Raises: `<bad-state>` for the active root, a pushed slot, or an attached lower scope. The failure leaves the scope intact. A NULL `scope` does nothing. Before `Error` initialization it terminates at the error floor.

Source: `lib/scope/regions.x:264`

Scope.free

```
meta native void Scope.free(void *ptr)
```

Frees one scope-owned allocation before its scope ends. `ptr` must be a pointer returned by `Scope.malloc`, `Scope.calloc`, `Scope.memdup`, one of their `_in` forms, or `Scope.realloc`. It is unlinked from whichever scope owns it now, which after a `Scope.move` may not be the active one.

Shortening a lifetime this way is normal. Freeing anything else, such as a stack address, an interned `String`, or a plain `malloc` result, is undefined, as is freeing the same pointer twice. Nothing diagnoses either.

A NULL `ptr` does nothing.

Source: `lib/scope/allocation.x:224`

Scope.malloc

```
meta native void *Scope.malloc(size_t size)
```

Allocates `size` uninitialized bytes in the active scope. The result is managed memory. `Scope.realloc` resizes it, `Scope.free` ends its life early, `Scope.move` reassigns its owner, and `Scope.release` or `Scope.destroy` reclaims whatever is left. The active scope at the time of allocation owns the result; a later retain or push does not move it.

Raises: `<size-limit>` when the size would overflow the allocation header, or `<alloc-fail>` when the underlying allocation fails. Before `Error` initialization they terminate at the error floor.

Source: `lib/scope/allocation.x:48`

Scope.malloc_in

```
void *Scope.malloc_in(Scope *slot, size_t size)
```

Allocates `size` uninitialized bytes in the scope held by `slot`. `slot` is the address of a `Scope` variable; if it holds NULL, a fresh unnamed scope is created and stored there. Targeting a slot does not touch the active-scope stack, so an intervening `Scope.retain` or `Scope.push` cannot redirect the allocation. Pass `Scope.top()` to name the active slot explicitly.

Raises: `<bad-arg>` when `slot` is NULL, `<size-limit>` when the size overflows, or `<alloc-fail>` when allocation fails. Before `Error` initialization they terminate at the error floor.

Source: `lib/scope/allocation.x:89`

Scope.memdup

```
meta native void *Scope.memdup(const void *ptr, size_t size)
```

Copies `size` bytes from `ptr` into a new allocation in the active scope. The copy is ordinary scope-owned memory, freed by `Scope.free` or by the release that ends the region. Nothing about the source is remembered, so duplicating a C string means copying its terminator too: `Scope.memdup(text, strlen(text) + 1)`.

A NULL `ptr` or a zero `size` returns NULL and allocates nothing, so a duplicate of nothing is indistinguishable from failure; check the arguments when that distinction matters.

Raises: `<size-limit>` or `<alloc-fail>` from the underlying allocation. A NULL `ptr` or zero `size` returns NULL without raising.

Source: `lib/scope/allocation.x:173`

Scope.memdup_in

```
void *Scope.memdup_in(Scope *slot, const void *ptr, size_t size)
```

Copies `size` bytes from `ptr` into the scope held by `slot`. The slot-targeted form of `Scope.memdup`, with the same NULL-for-nothing rule and the same lazy scope creation as `Scope.malloc_in`. It is the usual way to hand a snapshot of caller data to a scope that outlives the current region.

Raises: `<bad-arg>` when `slot` is NULL, or `<size-limit>` or `<alloc-fail>` from the underlying allocation. A NULL `ptr` or zero `size` returns NULL without raising.

Source: `lib/scope/allocation.x:187`

Scope.move

```
meta native void Scope.move(void *ptr, Scope *slot)
```

Relinks one allocation onto the scope held by `slot`. The bytes are not copied and the pointer does not change; only ownership moves, so a temporary region can compute one result that outlives it. If `slot` holds NULL a fresh unnamed scope is created there, as the `_in` allocators do.

```
Scope keep = Scope.new_named("results");
Scope.retain();
char *text = Scope.memdup("survivor", 9);
Scope.move(text, &keep);
Scope.release();
puts(text);
Scope.destroy(keep);
```

You do not need this for a `String`, a `List`, or a `Symbol`. Canonical values already outlive the scope that was active when they were built.

Raises: `<bad-arg>` when `slot` is NULL, or `<alloc-fail>` when a new destination scope cannot be allocated. These failures leave ownership unchanged. A NULL `ptr` does nothing. Before `Error` initialization they terminate at the error floor.

Source: `lib/scope/allocation.x:270`

Scope.name

```
const char *Scope.name(Scope scope)
```

Returns the diagnostic name of `scope`, or NULL if it has none. The string belongs to the allocator's name registry and stays valid until the scope is destroyed; do not free it. A scope from `Scope.new`, and one the runtime created implicitly for the first allocation into an empty slot, both have no name.

Source: `lib/scope/accounting.x:13`

Scope.new_named

```
meta native Scope Scope.new_named(const char *name)
```

Creates a detached scope carrying a copy of `name` for diagnostics. The name is copied, so a temporary buffer is fine. `Scope.name` reports it, and the allocator prints it at exit if anything the scope owns is still alive, as a line like `scope "request": 3 allocations`. A NULL `name` behaves like `Scope.new`.

```
Scope work = Scope.new_named("request");
char *copy = Scope.memdup_in(&work, "payload", 8);
puts(copy);
Scope.destroy(work);
```

Raises: `<alloc-fail>` when the scope or name copy cannot be allocated, or `<size-limit>` when the name is too large. Before `Error` initialization these failures terminate the process.

Source: `lib/scope/regions.x:243`

Scope.realloc

```
meta native void *Scope.realloc(void *ptr, size_t size)
```

Resizes one scope-owned allocation and returns the new pointer. Ownership does not change: the allocation stays with the scope that already held it, even if that is not the active one. Two edge cases follow C's `realloc`: a NULL `ptr` allocates `size` bytes in the active scope, and a `size` of zero frees the allocation and returns NULL. A NULL result is not by itself a failure.

As with C, the old pointer must be treated as dead once a resize succeeds.

Raises: `<size-limit>` when the size overflows, or `<alloc-fail>` when allocation fails. Before `Error` initialization these failures terminate at the error floor.

Source: `lib/scope/allocation.x:292`

Scope.release

```
meta native void Scope.release(void)
```

Destroys the scope in the active slot and frees everything it owns. Every allocation charged to that scope is freed: `Scope.malloc`, `Scope.calloc`, and `Scope.memdup` results, and the backing storage of the `Blocks`, `Arrays`, `Buffers`, and boxed wide `Vars` built while it was active. The scope linked below it then becomes active again.

Canonical values are not reclaimed. Interned `Strings`, `List` cells, and `Symbols` built inside the region remain valid after release. An allocation relinked with `Scope.move` now belongs to its new scope and survives. Anything owned by another scope or another slot is untouched, and memory from plain `malloc` is unaffected.

The active region must be the one opened by a matching `Scope.retain` on this same slot. Releasing an empty slot, a directly allocated root, or a region retained on another pushed slot is an imbalance and raises. An extra release cannot destroy the surrounding region, and a retained empty scope still closes normally.

A pointer into a released region is not diagnosed. Neither the compiler nor the runtime tracks it, and using it afterwards is undefined behavior. `Scope.stats` confirms that a routine leaves nothing live.

Raises: `<bad-state>` when no matching retain owns the active region. The failure leaves the active region intact. Before `Error` initialization it terminates at the error floor.

Source: `lib/scope/regions.x:155`

Scope.retain

```
meta native void Scope.retain(void)
```

Opens a new scope in the active slot and makes it the current one. Allocations that follow are charged to the new scope. The scope that was there is linked below it and becomes current again on release, so retained regions nest. Open a nested region when a group of temporaries should die before the surrounding work does. Each `Scope.retain` must be paired with exactly one `Scope.release`.

Pair them with `defer`. `Error` transfer does not release a scope. A raise that jumps past a plain `Scope.release` leaves that region live for the rest of the process, so use `defer` in any function that both retains and can raise.

```
Scope.retain();
defer Scope.release();
char *line = Scope.malloc(width + 1);
snprintf(line, width + 1, "%d bytes", width);
puts(line);
```

A scope wrapped around nothing but canonical values does nothing, since those values already outlive it, and adds a release that is easy to omit.

Raises: `<alloc-fail>` when the scope or its retain record cannot be allocated, or `<size-limit>` when that record cannot grow. Before `Error` initialization they terminate at the error floor.

Source: `lib/scope/regions.x:117`

Scope.stats

```
ScopeStats Scope.stats(void)
```

Returns a snapshot of the allocator's counters. `live_allocations` and `live_scopes` are derived from the call counts, so read them before and after a routine to check that it leaves nothing behind. The snapshot also carries `allocation_calls`, `reallocation_calls`, `free_calls`, `scope_creations`, `scope_destructions`, `requested_bytes`, `largest_request`, `live_requested_bytes`, and `peak_live_requested_bytes`. Requested bytes are cumulative traffic; live requested bytes are the exact current managed payload total. Stats remain valid after shutdown.

```
size_t before = Scope.stats().live_allocations;
Scope.retain();
char *scratch = Scope.malloc(128);
scratch[0] = 0;
Scope.release();
printf("reclaimed = %d\n", Scope.stats().live_allocations == before);
```

Source: `lib/scope/accounting.x:107`

Scope.top

```
Scope *Scope.top(void)
```

Returns the address of the active scope slot. The result is the slot, not the scope. Pass it to the `_in` allocators to name the active slot explicitly. Dereferencing it gives the scope, which may be NULL before anything has been allocated into the slot. It names the slot that was active at the call; a later `Scope.push`, `Scope.pop`, `Scope.retain`, or `Scope.release` changes which slot is active, so read it again instead of caching it.

Source: `lib/scope/regions.x:58`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`Scope.initialize`	Initializes the process-wide `Scope` runtime owner.
`Scope.owner`	Returns the `Scope` that currently owns `ptr`.
`Scope.shutdown_hook`	Registers `hook` to run during process-wide `Scope` shutdown.

`Scope`

Scope.initialize

```
void Scope.initialize(void)
```

Initializes the process-wide `Scope` runtime owner.

Source: `lib/scope/runtime.x:49`

Scope.owner

```
Scope Scope.owner(void *ptr)
```

Returns the `Scope` that currently owns `ptr`. `ptr` must be a live pointer returned by a `Scope` allocator. `Context` uses this to leave ancestor-owned objects where they are while moving results out of its own `Scope` chain. Passing any other nonnull pointer is undefined behavior, matching `Scope.free` and `Scope.move`.

Source: `lib/scope/allocation.x:236`

Scope.shutdown_hook

`void Scope.shutdown_hook(void (*hook)(void))`

Registers `hook` to run during process-wide `Scope` shutdown. The function pointer is retained without being invoked. Shutdown invokes registrations once in reverse order while `Scope` storage is still available, including one registered by a hook that shutdown is already running.

Raises: `<bad-arg>` for a null hook, `<size-limit>` when the registry cannot grow, or `<alloc-fail>` when its storage cannot be allocated.

Source: `lib/scope/runtime.x:83`

Public types

Type	Kind	Summary
`Scope`	struct	Handle for a region and any retained regions linked below it.
`ScopeAlloc`	struct	Intrusive header stored immediately before each `Scope`-owned allocation.
`ScopeStats`	struct	Value snapshot of process-wide `Scope` allocation and lifetime counters.

Scope

`typedef struct Scope { ScopeAlloc first; struct Scope *down, *up; } *Scope`

Handle for a region and any retained regions linked below it. The layout is public; only `Scope` operations create or mutate valid allocation and region links.

Source: `lib/scope/scope.x:34`

ScopeAlloc

`typedef struct ScopeAlloc { struct ScopeAlloc *prev, *next; } *ScopeAlloc`

Intrusive header stored immediately before each `Scope`-owned allocation. Its links are allocator-managed; callers must not construct or mutate it.

Source: `lib/scope/scope.x:26`

ScopeStats

`typedef struct ScopeStats { size_t allocation_calls, reallocation_calls, free_calls, live_allocations; size_t scope_creations, scope_destructions, live_scopes, requested_bytes; size_t largest_request, live_requested_bytes, peak_live_requested_bytes; } ScopeStats`

Value snapshot of process-wide `Scope` allocation and lifetime counters. It owns no storage; live object and scope counts are derived when the snapshot is taken. Live and peak requested bytes count public payload only, excluding Scope metadata, native allocator overhead, and Pool's direct small-object blocks.

Source: `lib/scope/scope.x:45`

Design notes

A `Scope` owns a group of individually managed allocations. Callers use the active scope, target an explicit scope slot, or retain and release a nested lifetime. Any block can still be freed, resized, or relinked onto another scope without copying.

Public operations initialize the module on first use, so they are safe before the runtime calls `Scope.initialize`. Shutdown runs its hooks newest first and leaves the module terminal; `Scope.stats` still reports the final counts.

Tests and examples

`make verify` (`unittest/test-scope.x`) and `make examples` (`docs-tour`).

lib/scripting.x

The modules every script unit includes.

Design notes

A source file whose first line is a `#!` line is a script unit, and the compiler reads that line as an include of this module. Any program may include it to get commands, pipelines, jobs, path operations, argument parsing, digests, regular expressions, and line differences together.

`lib/split.x`

`String` field splitting and repeatable typed cursors.

Primary API

Function	Summary
`Split.iter`	Returns an iterator over a lazy `String` cursor.
`Split.try_next`	Yields the next field and advances a caller-owned position on success.
`Split.var`	Boxes a borrowed descriptor as a `<split> Var`, so a `Split` can cross into compile-time code or a `Var` container.
`String.lines`	Returns a lazy cursor over lines in `str`, with endings removed.
`String.split`	Splits `str` on every occurrence of `sep` into a `List` of `String`s.
`String.split_lines`	Splits `str` into a `List` of lines.
`String.split_n`	Splits `str` at no more than `max_splits` separators.
`String.splits`	Returns a lazy cursor over fields separated by `sep`.
`String.words`	Returns a lazy cursor over whitespace-delimited words in `str`.
`Var.split`	Unboxes a descriptor from a `Var` produced by `Split.var`.

`Split`

Split.iter

```
Iter Split.iter(Split split, Iter dest)
```

Returns an iterator over a lazy `String` cursor. This adapter over `Split.try_next` boxes each field into a `Var` and keeps the `Iter` state in step. Use the typed cursor in hot code and this method when you need the `Iter` protocol.

The cursor descriptor is `Scope`-owned; `dest` is caller-supplied iterator storage. Every iterator keeps its own byte position, so cursors can be nested or traversed concurrently. The `Split` descriptor, its borrowed input `String`s, and `dest` must remain live until iteration ends.

Constructing the iterator does not raise. Pulling may raise `<alloc-fail>` as `Split.try_next` does. A null `dest` returns NULL.

Source: `lib/split.x:278`

Split.try_next

```
int Split.try_next(Split split, int &?cursor, String &?out)
```

Yields the next field and advances a caller-owned position on success. Position must start at zero and thereafter retain only values written by this method. Both it and `out` are written only on success, so an empty `String` field stays distinct from exhaustion and a walk that has ended leaves the last field in place. The position belongs to the caller, so one descriptor supports repeated, nested, and concurrent walks, including alongside a `Split.iter` iterator over the same descriptor. A yielded canonical `String` remains live until its actual owning pool is released.

```
Split words = "ada lovelace".words();
int cursor = 0;
String word;
while (words.try_next(cursor, word)) printf("%s\n", word);
```

This is what `foreach(String word, split)` lowers to; `Split.iter` is the boxing adapter for every other binder.

Raises: `<alloc-fail>` while canonicalizing a nonempty field. `Null` arguments produce exhaustion without raising.

Source: `lib/split.x:252`

Split.var

```
Var Split.var(Split split)
```

Boxes a borrowed descriptor as a `<split>` `Var`, so a `Split` can cross into compile-time code or a `Var` container. The box does not own the descriptor, which stays live only as long as its `Scope`.

Source: `lib/split.x:30`

String

String.lines

```
meta native Split String.lines(String str)
```

Returns a lazy cursor over lines in `str`, with endings removed. LF, CR, and CRLF end a line, with CRLF counted as one ending. The yielded fields agree with `str.split_lines(0)`, including empty interior lines and the rule that a trailing ending does not add a final empty line.

Each yielded field is a canonical `String`. Distinct fields remain resident in the active `String` pool; bracket bulk traversal with `Pool.open` / `Pool.close` and promote retained values when that residency should be temporary. The cursor borrows `str`, whose actual owning pool must remain live through traversal.

Raises: `<alloc-fail>` when the cursor descriptor cannot be allocated. An empty `String` produces an exhausted cursor.

Source: `lib/split.x:214`

String.split

```
meta native List String.split(String str, String sep)
```

Splits `str` on every occurrence of `sep` into a `List` of `String`s. Separators are not coalesced, so adjacent ones produce empty fields and the result holds one more element than the number of separators found. An empty field is the empty `String`, the null pointer. An empty or null `sep` yields a one-element `List` holding `str`, and splitting the empty `String` yields the empty `List`. The canonical fields and `List` remain live until their actual `String` and `List` pools are released.

```
printf("%s\n", "a:b:c".split(":").repr());
printf("%s\n", "a::b".split(":").repr());
```

Raises: the same causes as `String.split_n`.

Source: `lib/split.x:120`

String.split_lines

```
meta native List String.split_lines(String str, int keep_ends)
```

Splits `str` into a `List` of lines. LF, CR, and CRLF all end a line, and CRLF counts as one ending. A nonzero `keep_ends` leaves each line's ending attached to it. A trailing line ending does not produce a final empty line, so text that ends in a newline yields as many lines as it has endings. A null or empty `str` returns `nil`. The canonical fields and `List` remain live until their actual `String` and `List` pools are released.

Raises: `<alloc-fail>` or `<size-limit>` while constructing fields or the result.

Source: `lib/split.x:133`

String.split_n

```
meta native List String.split_n(String str, String sep, int max_splits)
```

Splits `str` at no more than `max_splits` separators. A negative limit splits every occurrence; zero returns `str` as one field. A null `str` returns `nil`. A null or empty `sep` returns `str` as one field. The canonical fields and `List` remain live until their actual `String` and `List` pools are released.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the result.

Source: `lib/split.x:90`

String.splits

```
meta native Split String.splits(String str, String sep)
```

Returns a lazy cursor over fields separated by `sep`. Separators are not coalesced, so adjacent separators produce empty fields. The yielded fields agree with `str.split(sep)`. A null or empty separator yields `str` once, while an empty `str` yields nothing.

Each yielded field is a canonical `String`. Distinct fields remain resident in the active `String` pool; bracket bulk traversal with `Pool.open` / `Pool.close` and promote retained values when that residency should be temporary. The cursor borrows `str` and `sep`; both actual owning pools must remain live through traversal.

Raises: `<alloc-fail>` when the cursor descriptor cannot be allocated.

Source: `lib/split.x:228`

String.words

```
meta native Split String.words(String str)
```

Returns a lazy cursor over whitespace-delimited words in `str`. Runs of C `isspace` bytes are coalesced, leading and trailing whitespace is ignored, and no empty `String` is yielded. `str.splits(" ")` instead preserves empty fields around every explicit separator.

Each yielded field is a canonical `String`. Distinct fields remain resident in the active `String` pool; bracket bulk traversal with `Pool.open` / `Pool.close` and promote retained values when that residency should be temporary. The cursor borrows `str`, whose actual owning pool must remain live through traversal.

Raises: `<alloc-fail>` when the cursor descriptor cannot be allocated. An empty `String` produces an exhausted cursor.

Source: `lib/split.x:199`

Var

Var.split

```
Split Var.split(Var value)
```

Unboxes a descriptor from a `Var` produced by `Split.var`.

Source: `lib/split.x:33`

Public types

Type	Kind	Summary
`Split`	struct	Describes an immutable, repeatable lazy traversal of a borrowed `String`.

Split

```
typedef struct Split *Split
```

Describes an immutable, repeatable lazy traversal of a borrowed `String`. The descriptor is `Scope`-owned and keeps no position; each caller or `Iter` keeps its own cursor. The input and separator are not retained, so their actual `String` pools must remain live through every traversal. No cleanup is needed before the descriptor's `Scope` is released.

Source: `lib/split.x:22`

Design notes

A Split description is immutable and `Scope`-owned. It keeps no position; the caller holds the cursor, so one Split supports repeated, nested, and concurrent walks. Eager and lazy splitting share the same separator and line-ending scans.

Tests and examples

`make verify` (`unittest/test-split.x`).

`lib/string-classify.x`

Byte classification for canonical strings.

Primary API

Function	Summary
`String.contains_digit`	Reports whether `str` contains a decimal digit.
`String.is_alnum`	Reports whether every byte of nonempty `s` is alphanumeric.
`String.is_alnum_under`	Reports whether nonempty `s` is alphanumeric or underscore.
`String.is_alpha`	Reports whether every byte of nonempty `s` is alphabetic.
`String.is_alpha_under`	Reports whether every byte of nonempty `s` is alphabetic or underscore.
`String.is_digit`	Reports whether every byte of nonempty `s` is a decimal digit.
`String.is_identifier`	Reports whether `s` is a C identifier under C character classes.
`String.is_lower`	Reports whether every byte of nonempty `s` is lower-case.
`String.is_lower_under`	Reports whether every byte of nonempty `s` is lower-case or underscore.
`String.is_space`	Reports whether every byte of nonempty `s` is whitespace.
`String.is_upper`	Reports whether every byte of nonempty `s` is upper-case.
`String.is_upper_under`	Reports whether every byte of nonempty `s` is upper-case or underscore.

`String`

String.contains_digit

```
meta native int String.contains_digit(String str)
```

Reports whether `str` contains a decimal digit. A null or empty `String` returns false.

Source: `lib/string-classify.x:17`

String.is_alnum

```
meta native int String.is_alnum(String s)
```

Reports whether every byte of nonempty `s` is alphanumeric.

Source: `lib/string-classify.x:51`

String.is_alnum_under

```
meta native int String.is_alnum_under(String s)
```

Reports whether nonempty `s` is alphanumeric or underscore.

Source: `lib/string-classify.x:56`

String.is_alpha

```
meta native int String.is_alpha(String s)
```

Reports whether every byte of nonempty `s` is alphabetic.

Source: `lib/string-classify.x:36`

String.is_alpha_under

```
meta native int String.is_alpha_under(String s)
```

Reports whether every byte of nonempty `s` is alphabetic or underscore.

Source: `lib/string-classify.x:41`

String.is_digit

```
meta native int String.is_digit(String s)
```

Reports whether every byte of nonempty `s` is a decimal digit.

Source: `lib/string-classify.x:46`

String.is_identifier

```
meta native int String.is_identifier(String s)
```

Reports whether `s` is a C identifier under C character classes. The first byte must be alphabetic or underscore; later bytes may also be decimal digits. A null or empty `String` returns false.

Source: `lib/string-classify.x:64`

String.is_lower

```
meta native int String.is_lower(String s)
```

Reports whether every byte of nonempty `s` is lower-case.

Source: `lib/string-classify.x:79`

String.is_lower_under

```
meta native int String.is_lower_under(String s)
```

Reports whether every byte of nonempty `s` is lower-case or underscore.

Source: `lib/string-classify.x:84`

String.is_space

```
meta native int String.is_space(String s)
```

Reports whether every byte of nonempty `s` is whitespace.

Source: `lib/string-classify.x:74`

String.is_upper

```
meta native int String.is_upper(String s)
```

Reports whether every byte of nonempty `s` is upper-case.

Source: `lib/string-classify.x:89`

String.is_upper_under

```
meta native int String.is_upper_under(String s)
```

Reports whether every byte of nonempty `s` is upper-case or underscore.

Source: `lib/string-classify.x:94`

Design notes

Predicates classify unsigned bytes with C's current locale. The canonical empty `String` is NULL and every predicate reports false for it. No predicate allocates or mutates its input.

Tests and examples

`make verify` (`unittest/test-string-classify.x`).

`lib/string-escape.x`

Escaped spelling of canonical strings.

Primary API

Function	Summary
`String.unescape`	Decodes supported backslash escapes in `str` into a canonical `String`.

`String`

String.unescape

```
meta native String String.unescape(String str)
```

Decodes supported backslash escapes in `str` into a canonical `String`. Standard single-byte escapes, up to two hexadecimal digits after `x`, `u`, or `U`, and up to three octal digits are consumed. A backslash-newline is removed, an unknown escape yields its following byte, and a trailing backslash is dropped. `Null` input returns NULL and input without a backslash is returned unchanged.

Raises: `<bad-arg>` for an octal escape above `\377`, which does not fit a byte, or `<alloc-fail>` while constructing a changed result.

Source: `lib/string-escape.x:122`

Advanced and interop API

Function	Summary
`String.escape`	Returns a canonical escaped representation of the bytes in `str`.
`String.parse`	Returns the canonical unescaped contents of `str`.
`String.parse_char`	Parses one leading single-quoted escaped or literal byte, or returns −1.
`String.repr`	Returns a canonical quoted and escaped representation of `str`.
`String.write_repr`	Appends a quoted escaped representation of `str` to borrowed `out`.

`String`

String.escape

```
meta native String String.escape(String str)
```

Returns a canonical escaped representation of the bytes in `str`. Common control and delimiter bytes use named escapes, printable ASCII is copied, and every other byte uses a three-digit octal escape. `Null` input or an oversized result returns NULL.

Raises: `<alloc-fail>` while constructing the result.

Source: `lib/string-escape.x:31`

String.parse

```
meta native String String.parse(String str)
```

Returns the canonical unescaped contents of `str`. Matching outer `%"..."` or `"..."` delimiters are removed; unquoted input is unescaped directly. `Null` or empty input returns NULL. An unquoted input without backslashes is returned unchanged.

Raises: `<alloc-fail>` while copying or decoding.

Source: `lib/string-escape.x:230`

String.parse_char

`meta native int String.parse_char(String str)`

Parses one leading single-quoted escaped or literal byte, or returns −1. The opening quote, one decoded byte, and a closing quote are required. Text after that closing quote is ignored. A decoded NUL is returned as zero; malformed and null input returns −1, as does an octal escape above `\377`, which does not fit a byte.

Source: `lib/string-escape.x:211`

String.repr

`String String.repr(String str)`

Returns a canonical quoted and escaped representation of `str`. Empty input returns the canonical literal spelling `"\"\""`.

Raises: `<alloc-fail>` while escaping or formatting a nonempty `String`.

Source: `lib/string-escape.x:90`

String.write_repr

`Buffer String.write_repr(String str, Buffer out)`

Appends a quoted escaped representation of `str` to borrowed `out`. Bytes are streamed without first allocating an intermediate `String`. The same `out` is returned and not retained. Text written before a failure remains in the `Buffer`.

Raises: any cause from `Buffer.write_char` or `Buffer.write_len`.

Source: `lib/string-escape.x:101`

Design notes

This module translates between a `String`'s bytes and their escaped spelling inside a C or x2c string literal: `escape`, `repr`, and `write_repr` write it, and `unescape`, `parse`, and `parse_char` read it. A changed result is built in a `String.malloc` buffer and finalized by `String.intern_free`, so the canonical header stays private to `string.x`.

Tests and examples

`make verify` (`unittest/test-string.x`).

lib/string-format.x

Checked formatting of `Var` values into a `String`.

Primary API

Function	Summary
`String.format`	Formats `values` through a checked, C-style subset of `fmt`.

String

String.format

`String String.format(String fmt, List values)`

Formats `values` through a checked, C-style subset of `fmt`. The receiver is decoded runtime text, so this fixed-signature operation is safe to call through the interpreter as `fmt.format(values)`. It supports `%%`, flags `-+ #0`, numeric or `*` width and precision, integer conversions `d i o u x X` with `hh h l ll`, floating conversions `f F e E g G a A` with default, `l`, or `L`, and `%c` and `%s`. Numeric values are converted with `Var.convert`; `%s` uses `Var.str`. Pointer and write-count conversions, wide strings and characters, positional arguments, `j z t` lengths, malformed formats, and missing or excess values are rejected. `%c` also rejects NUL because canonical `String`s cannot contain it. Output is staged privately and no result is published on failure. Formatting follows the process locale.

Raises: `<format>` with byte `offset` and `reason`; numeric and string conversion failures are nested as `cause`. Allocation failures may also transfer while staging or canonicalizing the result.

Source: `lib/string-format.x:94`

Design notes

`String.format` parses a checked subset of C's format syntax itself, converts each `Var` value to the C type its conversion names, and prints it through `Buffer.printf`, so a format read at run time cannot pass C a mismatched argument. `String.printf`, which hands C arguments straight to `vsnprintf`, stays with the other constructors in `string.x`.

Tests and examples

`make verify` (`unittest/test-string.x`).

`lib/string-number.x`

Numeric parsing from canonical byte strings.

Advanced and interop API

Function	Summary
`String.try_double`	Parses all of `str` as a floating-point number, writing `out`.
`String.try_long`	Parses all of `str` as an integer, writing it through `out`.

`String`

String.try_double

```
meta native int String.try_double(String str, double *out)
```

Parses all of `str` as a floating-point number, writing `out`. Returns 1 and writes `out` on success; returns 0 and leaves `out` untouched on failure.

The whole string must be the number, on the same terms as `String.try_long`: leading and trailing whitespace are allowed and any other unconsumed character is a failure, so `1.5junk` does not parse.

It accepts what C's `strtod` accepts, including hex-float literals such as `0x1p4` and the `inf` and `nan` spellings. A value that overflows or underflows `double` fails. It does not know the `0o` and `0b` prefixes that the integer parser adds, so `0b101` fails instead of yielding the leading zero.

Source: `lib/string-number.x:87`

String.try_long

```
meta native int String.try_long(String str, long *out)
```

Parses all of `str` as an integer, writing it through `out`. Returns 1 and writes `out` on success; returns 0 and leaves `out` untouched on failure.

The whole string must be the number. Leading and trailing whitespace are allowed and any other unconsumed character is a failure, so `42` succeeds and `42junk` does not. A `+` or `-` sign is accepted, and the radix prefixes `0x`, `0o`, `0b`, and C's leading-zero octal are recognized. A value outside the range of `long` fails.

```
printf("%d %ld\n", " -0b101 ".try_long(&value), value);
printf("%d\n", "42junk".try_long(&value));
```

Source: `lib/string-number.x:37`

Design notes

The readers require a complete parse and leave caller output untouched on failure. Parsing allocates no storage.

`lib/string/string.x`

Canonical byte strings and core text operations.

Primary API

Function	Summary
`String.add`	Returns `str` followed by `other`.
`String.capitalize`	Upper-cases the first byte of `str` and lower-cases the remainder.
`String.contains`	Reports whether the bytes of `sub` occur anywhere in `str`.
`String.count`	Returns the non-overlapping count of `sub` in `str`.
`String.dedent`	Removes the indentation the text was written with.
`String.endswith`	Reports whether `str` ends with `suffix`.
`String.filter`	Returns the bytes of `str` accepted by ordinary `Var` truthiness of `fn`.
`String.find`	Returns the index of the first occurrence of `sub` in `str`, or −1.
`String.find_all`	Returns each non-overlapping starting index at which `sub` occurs.
`String.find_within`	Returns the first index of `sub` within `str[start:end]`, or −1.
`String.getindex`	Returns the byte at `index` in `str` as an int, or −1 if out of range.
`String.getslice`	Returns the canonical `String` `s[start:stop:step]`.
`String.iter`	Initializes `dest` as a lazy iterator over the bytes of `x`.
`String.join`	Joins `strings` into one canonical `String` with `sep` between elements.
`String.keep`	Returns a canonical `String` containing only bytes found in `chars`.
`String.len`	Returns the byte length of `str`, excluding the terminating NUL.
`String.lower`	Returns `str` with every upper-case byte lowered.
`String.lstrip`	Removes leading bytes found in the C string `negChars`.
`String.map`	Returns `str` with `fn` applied to every byte.
`String.new`	Returns the canonical `String` holding the bytes of the C string `str`.
`String.pad_center`	Pads both sides of `str` to the requested width.
`String.pad_left`	Pads the left side of `str` to the requested width.
`String.pad_right`	Pads the right side of `str` to the requested width.
`String.partition`	Splits `str` at the first `sep` into a three-element `List`.
`String.reject`	Returns a canonical `String` after removing bytes found in `chars`.
`String.remove_prefix`	Removes `prefix` when `str` starts with it and returns a canonical `String`.
`String.remove_suffix`	Removes `suffix` when `str` ends with it and returns a canonical `String`.
`String.repeat`	Returns a canonical `String` containing `count` copies of `str`.
`String.replace`	Returns `str` with every occurrence of `old` replaced by `replacement`.
`String.replace_n`	Replaces at most `max_replacements` non-overlapping occurrences of `old`.
`String.rfind`	Returns the index of the last occurrence of `sub` in `str`, or −1.
`String.rpartition`	Splits `str` around its final occurrence of `sep`.
`String.rstrip`	Removes trailing bytes found in the C string `negChars`.
`String.squeeze`	Collapses adjacent runs of each byte listed in `chars`.
`String.startswith`	Reports whether `str` starts with `prefix`.
`String.strip`	Returns `str` with leading and trailing bytes in `negChars` removed.
`String.try_next`	Writes the next byte, advances `cursor`, and returns one.
`String.upper`	Returns `str` with every lower-case byte raised.
`String.withindex`	Returns a canonical copy of `str` with the byte at `index` set.
`String.write_str`	Appends `str` to `out` unchanged.

`String`

String.add

```
String String.add(String str, String other)
```

Returns `str` followed by `other`. This implements the `add` row of `protocol Var(String)`, so it is what `+` on two `String`s lowers to, and because the result is interned, `==` on a newly built result is a content comparison. Either operand may be the empty `String`, in which case the other pointer is returned unchanged with its existing ownership and canonical or transient state.

Every concatenation interns its result, so building text by repeated concatenation in a loop allocates and hashes at every step. Use `Buffer` for that, or collect the pieces and call `String.join` once.

Raises: `<size-limit>` when the result cannot fit the `String` representation. Allocation failures propagate from the owning allocator.

Source: `lib/string/building.x:18`

String.capitalize

```
meta native String String.capitalize(String str)
```

Upper-cases the first byte of `str` and lower-cases the remainder. Mapping is bytewise through C's `toupper` and `tolower`. `Null`, empty, and unchanged inputs are returned as-is.

Raises: `<alloc-fail>` while constructing the result.

Source: `lib/string/mapping.x:54`

String.contains

```
meta native int String.contains(String str, String sub)
```

Reports whether the bytes of `sub` occur anywhere in `str`. An empty `sub` is contained in every `String`, including the empty one, so a truth test on user-supplied text should check for emptiness separately if that matters.

Source: `lib/string/search.x:119`

String.count

```
meta native int String.count(String str, String sub)
```

Returns the non-overlapping count of `sub` in `str`. A null or empty `str` or `sub` returns zero.

Source: `lib/string/search.x:96`

String.dedent

```
meta native String String.dedent(String str)
```

Removes the indentation the text was written with. The prefix is the run of spaces and tabs that opens the first content line, after one leading newline is dropped. Every following line that starts with that prefix loses exactly it, so indentation written past the prefix survives and the block renormalizes as a unit. A line that does not carry the prefix, including a blank one, is left alone, except that a final line of only spaces and tabs is removed so the closing quote's own indentation does not reach the result. `\r\n` is preserved because only the prefix after a newline is removed.

Raises: `<alloc-fail>` while constructing a changed result. Null input returns NULL and text with no prefix is returned as-is.

Source: `lib/string/splitting.x:62`

String.endswith

```
meta native int String.endswith(String str, String suffix)
```

Reports whether `str` ends with `suffix`. An empty `suffix` is a suffix of every `String`. The comparison is bytewise, so this is a safe test for a file extension but not for a case-insensitive one; lower both sides first.

Source: `lib/string/search.x:142`

String.filter

```
String String.filter(String str, Func fn)
```

Returns the bytes of `str` accepted by ordinary `Var` truthiness of `fn`. Each byte is boxed from `char` and passed by value. A null or empty `str` returns `str` without invoking the callback. Otherwise `fn` is called once per byte from left to right and is not retained. A null `fn` raises `<bad-arg>` on the first invocation.

Raises: whatever `Func.apply`, `fn`, or the returned `Var`'s truth operation raises, or `<alloc-fail>` when the result cannot be allocated.

Source: `lib/string/mapping.x:81`

String.find

```
meta native int String.find(String str, String sub)
```

Returns the index of the first occurrence of `sub` in `str`, or −1. The search compares bytes, so an index may land inside a multibyte sequence. An empty `sub` matches at index 0. Use `String.find_within` to bound the search to a range, or `String.rfind` to scan from the end.

Source: `lib/string/search.x:10`

String.find_all

```
meta native List String.find_all(String str, String sub, int start, int end)
```

Returns each non-overlapping starting index at which `sub` occurs. The search begins at `start` and uses the same normalized exclusive `end` as `String.find_within`. `Null` or empty `str` or `sub` returns `nil`. The result is a canonical `List` whose cells follow their owning pools.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the result.

Source: `lib/string/search.x:81`

String.find_within

```
meta native int String.find_within( String str, String sub, int start, int end)
```

Returns the first index of `sub` within `str[start:end]`, or −1. The returned index is absolute and counts from the start of `str`. Negative `start` and `end` count from the end of `str`, and both are then clamped to the `String`.

An `end` of −1 is the sentinel for "to the end of `str`", not "one byte before the end". There is therefore no negative `end` that excludes only the last byte; pass a nonnegative `end` for that. An empty `sub` matches at the normalized `start`.

```
printf("%d %d %d\n", text.find_within("c", 0, -1),
       text.find_within("c", 0, 2), text.find_within("a", -3, -1));
```

Source: `lib/string/search.x:29`

String.getindex

```
int String.getindex(String str, int index)
```

Returns the byte at `index` in `str` as an int, or −1 if out of range. This is what `str[index]` lowers to on a canonical `String`, so indexing yields an int byte value. A negative `index` counts from the end, so −1 is the last byte.

The byte is unsigned, so the result is 0 through 255 on every platform and −1 means out of range and nothing else. An `index` at or beyond the length is out of range.

Source: `lib/string/editing.x:83`

String.getslice

```
meta native String String.getslice(String s, int start, int stop, int step)
```

Returns the canonical `String` `s[start:stop:step]`. This is what slice syntax lowers to. `stop` is exclusive, negative `start` and `stop` count from the end, and a negative `step` walks backwards, so `s[::-1]` reverses. Indices are byte positions, so a slice can split a multibyte sequence. A full unit-step slice may return `s`; other nonempty slices return their canonical `String`.

Raises: `<alloc-fail>` while constructing a nonempty result. An empty range, a range that runs the wrong way for its `step`, or a `step` of zero also returns NULL, the empty `String`, without raising.

Source: `lib/string/editing.x:99`

String.iter

```
Iter String.iter(String x, Iter dest)
```

Initializes `dest` as a lazy iterator over the bytes of `x`. The caller owns `dest`; it borrows `x`, which must remain live through traversal. A null `dest` returns NULL, and null `x` is exhausted. Each pull yields the next byte as an `<i32>` `Var` in index order. The function retains neither argument. Foreach may convert each yielded byte to either `int` or `char`:

```
foreach (int byte, "abc") printf("%d\n", byte);
foreach (char ch, "abc") printf("%c\n", ch);
```

This is byte traversal, not Unicode character iteration.

Source: `lib/string/adapters.x:44`

String.join

```
meta native String String.join(String sep, List strings)
```

Joins `strings` into one canonical `String` with `sep` between elements. The receiver is the separator, not the sequence, so this reads `sep.join(parts)`. A null `sep` joins with nothing between elements. Empty elements contribute no bytes but still count as positions, so joining a split result with the same separator reproduces the original text, adjacent separators included. `List` elements that are not `String`s convert to the empty `String`.

Raises: `<alloc-fail>` when result storage cannot be allocated. A null or empty `List`, or an oversized result, also returns NULL, the empty `String`, without raising.

Source: `lib/string/building.x:83`

String.keep

```
meta native String String.keep(String str, String chars)
```

Returns a canonical `String` containing only bytes found in `chars`. `Null` `str` returns NULL; null `chars` returns NULL for any nonnull input.

Raises: `<alloc-fail>` while constructing the result.

Source: `lib/string/mapping.x:119`

String.len

```
int String.len(String str)
```

Returns the byte length of `str`, excluding the terminating NUL. Constant time: the length is cached in the `String`'s private header. Lengths count bytes, so a multibyte UTF-8 sequence counts once per byte. On a transient `String.malloc` buffer this reports the writable byte count.

The empty `String` is the null pointer, whose length is 0.

Source: `lib/string/storage.x:43`

String.lower

```
meta native String String.lower(String str)
```

Returns `str` with every upper-case byte lowered. Case mapping runs byte by byte through C's `tolower`, so it covers ASCII in the default locale and does not case-fold multibyte text. When no byte would change, `str` itself is returned after the unchanged temporary buffer is released.

Raises: `<alloc-fail>` while constructing the result.

Source: `lib/string/mapping.x:35`

String.lstrip

```
String String.lstrip(String str, char *negChars)
```

Removes leading bytes found in the C string `negChars`. Passing NULL uses `" \t\n\v\f\r"`. The result is canonical; null input returns NULL and an unchanged input is returned as-is.

Raises: `<alloc-fail>` while constructing a changed result.

Source: `lib/string/splitting.x:29`

String.map

```
String String.map(String str, Func fn)
```

Returns `str` with `fn` applied to every byte. Each byte is boxed from `char` and passed by value. Each result is converted to `int` and truncated to the byte that is stored, and that byte is what is checked, so a result such as 256 raises because its stored byte would be NUL. A null or empty `str` returns `str` without invoking the callback. Otherwise `fn` is called once per byte from left to right and is not retained. Each result must convert to a non-NUL byte. A null `fn` raises `<bad-arg>` on the first invocation.

Raises: whatever `Func.apply`, `fn`, or result conversion raises, `<bad-result>` when the converted result is zero, or `<alloc-fail>` when the result cannot be allocated.

Source: `lib/string/mapping.x:99`

String.new

```
String String.new(const char *str)
```

Returns the canonical `String` holding the bytes of the C string `str`. The input is borrowed and copied, so `str` may be a stack buffer and mutating it afterwards does not disturb the result. Equal nonempty content visible in the active pool chain yields the same pointer, which is why `==` on canonical `String`s from that chain is a content comparison. A detached or sibling pool may hold a distinct equal pointer. Empty input canonicalizes to the null pointer, the empty `String`'s only representation.

Raises: `<alloc-fail>` when canonical storage cannot be allocated. A null, empty, or oversized input returns NULL, which is indistinguishable from the empty `String`, without raising.

Source: `lib/string/storage.x:100`

String.pad_center

```
meta native String String.pad_center(String str, int width, char fill)
```

Pads both sides of `str` to the requested width.

Raises: the same causes as `String.pad_left`.

Source: `lib/string/building.x:146`

String.pad_left

```
meta native String String.pad_left(String str, int width, char fill)
```

Pads the left side of `str` to the requested width.

Raises: `<bad-arg>` when `fill` is NUL, `<size-limit>` when `width` cannot be represented, or `<alloc-fail>` when result storage cannot be allocated.

Source: `lib/string/building.x:134`

String.pad_right

```
meta native String String.pad_right(String str, int width, char fill)
```

Pads the right side of `str` to the requested width.

Raises: the same causes as `String.pad_left`.

Source: `lib/string/building.x:140`

String.partition

```
meta native List String.partition(String str, String sep)
```

Splits `str` at the first `sep` into a three-element `List`. The elements are the text before the separator, the separator itself, and the text after it. When `sep` does not occur, the result is `str` followed by two empty `String`s, so the shape is three elements either way and a caller can destructure it without testing for the separator first. A leading separator gives an empty first element and a missing separator gives two empty trailing elements, so compare the middle element against `sep` if you need to tell them apart. The result and any new substrings are canonical and follow their owning `List` and `String` pools. Unchanged `str` and `sep` elements are borrowed into the result, so a transient input must outlive the returned `List`.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the result.

Source: `lib/string/splitting.x:117`

String.reject

```
meta native String String.reject(String str, String chars)
```

Returns a canonical `String` after removing bytes found in `chars`. `Null` or empty `str`, or null `chars`, returns `str` unchanged.

Raises: `<alloc-fail>` while constructing a changed result.

Source: `lib/string/mapping.x:130`

String.remove_prefix

```
meta native String String.remove_prefix(String str, String prefix)
```

Removes `prefix` when `str` starts with it and returns a canonical `String`. A null or absent prefix returns `str` unchanged; removing the complete `String` returns NULL.

Raises: `<alloc-fail>` while constructing a changed result.

Source: `lib/string/splitting.x:89`

String.remove_suffix

```
meta native String String.remove_suffix(String str, String suffix)
```

Removes `suffix` when `str` ends with it and returns a canonical `String`. A null or absent suffix returns `str` unchanged; removing the complete `String` returns NULL.

Raises: `<alloc-fail>` while constructing a changed result.

Source: `lib/string/splitting.x:99`

String.repeat

```
meta native String String.repeat(String str, int count)
```

Returns a canonical `String` containing `count` copies of `str`. `Null` or empty input, a nonpositive count, or an oversized result returns NULL. A count of one returns `str` when it is already visible in the active pool; a transient buffer is copied and interned instead.

Raises: <alloc-fail> while constructing a nonempty result.

Source: lib/string/building.x:42

String.replace

```
meta native String String.replace( String str, String old, String replacement)
```

Returns str with every occurrence of old replaced by replacement. Scanning runs left to right and matches do not overlap: each one resumes after the text it consumed, so replacing aa in aaaa performs two replacements, not three. Replaced text is never rescanned, so a replacement that contains old does not loop. An empty replacement deletes the matches.

When there is nothing to replace, str itself is returned, so the result may be the same pointer as the input.

Raises: <alloc-fail> when result storage cannot be allocated. An empty or null old returns str, and an oversized result returns NULL, without raising.

Source: lib/string/editing.x:18

String.replace_n

```
meta native String String.replace_n( String str, String old, String replacement, int max_replacements)
```

Replaces at most max_replacements non-overlapping occurrences of old. Scanning proceeds left to right and does not rescan replacement text. A negative limit replaces all matches; zero, null or empty old, null input, or no match returns str unchanged. Null replacement deletes matches. An oversized result returns NULL.

Raises: <alloc-fail> while constructing a changed result.

Source: lib/string/editing.x:29

String.rfind

```
meta native int String.rfind(String str, String sub)
```

Returns the index of the last occurrence of sub in str, or −1. Scanning runs backwards from the end, and the returned index still measures from the start of str. An empty sub reports the length of str, matching after the last byte and mirroring the forward search reporting 0.

Source: lib/string/search.x:67

String.rpartition

```
meta native List String.rpartition(String str, String sep)
```

Splits str around its final occurrence of sep. The three elements are the text before the separator, the separator itself, and the text after it. When sep is null or absent, two empty Strings precede str. New substrings are canonical; the result follows its owning List and String pools. Unchanged str and sep elements are borrowed into the result, so a transient input must outlive the returned List.

Raises: <alloc-fail> or <size-limit> while constructing the result.

Source: lib/string/splitting.x:137

String.rstrip

```
String String.rstrip(String str, char *negChars)
```

Removes trailing bytes found in the C string negChars. Passing NULL uses " \t\n\v\f\r". The result is canonical; null input returns NULL and an unchanged input is returned as-is.

Raises: <alloc-fail> while constructing a changed result.

Source: lib/string/splitting.x:42

String.squeeze

```
meta native String String.squeeze(String str, String chars)
```

Collapses adjacent runs of each byte listed in `chars`. Bytes outside `chars` are preserved even when repeated. `Null` or empty `str`, or null `chars`, returns `str` unchanged.

Raises: `<alloc-fail>` while constructing a changed result.

Source: `lib/string/mapping.x:140`

String.startswith

```
meta native int String.startswith(String str, String prefix)
```

Reports whether `str` starts with `prefix`. An empty `prefix` is a prefix of every `String`. `String.remove_prefix` performs the same test and returns the remainder, so there is rarely a reason to run both.

Source: `lib/string/search.x:131`

String.strip

```
String String.strip(String str, char *negChars)
```

Returns `str` with leading and trailing bytes in `negChars` removed. `negChars` is a NUL-terminated C string listing the bytes to remove, not a substring and not a pattern; order and repetition in it are irrelevant. Passing NULL uses the default whitespace set " \t\n\v\f\r". Trimming stops at each end on the first byte not in the set, and `str` itself is returned when nothing is trimmed.

Raises: `<alloc-fail>` while constructing the result. A `String` made entirely of removable bytes trims to NULL, the empty `String`, without raising.

Source: `lib/string/splitting.x:15`

String.try_next

```
int String.try_next(String str, int &?cursor, int &?out)
```

Writes the next byte, advances `cursor`, and returns one. Initialize the caller-owned cursor to zero. A null `String`, a null pointer, a negative cursor, or exhaustion returns zero without changing `cursor` or `out`. This is byte traversal, not Unicode characters.

`foreach (int byte, str)` and `foreach (char ch, str)` compile to this loop.

Source: `lib/string/adapters.x:65`

String.upper

```
meta native String String.upper(String str)
```

Returns `str` with every lower-case byte raised. Like `String.lower`, mapping runs byte by byte through C's `toupper` and covers ASCII in the default locale. `str` itself is returned when nothing would change.

Raises: `<alloc-fail>` while constructing the result.

Source: `lib/string/mapping.x:45`

String.withindex

```
meta native String String.withindex(String str, int index, char value)
```

Returns a canonical copy of `str` with the byte at `index` set. Canonical `Strings` are immutable, so `str` is not modified. When the byte already has `value`, this may return `str`; otherwise a new `String` is built and interned. Native assignment `str[index] = value` writes through shared canonical storage and belongs only on a transient `String.malloc` buffer. A negative `index` counts from the end.

```
String word = "hello";
String capital = word.withindex(0, 'H');
printf("%s %s\n", word, capital);
```

Raises: <bad-arg> when value is NUL, which a String cannot contain, or <alloc-fail> while constructing the result. An out-of-range index returns str unchanged.

Source: lib/string/editing.x:127

String.write_str

```
Buffer String.write_str(String str, Buffer out)
```

Appends str to out unchanged. A String is already its own display text, so this writes it directly instead of routing through String.str. Null str is a no-op. The borrowed out is returned and not retained.

Raises: any cause from Buffer.write.

Source: lib/string/adapters.x:16

Advanced and interop API

Function	Summary
String.compare	Compares x and y bytewise, returning negative, zero, or positive.
String.equal	Reports whether x and y contain the same bytes.
String.free	Releases a transient String.malloc buffer early.
String.hash	Returns the content hash of str, or zero for the empty String.
String.intern	Returns the canonical String for the borrowed bytes of string.
String.intern_free	Finalizes an owned String.malloc buffer into a canonical String.
String.malloc	Allocates a transient mutable buffer of len bytes, not a String.
String.new_fill	Returns the canonical String containing count copies of fill.
String.new_in	Returns the canonical String for at most length borrowed bytes in pool.
String.new_len	Returns the canonical String holding at most len bytes of str.
String.printf	Formats a canonical String from fmt and the trailing arguments.
String.promote	Moves str from the active pool to its parent and returns the same pointer.
String.str	Returns str itself as its display String without copying or retaining it.
String.symbol	Returns the compact Symbol encoded from str, or zero for empty input.

String

String.compare

```
meta native int String.compare(String x, String y)
```

Compares x and y bytewise, returning negative, zero, or positive. The ordering is C's strcmp on the raw bytes, so it is neither locale-aware nor Unicode collation, and only the sign of the result is meaningful. The empty String, being the null pointer, sorts before every non-empty String, and two empty Strings compare equal.

Source: lib/string/storage.x:78

String.equal

```
int String.equal(String x, String y)
```

Reports whether x and y contain the same bytes. Interning already makes x == y a content test for two canonical Strings, so use this when one side may be a transient String.malloc buffer or when null has to compare cleanly. Two nulls are equal, since the null pointer is the empty String, and a null equals no non-empty String.

Source: lib/string/storage.x:66

String.free

```
meta native void String.free(String str)
```

Releases a transient `String.malloc` buffer early. The pointer is first looked up in the intern table, including the enclosing pools, and the call does nothing if it is the canonical `String` visible from the active pool. Only a transient buffer allocated in the active pool chain is released; a canonical pointer from a detached or unrelated pool is not a valid argument.

A null argument is ignored.

Source: `lib/string/storage.x:302`

String.hash

```
meta native unsigned String.hash(String str)
```

Returns the content hash of `str`, or zero for the empty `String`. Canonical `Strings` use the cached hash; transient buffers are hashed from their current NUL-terminated contents.

Source: `lib/string/storage.x:53`

String.intern

```
meta native Self String.intern(Self string)
```

Returns the canonical `String` for the borrowed bytes of `string`. This is `String.new` under another name. The bytes are copied into canonical storage and the caller keeps ownership of whatever buffer it passed in, so a stack array or a C library return value is a fine argument. It is not the finalizer for a `String.malloc` buffer: handing one here interns a second copy and leaves the buffer for the caller to free. Use `String.intern_free` for an owned buffer.

Raises: `<alloc-fail>` when canonical storage cannot be allocated. `Null` or empty input returns NULL, the empty `String`, without raising.

Source: `lib/string/storage.x:149`

String.intern_free

```
Self String.intern_free(Self string)
```

Finalizes an owned `String.malloc` buffer into a canonical `String`. This takes ownership. The length and hash are computed from the buffer's NUL-terminated contents, then the buffer either becomes the canonical `String` for that content or, when an equal canonical `String` already exists, is released in favor of the existing one. Either way the argument must not be used again; keep the return value. Passing an already canonical `String` visible in the active pool chain is a harmless no-op that returns it unchanged. The argument and its storage must be visible in that chain; a canonical `String` from a detached or sibling pool is not valid here because installing its pointer would outlive its storage.

```
String canonical = buf.intern_free();
printf("%s %d\n", canonical, canonical == "abcde");
```

Raises: `<alloc-fail>` when the canonical value cannot be registered. A null argument returns NULL, and a buffer empty at its first byte is released and reported as NULL, the empty `String`, without raising.

Source: `lib/string/storage.x:246`

String.malloc

```
String String.malloc(int len)
```

Allocates a transient mutable buffer of `len` bytes, not a `String`. The byte count includes room for the terminating NUL, so `len - 1` bytes are writable and `String.len` on a fresh buffer reports `len - 1`. The buffer is not interned and has no cached hash, so == against a canonical `String` is meaningless until it is finalized. Fill it with native

indexing, then call `String.intern_free` to canonicalize and release it, or `String.free` to discard it. The backing allocation belongs to the active `String`/`List` pool and is invalidated when that pool is released, even though the caller controls finalization.

```
String buf = String.malloc(6);
printf("%d writable bytes\n", buf.len());
```

Raises: `<alloc-fail>` when storage cannot be allocated. A nonpositive or oversized `len` returns NULL without raising.

Source: `lib/string/storage.x:213`

String.new_fill

```
meta native String String.new_fill(char fill, int count)
```

Returns the canonical `String` containing `count` copies of `fill`.

Raises: `<bad-arg>` when `fill` is NUL, `<size-limit>` when `count` is `INT_MAX`, because the allocation includes one trailing NUL byte, or `<alloc-fail>` when canonical storage cannot be allocated. A nonpositive `count` returns NULL without raising.

Source: `lib/string/building.x:62`

String.new_in

```
String String.new_in(Pool pool, const char *bytes, int length)
```

Returns the canonical `String` for at most `length` borrowed bytes in `pool`. The active pool does not change. Copying stops at the first NUL. An existing equal `String` in `pool` or an ancestor is returned with that owner's lifetime; otherwise the new value is owned by `pool`. A null argument, nonpositive length, or empty input returns NULL.

Raises: `<alloc-fail>`, `<size-limit>`, or `<invariant>` while interning.

Source: `lib/string/storage.x:132`

String.new_len

```
String String.new_len(const char *str, int len)
```

Returns the canonical `String` holding at most `len` bytes of `str`. Copying stops at `len` bytes or at the first NUL, whichever comes first, because a `String` cannot carry embedded NUL bytes: `String.new_len("ab\0cd", 5)` is the two-byte `String` `ab`. `str` need not be NUL-terminated, so it may be bounded C input or a window into a larger buffer.

Raises: `<alloc-fail>` when canonical storage cannot be allocated. A null `str`, nonpositive `len`, or leading NUL returns NULL, the empty `String`, without raising.

Source: `lib/string/storage.x:117`

String.printf

```
String String.printf(String fmt, ...)
```

Formats a canonical `String` from `fmt` and the trailing arguments. The receiver is the format `String`, so format-dependent construction reads `%"%-12s %.2f".printf(name, score)`. Conversions, promotion rules, and argument matching are C's, since the work is done by `vsnprintf`; canonical `String`s are NUL-terminated and satisfy `%s` directly. Prefer `%"$name has ${name.len()} bytes"` when interpolation already says what you want, and `String.format` when the arguments must be checked.

Raises: `<alloc-fail>` when result storage cannot be allocated. An empty `fmt` or formatting error also returns NULL without raising. Arguments that do not match the conversions are undefined behavior as in C.

Source: `lib/string/building.x:177`

String.promote

```
Self String.promote(Self str)
```

Moves `str` from the active pool to its parent and returns the same pointer. Empty, transient, and ancestor-owned `String`s are returned unchanged.

Raises: `<alloc-fail>`, `<size-limit>`, or `<invariant>` while recording the promotion.

Source: `lib/string/storage.x:325`

String.str

```
String String.str(String str)
```

Returns `str` itself as its display `String` without copying or retaining it.

Source: `lib/string/adapters.x:8`

String.symbol

```
meta native Symbol String.symbol(String str)
```

Returns the compact `Symbol` encoded from `str`, or zero for empty input. `Symbol`'s restricted spelling folds case and _ with -; other spellings use seven-bit bytes, and input beyond the selected encoding's capacity is truncated. Use `Symbol.try_new` when every byte must be preserved.

Source: `lib/string/adapters.x:23`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`String.is_permanent`	Asks whether `str` already outlives every canonical `String` pool.
`String.try_own`	Proves `str` safe beyond every active canonical `String` pool.

`String`

String.is_permanent

```
int String.is_permanent(String str)
```

Asks whether `str` already outlives every canonical `String` pool. Returns 1 when `str` is empty or the outermost pool owns it, and 0 for a transient buffer or a `String` a nested pool can still reclaim. Unlike `String.try_own` it neither promotes nor allocates, so a caller may ask about a `String` it does not own.

Source: `lib/string/storage.x:350`

String.try_own

```
int String.try_own(String str)
```

Proves `str` safe beyond every active canonical `String` pool. Returns 1 when `str` is empty, already permanent, or can be promoted to the outermost pool. Returns 0 when no active pool owns it. Promotion proceeds one pool at a time; if a later step fails, the earlier promotions remain.

Raises: `<alloc-fail>`, `<size-limit>`, or `<invariant>` while recording a promotion.

Source: `lib/string/storage.x:339`

Public types

Type	Kind	Summary
`String`	alias	An immutable canonical NUL-terminated byte string, or NULL for empty.

String

`typedef char *String`

An immutable canonical NUL-terminated byte string, or NULL for empty. Equal nonempty contents visible in one `String`/`List` pool chain share a pointer. Assignment borrows that pointer; it does not copy or extend the owning pool's lifetime. `String.malloc` is the mutable exception; its backing allocation still belongs to the active pool, and must be finalized with `String.intern_free` or released with `String.free`.

Source: `lib/string/string.x:29`

Design notes

String owns canonical immutable byte strings. Equal nonempty contents in one pool chain share one pointer, the empty String is native zero, and a private header before the bytes holds the exact length and a cached content hash. A String cannot hold an embedded NUL, and every operation here works on bytes, without Unicode character semantics.

A canonical String belongs to the pool that interned it or to an ancestor, and releasing that pool invalidates it unless it was promoted. A transient `String.malloc` buffer belongs to the active pool until it is interned or freed. Pool storage and registration causes transfer and never return to the constructing operation.

Tests and examples

`make verify` (`unittest/test-string.x`) and `make examples` (`docs-word-count`).

lib/symbol.x

Immediate encoded names.

Primary API

Function	Summary
`Symbol.last`	Returns the final decoded byte of `symbol`, or NUL for zero.
`Symbol.len`	Returns the number of decoded bytes in `symbol`.
`Symbol.new`	Encodes the nonnull NUL-terminated spelling `str` as a compact `Symbol`.
`Symbol.try_new`	Encodes `spelling` only when the `Symbol` preserves every byte.
`Symbol.write_str`	Appends the decoded spelling of `symbol` to `out`.

Symbol

Symbol.last

`meta native char Symbol.last(Symbol symbol)`

Returns the final decoded byte of `symbol`, or NUL for zero.

Source: `lib/symbol.x:245`

Symbol.len

`meta native int Symbol.len(Symbol symbol)`

Returns the number of decoded bytes in `symbol`.

Source: `lib/symbol.x:122`

Symbol.new

`Symbol Symbol.new(const char *str)`

Encodes the nonnull NUL-terminated spelling `str` as a compact `Symbol`. Encoding and truncation follow `Symbol.new_len`.

Source: `lib/symbol.x:103`

Symbol.try_new

```
meta native int Symbol.try_new(String spelling, Symbol *out)
```

Encodes `spelling` only when the `Symbol` preserves every byte. Returns 1 and writes `out` on success; returns 0 and leaves `out` untouched when case folding, _/- folding, or truncation would change the spelling. The null `String` is the empty `Symbol`.

Raises: `<alloc-fail>` while checking the decoded spelling.

Source: `lib/symbol.x:111`

Symbol.write_str

```
Buffer Symbol.write_str(Symbol symbol, Buffer out)
```

Appends the decoded spelling of `symbol` to `out`. The bytes go straight into `out`, allocating no `String`. A zero `Symbol` appends nothing.

Raises: `<size-limit>` or `<alloc-fail>` when `out` cannot grow.

Source: `lib/symbol.x:204`

Advanced and interop API

Function	Summary
`Symbol.compare`	Compares decoded `Symbol` spellings bytewise.
`Symbol.decode`	Decodes `symbol` into caller-owned byte storage.
`Symbol.first`	Returns the first decoded byte of `symbol`, or NUL for zero.
`Symbol.new_len`	Encodes at most `len` bytes of `str` as a compact `Symbol`.
`Symbol.parse`	Parses the first compact `Symbol` spelling from `text`.
`Symbol.repr`	Returns the canonical readable representation of `symbol`.
`Symbol.str`	Returns the decoded spelling as a canonical `String`.
`Symbol.write_repr`	Appends the readable representation of `symbol` to `out`.

`Symbol`

Symbol.compare

```
meta native int Symbol.compare(Symbol a, Symbol b)
```

Compares decoded `Symbol` spellings bytewise. Zero sorts before nonzero values. Equal decoded lengths and bytes are ordered by the encoded value, so distinct encodings still have a total order. The result is −1, 0, or 1.

Source: `lib/symbol.x:173`

Symbol.decode

```
void Symbol.decode(Symbol symbol, char *dest)
```

Decodes `symbol` into caller-owned byte storage. `dest` must hold at least `SYMBOL_MAX_5BIT + 1` bytes. A nonzero `Symbol` is NUL-terminated there. A null destination or zero `Symbol` leaves storage unchanged.

Source: `lib/symbol.x:133`

Symbol.first

```
meta native char Symbol.first(Symbol symbol)
```

Returns the first decoded byte of `symbol`, or NUL for zero.

Source: `lib/symbol.x:234`

Symbol.new_len

```
Symbol Symbol.new_len(const char *str, int len)
```

Encodes at most `len` bytes of `str` as a compact `Symbol`. A null `str` or nonpositive `len` returns zero. All `len` readable bytes select the encoding before the result is truncated to ten restricted or seven general bytes. The restricted encoding folds ASCII case and treats underscore as hyphen; the general encoding retains each low seven bits, including embedded NUL. `String` and `Buffer` conversions stop at the first decoded NUL.

Source: `lib/symbol.x:77`

Symbol.parse

```
Symbol Symbol.parse(char *text)
```

Parses the first compact `Symbol` spelling from `text`. The parser accepts an angled literal or a bare `Atom` prefix and ignores trailing text. Quoted angled literals use `String` escape rules. The parsed bytes then take `Symbol.new_len` folding and truncation. `Null`, empty, or malformed input returns zero. Zero is also the empty `Symbol`.

Raises: `<alloc-fail>` while unescaping a quoted literal.

Source: `lib/symbol.x:261`

Symbol.repr

```
String Symbol.repr(Symbol symbol)
```

Returns the canonical readable representation of `symbol`. Zero becomes `<>`; other values use the form emitted by `Symbol.write_repr`. The result follows the canonical pool-chain lifetime described by `Symbol.str`.

Raises: `<alloc-fail>` while constructing the result.

Source: `lib/symbol.x:193`

Symbol.str

```
String Symbol.str(Symbol symbol)
```

Returns the decoded spelling as a canonical `String`. Conversion stops at the first decoded NUL. The result follows the canonical pool chain: it may already belong to an ancestor and lives until its actual owning pool is released. A zero `Symbol` returns NULL, the empty `String`.

Raises: `<alloc-fail>` while canonicalizing the spelling.

Source: `lib/symbol.x:161`

Symbol.write_repr

```
Buffer Symbol.write_repr(Symbol symbol, Buffer out)
```

Appends the readable representation of `symbol` to `out`. Zero writes `<>`; restricted values use a bare angled spelling and general values use a quoted angled spelling, escaping backslash and double quote.

Raises: `<size-limit>` or `<alloc-fail>` when `out` cannot grow.

Source: `lib/symbol.x:216`

Design notes

`Symbol` stores closed-vocabulary names directly in an integer. Restricted spellings use ten 5-bit characters; other spellings use seven 7-bit bytes. The restricted form folds ASCII case and aliases underscore with hyphen; the general form retains only the low seven bits. Construction truncates beyond the selected capacity. Use `Atom` when every input byte must round-trip.

Tests and examples

`make verify` (`unittest/test-symbol.x`) and `make examples` (`symbol-literal-demo`).

lib/symbolset.x

Immutable ordered sets of compact Symbols.

Primary API

Function	Summary
SymbolSet.contains	Reports whether symbol belongs to set.
SymbolSet.encode	Returns the bytes of a SymbolSet holding symbols in order, in a new byte Block owned by the active Scope.
SymbolSet.getindex	Returns the Symbol at index, or zero when the index is out of range.
SymbolSet.index	Returns symbol's source-order index, or −1 when it is absent.
SymbolSet.iter	Initializes dest to iterate over the Symbols in source order.
SymbolSet.len	Returns the number of Symbols in set, or zero for a null set.

SymbolSet

SymbolSet.contains

```
int SymbolSet.contains(SymbolSet x, Symbol symbol)
```

Reports whether symbol belongs to set.

Source: lib/symbolset.x:112

SymbolSet.encode

```
Block SymbolSet.encode(Array symbols)
```

Returns the bytes of a SymbolSet holding symbols in order, in a new byte Block owned by the active Scope. The compiler emits them for each %<<...>> literal. Members must be distinct: the table search does not end for a repeated Symbol.

Source: lib/symbolset.x:176

SymbolSet.getindex

```
Symbol SymbolSet.getindex(SymbolSet x, int index)
```

Returns the Symbol at index, or zero when the index is out of range. Negative indices count from the end. Zero is also a valid empty Symbol, so use SymbolSet.len when an out-of-range result must be distinguished.

Source: lib/symbolset.x:119

SymbolSet.index

```
int SymbolSet.index(SymbolSet x, Symbol symbol)
```

Returns symbol's source-order index, or −1 when it is absent. A null or empty set returns −1. The ordered table verifies the hash candidate, so a nonmember collision is not reported as membership.

Source: lib/symbolset.x:97

SymbolSet.iter

```
Iter SymbolSet.iter(SymbolSet x, Iter dest)
```

Initializes dest to iterate over the Symbols in source order. The caller owns dest, which must remain valid through every pull. The iterator borrows the generated set storage and allocates nothing. A null dest returns NULL; a null set produces an empty iterator.

Source: lib/symbolset.x:140

SymbolSet.len

`size_t SymbolSet.len(SymbolSet x)`

Returns the number of `Symbol`s in `set`, or zero for a null set.

Source: `lib/symbolset.x:91`

Design notes

A `SymbolSet` is a compiler-generated byte string. Its perfect hash maps each member to its source-order index; the ordered `Symbol` table also rejects nonmembers that happen to reach the same index. The set borrows its generated static storage; no operation retains or frees it. `SymbolSet.encode` builds the bytes the compiler emits, so this module owns both the layout and the hash.

Tests and examples

`make verify` (compiler `SymbolSet` fixtures).

lib/system-macros.x

Convenience macros for text, timing, and processes.

Design notes

Include this module to use its declarations.

lib/thread.x

`Context`-backed native worker threads.

Primary API

Function	Summary
`Thread.free`	Frees a `Thread` handle after its consuming join has completed.
`Thread.join`	Joins one worker and exports its result into the joining `Scope` and pools.
`Thread.start`	Starts one worker and returns its heap-owned handle.

Thread

Thread.free

`void Thread.free(Thread t)`

Frees a `Thread` handle after its consuming join has completed.

Raises: `<bad-state>` for NULL, running, or joining handles.

Source: `lib/thread.x:326`

Thread.join

`Var Thread.join(Thread t)`

Joins one worker and exports its result into the joining `Scope` and pools. A failure during export still consumes the sealed worker storage. A native join failure leaves the `Thread` joinable for a retry. Concurrent joins are rejected while one is active; once native join succeeds, export cannot be retried.

Raises: `<bad-arg>` for a NULL handle, `<bad-state>` when another join is active or the worker was already joined, `<io-fail>` when the native join fails, any cause from result or error export, and `<join-fail>` carrying the worker's exported errors. None of them return here. A callback that returns `void` joins as `void`; a handled worker failure transfers instead of returning a sentinel.

Source: `lib/thread.x:286`

Thread.start

`Thread Thread.start(ThreadFn function, const void *input, size_t input_size)`

Starts one worker and returns its heap-owned handle. `input_size` bytes are copied before native start and passed once to the callback. Pointees within those bytes remain shared and must outlive the worker. Copied storage has `max_align_t` alignment, so over-aligned input types are unsupported. A worker runs on an 8 MiB stack on every platform, so library recursion limits are reached the same way on a worker as on the main thread. The worker also adopts this thread's `Error` policy, so a cause set to `<ignore>`, `<log>`, or `<collect>` resumes inside the callback as it does here. An attempt that reaches `pthread_create` permanently enables canonical-pool locking; the first successful start also freezes `Var` descriptor registration.

Raises: `<bad-arg>` for a NULL function or missing nonempty input, `<size-limit>` when the handle size or shutdown-hook registry overflows, `<alloc-fail>` when the handle or shutdown hook cannot be allocated, or `<io-fail>` when `pthread_create` fails. Failure during native once initialization or mutex setup aborts the process.

Source: `lib/thread.x:91`

Public types

Type	Kind	Summary
`Thread`	struct	A heap-owned native worker handle with one consuming join.
`ThreadFn`	callback	Computes one `Thread` result from a borrowed copy of the start input bytes.

Thread

```
typedef struct Thread *Thread
```

A heap-owned native worker handle with one consuming join. A started `Thread` must be joined, including after its callback finishes, and then released with `Thread.free`.

Source: `lib/thread.x:27`

ThreadFn

```
typedef Var (*ThreadFn)(const void *input, size_t input_size)
```

Computes one `Thread` result from a borrowed copy of the start input bytes. The input is valid only during the call. The returned value is exported by `Thread.join`; an `Error` escaping the callback is reported as `<join-fail>`. A cause the worker's `Error` policy resolves returns to its raise and does not end the callback.

Source: `lib/thread.x:21`

Design notes

A `Thread` copies its fixed-size input with max_align_t alignment, runs one callback in an isolated `Context`, and seals the exported result or error snapshot until join. Over-aligned input types are not supported. Shared objects require their own coordination.

Tests and examples

`make verify` (`unittest/test-thread.x`).

lib/type.x

Semantic type values and their structural operations.

Primary API

Function	Summary
`List.type`	Views `x` as a `Type` without validating its type shape.
`Symbol.is_builtin_type`	Returns whether `sym` can begin a builtin C type specifier.
`Symbol.is_inline`	Returns whether `sym` is the `inline` function specifier.
`Symbol.is_storage_class`	Returns whether `sym` is a storage-class specifier.
`Symbol.is_type_modifier`	Returns whether `sym` modifies the width or signedness of a scalar.

Function	Summary
`Symbol.is_type_qualifier`	Returns whether `sym` is `const`, `restrict`, or `volatile`.
`Type.apply`	Returns the result `Type` of a function `Type`, following pointer and array modifiers, or `NULL` when the chain does not end at a function.
`Type.base_type`	Returns the suffix of `type` beginning at its builtin or typedef base.
`Type.body`	Returns the stored body portion of an enum, struct, or union `Type`.
`Type.canonicalize`	Removes non-typedef storage classes, `inline`, and type qualifiers from `type`.
`Type.declared`	Returns the stored declaration `Type` after removing non-typedef storage classes and `inline`.
`Type.dereference`	Removes one outer pointer-like or array modifier, or returns `NULL`.
`Type.discards_qualifiers`	Returns whether handing a `source` value to a `target` declaration would silently drop a qualifier the target does not keep.
`Type.function_parts`	Splits a direct or pointer function `type` into its fixed `params`, which are `NULL` for `(void)`, and its `return_type` when one is asked for.
`Type.integer_literal_magnitude`	Reads an integer literal of `type` spelled in digits, with any suffix, into `value`.
`Type.is_aggregate`	Returns whether `type` is any struct or union shape.
`Type.is_aggregate_tag`	Returns whether `t` is a body-free struct or union tag reference.
`Type.is_aggregate_tag_body`	Returns whether `type` is a tagged struct or union definition.
`Type.is_array`	Returns whether the outer declarator represented by `type` is an array.
`Type.is_bare_typedef_name`	Returns whether `type` is one bare typedef-name `String`.
`Type.is_bitfield`	Returns whether the outer declarator represented by `type` is a bitfield.
`Type.is_builtin`	Returns whether `type` is a builtin scalar, struct, union, or enum.
`Type.is_char_pointer_like`	Returns whether `type` is exactly a pointer or array of `char` or `const char`, without resolving typedefs or removing qualifiers.
`Type.is_enum`	Returns whether `type` is any enum shape.
`Type.is_enum_tag`	Returns whether `type` is a body-free enum tag reference.
`Type.is_enum_tag_body`	Returns whether `type` is a tagged enum definition.
`Type.is_extern`	Returns whether `type` carries the `extern` storage class.
`Type.is_function`	Returns whether the outer declarator is a function or inline function.
`Type.is_inline`	Returns whether `type` carries the `inline` function specifier.
`Type.is_integral`	Returns whether `type` is a fixed integral scalar or an enum.
`Type.is_number`	Returns whether `type` is a fixed numeric scalar or an enum.
`Type.is_pointer`	Returns whether `t` begins with a pointer-like modifier.
`Type.is_reference`	Returns whether `t` is a reference or an optional reference.
`Type.is_static`	Returns whether `type` carries the `static` storage class.
`Type.is_threaded`	Returns whether `type` carries the `threaded` storage class.
`Type.is_typedef`	Returns whether `type` begins with the `typedef` storage class.
`Type.is_typedef_name`	Returns whether the base of `type` is exactly one typedef-name `String`.
`Type.list`	Views `x` as its underlying `List` without validating its type shape.
`Type.numeric_literal`	Returns the native type selected by a validated numeric token.
`Type.numeric_literal_value`	Reads a validated numeric literal at its semantic type's precision.
`Type.promote`	Applies integer promotion to `type`.
`Type.qualify`	Applies `source`'s outer qualifiers to `type`.
`Type.reference`	Returns the pointer `Type` formed by prefixing `type` with *.
`Type.scalar`	Returns the normalized builtin scalar spelling, or `NULL` when `t` is not one valid scalar combination.

Function	Summary
Type.scalar_tag	Returns the fixed Var numeric tag for type, or zero when none exists.
Type.tag	Returns the one-element tag List of an enum, struct, or union Type.
Type.var_numeric_extractor	Returns the numeric Var reader for type, or NULL when unsupported.
Type.var_numeric_update_helper	Returns the native numeric update helper for type, or NULL when the scalar has no registered update helper.
Type.widest	Returns the usual arithmetic result Type for two scalar operands.
Var.type	Returns the List payload of x as a Type, or NULL for another tag.

List

List.type

```
inline Type List.type(List x)
```

Views x as a Type without validating its type shape.

Source: lib/type.x:28

Symbol

Symbol.is_builtin_type

```
int Symbol.is_builtin_type(Symbol sym)
```

Returns whether sym can begin a builtin C type specifier.

Source: lib/type.x:57

Symbol.is_inline

```
int Symbol.is_inline(Symbol sym)
```

Returns whether sym is the inline function specifier.

Source: lib/type.x:44

Symbol.is_storage_class

```
int Symbol.is_storage_class(Symbol sym)
```

Returns whether sym is a storage-class specifier.

Source: lib/type.x:41

Symbol.is_type_modifier

```
int Symbol.is_type_modifier(Symbol sym)
```

Returns whether sym modifies the width or signedness of a scalar.

Source: lib/type.x:50

Symbol.is_type_qualifier

```
int Symbol.is_type_qualifier(Symbol sym)
```

Returns whether sym is const, restrict, or volatile.

Source: lib/type.x:47

Type

Type.apply

```
Type Type.apply(Type type)
```

Returns the result `Type` of a function `Type`, following pointer and array modifiers, or `NULL` when the chain does not end at a function.

Source: `lib/type.x:164`

Type.base_type

```
Type Type.base_type(Type type)
```

Returns the suffix of `type` beginning at its builtin or typedef base. The result shares the original `List` and is `NULL` when no base is present.

Source: `lib/type.x:217`

Type.body

```
List Type.body(Type t)
```

Returns the stored body portion of an enum, struct, or union `Type`. Tag references and `Type`s without a stored body return `NULL`.

Source: `lib/type.x:102`

Type.canonicalize

```
Type Type.canonicalize(Type type)
```

Removes non-typedef storage classes, `inline`, and type qualifiers from `type`.

Source: `lib/type.x:222`

Type.declared

```
Type Type.declared(Type type)
```

Returns the stored declaration `Type` after removing non-typedef storage classes and `inline`. Those specifiers describe declaration placement; `const`, `restrict`, and `volatile` describe the stored value and remain.

Source: `lib/type.x:228`

Type.dereference

```
Type Type.dereference(Type type)
```

Removes one outer pointer-like or array modifier, or returns `NULL`.

Source: `lib/type.x:137`

Type.discards_qualifiers

```
int Type.discards_qualifiers(Type source, Type target)
```

Returns whether handing a `source` value to a `target` declaration would silently drop a qualifier the target does not keep. The leading qualifiers of each type describe the copied value, not what it points at, so only the deeper levels are compared. Callers use this where the two types are otherwise the same, or where one points at `void`, which has no deeper levels to compare; a conversion through a converter function copies instead of aliasing.

Source: `lib/type.x:256`

Type.function_parts

```
int Type.function_parts(Type type, List &params, Type &?return_type)
```

Splits a direct or pointer function `type` into its fixed `params`, which are `NULL` for `(void)`, and its `return_type` when one is asked for. Returns 0 and leaves both unchanged when `type` is not a function.
Source: `lib/type.x:177`

Type.integer_literal_magnitude

```
int Type.integer_literal_magnitude( Type type, String text, unsigned long long &value)
```

Reads an integer literal of `type` spelled in digits, with any suffix, into `value`. Returns 0 for any other literal.
Source: `lib/type.x:545`

Type.is_aggregate

```
int Type.is_aggregate(Type type)
```

Returns whether `type` is any struct or union shape.
Source: `lib/type.x:63`

Type.is_aggregate_tag

```
int Type.is_aggregate_tag(Type t)
```

Returns whether `t` is a body-free struct or union tag reference.
Source: `lib/type.x:66`

Type.is_aggregate_tag_body

```
int Type.is_aggregate_tag_body(Type type)
```

Returns whether `type` is a tagged struct or union definition.
Source: `lib/type.x:73`

Type.is_array

```
int Type.is_array(Type type)
```

Returns whether the outer declarator represented by `type` is an array.
Source: `lib/type.x:118`

Type.is_bare_typedef_name

```
int Type.is_bare_typedef_name(Type type)
```

Returns whether `type` is one bare typedef-name `String`. Unlike `is_typedef_name`, this rejects pointer and array wrappers.
Source: `lib/type.x:318`

Type.is_bitfield

```
int Type.is_bitfield(Type type)
```

Returns whether the outer declarator represented by `type` is a bitfield.
Source: `lib/type.x:129`

Type.is_builtin

```
int Type.is_builtin(Type type)
```

Returns whether `type` is a builtin scalar, struct, union, or enum.
Source: `lib/type.x:300`

Type.is_char_pointer_like

`int Type.is_char_pointer_like(Type type)`

Returns whether `type` is exactly a pointer or array of `char` or `const char`, without resolving typedefs or removing qualifiers.

Source: `lib/type.x:293`

Type.is_enum

`int Type.is_enum(Type type)`

Returns whether `type` is any enum shape.

Source: `lib/type.x:77`

Type.is_enum_tag

`int Type.is_enum_tag(Type type)`

Returns whether `type` is a body-free enum tag reference.

Source: `lib/type.x:80`

Type.is_enum_tag_body

`int Type.is_enum_tag_body(Type type)`

Returns whether `type` is a tagged enum definition.

Source: `lib/type.x:86`

Type.is_extern

`int Type.is_extern(Type type)`

Returns whether `type` carries the `extern` storage class.

Source: `lib/type.x:203`

Type.is_function

`int Type.is_function(Type type)`

Returns whether the outer declarator is a function or inline function.

Source: `lib/type.x:121`

Type.is_inline

`int Type.is_inline(Type type)`

Returns whether `type` carries the `inline` function specifier.

Source: `lib/type.x:200`

Type.is_integral

`int Type.is_integral(Type type)`

Returns whether `type` is a fixed integral scalar or an enum.

Source: `lib/type.x:325`

Type.is_number

`int Type.is_number(Type type)`

Returns whether `type` is a fixed numeric scalar or an enum.
Source: `lib/type.x:322`

Type.is_pointer

```
int Type.is_pointer(Type t)
```

Returns whether `t` begins with a pointer-like modifier.
Source: `lib/type.x:112`

Type.is_reference

```
int Type.is_reference(Type t)
```

Returns whether `t` is a reference or an optional reference.
Source: `lib/type.x:115`

Type.is_static

```
int Type.is_static(Type type)
```

Returns whether `type` carries the `static` storage class.
Source: `lib/type.x:197`

Type.is_threaded

```
int Type.is_threaded(Type type)
```

Returns whether `type` carries the `threaded` storage class.
Source: `lib/type.x:206`

Type.is_typedef

```
int Type.is_typedef(Type type)
```

Returns whether `type` begins with the `typedef` storage class.
Source: `lib/type.x:209`

Type.is_typedef_name

```
int Type.is_typedef_name(Type type)
```

Returns whether the base of `type` is exactly one typedef-name `String`.
Source: `lib/type.x:310`

Type.list

```
inline List Type.list(Type x)
```

Views `x` as its underlying `List` without validating its type shape.
Source: `lib/type.x:25`

Type.numeric_literal

```
Type Type.numeric_literal(String text, int floating)
```

Returns the native type selected by a validated numeric token. `floating` selects floating suffix rules; an integer outside all supported native families returns `NULL`. A signed data token includes its sign when selecting a family, so a signed minimum fits that family.
Source: `lib/type.x:497`

Type.numeric_literal_value

```
Var Type.numeric_literal_value(Type type, String text)
```

Reads a validated numeric literal at its semantic type's precision. Returns `void` when its magnitude exceeds the integer representation.

Source: `lib/type.x:525`

Type.promote

```
Type Type.promote(Type type)
```

Applies integer promotion to `type`. Enums and narrow integers become `int`; other scalars retain their canonical spelling, and a non-scalar returns `NULL`.

Source: `lib/type.x:445`

Type.qualify

```
Type Type.qualify(Type type, Type source)
```

Applies `source`'s outer qualifiers to `type`.

Source: `lib/type.x:146`

Type.reference

```
Type Type.reference(Type type)
```

Returns the pointer `Type` formed by prefixing `type` with `*`.

Source: `lib/type.x:159`

Type.scalar

```
Type Type.scalar(Type t)
```

Returns the normalized builtin scalar spelling, or `NULL` when `t` is not one valid scalar combination. Storage classes and qualifiers do not affect the result.

Source: `lib/type.x:343`

Type.scalar_tag

```
Symbol Type.scalar_tag(Type type)
```

Returns the fixed `Var` numeric tag for `type`, or zero when none exists.

Source: `lib/type.x:417`

Type.tag

```
List Type.tag(Type type)
```

Returns the one-element tag `List` of an enum, struct, or union `Type`. For a compiler-generated anonymous tag, that element is a gensym node. A shape with no tag slot returns `NULL`.

Source: `lib/type.x:92`

Type.var_numeric_extractor

```
String Type.var_numeric_extractor(Type type)
```

Returns the numeric `Var` reader for `type`, or `NULL` when unsupported. Enums use `Var_int` after conversion to their shared integer tag.

Source: `lib/type.x:425`

Type.var_numeric_update_helper

`String Type.var_numeric_update_helper(Type type)`

Returns the native numeric update helper for `type`, or `NULL` when the scalar has no registered update helper.

Source: `lib/type.x:434`

Type.widest

`Type Type.widest(Type a, Type b)`

Returns the usual arithmetic result `Type` for two scalar operands. A missing or non-scalar operand produces `NULL`.

Source: `lib/type.x:458`

`Var`

Var.type

`inline Type Var.type(Var x)`

Returns the `List` payload of `x` as a `Type`, or `NULL` for another tag.

Source: `lib/type.x:22`

Design notes

These operations inspect captured types without compiler state. Name resolution and translation-unit registrations belong to the compiler.

Tests and examples

`make verify` (`meta-type-values` compiler fixture).

`lib/typed-array.x`

Packed typed `Arrays` generated from shared methods.

Primary API

Function	Summary
`Array.arraychar`	Packs `values` into a fresh array in source order.
`Array.arraydbl`	Packs `values` into a fresh array in source order.
`Array.arrayfloat`	Packs `values` into a fresh array in source order.
`Array.arrayint`	Packs `values` into a fresh array in source order.
`Array.arraylong`	Packs `values` into a fresh array in source order.
`Array.arrayshort`	Packs `values` into a fresh array in source order.
`Array.arraystring`	Packs `values` into a fresh array in source order.
`ArrayChar.array`	Returns a fresh ordinary Array containing boxed elements in source order.
`ArrayChar.block`	Views `array` as the same underlying `Block` storage.
`ArrayChar.compare`	Compares packed Arrays lexicographically with the element family's ordering.
`ArrayChar.concat`	Returns a fresh array holding `a` followed by `b`.
`ArrayChar.contains`	Reports whether a native element equals `value`.
`ArrayChar.copy`	Returns a fresh packed copy, or NULL for a null array.
`ArrayChar.count`	Returns how many native elements equal `value`.
`ArrayChar.equal`	Reports elementwise native equality.

Function	Summary
ArrayChar.find	Returns the first index whose native element equals value, or −1.
ArrayChar.getindex	Reads raw index with the bounds and null preconditions of a C pointer.
ArrayChar.getslice	Returns a fresh packed array for normalized [start:end:step].
ArrayChar.indexof	Returns the same first-match index as ArrayChar.find.
ArrayChar.insert	Inserts value at normalized index and returns it.
ArrayChar.iter	Initializes dest to lazily yield boxed elements in index order.
ArrayChar.new	Returns a fresh empty packed array owned by the current Scope.
ArrayChar.postfixindex	Applies ++ or -- at raw index and returns the previous value.
ArrayChar.push	Appends value and returns it.
ArrayChar.remove	Removes and returns the element at normalized index.
ArrayChar.remslice	Removes the region array[start:end] and returns a fresh packed array.
ArrayChar.repr	Returns the readable packed-Array representation.
ArrayChar.reverse	Reverses array in place and returns that same array.
ArrayChar.setindex	Stores and returns value at raw index, with C-pointer preconditions.
ArrayChar.setslice	Replaces the region array[start:end] and returns array.
ArrayChar.shift	Removes and returns the first element, shifting the remainder left.
ArrayChar.splice	Replaces remove_count values at normalized index and returns them.
ArrayChar.str	Returns the packed-Array display String.
ArrayChar.take_last	Removes and returns the last element.
ArrayChar.try_get	Reads normalized index into out and reports whether it exists.
ArrayChar.try_next	Writes the next native element, advances cursor, and returns one.
ArrayChar.try_take_last	Removes the last element into out and returns one when present.
ArrayChar.unshift	Inserts value at the front and returns it, shifting elements right.
ArrayChar.updateindex	Applies integer op to raw element index and returns the stored value.
ArrayChar.var	Boxes array with its registered packed-array tag without copying.
ArrayChar.write_repr	Appends the readable packed-Array representation to out.
ArrayChar.write_str	Appends the packed-Array display text to out.
ArrayDbl.array	Returns a fresh ordinary Array containing boxed elements in source order.
ArrayDbl.block	Views array as the same underlying Block storage.
ArrayDbl.compare	Compares packed Arrays lexicographically with the element family's ordering.
ArrayDbl.concat	Returns a fresh array holding a followed by b.
ArrayDbl.contains	Reports whether a native element equals value.
ArrayDbl.copy	Returns a fresh packed copy, or NULL for a null array.
ArrayDbl.count	Returns how many native elements equal value.
ArrayDbl.equal	Reports elementwise native equality.
ArrayDbl.find	Returns the first index whose native element equals value, or −1.
ArrayDbl.getindex	Reads raw index with the bounds and null preconditions of a C pointer.
ArrayDbl.getslice	Returns a fresh packed array for normalized [start:end:step].
ArrayDbl.indexof	Returns the same first-match index as ArrayDbl.find.
ArrayDbl.insert	Inserts value at normalized index and returns it.
ArrayDbl.iter	Initializes dest to lazily yield boxed elements in index order.
ArrayDbl.new	Returns a fresh empty packed array owned by the current Scope.
ArrayDbl.postfixindex	Applies ++ or -- at raw index and returns the previous value.

Function	Summary
ArrayDbl.push	Appends value and returns it.
ArrayDbl.remove	Removes and returns the element at normalized index.
ArrayDbl.remslice	Removes the region array[start:end] and returns a fresh packed array.
ArrayDbl.repr	Returns the readable packed-Array representation.
ArrayDbl.reverse	Reverses array in place and returns that same array.
ArrayDbl.setindex	Stores and returns value at raw index, with C-pointer preconditions.
ArrayDbl.setslice	Replaces the region array[start:end] and returns array.
ArrayDbl.shift	Removes and returns the first element, shifting the remainder left.
ArrayDbl.splice	Replaces remove_count values at normalized index and returns them.
ArrayDbl.str	Returns the packed-Array display String.
ArrayDbl.take_last	Removes and returns the last element.
ArrayDbl.try_get	Reads normalized index into out and reports whether it exists.
ArrayDbl.try_next	Writes the next native element, advances cursor, and returns one.
ArrayDbl.try_take_last	Removes the last element into out and returns one when present.
ArrayDbl.unshift	Inserts value at the front and returns it, shifting elements right.
ArrayDbl.updateindex	Applies a floating arithmetic op and returns the stored value.
ArrayDbl.var	Boxes array with its registered packed-array tag without copying.
ArrayDbl.write_repr	Appends the readable packed-Array representation to out.
ArrayDbl.write_str	Appends the packed-Array display text to out.
ArrayFloat.array	Returns a fresh ordinary Array containing boxed elements in source order.
ArrayFloat.block	Views array as the same underlying Block storage.
ArrayFloat.compare	Compares packed Arrays lexicographically with the element family's ordering.
ArrayFloat.concat	Returns a fresh array holding a followed by b.
ArrayFloat.contains	Reports whether a native element equals value.
ArrayFloat.copy	Returns a fresh packed copy, or NULL for a null array.
ArrayFloat.count	Returns how many native elements equal value.
ArrayFloat.equal	Reports elementwise native equality.
ArrayFloat.find	Returns the first index whose native element equals value, or −1.
ArrayFloat.getindex	Reads raw index with the bounds and null preconditions of a C pointer.
ArrayFloat.getslice	Returns a fresh packed array for normalized [start:end:step].
ArrayFloat.indexof	Returns the same first-match index as ArrayFloat.find.
ArrayFloat.insert	Inserts value at normalized index and returns it.
ArrayFloat.iter	Initializes dest to lazily yield boxed elements in index order.
ArrayFloat.new	Returns a fresh empty packed array owned by the current Scope.
ArrayFloat.postfixindex	Applies ++ or -- at raw index and returns the previous value.
ArrayFloat.push	Appends value and returns it.
ArrayFloat.remove	Removes and returns the element at normalized index.
ArrayFloat.remslice	Removes the region array[start:end] and returns a fresh packed array.
ArrayFloat.repr	Returns the readable packed-Array representation.
ArrayFloat.reverse	Reverses array in place and returns that same array.
ArrayFloat.setindex	Stores and returns value at raw index, with C-pointer preconditions.
ArrayFloat.setslice	Replaces the region array[start:end] and returns array.
ArrayFloat.shift	Removes and returns the first element, shifting the remainder left.

Function	Summary
ArrayFloat.splice	Replaces remove_count values at normalized index and returns them.
ArrayFloat.str	Returns the packed-Array display String.
ArrayFloat.take_last	Removes and returns the last element.
ArrayFloat.try_get	Reads normalized index into out and reports whether it exists.
ArrayFloat.try_next	Writes the next native element, advances cursor, and returns one.
ArrayFloat.try_take_last	Removes the last element into out and returns one when present.
ArrayFloat.unshift	Inserts value at the front and returns it, shifting elements right.
ArrayFloat.updateindex	Applies a floating arithmetic op and returns the stored value.
ArrayFloat.var	Boxes array with its registered packed-array tag without copying.
ArrayFloat.write_repr	Appends the readable packed-Array representation to out.
ArrayFloat.write_str	Appends the packed-Array display text to out.
ArrayInt.array	Returns a fresh ordinary Array containing boxed elements in source order.
ArrayInt.block	Views array as the same underlying Block storage.
ArrayInt.compare	Compares packed Arrays lexicographically with the element family's ordering.
ArrayInt.concat	Returns a fresh array holding a followed by b.
ArrayInt.contains	Reports whether a native element equals value.
ArrayInt.copy	Returns a fresh packed copy, or NULL for a null array.
ArrayInt.count	Returns how many native elements equal value.
ArrayInt.equal	Reports elementwise native equality.
ArrayInt.find	Returns the first index whose native element equals value, or −1.
ArrayInt.getindex	Reads raw index with the bounds and null preconditions of a C pointer.
ArrayInt.getslice	Returns a fresh packed array for normalized [start:end:step].
ArrayInt.indexof	Returns the same first-match index as ArrayInt.find.
ArrayInt.insert	Inserts value at normalized index and returns it.
ArrayInt.iter	Initializes dest to lazily yield boxed elements in index order.
ArrayInt.new	Returns a fresh empty packed array owned by the current Scope.
ArrayInt.postfixindex	Applies ++ or -- at raw index and returns the previous value.
ArrayInt.push	Appends value and returns it.
ArrayInt.remove	Removes and returns the element at normalized index.
ArrayInt.remslice	Removes the region array[start:end] and returns a fresh packed array.
ArrayInt.repr	Returns the readable packed-Array representation.
ArrayInt.reverse	Reverses array in place and returns that same array.
ArrayInt.setindex	Stores and returns value at raw index, with C-pointer preconditions.
ArrayInt.setslice	Replaces the region array[start:end] and returns array.
ArrayInt.shift	Removes and returns the first element, shifting the remainder left.
ArrayInt.splice	Replaces remove_count values at normalized index and returns them.
ArrayInt.str	Returns the packed-Array display String.
ArrayInt.take_last	Removes and returns the last element.
ArrayInt.try_get	Reads normalized index into out and reports whether it exists.
ArrayInt.try_next	Writes the next native element, advances cursor, and returns one.
ArrayInt.try_take_last	Removes the last element into out and returns one when present.
ArrayInt.unshift	Inserts value at the front and returns it, shifting elements right.
ArrayInt.updateindex	Applies integer op to raw element index and returns the stored value.

Function	Summary
ArrayInt.var	Boxes array with its registered packed-array tag without copying.
ArrayInt.write_repr	Appends the readable packed-Array representation to out.
ArrayInt.write_str	Appends the packed-Array display text to out.
ArrayLong.array	Returns a fresh ordinary Array containing boxed elements in source order.
ArrayLong.block	Views array as the same underlying Block storage.
ArrayLong.compare	Compares packed Arrays lexicographically with the element family's ordering.
ArrayLong.concat	Returns a fresh array holding a followed by b.
ArrayLong.contains	Reports whether a native element equals value.
ArrayLong.copy	Returns a fresh packed copy, or NULL for a null array.
ArrayLong.count	Returns how many native elements equal value.
ArrayLong.equal	Reports elementwise native equality.
ArrayLong.find	Returns the first index whose native element equals value, or −1.
ArrayLong.getindex	Reads raw index with the bounds and null preconditions of a C pointer.
ArrayLong.getslice	Returns a fresh packed array for normalized [start:end:step].
ArrayLong.indexof	Returns the same first-match index as ArrayLong.find.
ArrayLong.insert	Inserts value at normalized index and returns it.
ArrayLong.iter	Initializes dest to lazily yield boxed elements in index order.
ArrayLong.new	Returns a fresh empty packed array owned by the current Scope.
ArrayLong.postfixindex	Applies ++ or -- at raw index and returns the previous value.
ArrayLong.push	Appends value and returns it.
ArrayLong.remove	Removes and returns the element at normalized index.
ArrayLong.remslice	Removes the region array[start:end] and returns a fresh packed array.
ArrayLong.repr	Returns the readable packed-Array representation.
ArrayLong.reverse	Reverses array in place and returns that same array.
ArrayLong.setindex	Stores and returns value at raw index, with C-pointer preconditions.
ArrayLong.setslice	Replaces the region array[start:end] and returns array.
ArrayLong.shift	Removes and returns the first element, shifting the remainder left.
ArrayLong.splice	Replaces remove_count values at normalized index and returns them.
ArrayLong.str	Returns the packed-Array display String.
ArrayLong.take_last	Removes and returns the last element.
ArrayLong.try_get	Reads normalized index into out and reports whether it exists.
ArrayLong.try_next	Writes the next native element, advances cursor, and returns one.
ArrayLong.try_take_last	Removes the last element into out and returns one when present.
ArrayLong.unshift	Inserts value at the front and returns it, shifting elements right.
ArrayLong.updateindex	Applies integer op to raw element index and returns the stored value.
ArrayLong.var	Boxes array with its registered packed-array tag without copying.
ArrayLong.write_repr	Appends the readable packed-Array representation to out.
ArrayLong.write_str	Appends the packed-Array display text to out.
ArrayShort.array	Returns a fresh ordinary Array containing boxed elements in source order.
ArrayShort.block	Views array as the same underlying Block storage.
ArrayShort.compare	Compares packed Arrays lexicographically with the element family's ordering.
ArrayShort.concat	Returns a fresh array holding a followed by b.
ArrayShort.contains	Reports whether a native element equals value.

Function	Summary
ArrayShort.copy	Returns a fresh packed copy, or NULL for a null array.
ArrayShort.count	Returns how many native elements equal value.
ArrayShort.equal	Reports elementwise native equality.
ArrayShort.find	Returns the first index whose native element equals value, or −1.
ArrayShort.getindex	Reads raw index with the bounds and null preconditions of a C pointer.
ArrayShort.getslice	Returns a fresh packed array for normalized [start:end:step].
ArrayShort.indexof	Returns the same first-match index as ArrayShort.find.
ArrayShort.insert	Inserts value at normalized index and returns it.
ArrayShort.iter	Initializes dest to lazily yield boxed elements in index order.
ArrayShort.new	Returns a fresh empty packed array owned by the current Scope.
ArrayShort.postfixindex	Applies ++ or -- at raw index and returns the previous value.
ArrayShort.push	Appends value and returns it.
ArrayShort.remove	Removes and returns the element at normalized index.
ArrayShort.remslice	Removes the region array[start:end] and returns a fresh packed array.
ArrayShort.repr	Returns the readable packed-Array representation.
ArrayShort.reverse	Reverses array in place and returns that same array.
ArrayShort.setindex	Stores and returns value at raw index, with C-pointer preconditions.
ArrayShort.setslice	Replaces the region array[start:end] and returns array.
ArrayShort.shift	Removes and returns the first element, shifting the remainder left.
ArrayShort.splice	Replaces remove_count values at normalized index and returns them.
ArrayShort.str	Returns the packed-Array display String.
ArrayShort.take_last	Removes and returns the last element.
ArrayShort.try_get	Reads normalized index into out and reports whether it exists.
ArrayShort.try_next	Writes the next native element, advances cursor, and returns one.
ArrayShort.try_take_last	Removes the last element into out and returns one when present.
ArrayShort.unshift	Inserts value at the front and returns it, shifting elements right.
ArrayShort.updateindex	Applies integer op to raw element index and returns the stored value.
ArrayShort.var	Boxes array with its registered packed-array tag without copying.
ArrayShort.write_repr	Appends the readable packed-Array representation to out.
ArrayShort.write_str	Appends the packed-Array display text to out.
ArrayString.array	Returns a fresh ordinary Array containing boxed elements in source order.
ArrayString.block	Views array as the same underlying Block storage.
ArrayString.compare	Compares packed Arrays lexicographically with the element family's ordering.
ArrayString.concat	Returns a fresh array holding a followed by b.
ArrayString.contains	Reports whether a native element equals value.
ArrayString.copy	Returns a fresh packed copy, or NULL for a null array.
ArrayString.count	Returns how many native elements equal value.
ArrayString.equal	Reports elementwise native equality.
ArrayString.find	Returns the first index whose native element equals value, or −1.
ArrayString.getindex	Reads raw index with the bounds and null preconditions of a C pointer.
ArrayString.getslice	Returns a fresh packed array for normalized [start:end:step].
ArrayString.indexof	Returns the same first-match index as ArrayString.find.
ArrayString.insert	Inserts value at normalized index and returns it.

Function	Summary
ArrayString.iter	Initializes dest to lazily yield boxed elements in index order.
ArrayString.new	Returns a fresh empty packed array owned by the current Scope.
ArrayString.postfixindex	Raises <bad-op> because String elements have no postfix operation.
ArrayString.push	Appends value and returns it.
ArrayString.remove	Removes and returns the element at normalized index.
ArrayString.remslice	Removes the region array[start:end] and returns a fresh packed array.
ArrayString.repr	Returns the readable packed-Array representation.
ArrayString.reverse	Reverses array in place and returns that same array.
ArrayString.setindex	Stores and returns value at raw index, with C-pointer preconditions.
ArrayString.setslice	Replaces the region array[start:end] and returns array.
ArrayString.shift	Removes and returns the first element, shifting the remainder left.
ArrayString.splice	Replaces remove_count values at normalized index and returns them.
ArrayString.str	Returns the packed-Array display String.
ArrayString.take_last	Removes and returns the last element.
ArrayString.try_get	Reads normalized index into out and reports whether it exists.
ArrayString.try_next	Writes the next native element, advances cursor, and returns one.
ArrayString.try_take_last	Removes the last element into out and returns one when present.
ArrayString.unshift	Inserts value at the front and returns it, shifting elements right.
ArrayString.updateindex	Concatenates rhs onto the String at raw index and returns the stored canonical String.
ArrayString.var	Boxes array with its registered packed-array tag without copying.
ArrayString.write_repr	Appends the readable packed-Array representation to out.
ArrayString.write_str	Appends the packed-Array display text to out.
Var.arraychar	Returns the packed-array pointer carried by value.
Var.arraydbl	Returns the packed-array pointer carried by value.
Var.arrayfloat	Returns the packed-array pointer carried by value.
Var.arrayint	Returns the packed-array pointer carried by value.
Var.arraylong	Returns the packed-array pointer carried by value.
Var.arrayshort	Returns the packed-array pointer carried by value.
Var.arraystring	Returns the packed-array pointer carried by value.

Array

Array.arraychar

```
ArrayChar Array.arraychar(Array values)
```

Packs values into a fresh array in source order. A null input returns NULL. Each element goes through its Var conversion; the source is unchanged and a failing conversion or allocation releases the partial result before transferring its cause.

Source: lib/typed-array.x:108

Array.arraydbl

```
ArrayDbl Array.arraydbl(Array values)
```

Packs values into a fresh array in source order. A null input returns NULL. Each element goes through its Var conversion; the source is unchanged and a failing conversion or allocation releases the partial result before transferring its cause.

Source: `lib/typed-array.x:153`

Array.arrayfloat

```
ArrayFloat Array.arrayfloat(Array values)
```

Packs `values` into a fresh array in source order. A null input returns NULL. Each element goes through its `Var` conversion; the source is unchanged and a failing conversion or allocation releases the partial result before transferring its cause.

Source: `lib/typed-array.x:144`

Array.arrayint

```
ArrayInt Array.arrayint(Array values)
```

Packs `values` into a fresh array in source order. A null input returns NULL. Each element goes through its `Var` conversion; the source is unchanged and a failing conversion or allocation releases the partial result before transferring its cause.

Source: `lib/typed-array.x:126`

Array.arraylong

```
ArrayLong Array.arraylong(Array values)
```

Packs `values` into a fresh array in source order. A null input returns NULL. Each element goes through its `Var` conversion; the source is unchanged and a failing conversion or allocation releases the partial result before transferring its cause.

Source: `lib/typed-array.x:135`

Array.arrayshort

```
ArrayShort Array.arrayshort(Array values)
```

Packs `values` into a fresh array in source order. A null input returns NULL. Each element goes through its `Var` conversion; the source is unchanged and a failing conversion or allocation releases the partial result before transferring its cause.

Source: `lib/typed-array.x:117`

Array.arraystring

```
ArrayString Array.arraystring(Array values)
```

Packs `values` into a fresh array in source order. A null input returns NULL. Each element goes through its `Var` conversion; the source is unchanged and a failing conversion or allocation releases the partial result before transferring its cause.

Source: `lib/typed-array.x:162`

ArrayChar

ArrayChar.array

```
Array ArrayChar.array(ArrayChar array)
```

Returns a fresh ordinary Array containing boxed elements in source order. A null input returns NULL. The typed Array and its storage are unchanged.

Raises: any cause from boxing or appending an element.

Source: `lib/typed-array.x:106`

ArrayChar.block

`Block ArrayChar.block(ArrayChar array)`

Views `array` as the same underlying `Block` storage.

Source: `lib/typed-array.x:108`

ArrayChar.compare

`int ArrayChar.compare(ArrayChar a, ArrayChar b)`

Compares packed Arrays lexicographically with the element family's ordering. Identical handles compare equal and NULL sorts first.

Source: `lib/typed-array.x:105`

ArrayChar.concat

`ArrayChar ArrayChar.concat(ArrayChar a, ArrayChar b)`

Returns a fresh array holding `a` followed by `b`. Neither input is modified. A null `a` returns NULL, while a null `b` produces an independent copy of `a`.

Raises: `<size-limit>` or `<alloc-fail>` when the result cannot be represented or allocated.

Source: `lib/typed-array.x:104`

ArrayChar.contains

`int ArrayChar.contains(ArrayChar array, char value)`

Reports whether a native element equals `value`. The scan is linear. Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:104`

ArrayChar.copy

`ArrayChar ArrayChar.copy(ArrayChar array)`

Returns a fresh packed copy, or NULL for a null array. The result has independent mutable storage containing the same stored values.

Raises: `<size-limit>` or `<alloc-fail>` when the copy cannot be represented or allocated.

Source: `lib/typed-array.x:104`

ArrayChar.count

`int ArrayChar.count(ArrayChar array, char value)`

Returns how many native elements equal `value`. A null array returns zero; the scan is linear.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:104`

ArrayChar.equal

`int ArrayChar.equal(ArrayChar a, ArrayChar b)`

Reports elementwise native equality. Two null arrays compare equal; one null array or different lengths compare unequal.

Source: `lib/typed-array.x:104`

ArrayChar.find

`int ArrayChar.find(ArrayChar array, char value)`

Returns the first index whose native element equals `value`, or −1. A null array returns −1; the scan is linear and preserves source order.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:104`

ArrayChar.getindex

```
char ArrayChar.getindex(ArrayChar array, int index)
```

Reads raw `index` with the bounds and null preconditions of a C pointer. Negative indices are not normalized. Use `try_get` for a checked read.

Source: `lib/typed-array.x:104`

ArrayChar.getslice

```
ArrayChar ArrayChar.getslice(ArrayChar array, int start, int end, int step)
```

Returns a fresh packed array for normalized `[start:end:step]`. Negative bounds count from the end and a negative step walks backward.

Raises: `<bad-arg>` for a null array or zero step, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:104`

ArrayChar.indexof

```
int ArrayChar.indexof(ArrayChar array, char value)
```

Returns the same first-match index as `ArrayChar.find`.

Source: `lib/typed-array.x:104`

ArrayChar.insert

```
char ArrayChar.insert(ArrayChar array, int index, char value)
```

Inserts `value` at normalized `index` and returns it. The accepted range includes the position after the last element; negative indices are normalized against that wider range. Boxed `Array` returns `void` for a null array or out-of-range index without mutation.

Raises: `<bad-arg>` for a null typed array or out-of-range typed-array index, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:104`

ArrayChar.iter

```
Iter ArrayChar.iter(ArrayChar array, Iter dest)
```

Initializes `dest` to lazily yield boxed elements in index order. Returns `dest`, whose storage is caller-owned. The iterator borrows `array` and reads its current storage on each step, so both must remain live and mutation can change the remaining traversal. A null `dest` returns NULL. A null array, or one longer than the integer index domain, is exhausted. A boxed wide element is `Scope`-owned; retaining one after a pull requires its owning `Scope` to remain live.

Raises: any cause from boxing an element.

Source: `lib/typed-array.x:110`

ArrayChar.new

```
ArrayChar ArrayChar.new(void)
```

Returns a fresh empty packed array owned by the current `Scope`.

Raises: <alloc-fail> when storage cannot be allocated.

Source: lib/typed-array.x:104

ArrayChar.postfixindex

```
char ArrayChar.postfixindex(ArrayChar array, int index, Symbol op)
```

Applies ++ or -- at raw index and returns the previous value. The index has the unchecked C-pointer preconditions of bracket access.

Raises: <bad-op> for another operator. The failure does not return.

Source: lib/typed-array.x:107

ArrayChar.push

```
char ArrayChar.push(ArrayChar array, char value)
```

Appends value and returns it. Boxed Array returns void for a null array.

Raises: <bad-arg> for a null typed array, <size-limit> at the integer index limit, or <alloc-fail> when storage cannot grow.

Source: lib/typed-array.x:104

ArrayChar.remove

```
char ArrayChar.remove(ArrayChar array, int index)
```

Removes and returns the element at normalized index. Negative indices count from the end; following elements shift left. Boxed Array returns void for a null array or out-of-range index.

Raises: <bad-arg> for a null typed array or out-of-range typed-array index, or <size-limit> when the length exceeds the integer index domain in either family.

Source: lib/typed-array.x:104

ArrayChar.remslice

```
ArrayChar ArrayChar.remslice(ArrayChar array, int start, int end)
```

Removes the region array[start:end] and returns a fresh packed array. Bounds normalize as slice bounds, so a negative end is a stop. Reversed bounds are swapped, and the removed values retain their order.

Raises: <bad-arg> for a null array, <size-limit> outside the integer index domain, or <alloc-fail> on allocation failure.

Source: lib/typed-array.x:104

ArrayChar.repr

```
String ArrayChar.repr(ArrayChar array)
```

Returns the readable packed-Array representation.

Source: lib/typed-array.x:105

ArrayChar.reverse

```
ArrayChar ArrayChar.reverse(ArrayChar array)
```

Reverses array in place and returns that same array. A null or one-element array is returned unchanged.

Source: lib/typed-array.x:104

ArrayChar.setindex

```
char ArrayChar.setindex(ArrayChar array, int index, char value)
```

Stores and returns `value` at raw `index`, with C-pointer preconditions. Negative indices are not normalized; the caller supplies a valid index.

Source: `lib/typed-array.x:104`

ArrayChar.setslice

```
ArrayChar ArrayChar.setslice(ArrayChar array, int start, int end, ArrayChar values)
```

Replaces the region `array[start:end]` and returns `array`. Bounds normalize as slice bounds, so a negative `end` is a stop. Reversed bounds are swapped; null `values` deletes the region. Passing `array` itself as `values` is supported and copies before mutation.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` while copying self-aliased values or growing storage for the replacement. These failures leave `array` unchanged.

Source: `lib/typed-array.x:104`

ArrayChar.shift

```
char ArrayChar.shift(ArrayChar array)
```

Removes and returns the first element, shifting the remainder left. This operation is O(n). Boxed `Array` returns `void` for a null or empty array.

Raises: `<bad-arg>` for a null or empty typed array, or `<size-limit>` when the length exceeds the integer index domain in either family.

Source: `lib/typed-array.x:104`

ArrayChar.splice

```
ArrayChar ArrayChar.splice(ArrayChar array, int index, int remove_count, ArrayChar values)
```

Replaces `remove_count` values at normalized `index` and returns them. The result is a fresh packed array. A nonpositive count removes nothing; null `values` performs deletion, and self-splicing is supported.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:104`

ArrayChar.str

```
String ArrayChar.str(ArrayChar array)
```

Returns the packed-Array display String.

Source: `lib/typed-array.x:105`

ArrayChar.take_last

```
char ArrayChar.take_last(ArrayChar array)
```

Removes and returns the last element. Boxed `Array` returns `void` for a null or empty array.

Raises: `<bad-arg>` for a null or empty typed array.

Source: `lib/typed-array.x:104`

ArrayChar.try_get

```
int ArrayChar.try_get(ArrayChar array, int index, char &?out)
```

Reads normalized `index` into `out` and reports whether it exists. Negative indices count from the end. A null array, null `out`, or out-of-range index returns zero without writing; success returns one.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:104`

ArrayChar.try_next

```
int ArrayChar.try_next(ArrayChar array, int &?cursor, char &?out)
```

Writes the next native element, advances `cursor`, and returns one. Initialize the caller-owned cursor to zero. A null argument, negative cursor, exhaustion, or an array outside the integer index domain returns zero without changing `cursor` or `out`. Structural mutation invalidates the cursor.

Source: `lib/typed-array.x:104`

ArrayChar.try_take_last

```
int ArrayChar.try_take_last(ArrayChar array, char &?out)
```

Removes the last element into `out` and returns one when present. A null or empty array, or null `out`, returns zero without writing or changing the array.

Source: `lib/typed-array.x:104`

ArrayChar.unshift

```
char ArrayChar.unshift(ArrayChar array, char value)
```

Inserts `value` at the front and returns it, shifting elements right. This operation is O(n). Boxed `Array` returns `void` for a null array.

Raises: `<bad-arg>` for a null typed array, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:104`

ArrayChar.updateindex

```
char ArrayChar.updateindex(ArrayChar array, int index, Symbol op, char rhs)
```

Applies integer `op` to raw element `index` and returns the stored value. The index has the unchecked C-pointer preconditions of bracket access. Arithmetic follows the generated C operators, integer promotions, and result conversion. A failing operation leaves the element unchanged.

Raises: `<div-zero>`, `<bad-shift>`, or `<bad-op>` for its corresponding invalid operation.

Source: `lib/typed-array.x:107`

ArrayChar.var

```
Var ArrayChar.var(ArrayChar array)
```

Boxes `array` with its registered packed-array tag without copying. The boxed and typed values share the same mutable object identity.

Source: `lib/typed-array.x:108`

ArrayChar.write_repr

```
Buffer ArrayChar.write_repr(ArrayChar array, Buffer out)
```

Appends the readable packed-Array representation to `out`.

Source: `lib/typed-array.x:105`

ArrayChar.write_str

```
Buffer ArrayChar.write_str(ArrayChar array, Buffer out)
```

Appends the packed-Array display text to `out`.

Source: `lib/typed-array.x:105`

ArrayDbl

ArrayDbl.array

```
Array ArrayDbl.array(ArrayDbl array)
```

Returns a fresh ordinary Array containing boxed elements in source order. A null input returns NULL. The typed Array and its storage are unchanged.

Raises: any cause from boxing or appending an element.

Source: `lib/typed-array.x:151`

ArrayDbl.block

```
Block ArrayDbl.block(ArrayDbl array)
```

Views `array` as the same underlying `Block` storage.

Source: `lib/typed-array.x:153`

ArrayDbl.compare

```
int ArrayDbl.compare(ArrayDbl a, ArrayDbl b)
```

Compares packed Arrays lexicographically with the element family's ordering. Identical handles compare equal and NULL sorts first.

Source: `lib/typed-array.x:150`

ArrayDbl.concat

```
ArrayDbl ArrayDbl.concat(ArrayDbl a, ArrayDbl b)
```

Returns a fresh array holding `a` followed by `b`. Neither input is modified. A null `a` returns NULL, while a null `b` produces an independent copy of `a`.

Raises: `<size-limit>` or `<alloc-fail>` when the result cannot be represented or allocated.

Source: `lib/typed-array.x:149`

ArrayDbl.contains

```
int ArrayDbl.contains(ArrayDbl array, double value)
```

Reports whether a native element equals `value`. The scan is linear. Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:149`

ArrayDbl.copy

```
ArrayDbl ArrayDbl.copy(ArrayDbl array)
```

Returns a fresh packed copy, or NULL for a null array. The result has independent mutable storage containing the same stored values.

Raises: `<size-limit>` or `<alloc-fail>` when the copy cannot be represented or allocated.

Source: `lib/typed-array.x:149`

ArrayDbl.count

```
int ArrayDbl.count(ArrayDbl array, double value)
```

Returns how many native elements equal `value`. A null array returns zero; the scan is linear.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:149`

ArrayDbl.equal

```
int ArrayDbl.equal(ArrayDbl a, ArrayDbl b)
```

Reports elementwise native equality. Two null arrays compare equal; one null array or different lengths compare unequal.

Source: `lib/typed-array.x:149`

ArrayDbl.find

```
int ArrayDbl.find(ArrayDbl array, double value)
```

Returns the first index whose native element equals `value`, or −1. A null array returns −1; the scan is linear and preserves source order.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:149`

ArrayDbl.getindex

```
double ArrayDbl.getindex(ArrayDbl array, int index)
```

Reads raw `index` with the bounds and null preconditions of a C pointer. Negative indices are not normalized. Use `try_get` for a checked read.

Source: `lib/typed-array.x:149`

ArrayDbl.getslice

```
ArrayDbl ArrayDbl.getslice(ArrayDbl array, int start, int end, int step)
```

Returns a fresh packed array for normalized `[start:end:step]`. Negative bounds count from the end and a negative step walks backward.

Raises: `<bad-arg>` for a null array or zero step, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:149`

ArrayDbl.indexof

```
int ArrayDbl.indexof(ArrayDbl array, double value)
```

Returns the same first-match index as `ArrayDbl.find`.

Source: `lib/typed-array.x:149`

ArrayDbl.insert

```
double ArrayDbl.insert(ArrayDbl array, int index, double value)
```

Inserts `value` at normalized `index` and returns it. The accepted range includes the position after the last element; negative indices are normalized against that wider range. Boxed `Array` returns `void` for a null array or out-of-range index without mutation.

Raises: `<bad-arg>` for a null typed array or out-of-range typed-array index, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:149`

ArrayDbl.iter

```
Iter ArrayDbl.iter(ArrayDbl array, Iter dest)
```

Initializes `dest` to lazily yield boxed elements in index order. Returns `dest`, whose storage is caller-owned. The iterator borrows `array` and reads its current storage on each step, so both must remain live and mutation can change the remaining traversal. A null `dest` returns NULL. A null array, or one longer than the integer index domain, is exhausted. A boxed wide element is `Scope`-owned; retaining one after a pull requires its owning `Scope` to remain live.

Raises: any cause from boxing an element.

Source: `lib/typed-array.x:155`

ArrayDbl.new

```
ArrayDbl ArrayDbl.new(void)
```

Returns a fresh empty packed array owned by the current `Scope`.

Raises: `<alloc-fail>` when storage cannot be allocated.

Source: `lib/typed-array.x:149`

ArrayDbl.postfixindex

```
double ArrayDbl.postfixindex(ArrayDbl array, int index, Symbol op)
```

Applies `++` or `--` at raw `index` and returns the previous value. The index has the unchecked C-pointer preconditions of bracket access.

Raises: `<bad-op>` for another operator. The failure does not return.

Source: `lib/typed-array.x:152`

ArrayDbl.push

```
double ArrayDbl.push(ArrayDbl array, double value)
```

Appends `value` and returns it. Boxed `Array` returns `void` for a null array.

Raises: `<bad-arg>` for a null typed array, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:149`

ArrayDbl.remove

```
double ArrayDbl.remove(ArrayDbl array, int index)
```

Removes and returns the element at normalized `index`. Negative indices count from the end; following elements shift left. Boxed `Array` returns `void` for a null array or out-of-range index.

Raises: `<bad-arg>` for a null typed array or out-of-range typed-array index, or `<size-limit>` when the length exceeds the integer index domain in either family.

Source: `lib/typed-array.x:149`

ArrayDbl.remslice

```
ArrayDbl ArrayDbl.remslice(ArrayDbl array, int start, int end)
```

Removes the region `array[start:end]` and returns a fresh packed array. Bounds normalize as slice bounds, so a negative `end` is a stop. Reversed bounds are swapped, and the removed values retain their order.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:149`

ArrayDbl.repr

```
String ArrayDbl.repr(ArrayDbl array)
```

Returns the readable packed-Array representation.

Source: `lib/typed-array.x:150`

ArrayDbl.reverse

```
ArrayDbl ArrayDbl.reverse(ArrayDbl array)
```

Reverses `array` in place and returns that same array. A null or one-element array is returned unchanged.

Source: `lib/typed-array.x:149`

ArrayDbl.setindex

```
double ArrayDbl.setindex(ArrayDbl array, int index, double value)
```

Stores and returns `value` at raw `index`, with C-pointer preconditions. Negative indices are not normalized; the caller supplies a valid index.

Source: `lib/typed-array.x:149`

ArrayDbl.setslice

```
ArrayDbl ArrayDbl.setslice(ArrayDbl array, int start, int end, ArrayDbl values)
```

Replaces the region `array[start:end]` and returns `array`. Bounds normalize as slice bounds, so a negative `end` is a stop. Reversed bounds are swapped; null `values` deletes the region. Passing `array` itself as `values` is supported and copies before mutation.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` while copying self-aliased values or growing storage for the replacement. These failures leave `array` unchanged.

Source: `lib/typed-array.x:149`

ArrayDbl.shift

```
double ArrayDbl.shift(ArrayDbl array)
```

Removes and returns the first element, shifting the remainder left. This operation is O(n). Boxed `Array` returns `void` for a null or empty array.

Raises: `<bad-arg>` for a null or empty typed array, or `<size-limit>` when the length exceeds the integer index domain in either family.

Source: `lib/typed-array.x:149`

ArrayDbl.splice

```
ArrayDbl ArrayDbl.splice(ArrayDbl array, int index, int remove_count, ArrayDbl values)
```

Replaces `remove_count` values at normalized `index` and returns them. The result is a fresh packed array. A nonpositive count removes nothing; null `values` performs deletion, and self-splicing is supported.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:149`

ArrayDbl.str

```
String ArrayDbl.str(ArrayDbl array)
```

Returns the packed-Array display String.

Source: `lib/typed-array.x:150`

ArrayDbl.take_last

```
double ArrayDbl.take_last(ArrayDbl array)
```

Removes and returns the last element. Boxed `Array` returns `void` for a null or empty array.

Raises: `<bad-arg>` for a null or empty typed array.

Source: `lib/typed-array.x:149`

ArrayDbl.try_get

```
int ArrayDbl.try_get(ArrayDbl array, int index, double &?out)
```

Reads normalized `index` into `out` and reports whether it exists. Negative indices count from the end. A null array, null `out`, or out-of-range index returns zero without writing; success returns one.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:149`

ArrayDbl.try_next

```
int ArrayDbl.try_next(ArrayDbl array, int &?cursor, double &?out)
```

Writes the next native element, advances `cursor`, and returns one. Initialize the caller-owned cursor to zero. A null argument, negative cursor, exhaustion, or an array outside the integer index domain returns zero without changing `cursor` or `out`. Structural mutation invalidates the cursor.

Source: `lib/typed-array.x:149`

ArrayDbl.try_take_last

```
int ArrayDbl.try_take_last(ArrayDbl array, double &?out)
```

Removes the last element into `out` and returns one when present. A null or empty array, or null `out`, returns zero without writing or changing the array.

Source: `lib/typed-array.x:149`

ArrayDbl.unshift

```
double ArrayDbl.unshift(ArrayDbl array, double value)
```

Inserts `value` at the front and returns it, shifting elements right. This operation is O(n). Boxed `Array` returns `void` for a null array.

Raises: `<bad-arg>` for a null typed array, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:149`

ArrayDbl.updateindex

```
double ArrayDbl.updateindex(ArrayDbl array, int index, Symbol op, double rhs)
```

Applies a floating arithmetic `op` and returns the stored value. The supported operators are `+`, `-`, `*`, and `/`; the raw index has the unchecked C-pointer preconditions of bracket access.

Raises: `<bad-types>` for another operator, before changing the element.

Source: `lib/typed-array.x:152`

ArrayDbl.var

```
Var ArrayDbl.var(ArrayDbl array)
```

Boxes `array` with its registered packed-array tag without copying. The boxed and typed values share the same mutable object identity.

Source: `lib/typed-array.x:153`

ArrayDbl.write_repr

```
Buffer ArrayDbl.write_repr(ArrayDbl array, Buffer out)
```

Appends the readable packed-Array representation to `out`.

Source: `lib/typed-array.x:150`

ArrayDbl.write_str

```
Buffer ArrayDbl.write_str(ArrayDbl array, Buffer out)
```

Appends the packed-Array display text to `out`.

Source: `lib/typed-array.x:150`

ArrayFloat

ArrayFloat.array

```
Array ArrayFloat.array(ArrayFloat array)
```

Returns a fresh ordinary Array containing boxed elements in source order. A null input returns NULL. The typed Array and its storage are unchanged.

Raises: any cause from boxing or appending an element.

Source: `lib/typed-array.x:142`

ArrayFloat.block

```
Block ArrayFloat.block(ArrayFloat array)
```

Views `array` as the same underlying `Block` storage.

Source: `lib/typed-array.x:144`

ArrayFloat.compare

```
int ArrayFloat.compare(ArrayFloat a, ArrayFloat b)
```

Compares packed Arrays lexicographically with the element family's ordering. Identical handles compare equal and NULL sorts first.

Source: `lib/typed-array.x:141`

ArrayFloat.concat

```
ArrayFloat ArrayFloat.concat(ArrayFloat a, ArrayFloat b)
```

Returns a fresh array holding `a` followed by `b`. Neither input is modified. A null `a` returns NULL, while a null `b` produces an independent copy of `a`.

Raises: `<size-limit>` or `<alloc-fail>` when the result cannot be represented or allocated.

Source: `lib/typed-array.x:140`

ArrayFloat.contains

```
int ArrayFloat.contains(ArrayFloat array, float value)
```

Reports whether a native element equals `value`. The scan is linear. Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:140`

ArrayFloat.copy

```
ArrayFloat ArrayFloat.copy(ArrayFloat array)
```

Returns a fresh packed copy, or NULL for a null array. The result has independent mutable storage containing the same stored values.

Raises: `<size-limit>` or `<alloc-fail>` when the copy cannot be represented or allocated.

Source: `lib/typed-array.x:140`

ArrayFloat.count

```
int ArrayFloat.count(ArrayFloat array, float value)
```

Returns how many native elements equal `value`. A null array returns zero; the scan is linear.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:140`

ArrayFloat.equal

```
int ArrayFloat.equal(ArrayFloat a, ArrayFloat b)
```

Reports elementwise native equality. Two null arrays compare equal; one null array or different lengths compare unequal.

Source: `lib/typed-array.x:140`

ArrayFloat.find

```
int ArrayFloat.find(ArrayFloat array, float value)
```

Returns the first index whose native element equals `value`, or −1. A null array returns −1; the scan is linear and preserves source order.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:140`

ArrayFloat.getindex

```
float ArrayFloat.getindex(ArrayFloat array, int index)
```

Reads raw `index` with the bounds and null preconditions of a C pointer. Negative indices are not normalized. Use `try_get` for a checked read.

Source: `lib/typed-array.x:140`

ArrayFloat.getslice

```
ArrayFloat ArrayFloat.getslice(ArrayFloat array, int start, int end, int step)
```

Returns a fresh packed array for normalized `[start:end:step]`. Negative bounds count from the end and a negative step walks backward.

Raises: `<bad-arg>` for a null array or zero step, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:140`

ArrayFloat.indexof

```
int ArrayFloat.indexof(ArrayFloat array, float value)
```

Returns the same first-match index as `ArrayFloat.find`.

Source: `lib/typed-array.x:140`

ArrayFloat.insert

```
float ArrayFloat.insert(ArrayFloat array, int index, float value)
```

Inserts `value` at normalized `index` and returns it. The accepted range includes the position after the last element; negative indices are normalized against that wider range. Boxed `Array` returns `void` for a null array or out-of-range index without mutation.

Raises: `<bad-arg>` for a null typed array or out-of-range typed-array index, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:140`

ArrayFloat.iter

```
Iter ArrayFloat.iter(ArrayFloat array, Iter dest)
```

Initializes `dest` to lazily yield boxed elements in index order. Returns `dest`, whose storage is caller-owned. The iterator borrows `array` and reads its current storage on each step, so both must remain live and mutation can change the remaining traversal. A null `dest` returns NULL. A null array, or one longer than the integer index domain, is exhausted. A boxed wide element is `Scope`-owned; retaining one after a pull requires its owning `Scope` to remain live.

Raises: any cause from boxing an element.

Source: `lib/typed-array.x:146`

ArrayFloat.new

```
ArrayFloat ArrayFloat.new(void)
```

Returns a fresh empty packed array owned by the current `Scope`.

Raises: `<alloc-fail>` when storage cannot be allocated.

Source: `lib/typed-array.x:140`

ArrayFloat.postfixindex

```
float ArrayFloat.postfixindex(ArrayFloat array, int index, Symbol op)
```

Applies `++` or `--` at raw `index` and returns the previous value. The index has the unchecked C-pointer preconditions of bracket access.

Raises: `<bad-op>` for another operator. The failure does not return.

Source: `lib/typed-array.x:143`

ArrayFloat.push

```
float ArrayFloat.push(ArrayFloat array, float value)
```

Appends `value` and returns it. Boxed `Array` returns `void` for a null array.

Raises: `<bad-arg>` for a null typed array, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:140`

ArrayFloat.remove

```
float ArrayFloat.remove(ArrayFloat array, int index)
```

Removes and returns the element at normalized `index`. Negative indices count from the end; following elements shift left. Boxed `Array` returns `void` for a null array or out-of-range index.

Raises: `<bad-arg>` for a null typed array or out-of-range typed-array index, or `<size-limit>` when the length exceeds the integer index domain in either family.

Source: `lib/typed-array.x:140`

ArrayFloat.remslice

```
ArrayFloat ArrayFloat.remslice(ArrayFloat array, int start, int end)
```

Removes the region `array[start:end]` and returns a fresh packed array. Bounds normalize as slice bounds, so a negative `end` is a stop. Reversed bounds are swapped, and the removed values retain their order.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:140`

ArrayFloat.repr

```
String ArrayFloat.repr(ArrayFloat array)
```

Returns the readable packed-Array representation.

Source: `lib/typed-array.x:141`

ArrayFloat.reverse

```
ArrayFloat ArrayFloat.reverse(ArrayFloat array)
```

Reverses `array` in place and returns that same array. A null or one-element array is returned unchanged.

Source: `lib/typed-array.x:140`

ArrayFloat.setindex

```
float ArrayFloat.setindex(ArrayFloat array, int index, float value)
```

Stores and returns `value` at raw `index`, with C-pointer preconditions. Negative indices are not normalized; the caller supplies a valid index.

Source: `lib/typed-array.x:140`

ArrayFloat.setslice

```
ArrayFloat ArrayFloat.setslice(ArrayFloat array, int start, int end, ArrayFloat values)
```

Replaces the region `array[start:end]` and returns `array`. Bounds normalize as slice bounds, so a negative `end` is a stop. Reversed bounds are swapped; null `values` deletes the region. Passing `array` itself as `values` is supported and copies before mutation.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` while copying self-aliased values or growing storage for the replacement. These failures leave `array` unchanged.

Source: `lib/typed-array.x:140`

ArrayFloat.shift

```
float ArrayFloat.shift(ArrayFloat array)
```

Removes and returns the first element, shifting the remainder left. This operation is O(n). Boxed `Array` returns `void` for a null or empty array.

Raises: `<bad-arg>` for a null or empty typed array, or `<size-limit>` when the length exceeds the integer index domain in either family.

Source: `lib/typed-array.x:140`

ArrayFloat.splice

```
ArrayFloat ArrayFloat.splice(ArrayFloat array, int index, int remove_count, ArrayFloat values)
```

Replaces `remove_count` values at normalized `index` and returns them. The result is a fresh packed array. A nonpositive count removes nothing; null `values` performs deletion, and self-splicing is supported.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:140`

ArrayFloat.str

```
String ArrayFloat.str(ArrayFloat array)
```

Returns the packed-Array display String.

Source: `lib/typed-array.x:141`

ArrayFloat.take_last

```
float ArrayFloat.take_last(ArrayFloat array)
```

Removes and returns the last element. Boxed `Array` returns `void` for a null or empty array.

Raises: `<bad-arg>` for a null or empty typed array.

Source: `lib/typed-array.x:140`

ArrayFloat.try_get

```
int ArrayFloat.try_get(ArrayFloat array, int index, float &?out)
```

Reads normalized `index` into `out` and reports whether it exists. Negative indices count from the end. A null array, null `out`, or out-of-range index returns zero without writing; success returns one.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:140`

ArrayFloat.try_next

```
int ArrayFloat.try_next(ArrayFloat array, int &?cursor, float &?out)
```

Writes the next native element, advances `cursor`, and returns one. Initialize the caller-owned cursor to zero. A null argument, negative cursor, exhaustion, or an array outside the integer index domain returns zero without changing `cursor` or `out`. Structural mutation invalidates the cursor.

Source: `lib/typed-array.x:140`

ArrayFloat.try_take_last

```
int ArrayFloat.try_take_last(ArrayFloat array, float &?out)
```

Removes the last element into `out` and returns one when present. A null or empty array, or null `out`, returns zero without writing or changing the array.

Source: `lib/typed-array.x:140`

ArrayFloat.unshift

```
float ArrayFloat.unshift(ArrayFloat array, float value)
```

Inserts `value` at the front and returns it, shifting elements right. This operation is O(n). Boxed `Array` returns `void` for a null array.

Raises: `<bad-arg>` for a null typed array, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:140`

ArrayFloat.updateindex

```
float ArrayFloat.updateindex(ArrayFloat array, int index, Symbol op, float rhs)
```

Applies a floating arithmetic `op` and returns the stored value. The supported operators are `+`, `-`, `*`, and `/`; the raw index has the unchecked C-pointer preconditions of bracket access.

Raises: `<bad-types>` for another operator, before changing the element.

Source: `lib/typed-array.x:143`

ArrayFloat.var

```
Var ArrayFloat.var(ArrayFloat array)
```

Boxes `array` with its registered packed-array tag without copying. The boxed and typed values share the same mutable object identity.

Source: `lib/typed-array.x:144`

ArrayFloat.write_repr

```
Buffer ArrayFloat.write_repr(ArrayFloat array, Buffer out)
```

Appends the readable packed-Array representation to `out`.

Source: `lib/typed-array.x:141`

ArrayFloat.write_str

```
Buffer ArrayFloat.write_str(ArrayFloat array, Buffer out)
```

Appends the packed-Array display text to `out`.

Source: `lib/typed-array.x:141`

ArrayInt

ArrayInt.array

```
Array ArrayInt.array(ArrayInt array)
```

Returns a fresh ordinary Array containing boxed elements in source order. A null input returns NULL. The typed Array and its storage are unchanged.

Raises: any cause from boxing or appending an element.

Source: `lib/typed-array.x:124`

ArrayInt.block

```
Block ArrayInt.block(ArrayInt array)
```

Views `array` as the same underlying `Block` storage.

Source: `lib/typed-array.x:126`

ArrayInt.compare

```
int ArrayInt.compare(ArrayInt a, ArrayInt b)
```

Compares packed Arrays lexicographically with the element family's ordering. Identical handles compare equal and NULL sorts first.

Source: `lib/typed-array.x:123`

ArrayInt.concat

```
ArrayInt ArrayInt.concat(ArrayInt a, ArrayInt b)
```

Returns a fresh array holding `a` followed by `b`. Neither input is modified. A null `a` returns NULL, while a null `b` produces an independent copy of `a`.

Raises: `<size-limit>` or `<alloc-fail>` when the result cannot be represented or allocated.

Source: `lib/typed-array.x:122`

ArrayInt.contains

```
int ArrayInt.contains(ArrayInt array, int value)
```

Reports whether a native element equals `value`. The scan is linear. Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:122`

ArrayInt.copy

```
ArrayInt ArrayInt.copy(ArrayInt array)
```

Returns a fresh packed copy, or NULL for a null array. The result has independent mutable storage containing the same stored values.

Raises: `<size-limit>` or `<alloc-fail>` when the copy cannot be represented or allocated.

Source: `lib/typed-array.x:122`

ArrayInt.count

```
int ArrayInt.count(ArrayInt array, int value)
```

Returns how many native elements equal `value`. A null array returns zero; the scan is linear.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:122`

ArrayInt.equal

```
int ArrayInt.equal(ArrayInt a, ArrayInt b)
```

Reports elementwise native equality. Two null arrays compare equal; one null array or different lengths compare unequal.

Source: `lib/typed-array.x:122`

ArrayInt.find

```
int ArrayInt.find(ArrayInt array, int value)
```

Returns the first index whose native element equals `value`, or −1. A null array returns −1; the scan is linear and preserves source order.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:122`

ArrayInt.getindex

```
int ArrayInt.getindex(ArrayInt array, int index)
```

Reads raw `index` with the bounds and null preconditions of a C pointer. Negative indices are not normalized. Use `try_get` for a checked read.

Source: `lib/typed-array.x:122`

ArrayInt.getslice

```
ArrayInt ArrayInt.getslice(ArrayInt array, int start, int end, int step)
```

Returns a fresh packed array for normalized `[start:end:step]`. Negative bounds count from the end and a negative step walks backward.

Raises: `<bad-arg>` for a null array or zero step, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:122`

ArrayInt.indexof

```
int ArrayInt.indexof(ArrayInt array, int value)
```

Returns the same first-match index as `ArrayInt.find`.

Source: `lib/typed-array.x:122`

ArrayInt.insert

```
int ArrayInt.insert(ArrayInt array, int index, int value)
```

Inserts `value` at normalized `index` and returns it. The accepted range includes the position after the last element; negative indices are normalized against that wider range. Boxed `Array` returns `void` for a null array or out-of-range index without mutation.

Raises: `<bad-arg>` for a null typed array or out-of-range typed-array index, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:122`

ArrayInt.iter

```
Iter ArrayInt.iter(ArrayInt array, Iter dest)
```

Initializes `dest` to lazily yield boxed elements in index order. Returns `dest`, whose storage is caller-owned. The iterator borrows `array` and reads its current storage on each step, so both must remain live and mutation can change the remaining traversal. A null `dest` returns NULL. A null array, or one longer than the integer index domain, is exhausted. A boxed wide element is `Scope`-owned; retaining one after a pull requires its owning `Scope` to remain live.

Raises: any cause from boxing an element.

Source: `lib/typed-array.x:128`

ArrayInt.new

```
ArrayInt ArrayInt.new(void)
```

Returns a fresh empty packed array owned by the current `Scope`.

Raises: `<alloc-fail>` when storage cannot be allocated.

Source: `lib/typed-array.x:122`

ArrayInt.postfixindex

```
int ArrayInt.postfixindex(ArrayInt array, int index, Symbol op)
```

Applies `++` or `--` at raw `index` and returns the previous value. The index has the unchecked C-pointer preconditions of bracket access.

Raises: `<bad-op>` for another operator. The failure does not return.

Source: `lib/typed-array.x:125`

ArrayInt.push

```
int ArrayInt.push(ArrayInt array, int value)
```

Appends `value` and returns it. Boxed `Array` returns `void` for a null array.

Raises: <bad-arg> for a null typed array, <size-limit> at the integer index limit, or <alloc-fail> when storage cannot grow.

Source: lib/typed-array.x:122

ArrayInt.remove

```
int ArrayInt.remove(ArrayInt array, int index)
```

Removes and returns the element at normalized index. Negative indices count from the end; following elements shift left. Boxed Array returns void for a null array or out-of-range index.

Raises: <bad-arg> for a null typed array or out-of-range typed-array index, or <size-limit> when the length exceeds the integer index domain in either family.

Source: lib/typed-array.x:122

ArrayInt.remslice

```
ArrayInt ArrayInt.remslice(ArrayInt array, int start, int end)
```

Removes the region array[start:end] and returns a fresh packed array. Bounds normalize as slice bounds, so a negative end is a stop. Reversed bounds are swapped, and the removed values retain their order.

Raises: <bad-arg> for a null array, <size-limit> outside the integer index domain, or <alloc-fail> on allocation failure.

Source: lib/typed-array.x:122

ArrayInt.repr

```
String ArrayInt.repr(ArrayInt array)
```

Returns the readable packed-Array representation.

Source: lib/typed-array.x:123

ArrayInt.reverse

```
ArrayInt ArrayInt.reverse(ArrayInt array)
```

Reverses array in place and returns that same array. A null or one-element array is returned unchanged.

Source: lib/typed-array.x:122

ArrayInt.setindex

```
int ArrayInt.setindex(ArrayInt array, int index, int value)
```

Stores and returns value at raw index, with C-pointer preconditions. Negative indices are not normalized; the caller supplies a valid index.

Source: lib/typed-array.x:122

ArrayInt.setslice

```
ArrayInt ArrayInt.setslice(ArrayInt array, int start, int end, ArrayInt values)
```

Replaces the region array[start:end] and returns array. Bounds normalize as slice bounds, so a negative end is a stop. Reversed bounds are swapped; null values deletes the region. Passing array itself as values is supported and copies before mutation.

Raises: <bad-arg> for a null array, <size-limit> outside the integer index domain, or <alloc-fail> while copying self-aliased values or growing storage for the replacement. These failures leave array unchanged.

Source: lib/typed-array.x:122

ArrayInt.shift

```
int ArrayInt.shift(ArrayInt array)
```

Removes and returns the first element, shifting the remainder left. This operation is O(n). Boxed `Array` returns `void` for a null or empty array.

Raises: `<bad-arg>` for a null or empty typed array, or `<size-limit>` when the length exceeds the integer index domain in either family.

Source: `lib/typed-array.x:122`

ArrayInt.splice

```
ArrayInt ArrayInt.splice(ArrayInt array, int index, int remove_count, ArrayInt values)
```

Replaces `remove_count` values at normalized `index` and returns them. The result is a fresh packed array. A nonpositive count removes nothing; null `values` performs deletion, and self-splicing is supported.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:122`

ArrayInt.str

```
String ArrayInt.str(ArrayInt array)
```

Returns the packed-Array display String.

Source: `lib/typed-array.x:123`

ArrayInt.take_last

```
int ArrayInt.take_last(ArrayInt array)
```

Removes and returns the last element. Boxed `Array` returns `void` for a null or empty array.

Raises: `<bad-arg>` for a null or empty typed array.

Source: `lib/typed-array.x:122`

ArrayInt.try_get

```
int ArrayInt.try_get(ArrayInt array, int index, int &?out)
```

Reads normalized `index` into `out` and reports whether it exists. Negative indices count from the end. A null array, null `out`, or out-of-range index returns zero without writing; success returns one.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:122`

ArrayInt.try_next

```
int ArrayInt.try_next(ArrayInt array, int &?cursor, int &?out)
```

Writes the next native element, advances `cursor`, and returns one. Initialize the caller-owned cursor to zero. A null argument, negative cursor, exhaustion, or an array outside the integer index domain returns zero without changing `cursor` or `out`. Structural mutation invalidates the cursor.

Source: `lib/typed-array.x:122`

ArrayInt.try_take_last

```
int ArrayInt.try_take_last(ArrayInt array, int &?out)
```

Removes the last element into `out` and returns one when present. A null or empty array, or null `out`, returns zero without writing or changing the array.

Source: `lib/typed-array.x:122`

ArrayInt.unshift

```
int ArrayInt.unshift(ArrayInt array, int value)
```

Inserts `value` at the front and returns it, shifting elements right. This operation is O(n). Boxed `Array` returns `void` for a null array.

Raises: `<bad-arg>` for a null typed array, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:122`

ArrayInt.updateindex

```
int ArrayInt.updateindex(ArrayInt array, int index, Symbol op, int rhs)
```

Applies integer `op` to raw element `index` and returns the stored value. The index has the unchecked C-pointer preconditions of bracket access. Arithmetic follows the generated C operators, integer promotions, and result conversion. A failing operation leaves the element unchanged.

Raises: `<div-zero>`, `<bad-shift>`, or `<bad-op>` for its corresponding invalid operation.

Source: `lib/typed-array.x:125`

ArrayInt.var

```
Var ArrayInt.var(ArrayInt array)
```

Boxes `array` with its registered packed-array tag without copying. The boxed and typed values share the same mutable object identity.

Source: `lib/typed-array.x:126`

ArrayInt.write_repr

```
Buffer ArrayInt.write_repr(ArrayInt array, Buffer out)
```

Appends the readable packed-Array representation to `out`.

Source: `lib/typed-array.x:123`

ArrayInt.write_str

```
Buffer ArrayInt.write_str(ArrayInt array, Buffer out)
```

Appends the packed-Array display text to `out`.

Source: `lib/typed-array.x:123`

ArrayLong

ArrayLong.array

```
Array ArrayLong.array(ArrayLong array)
```

Returns a fresh ordinary Array containing boxed elements in source order. A null input returns NULL. The typed Array and its storage are unchanged.

Raises: any cause from boxing or appending an element.

Source: `lib/typed-array.x:133`

ArrayLong.block

```
Block ArrayLong.block(ArrayLong array)
```

Views `array` as the same underlying `Block` storage.

Source: `lib/typed-array.x:135`

ArrayLong.compare

```
int ArrayLong.compare(ArrayLong a, ArrayLong b)
```

Compares packed Arrays lexicographically with the element family's ordering. Identical handles compare equal and NULL sorts first.

Source: `lib/typed-array.x:132`

ArrayLong.concat

```
ArrayLong ArrayLong.concat(ArrayLong a, ArrayLong b)
```

Returns a fresh array holding `a` followed by `b`. Neither input is modified. A null `a` returns NULL, while a null `b` produces an independent copy of `a`.

Raises: `<size-limit>` or `<alloc-fail>` when the result cannot be represented or allocated.

Source: `lib/typed-array.x:131`

ArrayLong.contains

```
int ArrayLong.contains(ArrayLong array, long value)
```

Reports whether a native element equals `value`. The scan is linear. Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:131`

ArrayLong.copy

```
ArrayLong ArrayLong.copy(ArrayLong array)
```

Returns a fresh packed copy, or NULL for a null array. The result has independent mutable storage containing the same stored values.

Raises: `<size-limit>` or `<alloc-fail>` when the copy cannot be represented or allocated.

Source: `lib/typed-array.x:131`

ArrayLong.count

```
int ArrayLong.count(ArrayLong array, long value)
```

Returns how many native elements equal `value`. A null array returns zero; the scan is linear.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:131`

ArrayLong.equal

```
int ArrayLong.equal(ArrayLong a, ArrayLong b)
```

Reports elementwise native equality. Two null arrays compare equal; one null array or different lengths compare unequal.

Source: `lib/typed-array.x:131`

ArrayLong.find

```
int ArrayLong.find(ArrayLong array, long value)
```

Returns the first index whose native element equals `value`, or −1. A null array returns −1; the scan is linear and preserves source order.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:131`

ArrayLong.getindex

```
long ArrayLong.getindex(ArrayLong array, int index)
```

Reads raw `index` with the bounds and null preconditions of a C pointer. Negative indices are not normalized. Use `try_get` for a checked read.

Source: `lib/typed-array.x:131`

ArrayLong.getslice

```
ArrayLong ArrayLong.getslice(ArrayLong array, int start, int end, int step)
```

Returns a fresh packed array for normalized `[start:end:step]`. Negative bounds count from the end and a negative step walks backward.

Raises: `<bad-arg>` for a null array or zero step, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:131`

ArrayLong.indexof

```
int ArrayLong.indexof(ArrayLong array, long value)
```

Returns the same first-match index as `ArrayLong.find`.

Source: `lib/typed-array.x:131`

ArrayLong.insert

```
long ArrayLong.insert(ArrayLong array, int index, long value)
```

Inserts `value` at normalized `index` and returns it. The accepted range includes the position after the last element; negative indices are normalized against that wider range. Boxed `Array` returns `void` for a null array or out-of-range index without mutation.

Raises: `<bad-arg>` for a null typed array or out-of-range typed-array index, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:131`

ArrayLong.iter

```
Iter ArrayLong.iter(ArrayLong array, Iter dest)
```

Initializes `dest` to lazily yield boxed elements in index order. Returns `dest`, whose storage is caller-owned. The iterator borrows `array` and reads its current storage on each step, so both must remain live and mutation can change the remaining traversal. A null `dest` returns NULL. A null array, or one longer than the integer index domain, is exhausted. A boxed wide element is `Scope`-owned; retaining one after a pull requires its owning `Scope` to remain live.

Raises: any cause from boxing an element.

Source: `lib/typed-array.x:137`

ArrayLong.new

```
ArrayLong ArrayLong.new(void)
```

Returns a fresh empty packed array owned by the current `Scope`.

Raises: `<alloc-fail>` when storage cannot be allocated.

Source: `lib/typed-array.x:131`

ArrayLong.postfixindex

```
long ArrayLong.postfixindex(ArrayLong array, int index, Symbol op)
```

Applies ++ or -- at raw index and returns the previous value. The index has the unchecked C-pointer preconditions of bracket access.

Raises: <bad-op> for another operator. The failure does not return.

Source: lib/typed-array.x:134

ArrayLong.push

```
long ArrayLong.push(ArrayLong array, long value)
```

Appends value and returns it. Boxed Array returns void for a null array.

Raises: <bad-arg> for a null typed array, <size-limit> at the integer index limit, or <alloc-fail> when storage cannot grow.

Source: lib/typed-array.x:131

ArrayLong.remove

```
long ArrayLong.remove(ArrayLong array, int index)
```

Removes and returns the element at normalized index. Negative indices count from the end; following elements shift left. Boxed Array returns void for a null array or out-of-range index.

Raises: <bad-arg> for a null typed array or out-of-range typed-array index, or <size-limit> when the length exceeds the integer index domain in either family.

Source: lib/typed-array.x:131

ArrayLong.remslice

```
ArrayLong ArrayLong.remslice(ArrayLong array, int start, int end)
```

Removes the region array[start:end] and returns a fresh packed array. Bounds normalize as slice bounds, so a negative end is a stop. Reversed bounds are swapped, and the removed values retain their order.

Raises: <bad-arg> for a null array, <size-limit> outside the integer index domain, or <alloc-fail> on allocation failure.

Source: lib/typed-array.x:131

ArrayLong.repr

```
String ArrayLong.repr(ArrayLong array)
```

Returns the readable packed-Array representation.

Source: lib/typed-array.x:132

ArrayLong.reverse

```
ArrayLong ArrayLong.reverse(ArrayLong array)
```

Reverses array in place and returns that same array. A null or one-element array is returned unchanged.

Source: lib/typed-array.x:131

ArrayLong.setindex

```
long ArrayLong.setindex(ArrayLong array, int index, long value)
```

Stores and returns value at raw index, with C-pointer preconditions. Negative indices are not normalized; the caller supplies a valid index.

Source: lib/typed-array.x:131

ArrayLong.setslice

```
ArrayLong ArrayLong.setslice(ArrayLong array, int start, int end, ArrayLong values)
```

Replaces the region `array[start:end]` and returns `array`. Bounds normalize as slice bounds, so a negative `end` is a stop. Reversed bounds are swapped; null `values` deletes the region. Passing `array` itself as `values` is supported and copies before mutation.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` while copying self-aliased values or growing storage for the replacement. These failures leave `array` unchanged.

Source: `lib/typed-array.x:131`

ArrayLong.shift

```
long ArrayLong.shift(ArrayLong array)
```

Removes and returns the first element, shifting the remainder left. This operation is O(n). Boxed `Array` returns `void` for a null or empty array.

Raises: `<bad-arg>` for a null or empty typed array, or `<size-limit>` when the length exceeds the integer index domain in either family.

Source: `lib/typed-array.x:131`

ArrayLong.splice

```
ArrayLong ArrayLong.splice(ArrayLong array, int index, int remove_count, ArrayLong values)
```

Replaces `remove_count` values at normalized `index` and returns them. The result is a fresh packed array. A nonpositive count removes nothing; null `values` performs deletion, and self-splicing is supported.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:131`

ArrayLong.str

```
String ArrayLong.str(ArrayLong array)
```

Returns the packed-Array display String.

Source: `lib/typed-array.x:132`

ArrayLong.take_last

```
long ArrayLong.take_last(ArrayLong array)
```

Removes and returns the last element. Boxed `Array` returns `void` for a null or empty array.

Raises: `<bad-arg>` for a null or empty typed array.

Source: `lib/typed-array.x:131`

ArrayLong.try_get

```
int ArrayLong.try_get(ArrayLong array, int index, long &?out)
```

Reads normalized `index` into `out` and reports whether it exists. Negative indices count from the end. A null array, null `out`, or out-of-range index returns zero without writing; success returns one.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:131`

ArrayLong.try_next

```
int ArrayLong.try_next(ArrayLong array, int &?cursor, long &?out)
```

Writes the next native element, advances `cursor`, and returns one. Initialize the caller-owned cursor to zero. A null argument, negative cursor, exhaustion, or an array outside the integer index domain returns zero without changing `cursor` or `out`. Structural mutation invalidates the cursor.

Source: `lib/typed-array.x:131`

ArrayLong.try_take_last

```
int ArrayLong.try_take_last(ArrayLong array, long &?out)
```

Removes the last element into `out` and returns one when present. A null or empty array, or null `out`, returns zero without writing or changing the array.

Source: `lib/typed-array.x:131`

ArrayLong.unshift

```
long ArrayLong.unshift(ArrayLong array, long value)
```

Inserts `value` at the front and returns it, shifting elements right. This operation is O(n). Boxed `Array` returns `void` for a null array.

Raises: `<bad-arg>` for a null typed array, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:131`

ArrayLong.updateindex

```
long ArrayLong.updateindex(ArrayLong array, int index, Symbol op, long rhs)
```

Applies integer `op` to raw element `index` and returns the stored value. The index has the unchecked C-pointer preconditions of bracket access. Arithmetic follows the generated C operators, integer promotions, and result conversion. A failing operation leaves the element unchanged.

Raises: `<div-zero>`, `<bad-shift>`, or `<bad-op>` for its corresponding invalid operation.

Source: `lib/typed-array.x:134`

ArrayLong.var

```
Var ArrayLong.var(ArrayLong array)
```

Boxes `array` with its registered packed-array tag without copying. The boxed and typed values share the same mutable object identity.

Source: `lib/typed-array.x:135`

ArrayLong.write_repr

```
Buffer ArrayLong.write_repr(ArrayLong array, Buffer out)
```

Appends the readable packed-Array representation to `out`.

Source: `lib/typed-array.x:132`

ArrayLong.write_str

```
Buffer ArrayLong.write_str(ArrayLong array, Buffer out)
```

Appends the packed-Array display text to `out`.

Source: `lib/typed-array.x:132`

ArrayShort

ArrayShort.array

```
Array ArrayShort.array(ArrayShort array)
```

Returns a fresh ordinary Array containing boxed elements in source order. A null input returns NULL. The typed Array and its storage are unchanged.

Raises: any cause from boxing or appending an element.

Source: `lib/typed-array.x:115`

ArrayShort.block

```
Block ArrayShort.block(ArrayShort array)
```

Views `array` as the same underlying `Block` storage.

Source: `lib/typed-array.x:117`

ArrayShort.compare

```
int ArrayShort.compare(ArrayShort a, ArrayShort b)
```

Compares packed Arrays lexicographically with the element family's ordering. Identical handles compare equal and NULL sorts first.

Source: `lib/typed-array.x:114`

ArrayShort.concat

```
ArrayShort ArrayShort.concat(ArrayShort a, ArrayShort b)
```

Returns a fresh array holding `a` followed by `b`. Neither input is modified. A null `a` returns NULL, while a null `b` produces an independent copy of `a`.

Raises: `<size-limit>` or `<alloc-fail>` when the result cannot be represented or allocated.

Source: `lib/typed-array.x:113`

ArrayShort.contains

```
int ArrayShort.contains(ArrayShort array, short value)
```

Reports whether a native element equals `value`. The scan is linear. Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:113`

ArrayShort.copy

```
ArrayShort ArrayShort.copy(ArrayShort array)
```

Returns a fresh packed copy, or NULL for a null array. The result has independent mutable storage containing the same stored values.

Raises: `<size-limit>` or `<alloc-fail>` when the copy cannot be represented or allocated.

Source: `lib/typed-array.x:113`

ArrayShort.count

```
int ArrayShort.count(ArrayShort array, short value)
```

Returns how many native elements equal `value`. A null array returns zero; the scan is linear.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:113`

ArrayShort.equal

```
int ArrayShort.equal(ArrayShort a, ArrayShort b)
```

Reports elementwise native equality. Two null arrays compare equal; one null array or different lengths compare unequal.

Source: `lib/typed-array.x:113`

ArrayShort.find

```
int ArrayShort.find(ArrayShort array, short value)
```

Returns the first index whose native element equals `value`, or −1. A null array returns −1; the scan is linear and preserves source order.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:113`

ArrayShort.getindex

```
short ArrayShort.getindex(ArrayShort array, int index)
```

Reads raw `index` with the bounds and null preconditions of a C pointer. Negative indices are not normalized. Use `try_get` for a checked read.

Source: `lib/typed-array.x:113`

ArrayShort.getslice

```
ArrayShort ArrayShort.getslice(ArrayShort array, int start, int end, int step)
```

Returns a fresh packed array for normalized `[start:end:step]`. Negative bounds count from the end and a negative step walks backward.

Raises: `<bad-arg>` for a null array or zero step, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:113`

ArrayShort.indexof

```
int ArrayShort.indexof(ArrayShort array, short value)
```

Returns the same first-match index as `ArrayShort.find`.

Source: `lib/typed-array.x:113`

ArrayShort.insert

```
short ArrayShort.insert(ArrayShort array, int index, short value)
```

Inserts `value` at normalized `index` and returns it. The accepted range includes the position after the last element; negative indices are normalized against that wider range. Boxed `Array` returns `void` for a null array or out-of-range index without mutation.

Raises: `<bad-arg>` for a null typed array or out-of-range typed-array index, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:113`

ArrayShort.iter

```
Iter ArrayShort.iter(ArrayShort array, Iter dest)
```

Initializes `dest` to lazily yield boxed elements in index order. Returns `dest`, whose storage is caller-owned. The iterator borrows `array` and reads its current storage on each step, so both must remain live and mutation can change the remaining traversal. A null `dest` returns NULL. A null array, or one longer than the integer index domain, is exhausted. A boxed wide element is `Scope`-owned; retaining one after a pull requires its owning `Scope` to remain live.

Raises: any cause from boxing an element.

Source: `lib/typed-array.x:119`

ArrayShort.new

```
ArrayShort ArrayShort.new(void)
```

Returns a fresh empty packed array owned by the current `Scope`.

Raises: `<alloc-fail>` when storage cannot be allocated.

Source: `lib/typed-array.x:113`

ArrayShort.postfixindex

```
short ArrayShort.postfixindex(ArrayShort array, int index, Symbol op)
```

Applies `++` or `--` at raw `index` and returns the previous value. The index has the unchecked C-pointer preconditions of bracket access.

Raises: `<bad-op>` for another operator. The failure does not return.

Source: `lib/typed-array.x:116`

ArrayShort.push

```
short ArrayShort.push(ArrayShort array, short value)
```

Appends `value` and returns it. Boxed `Array` returns `void` for a null array.

Raises: `<bad-arg>` for a null typed array, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:113`

ArrayShort.remove

```
short ArrayShort.remove(ArrayShort array, int index)
```

Removes and returns the element at normalized `index`. Negative indices count from the end; following elements shift left. Boxed `Array` returns `void` for a null array or out-of-range index.

Raises: `<bad-arg>` for a null typed array or out-of-range typed-array index, or `<size-limit>` when the length exceeds the integer index domain in either family.

Source: `lib/typed-array.x:113`

ArrayShort.remslice

```
ArrayShort ArrayShort.remslice(ArrayShort array, int start, int end)
```

Removes the region `array[start:end]` and returns a fresh packed array. Bounds normalize as slice bounds, so a negative `end` is a stop. Reversed bounds are swapped, and the removed values retain their order.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:113`

ArrayShort.repr

```
String ArrayShort.repr(ArrayShort array)
```

Returns the readable packed-Array representation.

Source: `lib/typed-array.x:114`

ArrayShort.reverse

```
ArrayShort ArrayShort.reverse(ArrayShort array)
```

Reverses `array` in place and returns that same array. A null or one-element array is returned unchanged.

Source: `lib/typed-array.x:113`

ArrayShort.setindex

```
short ArrayShort.setindex(ArrayShort array, int index, short value)
```

Stores and returns `value` at raw `index`, with C-pointer preconditions. Negative indices are not normalized; the caller supplies a valid index.

Source: `lib/typed-array.x:113`

ArrayShort.setslice

```
ArrayShort ArrayShort.setslice(ArrayShort array, int start, int end, ArrayShort values)
```

Replaces the region `array[start:end]` and returns `array`. Bounds normalize as slice bounds, so a negative `end` is a stop. Reversed bounds are swapped; null `values` deletes the region. Passing `array` itself as `values` is supported and copies before mutation.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` while copying self-aliased values or growing storage for the replacement. These failures leave `array` unchanged.

Source: `lib/typed-array.x:113`

ArrayShort.shift

```
short ArrayShort.shift(ArrayShort array)
```

Removes and returns the first element, shifting the remainder left. This operation is O(n). Boxed `Array` returns `void` for a null or empty array.

Raises: `<bad-arg>` for a null or empty typed array, or `<size-limit>` when the length exceeds the integer index domain in either family.

Source: `lib/typed-array.x:113`

ArrayShort.splice

```
ArrayShort ArrayShort.splice(ArrayShort array, int index, int remove_count, ArrayShort values)
```

Replaces `remove_count` values at normalized `index` and returns them. The result is a fresh packed array. A nonpositive count removes nothing; null `values` performs deletion, and self-splicing is supported.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:113`

ArrayShort.str

```
String ArrayShort.str(ArrayShort array)
```

Returns the packed-Array display String.

Source: `lib/typed-array.x:114`

ArrayShort.take_last

`short ArrayShort.take_last(ArrayShort array)`

Removes and returns the last element. Boxed `Array` returns `void` for a null or empty array.

Raises: `<bad-arg>` for a null or empty typed array.

Source: `lib/typed-array.x:113`

ArrayShort.try_get

`int ArrayShort.try_get(ArrayShort array, int index, short &?out)`

Reads normalized `index` into `out` and reports whether it exists. Negative indices count from the end. A null array, null `out`, or out-of-range index returns zero without writing; success returns one.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:113`

ArrayShort.try_next

`int ArrayShort.try_next(ArrayShort array, int &?cursor, short &?out)`

Writes the next native element, advances `cursor`, and returns one. Initialize the caller-owned cursor to zero. A null argument, negative cursor, exhaustion, or an array outside the integer index domain returns zero without changing `cursor` or `out`. Structural mutation invalidates the cursor.

Source: `lib/typed-array.x:113`

ArrayShort.try_take_last

`int ArrayShort.try_take_last(ArrayShort array, short &?out)`

Removes the last element into `out` and returns one when present. A null or empty array, or null `out`, returns zero without writing or changing the array.

Source: `lib/typed-array.x:113`

ArrayShort.unshift

`short ArrayShort.unshift(ArrayShort array, short value)`

Inserts `value` at the front and returns it, shifting elements right. This operation is O(n). Boxed `Array` returns `void` for a null array.

Raises: `<bad-arg>` for a null typed array, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:113`

ArrayShort.updateindex

`short ArrayShort.updateindex(ArrayShort array, int index, Symbol op, short rhs)`

Applies integer `op` to raw element `index` and returns the stored value. The index has the unchecked C-pointer preconditions of bracket access. Arithmetic follows the generated C operators, integer promotions, and result conversion. A failing operation leaves the element unchanged.

Raises: `<div-zero>`, `<bad-shift>`, or `<bad-op>` for its corresponding invalid operation.

Source: `lib/typed-array.x:116`

ArrayShort.var

`Var ArrayShort.var(ArrayShort array)`

Boxes `array` with its registered packed-array tag without copying. The boxed and typed values share the same mutable object identity.

Source: `lib/typed-array.x:117`

ArrayShort.write_repr

```
Buffer ArrayShort.write_repr(ArrayShort array, Buffer out)
```

Appends the readable packed-Array representation to `out`.

Source: `lib/typed-array.x:114`

ArrayShort.write_str

```
Buffer ArrayShort.write_str(ArrayShort array, Buffer out)
```

Appends the packed-Array display text to `out`.

Source: `lib/typed-array.x:114`

ArrayString

ArrayString.array

```
Array ArrayString.array(ArrayString array)
```

Returns a fresh ordinary Array containing boxed elements in source order. A null input returns NULL. The typed Array and its storage are unchanged.

Raises: any cause from boxing or appending an element.

Source: `lib/typed-array.x:160`

ArrayString.block

```
Block ArrayString.block(ArrayString array)
```

Views `array` as the same underlying `Block` storage.

Source: `lib/typed-array.x:162`

ArrayString.compare

```
int ArrayString.compare(ArrayString a, ArrayString b)
```

Compares packed Arrays lexicographically with the element family's ordering. Identical handles compare equal and NULL sorts first.

Source: `lib/typed-array.x:159`

ArrayString.concat

```
ArrayString ArrayString.concat(ArrayString a, ArrayString b)
```

Returns a fresh array holding `a` followed by `b`. Neither input is modified. A null `a` returns NULL, while a null `b` produces an independent copy of `a`.

Raises: `<size-limit>` or `<alloc-fail>` when the result cannot be represented or allocated.

Source: `lib/typed-array.x:158`

ArrayString.contains

```
int ArrayString.contains(ArrayString array, String value)
```

Reports whether a native element equals `value`. The scan is linear. Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:158`

ArrayString.copy

```
ArrayString ArrayString.copy(ArrayString array)
```

Returns a fresh packed copy, or NULL for a null array. The result has independent mutable storage containing the same stored values.

Raises: `<size-limit>` or `<alloc-fail>` when the copy cannot be represented or allocated.

Source: `lib/typed-array.x:158`

ArrayString.count

```
int ArrayString.count(ArrayString array, String value)
```

Returns how many native elements equal `value`. A null array returns zero; the scan is linear.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:158`

ArrayString.equal

```
int ArrayString.equal(ArrayString a, ArrayString b)
```

Reports elementwise native equality. Two null arrays compare equal; one null array or different lengths compare unequal.

Source: `lib/typed-array.x:158`

ArrayString.find

```
int ArrayString.find(ArrayString array, String value)
```

Returns the first index whose native element equals `value`, or −1. A null array returns −1; the scan is linear and preserves source order.

Raises: `<size-limit>` when the length exceeds the integer index domain.

Source: `lib/typed-array.x:158`

ArrayString.getindex

```
String ArrayString.getindex(ArrayString array, int index)
```

Reads raw `index` with the bounds and null preconditions of a C pointer. Negative indices are not normalized. Use `try_get` for a checked read.

Source: `lib/typed-array.x:158`

ArrayString.getslice

```
ArrayString ArrayString.getslice(ArrayString array, int start, int end, int step)
```

Returns a fresh packed array for normalized `[start:end:step]`. Negative bounds count from the end and a negative step walks backward.

Raises: `<bad-arg>` for a null array or zero step, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:158`

ArrayString.indexof

```
int ArrayString.indexof(ArrayString array, String value)
```

Returns the same first-match index as `ArrayString.find`.

Source: `lib/typed-array.x:158`

ArrayString.insert

```
String ArrayString.insert(ArrayString array, int index, String value)
```

Inserts `value` at normalized `index` and returns it. The accepted range includes the position after the last element; negative indices are normalized against that wider range. Boxed `Array` returns `void` for a null array or out-of-range index without mutation.

Raises: `<bad-arg>` for a null typed array or out-of-range typed-array index, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:158`

ArrayString.iter

```
Iter ArrayString.iter(ArrayString array, Iter dest)
```

Initializes `dest` to lazily yield boxed elements in index order. Returns `dest`, whose storage is caller-owned. The iterator borrows `array` and reads its current storage on each step, so both must remain live and mutation can change the remaining traversal. A null `dest` returns NULL. A null array, or one longer than the integer index domain, is exhausted. A boxed wide element is `Scope`-owned; retaining one after a pull requires its owning `Scope` to remain live.

Raises: any cause from boxing an element.

Source: `lib/typed-array.x:165`

ArrayString.new

```
ArrayString ArrayString.new(void)
```

Returns a fresh empty packed array owned by the current `Scope`.

Raises: `<alloc-fail>` when storage cannot be allocated.

Source: `lib/typed-array.x:158`

ArrayString.postfixindex

```
String ArrayString.postfixindex(ArrayString array, int index, Symbol op)
```

Raises `<bad-op>` because String elements have no postfix operation.

Source: `lib/typed-array.x:161`

ArrayString.push

```
String ArrayString.push(ArrayString array, String value)
```

Appends `value` and returns it. Boxed `Array` returns `void` for a null array.

Raises: `<bad-arg>` for a null typed array, `<size-limit>` at the integer index limit, or `<alloc-fail>` when storage cannot grow.

Source: `lib/typed-array.x:158`

ArrayString.remove

```
String ArrayString.remove(ArrayString array, int index)
```

Removes and returns the element at normalized `index`. Negative indices count from the end; following elements shift left. Boxed `Array` returns `void` for a null array or out-of-range index.

Raises: `<bad-arg>` for a null typed array or out-of-range typed-array index, or `<size-limit>` when the length exceeds the integer index domain in either family.

Source: `lib/typed-array.x:158`

ArrayString.remslice

```
ArrayString ArrayString.remslice(ArrayString array, int start, int end)
```

Removes the region `array[start:end]` and returns a fresh packed array. Bounds normalize as slice bounds, so a negative `end` is a stop. Reversed bounds are swapped, and the removed values retain their order.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` on allocation failure.

Source: `lib/typed-array.x:158`

ArrayString.repr

```
String ArrayString.repr(ArrayString array)
```

Returns the readable packed-Array representation.

Source: `lib/typed-array.x:159`

ArrayString.reverse

```
ArrayString ArrayString.reverse(ArrayString array)
```

Reverses `array` in place and returns that same array. A null or one-element array is returned unchanged.

Source: `lib/typed-array.x:158`

ArrayString.setindex

```
String ArrayString.setindex(ArrayString array, int index, String value)
```

Stores and returns `value` at raw `index`, with C-pointer preconditions. Negative indices are not normalized; the caller supplies a valid index.

Source: `lib/typed-array.x:158`

ArrayString.setslice

```
ArrayString ArrayString.setslice(ArrayString array, int start, int end, ArrayString values)
```

Replaces the region `array[start:end]` and returns `array`. Bounds normalize as slice bounds, so a negative `end` is a stop. Reversed bounds are swapped; null `values` deletes the region. Passing `array` itself as `values` is supported and copies before mutation.

Raises: `<bad-arg>` for a null array, `<size-limit>` outside the integer index domain, or `<alloc-fail>` while copying self-aliased values or growing storage for the replacement. These failures leave `array` unchanged.

Source: `lib/typed-array.x:158`

ArrayString.shift

```
String ArrayString.shift(ArrayString array)
```

Removes and returns the first element, shifting the remainder left. This operation is O(n). Boxed `Array` returns `void` for a null or empty array.

Raises: `<bad-arg>` for a null or empty typed array, or `<size-limit>` when the length exceeds the integer index domain in either family.

Source: `lib/typed-array.x:158`

ArrayString.splice

```
ArrayString ArrayString.splice(ArrayString array, int index, int remove_count, ArrayString values)
```

Replaces `remove_count` values at normalized `index` and returns them. The result is a fresh packed array. A nonpositive count removes nothing; null `values` performs deletion, and self-splicing is supported.

Raises: <bad-arg> for a null array, <size-limit> outside the integer index domain, or <alloc-fail> on allocation failure.

Source: lib/typed-array.x:158

ArrayString.str

```
String ArrayString.str(ArrayString array)
```

Returns the packed-Array display String.

Source: lib/typed-array.x:159

ArrayString.take_last

```
String ArrayString.take_last(ArrayString array)
```

Removes and returns the last element. Boxed Array returns void for a null or empty array.

Raises: <bad-arg> for a null or empty typed array.

Source: lib/typed-array.x:158

ArrayString.try_get

```
int ArrayString.try_get(ArrayString array, int index, String &?out)
```

Reads normalized index into out and reports whether it exists. Negative indices count from the end. A null array, null out, or out-of-range index returns zero without writing; success returns one.

Raises: <size-limit> when the length exceeds the integer index domain.

Source: lib/typed-array.x:158

ArrayString.try_next

```
int ArrayString.try_next(ArrayString array, int &?cursor, String &?out)
```

Writes the next native element, advances cursor, and returns one. Initialize the caller-owned cursor to zero. A null argument, negative cursor, exhaustion, or an array outside the integer index domain returns zero without changing cursor or out. Structural mutation invalidates the cursor.

Source: lib/typed-array.x:158

ArrayString.try_take_last

```
int ArrayString.try_take_last(ArrayString array, String &?out)
```

Removes the last element into out and returns one when present. A null or empty array, or null out, returns zero without writing or changing the array.

Source: lib/typed-array.x:158

ArrayString.unshift

```
String ArrayString.unshift(ArrayString array, String value)
```

Inserts value at the front and returns it, shifting elements right. This operation is O(n). Boxed Array returns void for a null array.

Raises: <bad-arg> for a null typed array, <size-limit> at the integer index limit, or <alloc-fail> when storage cannot grow.

Source: lib/typed-array.x:158

ArrayString.updateindex

```
String ArrayString.updateindex(ArrayString array, int index, Symbol op, String rhs)
```

Concatenates `rhs` onto the String at raw `index` and returns the stored canonical String. The index has the unchecked C-pointer preconditions of bracket access. Raises: `<bad-op>` for any operation other than +. A failure leaves the element unchanged.

Source: `lib/typed-array.x:161`

ArrayString.var

```
Var ArrayString.var(ArrayString array)
```

Boxes `array` with its registered packed-array tag without copying. The boxed and typed values share the same mutable object identity.

Source: `lib/typed-array.x:162`

ArrayString.write_repr

```
Buffer ArrayString.write_repr(ArrayString array, Buffer out)
```

Appends the readable packed-Array representation to `out`.

Source: `lib/typed-array.x:159`

ArrayString.write_str

```
Buffer ArrayString.write_str(ArrayString array, Buffer out)
```

Appends the packed-Array display text to `out`.

Source: `lib/typed-array.x:159`

Var

Var.arraychar

```
ArrayChar Var.arraychar(Var value)
```

Returns the packed-array pointer carried by `value`. `value` must carry this array family's registered tag; the converter does not validate the tag before reading its pointer payload.

Source: `lib/typed-array.x:108`

Var.arraydbl

```
ArrayDbl Var.arraydbl(Var value)
```

Returns the packed-array pointer carried by `value`. `value` must carry this array family's registered tag; the converter does not validate the tag before reading its pointer payload.

Source: `lib/typed-array.x:153`

Var.arrayfloat

```
ArrayFloat Var.arrayfloat(Var value)
```

Returns the packed-array pointer carried by `value`. `value` must carry this array family's registered tag; the converter does not validate the tag before reading its pointer payload.

Source: `lib/typed-array.x:144`

Var.arrayint

```
ArrayInt Var.arrayint(Var value)
```

Returns the packed-array pointer carried by `value`. `value` must carry this array family's registered tag; the converter does not validate the tag before reading its pointer payload.

Source: `lib/typed-array.x:126`

Var.arraylong

```
ArrayLong Var.arraylong(Var value)
```

Returns the packed-array pointer carried by `value`. `value` must carry this array family's registered tag; the converter does not validate the tag before reading its pointer payload.

Source: `lib/typed-array.x:135`

Var.arrayshort

```
ArrayShort Var.arrayshort(Var value)
```

Returns the packed-array pointer carried by `value`. `value` must carry this array family's registered tag; the converter does not validate the tag before reading its pointer payload.

Source: `lib/typed-array.x:117`

Var.arraystring

```
ArrayString Var.arraystring(Var value)
```

Returns the packed-array pointer carried by `value`. `value` must carry this array family's registered tag; the converter does not validate the tag before reading its pointer payload.

Source: `lib/typed-array.x:162`

Public types

Type	Kind	Summary
ArrayChar	alias	A mutable, `Scope`-owned packed `Array` of native `char` elements.
ArrayDbl	alias	A mutable, `Scope`-owned packed `Array` of native `double` elements.
ArrayFloat	alias	A mutable, `Scope`-owned packed `Array` of native `float` elements.
ArrayInt	alias	A mutable, `Scope`-owned packed `Array` of native `int` elements.
ArrayLong	alias	A mutable, `Scope`-owned packed `Array` of native `long` elements.
ArrayShort	alias	A mutable, `Scope`-owned packed `Array` of native `short` elements.
ArrayString	alias	A mutable, `Scope`-owned packed `Array` of canonical `String` pointers.

ArrayChar

```
typedef Block ArrayChar
```

A mutable, `Scope`-owned packed `Array` of native `char` elements. Assignment shares its storage; `ArrayChar.copy` makes an independent copy.

Source: `lib/typed-array.x:32`

ArrayDbl

```
typedef Block ArrayDbl
```

A mutable, `Scope`-owned packed `Array` of native `double` elements. Assignment shares its storage; `ArrayDbl.copy` makes an independent copy.

Source: `lib/typed-array.x:52`

ArrayFloat

```
typedef Block ArrayFloat
```

A mutable, `Scope`-owned packed `Array` of native `float` elements. Assignment shares its storage; `ArrayFloat.copy` makes an independent copy.

Source: `lib/typed-array.x:48`

ArrayInt

`typedef Block ArrayInt`

A mutable, `Scope`-owned packed `Array` of native `int` elements. Assignment shares its storage; `ArrayInt.copy` makes an independent copy.

Source: `lib/typed-array.x:40`

ArrayLong

`typedef Block ArrayLong`

A mutable, `Scope`-owned packed `Array` of native `long` elements. Assignment shares its storage; `ArrayLong.copy` makes an independent copy.

Source: `lib/typed-array.x:44`

ArrayShort

`typedef Block ArrayShort`

A mutable, `Scope`-owned packed `Array` of native `short` elements. Assignment shares its storage; `ArrayShort.copy` makes an independent copy.

Source: `lib/typed-array.x:36`

ArrayString

`typedef Block ArrayString`

A mutable, `Scope`-owned packed `Array` of canonical `String` pointers. Assignment shares its storage; `ArrayString.copy` makes an independent shallow copy. Exporting an owned array recanonicalizes its Strings before replacing the backing storage while preserving the array's identity. Borrowed or outer-owned Strings must outlive the array.

Source: `lib/typed-array.x:59`

Design notes

Typed `Arrays` own contiguous native storage over `Block`. Static element operations use native pointers. `Var` descriptors and `Iter` box a value only when it crosses into `Var`.

Bracket indexing is raw. `array[index]` is an inline indexed access on the concrete element pointer. There is no bounds test and no negative-index normalization, so an index outside the elements has the same undefined behavior as the equivalent C pointer. Bound loops with `len`, and use `try_get` for a checked read.

Numeric `array[index] += rhs` applies the generated C operator to native values, not a `Var` round trip; `ArrayString` uses canonical String concatenation. Integer promotions and result conversions follow C. Integer division by zero, invalid shifts, and unsupported operators still raise before changing the element; the boxed path's `<conv-range>` is absent.

Tests and examples

`make verify` (`unittest/test-typed-array.x`).

lib/typed-list.x

Typed cons chains generated from typed methods.

Primary API

Function	Summary
`List.listchar`	Validates `xs` as `ListChar` and returns the identical canonical chain.

Function	Summary
`List.listdbl`	Validates `xs` as `ListDbl` and returns the identical canonical chain.
`List.listfloat`	Validates `xs` as `ListFloat` and returns the identical canonical chain.
`List.listint`	Validates `xs` as `ListInt` and returns the identical canonical chain.
`List.listshort`	Validates `xs` as `ListShort` and returns the identical canonical chain.
`List.liststring`	Validates `xs` as `ListString` and returns the identical canonical chain.
`List.listsymbol`	Validates `xs` as `ListSymbol` and returns the identical canonical chain.
`ListChar.car`	Returns the first `char` in `xs`, or `0` when `xs` is `nil`.
`ListChar.cons`	Returns the canonical `ListChar` formed by prepending `value` to `tail`.
`ListChar.index`	Returns the first zero-based index of `value`, or −1 when absent.
`ListChar.last`	Returns the final `char` in `xs`, or `0` when `xs` is `nil`, after an O(n) walk.
`ListDbl.car`	Returns the first `double` in `xs`, or `0.0` when `xs` is `nil`.
`ListDbl.cons`	Returns the canonical `ListDbl` formed by prepending `value` to `tail`.
`ListDbl.index`	Returns the first zero-based index of `value`, or −1 when absent.
`ListDbl.last`	Returns the final `double` in `xs`, or `0.0` when `xs` is `nil`, after an O(n) walk.
`ListFloat.car`	Returns the first `float` in `xs`, or `0.0f` when `xs` is `nil`.
`ListFloat.cons`	Returns the canonical `ListFloat` formed by prepending `value` to `tail`.
`ListFloat.index`	Returns the first zero-based index of `value`, or −1 when absent.
`ListFloat.last`	Returns the final `float` in `xs`, or `0.0f` when `xs` is `nil`, after an O(n) walk.
`ListInt.car`	Returns the first `int` in `xs`, or `0` when `xs` is `nil`.
`ListInt.cons`	Returns the canonical `ListInt` formed by prepending `value` to `tail`.
`ListInt.index`	Returns the first zero-based index of `value`, or −1 when absent.
`ListInt.last`	Returns the final `int` in `xs`, or `0` when `xs` is `nil`, after an O(n) walk.
`ListShort.car`	Returns the first `short` in `xs`, or `0` when `xs` is `nil`.
`ListShort.cons`	Returns the canonical `ListShort` formed by prepending `value` to `tail`.
`ListShort.index`	Returns the first zero-based index of `value`, or −1 when absent.
`ListShort.last`	Returns the final `short` in `xs`, or `0` when `xs` is `nil`, after an O(n) walk.
`ListString.car`	Returns the first `String` in `xs`, or `NULL` when `xs` is `nil`.
`ListString.cons`	Returns the canonical `ListString` formed by prepending `value` to `tail`.
`ListString.index`	Returns the first zero-based index of `value`, or −1 when absent.
`ListString.last`	Returns the final `String` in `xs`, or `NULL` when `xs` is `nil`, after an O(n) walk.
`ListSymbol.car`	Returns the first `Symbol` in `xs`, or `0` when `xs` is `nil`.
`ListSymbol.cons`	Returns the canonical `ListSymbol` formed by prepending `value` to `tail`.
`ListSymbol.index`	Returns the first zero-based index of `value`, or −1 when absent.
`ListSymbol.last`	Returns the final `Symbol` in `xs`, or `0` when `xs` is `nil`, after an O(n) walk.
`Var.listchar`	Extracts and validates `value` as `ListChar` without copying its cells.
`Var.listdbl`	Extracts and validates `value` as `ListDbl` without copying its cells.
`Var.listfloat`	Extracts and validates `value` as `ListFloat` without copying its cells.
`Var.listint`	Extracts and validates `value` as `ListInt` without copying its cells.
`Var.listshort`	Extracts and validates `value` as `ListShort` without copying its cells.
`Var.liststring`	Extracts and validates `value` as `ListString` without copying its cells.
`Var.listsymbol`	Extracts and validates `value` as `ListSymbol` without copying its cells.

`List`

List.listchar

```
ListChar List.listchar(List xs)
```

Validates `xs` as `ListChar` and returns the identical canonical chain. Nil is valid. Every nonempty cell must hold a `<i8>` element; a foreign tag raises `<no-convert>` with its zero-based index. Validation is O(n), does not copy or mutate cells, and preserves their existing `List`-pool lifetime.

Raises: `<no-convert>` for the first foreign element.

Source: `lib/typed-list.x:199`

List.listdbl

```
ListDbl List.listdbl(List xs)
```

Validates `xs` as `ListDbl` and returns the identical canonical chain. Nil is valid. Every nonempty cell must hold a `<f64>` element; a foreign tag raises `<no-convert>` with its zero-based index. Validation is O(n), does not copy or mutate cells, and preserves their existing `List`-pool lifetime.

Raises: `<no-convert>` for the first foreign element.

Source: `lib/typed-list.x:215`

List.listfloat

```
ListFloat List.listfloat(List xs)
```

Validates `xs` as `ListFloat` and returns the identical canonical chain. Nil is valid. Every nonempty cell must hold a `<f32>` element; a foreign tag raises `<no-convert>` with its zero-based index. Validation is O(n), does not copy or mutate cells, and preserves their existing `List`-pool lifetime.

Raises: `<no-convert>` for the first foreign element.

Source: `lib/typed-list.x:211`

List.listint

```
ListInt List.listint(List xs)
```

Validates `xs` as `ListInt` and returns the identical canonical chain. Nil is valid. Every nonempty cell must hold a `<i32>` element; a foreign tag raises `<no-convert>` with its zero-based index. Validation is O(n), does not copy or mutate cells, and preserves their existing `List`-pool lifetime.

Raises: `<no-convert>` for the first foreign element.

Source: `lib/typed-list.x:207`

List.listshort

```
ListShort List.listshort(List xs)
```

Validates `xs` as `ListShort` and returns the identical canonical chain. Nil is valid. Every nonempty cell must hold a `<i16>` element; a foreign tag raises `<no-convert>` with its zero-based index. Validation is O(n), does not copy or mutate cells, and preserves their existing `List`-pool lifetime.

Raises: `<no-convert>` for the first foreign element.

Source: `lib/typed-list.x:203`

List.liststring

```
ListString List.liststring(List xs)
```

Validates `xs` as `ListString` and returns the identical canonical chain. Nil is valid. Every nonempty cell must hold a `<string>` element; a foreign tag raises `<no-convert>` with its zero-based index. Validation is O(n), does not copy or mutate cells, and preserves their existing `List`-pool lifetime.

Raises: `<no-convert>` for the first foreign element.

Source: `lib/typed-list.x:219`

List.listsymbol

```
ListSymbol List.listsymbol(List xs)
```

Validates `xs` as `ListSymbol` and returns the identical canonical chain. Nil is valid. Every nonempty cell must hold a `<symbol>` element; a foreign tag raises `<no-convert>` with its zero-based index. Validation is O(n), does not copy or mutate cells, and preserves their existing `List`-pool lifetime.

Raises: `<no-convert>` for the first foreign element.

Source: `lib/typed-list.x:223`

`ListChar`

ListChar.car

```
char ListChar.car(ListChar xs)
```

Returns the first `char` in `xs`, or `0` when `xs` is `nil`. A nonempty `xs` must retain the `ListChar` element-tag invariant.

Source: `lib/typed-list.x:199`

ListChar.cons

```
ListChar ListChar.cons(char value, ListChar tail)
```

Returns the canonical `ListChar` formed by prepending `value` to `tail`. The immutable tail is shared. The result follows the lifetime of its owning canonical `List` pool, which may be an ancestor of the current pool when an existing cell is reused.

Raises: `<alloc-fail>` or `<size-limit>` while installing a new canonical cell.

Source: `lib/typed-list.x:199`

ListChar.index

```
int ListChar.index(ListChar xs, char value)
```

Returns the first zero-based index of `value`, or −1 when absent.

Source: `lib/typed-list.x:199`

ListChar.last

```
char ListChar.last(ListChar xs)
```

Returns the final `char` in `xs`, or `0` when `xs` is `nil`, after an O(n) walk. A nonempty `xs` must retain the `ListChar` element-tag invariant.

Source: `lib/typed-list.x:199`

`ListDbl`

ListDbl.car

```
double ListDbl.car(ListDbl xs)
```

Returns the first `double` in `xs`, or `0.0` when `xs` is `nil`. A nonempty `xs` must retain the `ListDbl` element-tag invariant.

Source: `lib/typed-list.x:215`

ListDbl.cons

```
ListDbl ListDbl.cons(double value, ListDbl tail)
```

Returns the canonical `ListDbl` formed by prepending `value` to `tail`. The immutable tail is shared. The result follows the lifetime of its owning canonical `List` pool, which may be an ancestor of the current pool when an existing cell is reused.

Raises: `<alloc-fail>` or `<size-limit>` while installing a new canonical cell.

Source: `lib/typed-list.x:215`

ListDbl.index

```
int ListDbl.index(ListDbl xs, double value)
```

Returns the first zero-based index of `value`, or −1 when absent.

Source: `lib/typed-list.x:215`

ListDbl.last

```
double ListDbl.last(ListDbl xs)
```

Returns the final `double` in `xs`, or `0.0` when `xs` is `nil`, after an O(n) walk. A nonempty `xs` must retain the `ListDbl` element-tag invariant.

Source: `lib/typed-list.x:215`

ListFloat

ListFloat.car

```
float ListFloat.car(ListFloat xs)
```

Returns the first `float` in `xs`, or `0.0f` when `xs` is `nil`. A nonempty `xs` must retain the `ListFloat` element-tag invariant.

Source: `lib/typed-list.x:211`

ListFloat.cons

```
ListFloat ListFloat.cons(float value, ListFloat tail)
```

Returns the canonical `ListFloat` formed by prepending `value` to `tail`. The immutable tail is shared. The result follows the lifetime of its owning canonical `List` pool, which may be an ancestor of the current pool when an existing cell is reused.

Raises: `<alloc-fail>` or `<size-limit>` while installing a new canonical cell.

Source: `lib/typed-list.x:211`

ListFloat.index

```
int ListFloat.index(ListFloat xs, float value)
```

Returns the first zero-based index of `value`, or −1 when absent.

Source: `lib/typed-list.x:211`

ListFloat.last

```
float ListFloat.last(ListFloat xs)
```

Returns the final `float` in `xs`, or `0.0f` when `xs` is `nil`, after an O(n) walk. A nonempty `xs` must retain the `ListFloat` element-tag invariant.

Source: `lib/typed-list.x:211`

ListInt

ListInt.car

```
int ListInt.car(ListInt xs)
```

Returns the first `int` in `xs`, or `0` when `xs` is `nil`. A nonempty `xs` must retain the `ListInt` element-tag invariant.

Source: `lib/typed-list.x:207`

ListInt.cons

```
ListInt ListInt.cons(int value, ListInt tail)
```

Returns the canonical `ListInt` formed by prepending `value` to `tail`. The immutable tail is shared. The result follows the lifetime of its owning canonical `List` pool, which may be an ancestor of the current pool when an existing cell is reused.

Raises: `<alloc-fail>` or `<size-limit>` while installing a new canonical cell.

Source: `lib/typed-list.x:207`

ListInt.index

```
int ListInt.index(ListInt xs, int value)
```

Returns the first zero-based index of `value`, or −1 when absent.

Source: `lib/typed-list.x:207`

ListInt.last

```
int ListInt.last(ListInt xs)
```

Returns the final `int` in `xs`, or `0` when `xs` is `nil`, after an O(n) walk. A nonempty `xs` must retain the `ListInt` element-tag invariant.

Source: `lib/typed-list.x:207`

ListShort

ListShort.car

```
short ListShort.car(ListShort xs)
```

Returns the first `short` in `xs`, or `0` when `xs` is `nil`. A nonempty `xs` must retain the `ListShort` element-tag invariant.

Source: `lib/typed-list.x:203`

ListShort.cons

```
ListShort ListShort.cons(short value, ListShort tail)
```

Returns the canonical `ListShort` formed by prepending `value` to `tail`. The immutable tail is shared. The result follows the lifetime of its owning canonical `List` pool, which may be an ancestor of the current pool when an existing cell is reused.

Raises: `<alloc-fail>` or `<size-limit>` while installing a new canonical cell.

Source: `lib/typed-list.x:203`

ListShort.index

```
int ListShort.index(ListShort xs, short value)
```

Returns the first zero-based index of `value`, or −1 when absent.

Source: `lib/typed-list.x:203`

ListShort.last

```
short ListShort.last(ListShort xs)
```

Returns the final `short` in `xs`, or `0` when `xs` is `nil`, after an O(n) walk. A nonempty `xs` must retain the `ListShort` element-tag invariant.

Source: `lib/typed-list.x:203`

ListString

ListString.car

```
String ListString.car(ListString xs)
```

Returns the first `String` in `xs`, or `NULL` when `xs` is `nil`. A nonempty `xs` must retain the `ListString` element-tag invariant.

Source: `lib/typed-list.x:219`

ListString.cons

```
ListString ListString.cons(String value, ListString tail)
```

Returns the canonical `ListString` formed by prepending `value` to `tail`. The immutable tail is shared. The result follows the lifetime of its owning canonical `List` pool, which may be an ancestor of the current pool when an existing cell is reused.

Raises: `<alloc-fail>` or `<size-limit>` while installing a new canonical cell.

Source: `lib/typed-list.x:219`

ListString.index

```
int ListString.index(ListString xs, String value)
```

Returns the first zero-based index of `value`, or −1 when absent.

Source: `lib/typed-list.x:219`

ListString.last

```
String ListString.last(ListString xs)
```

Returns the final `String` in `xs`, or `NULL` when `xs` is `nil`, after an O(n) walk. A nonempty `xs` must retain the `ListString` element-tag invariant.

Source: `lib/typed-list.x:219`

ListSymbol

ListSymbol.car

```
Symbol ListSymbol.car(ListSymbol xs)
```

Returns the first `Symbol` in `xs`, or `0` when `xs` is `nil`. A nonempty `xs` must retain the `ListSymbol` element-tag invariant.

Source: `lib/typed-list.x:223`

ListSymbol.cons

```
ListSymbol ListSymbol.cons(Symbol value, ListSymbol tail)
```

Returns the canonical `ListSymbol` formed by prepending `value` to `tail`. The immutable tail is shared. The result follows the lifetime of its owning canonical `List` pool, which may be an ancestor of the current pool when an existing cell is reused.

Raises: `<alloc-fail>` or `<size-limit>` while installing a new canonical cell.

Source: `lib/typed-list.x:223`

ListSymbol.index

```
int ListSymbol.index(ListSymbol xs, Symbol value)
```

Returns the first zero-based index of `value`, or −1 when absent.

Source: `lib/typed-list.x:223`

ListSymbol.last

```
Symbol ListSymbol.last(ListSymbol xs)
```

Returns the final `Symbol` in `xs`, or `0` when `xs` is `nil`, after an O(n) walk. A nonempty `xs` must retain the `ListSymbol` element-tag invariant.

Source: `lib/typed-list.x:223`

Var

Var.listchar

```
ListChar Var.listchar(Var value)
```

Extracts and validates `value` as `ListChar` without copying its cells. A `Var` without the `<list>` tag becomes `nil`. A `List` payload follows the same element validation, identity, and lifetime rules as `List.listchar`.

Raises: `<no-convert>` for the first foreign element.

Source: `lib/typed-list.x:199`

Var.listdbl

```
ListDbl Var.listdbl(Var value)
```

Extracts and validates `value` as `ListDbl` without copying its cells. A `Var` without the `<list>` tag becomes `nil`. A `List` payload follows the same element validation, identity, and lifetime rules as `List.listdbl`.

Raises: `<no-convert>` for the first foreign element.

Source: `lib/typed-list.x:215`

Var.listfloat

```
ListFloat Var.listfloat(Var value)
```

Extracts and validates `value` as `ListFloat` without copying its cells. A `Var` without the `<list>` tag becomes `nil`. A `List` payload follows the same element validation, identity, and lifetime rules as `List.listfloat`.

Raises: `<no-convert>` for the first foreign element.

Source: `lib/typed-list.x:211`

Var.listint

```
ListInt Var.listint(Var value)
```

Extracts and validates `value` as `ListInt` without copying its cells. A `Var` without the `<list>` tag becomes `nil`. A `List` payload follows the same element validation, identity, and lifetime rules as `List.listint`.

Raises: `<no-convert>` for the first foreign element.

Source: `lib/typed-list.x:207`

Var.listshort

```
ListShort Var.listshort(Var value)
```

Extracts and validates `value` as `ListShort` without copying its cells. A `Var` without the `<list>` tag becomes `nil`. A `List` payload follows the same element validation, identity, and lifetime rules as `List.listshort`.

Raises: `<no-convert>` for the first foreign element.

Source: `lib/typed-list.x:203`

Var.liststring

```
ListString Var.liststring(Var value)
```

Extracts and validates `value` as `ListString` without copying its cells. A `Var` without the `<list>` tag becomes `nil`. A `List` payload follows the same element validation, identity, and lifetime rules as `List.liststring`.

Raises: `<no-convert>` for the first foreign element.

Source: `lib/typed-list.x:219`

Var.listsymbol

```
ListSymbol Var.listsymbol(Var value)
```

Extracts and validates `value` as `ListSymbol` without copying its cells. A `Var` without the `<list>` tag becomes `nil`. A `List` payload follows the same element validation, identity, and lifetime rules as `List.listsymbol`.

Raises: `<no-convert>` for the first foreign element.

Source: `lib/typed-list.x:223`

Public types

Type	Kind	Summary
`ListChar`	alias	Typed view of canonical `List` cells whose cars are `<i8>` char values.
`ListDbl`	alias	Typed view of canonical `List` cells whose cars are `<f64>` double values.
`ListFloat`	alias	Typed view of canonical `List` cells whose cars are `<f32>` float values.
`ListInt`	alias	Typed view of canonical `List` cells whose cars are `<i32>` int values.
`ListShort`	alias	Typed view of canonical `List` cells whose cars are `<i16>` short values.
`ListString`	alias	Typed view of canonical `List` cells whose cars are `<string>` `String`s.
`ListSymbol`	alias	Typed view of canonical `List` cells whose cars are `<symbol>` `Symbol`s.

ListChar

```
typedef List ListChar
```

Typed view of canonical `List` cells whose cars are `<i8>` char values.

Source: `lib/typed-list.x:130`

ListDbl

```
typedef List ListDbl
```

Typed view of canonical `List` cells whose cars are `<f64>` double values.

Source: `lib/typed-list.x:141`

ListFloat

```
typedef List ListFloat
```

Typed view of canonical `List` cells whose cars are `<f32>` float values.

Source: `lib/typed-list.x:138`

ListInt

`typedef List ListInt`

Typed view of canonical `List` cells whose cars are `<i32>` int values.

Source: `lib/typed-list.x:135`

ListShort

`typedef List ListShort`

Typed view of canonical `List` cells whose cars are `<i16>` short values.

Source: `lib/typed-list.x:133`

ListString

`typedef List ListString`

Typed view of canonical `List` cells whose cars are `<string>` `Strings`.

Source: `lib/typed-list.x:144`

ListSymbol

`typedef List ListSymbol`

Typed view of canonical `List` cells whose cars are `<symbol>` `Symbols`.

Source: `lib/typed-list.x:147`

Design notes

A typed list is a `List` whose car always carries one known `Var` tag. It shares `List`'s canonical pool, so a typed chain and the plain literal that spells it are the same cells, and a typed `cons` finds a cell an untyped `cons` already built.

A typed chain costs what an untyped one costs. `cons` searches the pool and, on a miss, allocates and inserts a cell. `ArrayInt.push` copies an element into contiguous storage. Use `typed-array.x` for indexing or bulk numeric storage, and typed lists for shared tails and canonical identity.

There is no long family. `Var.box_long` allocates a `Scope`-owned box, so a cell outliving that scope would hold a dangling car, and `List.equal` compares car bits. Two boxes of one number differ, so every cons would miss and the interning table would grow without bound.

Tests and examples

`make verify` (`unittest/test-typed-list.x`).

lib/typed-map.x

Typed `Maps` generated from shared storage.

Primary API

Function	Summary
`Map.mapintint`	Packs `entries` into a fresh typed map in the current `Scope`.
`Map.maplongdouble`	Packs `entries` into a fresh typed map in the current `Scope`.
`Map.mapstringint`	Packs `entries` into a fresh typed map in the current `Scope`.
`Map.mapstringstring`	Packs `entries` into a fresh typed map in the current `Scope`.
`MapIntInt.cleanup`	Releases the record and its backing Blocks, borrowing stored values.
`MapIntInt.compare`	Compares typed Maps by size and then sorted native key/value contents.
`MapIntInt.contains`	Returns one when `key` is present, or zero for a null map.
`MapIntInt.copy`	Returns a fresh shallow copy in the current `Scope`.
`MapIntInt.del`	Removes `key` and returns its value.

Function	Summary
MapIntInt.enumerate	Initializes dest to yield canonical (key value) Lists in bucket order.
MapIntInt.equal	Reports key/value equality independent of bucket and insertion order.
MapIntInt.get	Returns the value for key.
MapIntInt.getdefault	Returns the value for key, or defval without inserting it.
MapIntInt.getindex	Returns the value selected by bracket indexing.
MapIntInt.iter	Initializes dest as a single-pass iterator over boxed values.
MapIntInt.keys	Initializes dest as a single-pass iterator over boxed keys.
MapIntInt.len	Returns the number of entries, or zero for a null map.
MapIntInt.map	Returns a fresh ordinary Map containing boxed keys and values.
MapIntInt.merge	Copies every entry of other into map and returns the destination.
MapIntInt.new	Returns a fresh empty map owned by the current Scope.
MapIntInt.new_capacity	Returns a fresh empty map with exactly capacity buckets.
MapIntInt.postfixindex	Applies postfix ++ or -- and returns the previous value.
MapIntInt.repr	Returns the readable typed-Map representation.
MapIntInt.set	Copies key and val into the table, replacing an existing value.
MapIntInt.setdefault	Returns the value for key, inserting defval when absent.
MapIntInt.setindex	Stores val under key and returns val.
MapIntInt.str	Returns the typed-Map display String.
MapIntInt.truth	Returns one when map contains an entry.
MapIntInt.try_del	Removes key, writes its value to out, and returns one when present.
MapIntInt.try_get	Writes the value for key to out and returns one when present.
MapIntInt.try_next	Writes the next bucket's key and value, advances cursor, and returns one.
MapIntInt.updateindex	Applies op to the value for key and returns the stored result.
MapIntInt.var	Boxes map with its registered typed-map tag without copying it.
MapIntInt.write_repr	Appends the readable typed-Map representation in bucket order.
MapIntInt.write_str	Appends the typed-Map display text in bucket order.
MapLongDouble.cleanup	Releases the record and its backing Blocks, borrowing stored values.
MapLongDouble.compare	Compares typed Maps by size and then sorted native key/value contents.
MapLongDouble.contains	Returns one when key is present, or zero for a null map.
MapLongDouble.copy	Returns a fresh shallow copy in the current Scope.
MapLongDouble.del	Removes key and returns its value.
MapLongDouble.enumerate	Initializes dest to yield canonical (key value) Lists in bucket order.
MapLongDouble.equal	Reports key/value equality independent of bucket and insertion order.
MapLongDouble.get	Returns the value for key.
MapLongDouble.getdefault	Returns the value for key, or defval without inserting it.
MapLongDouble.getindex	Returns the value selected by bracket indexing.
MapLongDouble.iter	Initializes dest as a single-pass iterator over boxed values.
MapLongDouble.keys	Initializes dest as a single-pass iterator over boxed keys.
MapLongDouble.len	Returns the number of entries, or zero for a null map.
MapLongDouble.map	Returns a fresh ordinary Map containing boxed keys and values.
MapLongDouble.merge	Copies every entry of other into map and returns the destination.
MapLongDouble.new	Returns a fresh empty map owned by the current Scope.
MapLongDouble.new_capacity	Returns a fresh empty map with exactly capacity buckets.

Function	Summary
MapLongDouble.postfixindex	Applies postfix ++ or -- and returns the previous value.
MapLongDouble.repr	Returns the readable typed-Map representation.
MapLongDouble.set	Copies key and val into the table, replacing an existing value.
MapLongDouble.setdefault	Returns the value for key, inserting defval when absent.
MapLongDouble.setindex	Stores val under key and returns val.
MapLongDouble.str	Returns the typed-Map display String.
MapLongDouble.truth	Returns one when map contains an entry.
MapLongDouble.try_del	Removes key, writes its value to out, and returns one when present.
MapLongDouble.try_get	Writes the value for key to out and returns one when present.
MapLongDouble.try_next	Writes the next bucket's key and value, advances cursor, and returns one.
MapLongDouble.updateindex	Applies op to the value for key and returns the stored result.
MapLongDouble.var	Boxes map with its registered typed-map tag without copying it.
MapLongDouble.write_repr	Appends the readable typed-Map representation in bucket order.
MapLongDouble.write_str	Appends the typed-Map display text in bucket order.
MapStringInt.cleanup	Releases the record and its backing Blocks, borrowing stored values.
MapStringInt.compare	Compares typed Maps by size and then sorted native key/value contents.
MapStringInt.contains	Returns one when key is present, or zero for a null map.
MapStringInt.copy	Returns a fresh shallow copy in the current Scope.
MapStringInt.del	Removes key and returns its value.
MapStringInt.enumerate	Initializes dest to yield canonical (key value) Lists in bucket order.
MapStringInt.equal	Reports key/value equality independent of bucket and insertion order.
MapStringInt.get	Returns the value for key.
MapStringInt.getdefault	Returns the value for key, or defval without inserting it.
MapStringInt.getindex	Returns the value selected by bracket indexing.
MapStringInt.iter	Initializes dest as a single-pass iterator over boxed values.
MapStringInt.keys	Initializes dest as a single-pass iterator over boxed keys.
MapStringInt.len	Returns the number of entries, or zero for a null map.
MapStringInt.map	Returns a fresh ordinary Map containing boxed keys and values.
MapStringInt.merge	Copies every entry of other into map and returns the destination.
MapStringInt.new	Returns a fresh empty map owned by the current Scope.
MapStringInt.new_capacity	Returns a fresh empty map with exactly capacity buckets.
MapStringInt.postfixindex	Applies postfix ++ or -- and returns the previous value.
MapStringInt.repr	Returns the readable typed-Map representation.
MapStringInt.set	Copies key and val into the table, replacing an existing value.
MapStringInt.setdefault	Returns the value for key, inserting defval when absent.
MapStringInt.setindex	Stores val under key and returns val.
MapStringInt.str	Returns the typed-Map display String.
MapStringInt.truth	Returns one when map contains an entry.
MapStringInt.try_del	Removes key, writes its value to out, and returns one when present.
MapStringInt.try_get	Writes the value for key to out and returns one when present.
MapStringInt.try_next	Writes the next bucket's key and value, advances cursor, and returns one.
MapStringInt.updateindex	Applies op to the value for key and returns the stored result.
MapStringInt.var	Boxes map with its registered typed-map tag without copying it.

Function	Summary
MapStringInt.write_repr	Appends the readable typed-Map representation in bucket order.
MapStringInt.write_str	Appends the typed-Map display text in bucket order.
MapStringString.cleanup	Releases the record and its backing Blocks, borrowing stored values.
MapStringString.compare	Compares typed Maps by size and then sorted native key/value contents.
MapStringString.contains	Returns one when key is present, or zero for a null map.
MapStringString.copy	Returns a fresh shallow copy in the current Scope.
MapStringString.del	Removes key and returns its value.
MapStringString.enumerate	Initializes dest to yield canonical (key value) Lists in bucket order.
MapStringString.equal	Reports key/value equality independent of bucket and insertion order.
MapStringString.get	Returns the value for key.
MapStringString.getdefault	Returns the value for key, or defval without inserting it.
MapStringString.getindex	Returns the value selected by bracket indexing.
MapStringString.iter	Initializes dest as a single-pass iterator over boxed values.
MapStringString.keys	Initializes dest as a single-pass iterator over boxed keys.
MapStringString.len	Returns the number of entries, or zero for a null map.
MapStringString.map	Returns a fresh ordinary Map containing boxed keys and values.
MapStringString.merge	Copies every entry of other into map and returns the destination.
MapStringString.new	Returns a fresh empty map owned by the current Scope.
MapStringString.new_capacity	Returns a fresh empty map with exactly capacity buckets.
MapStringString.postfixindex	Applies postfix ++ or -- and returns the previous value.
MapStringString.repr	Returns the readable typed-Map representation.
MapStringString.set	Copies key and val into the table, replacing an existing value.
MapStringString.setdefault	Returns the value for key, inserting defval when absent.
MapStringString.setindex	Stores val under key and returns val.
MapStringString.str	Returns the typed-Map display String.
MapStringString.truth	Returns one when map contains an entry.
MapStringString.try_del	Removes key, writes its value to out, and returns one when present.
MapStringString.try_get	Writes the value for key to out and returns one when present.
MapStringString.try_next	Writes the next bucket's key and value, advances cursor, and returns one.
MapStringString.updateindex	Applies op to the value for key and returns the stored result.
MapStringString.var	Boxes map with its registered typed-map tag without copying it.
MapStringString.write_repr	Appends the readable typed-Map representation in bucket order.
MapStringString.write_str	Appends the typed-Map display text in bucket order.
Var.mapintint	Returns the typed-map pointer carried by value.
Var.maplongdouble	Returns the typed-map pointer carried by value.
Var.mapstringint	Returns the typed-map pointer carried by value.
Var.mapstringstring	Returns the typed-map pointer carried by value.

Map

Map.mapintint

```
MapIntInt Map.mapintint(Map entries)
```

Packs `entries` into a fresh typed map in the current `Scope`. A null input returns NULL. Each key and value goes through its `Var` conversion, and later entries in bucket order replace earlier equal keys. The input `Map` is unchanged.

Raises: any conversion cause, or `<alloc-fail>`, `<size-limit>`, or `<invariant>` while inserting.

Source: `lib/typed-map.x:232`

Map.maplongdouble

```
MapLongDouble Map.maplongdouble(Map entries)
```

Packs `entries` into a fresh typed map in the current `Scope`. A null input returns NULL. Each key and value goes through its `Var` conversion, and later entries in bucket order replace earlier equal keys. The input `Map` is unchanged.

Raises: any conversion cause, or `<alloc-fail>`, `<size-limit>`, or `<invariant>` while inserting.

Source: `lib/typed-map.x:252`

Map.mapstringint

```
MapStringInt Map.mapstringint(Map entries)
```

Packs `entries` into a fresh typed map in the current `Scope`. A null input returns NULL. Each key and value goes through its `Var` conversion, and later entries in bucket order replace earlier equal keys. The input `Map` is unchanged.

Raises: any conversion cause, or `<alloc-fail>`, `<size-limit>`, or `<invariant>` while inserting.

Source: `lib/typed-map.x:311`

Map.mapstringstring

```
MapStringString Map.mapstringstring(Map entries)
```

Packs `entries` into a fresh typed map in the current `Scope`. A null input returns NULL. Each key and value goes through its `Var` conversion, and later entries in bucket order replace earlier equal keys. The input `Map` is unchanged.

Raises: any conversion cause, or `<alloc-fail>`, `<size-limit>`, or `<invariant>` while inserting.

Source: `lib/typed-map.x:273`

MapIntInt

MapIntInt.cleanup

```
void MapIntInt.cleanup(MapIntInt map)
```

Releases the record and its backing Blocks, borrowing stored values.

Source: `lib/typed-map.x:241`

MapIntInt.compare

```
int MapIntInt.compare(MapIntInt a, MapIntInt b)
```

Compares typed Maps by size and then sorted native key/value contents. Identical handles compare equal and NULL sorts first. Neither Map is mutated. Raises: `<alloc-fail>` while creating temporary storage, or any cause from key or value comparison.

Source: `lib/typed-map.x:240`

MapIntInt.contains

```
int MapIntInt.contains(MapIntInt map, int key)
```

Returns one when `key` is present, or zero for a null map.

Source: `lib/typed-map.x:232`

MapIntInt.copy

```
MapIntInt MapIntInt.copy(MapIntInt map)
```

Returns a fresh shallow copy in the current `Scope`. The result has independent table storage but copies native key and value fields. A null input produces a fresh empty map.

Raises: an allocation, size, or invariant cause while copying.

Source: `lib/typed-map.x:232`

MapIntInt.del

```
int MapIntInt.del(MapIntInt map, int key)
```

Removes `key` and returns its value. Successful removal invalidates live cursors and iterators. Prefer `try_del` when absence is an ordinary outcome. Boxed families return `void` for absence; native families raise `<bad-arg>`.

Source: `lib/typed-map.x:232`

MapIntInt.enumerate

```
Iter MapIntInt.enumerate(MapIntInt map, Iter dest)
```

Initializes `dest` to yield canonical `(key value)` `Lists` in bucket order. The caller owns `dest`, and the iterator borrows `map`; both must remain live during traversal. A null `dest` returns NULL, and a null map is exhausted. Each step boxes both fields and interns two `List` cells. Each cell belongs to the pool owning its canonical match, which may be an ancestor pool, or to the current pool when newly allocated. Structural mutation invalidates the iterator. Each MapLongDouble pull allocates its long key as a `Var` in the current `Scope`; the yielded `List` and anything retaining it must not outlive that `Scope`.

Raises: `<alloc-fail>` or `<size-limit>` while interning the `List`, and for MapLongDouble, `<alloc-fail>` or `<bad-enc>` while boxing its key. Shared causes do not return from the pull.

Source: `lib/typed-map.x:241`

MapIntInt.equal

```
int MapIntInt.equal(MapIntInt a, MapIntInt b)
```

Reports key/value equality independent of bucket and insertion order. Two null maps compare equal; a null and a nonnull map compare unequal.

Source: `lib/typed-map.x:232`

MapIntInt.get

```
int MapIntInt.get(MapIntInt map, int key)
```

Returns the value for `key`. Prefer `try_get` when absence is an ordinary outcome. Boxed families return `void` for absence; native families raise `<bad-arg>`.

Source: `lib/typed-map.x:232`

MapIntInt.getdefault

```
int MapIntInt.getdefault(MapIntInt map, int key, int defval)
```

Returns the value for `key`, or `defval` without inserting it. A null map is treated as an empty one.

Source: `lib/typed-map.x:232`

MapIntInt.getindex

```
int MapIntInt.getindex(MapIntInt map, int key)
```

Returns the value selected by bracket indexing. This is the bracket-facing form of `MapIntInt.get`, with the same failure behavior. Prefer bracket indexing in ordinary code.

Source: `lib/typed-map.x:232`

MapIntInt.iter

```
Iter MapIntInt.iter(MapIntInt map, Iter dest)
```

Initializes `dest` as a single-pass iterator over boxed values. The caller owns `dest`, and the iterator borrows `map`; both must remain live during traversal. A null `dest` returns NULL. Values follow bucket order, a null map is exhausted, and structural mutation invalidates the iterator.

Source: `lib/typed-map.x:241`

MapIntInt.keys

```
Iter MapIntInt.keys(MapIntInt map, Iter dest)
```

Initializes `dest` as a single-pass iterator over boxed keys. The caller owns `dest`, and the iterator borrows `map`; both must remain live during traversal. A null `dest` returns NULL. Keys follow bucket order, a null map is exhausted, and structural mutation invalidates the iterator. MapIntInt and both String-keyed families box keys without allocation. Each MapLongDouble pull allocates its long key as a `Var` in the current `Scope`; the yielded `Var` and anything retaining it must not outlive that `Scope`.

Raises: MapLongDouble iteration may raise `<alloc-fail>` or `<bad-enc>` while boxing a key.

Source: `lib/typed-map.x:241`

MapIntInt.len

```
unsigned MapIntInt.len(MapIntInt map)
```

Returns the number of entries, or zero for a null map.

Source: `lib/typed-map.x:232`

MapIntInt.map

```
Map MapIntInt.map(MapIntInt map)
```

Returns a fresh ordinary Map containing boxed keys and values. A null input returns NULL. The typed Map and its storage are unchanged.

Raises: any cause from boxing or inserting an entry.

Source: `lib/typed-map.x:240`

MapIntInt.merge

```
MapIntInt MapIntInt.merge(MapIntInt map, MapIntInt other)
```

Copies every entry of `other` into `map` and returns the destination. `other` wins conflicts. A null `other` is a no-op; a null destination is replaced by a fresh map. When a supplied destination fails partway, entries already copied remain; a newly created destination is discarded.

Raises: an allocation, size, or invariant cause while inserting.

Source: `lib/typed-map.x:232`

MapIntInt.new

```
MapIntInt MapIntInt.new(void)
```

Returns a fresh empty map owned by the current `Scope`. The initial table has two buckets and grows automatically.

Raises: `<alloc-fail>` or `<size-limit>` when storage cannot be created.

Source: `lib/typed-map.x:232`

MapIntInt.new_capacity

```
MapIntInt MapIntInt.new_capacity(unsigned capacity)
```

Returns a fresh empty map with exactly `capacity` buckets. The result and its backing arrays belong to the current `Scope`. `capacity` must be a power of two of at least two.

Raises: `<bad-arg>` for another capacity, or `<alloc-fail>` / `<size-limit>` when storage cannot be created.

Source: `lib/typed-map.x:232`

MapIntInt.postfixindex

```
int MapIntInt.postfixindex(MapIntInt map, int key, Symbol op)
```

Applies postfix `++` or `--` and returns the previous value. The key must already exist, and only generated numeric families support these operations.

Raises: `<bad-arg>` for a null map or absent key, or `<bad-op>` for an unsupported operation. These failures leave the value unchanged.

Source: `lib/typed-map.x:232`

MapIntInt.repr

```
String MapIntInt.repr(MapIntInt map)
```

Returns the readable typed-Map representation.

Source: `lib/typed-map.x:240`

MapIntInt.set

```
void MapIntInt.set(MapIntInt map, int key, int val)
```

Copies `key` and `val` into the table, replacing an existing value. Insertion invalidates live cursors and iterators; replacement does not.

Raises: `<bad-arg>` for a null map, or an allocation, size, or invariant cause while inserting. Allocation and size failures leave the table unchanged.

Source: `lib/typed-map.x:232`

MapIntInt.setdefault

```
int MapIntInt.setdefault(MapIntInt map, int key, int defval)
```

Returns the value for `key`, inserting `defval` when absent. Insertion copies the native key and value fields and invalidates live cursors and iterators; an existing-key read does not mutate the table.

Raises: `<bad-arg>` for a null map, or an allocation, size, or invariant cause while inserting. Allocation and size failures leave the table unchanged.

Source: `lib/typed-map.x:232`

MapIntInt.setindex

```
int MapIntInt.setindex(MapIntInt map, int key, int val)
```

Stores `val` under `key` and returns `val`. This is the bracket-facing form of `MapIntInt.set`, with the same cursor invalidation and failure behavior.

Source: `lib/typed-map.x:232`

MapIntInt.str

```
String MapIntInt.str(MapIntInt map)
```

Returns the typed-Map display String.

Source: `lib/typed-map.x:240`

MapIntInt.truth

```
int MapIntInt.truth(MapIntInt map)
```

Returns one when `map` contains an entry.

Source: `lib/typed-map.x:232`

MapIntInt.try_del

```
int MapIntInt.try_del(MapIntInt map, int key, int &?out)
```

Removes `key`, writes its value to `out`, and returns one when present. A null map, null `out`, or absent key returns zero without writing or mutation. Successful removal invalidates live cursors and iterators.

Source: `lib/typed-map.x:232`

MapIntInt.try_get

```
int MapIntInt.try_get(MapIntInt map, int key, int &?out)
```

Writes the value for `key` to `out` and returns one when present. A null map, null `out`, or absent key returns zero without writing.

Source: `lib/typed-map.x:232`

MapIntInt.try_next

```
int MapIntInt.try_next(MapIntInt map, unsigned &?cursor, int &?key, int &?val)
```

Writes the next bucket's key and value, advances `cursor`, and returns one. Initialize the caller-owned cursor to zero. A null argument or exhaustion returns zero without changing `key` or `val`. Traversal follows bucket order, not insertion order, and structural mutation invalidates the cursor.

Source: `lib/typed-map.x:232`

MapIntInt.updateindex

```
int MapIntInt.updateindex(MapIntInt map, int key, Symbol op, int rhs)
```

Applies `op` to the value for `key` and returns the stored result. `+` inserts `rhs` when the key is absent; other operations require an existing key. Supported operations depend on the generated value type.

Raises: `<bad-arg>` for a null map or missing required key, a cause from the generated value operation, or an allocation, size, or invariant cause while inserting. Invalid operations leave an existing value unchanged. Allocation and size failures leave the table unchanged.

Source: `lib/typed-map.x:232`

MapIntInt.var

```
Var MapIntInt.var(MapIntInt map)
```

Boxes `map` with its registered typed-map tag without copying it. The typed pointer and `Var` share the same mutable map identity.

Source: `lib/typed-map.x:241`

MapIntInt.write_repr

```
Buffer MapIntInt.write_repr(MapIntInt map, Buffer out)
```

Appends the readable typed-Map representation in bucket order.

Source: `lib/typed-map.x:240`

MapIntInt.write_str

```
Buffer MapIntInt.write_str(MapIntInt map, Buffer out)
```

Appends the typed-Map display text in bucket order.

Source: `lib/typed-map.x:240`

MapLongDouble

MapLongDouble.cleanup

```
void MapLongDouble.cleanup(MapLongDouble map)
```

Releases the record and its backing Blocks, borrowing stored values.

Source: `lib/typed-map.x:261`

MapLongDouble.compare

```
int MapLongDouble.compare(MapLongDouble a, MapLongDouble b)
```

Compares typed Maps by size and then sorted native key/value contents. Identical handles compare equal and NULL sorts first. Neither Map is mutated. Raises: `<alloc-fail>` while creating temporary storage, or any cause from key or value comparison.

Source: `lib/typed-map.x:260`

MapLongDouble.contains

```
int MapLongDouble.contains(MapLongDouble map, long key)
```

Returns one when `key` is present, or zero for a null map.

Source: `lib/typed-map.x:252`

MapLongDouble.copy

```
MapLongDouble MapLongDouble.copy(MapLongDouble map)
```

Returns a fresh shallow copy in the current `Scope`. The result has independent table storage but copies native key and value fields. A null input produces a fresh empty map.

Raises: an allocation, size, or invariant cause while copying.

Source: `lib/typed-map.x:252`

MapLongDouble.del

```
double MapLongDouble.del(MapLongDouble map, long key)
```

Removes `key` and returns its value. Successful removal invalidates live cursors and iterators. Prefer `try_del` when absence is an ordinary outcome. Boxed families return `void` for absence; native families raise `<bad-arg>`.

Source: `lib/typed-map.x:252`

MapLongDouble.enumerate

```
Iter MapLongDouble.enumerate(MapLongDouble map, Iter dest)
```

Initializes `dest` to yield canonical `(key value)` `Lists` in bucket order. The caller owns `dest`, and the iterator borrows `map`; both must remain live during traversal. A null `dest` returns NULL, and a null map is exhausted.

Each step boxes both fields and interns two `List` cells. Each cell belongs to the pool owning its canonical match, which may be an ancestor pool, or to the current pool when newly allocated. Structural mutation invalidates the iterator. Each MapLongDouble pull allocates its long key as a `Var` in the current `Scope`; the yielded `List` and anything retaining it must not outlive that `Scope`.

Raises: `<alloc-fail>` or `<size-limit>` while interning the `List`, and for MapLongDouble, `<alloc-fail>` or `<bad-enc>` while boxing its key. Shared causes do not return from the pull.

Source: `lib/typed-map.x:261`

MapLongDouble.equal

```
int MapLongDouble.equal(MapLongDouble a, MapLongDouble b)
```

Reports key/value equality independent of bucket and insertion order. Two null maps compare equal; a null and a nonnull map compare unequal.

Source: `lib/typed-map.x:252`

MapLongDouble.get

```
double MapLongDouble.get(MapLongDouble map, long key)
```

Returns the value for `key`. Prefer `try_get` when absence is an ordinary outcome. Boxed families return `void` for absence; native families raise `<bad-arg>`.

Source: `lib/typed-map.x:252`

MapLongDouble.getdefault

```
double MapLongDouble.getdefault(MapLongDouble map, long key, double defval)
```

Returns the value for `key`, or `defval` without inserting it. A null map is treated as an empty one.

Source: `lib/typed-map.x:252`

MapLongDouble.getindex

```
double MapLongDouble.getindex(MapLongDouble map, long key)
```

Returns the value selected by bracket indexing. This is the bracket-facing form of `MapLongDouble.get`, with the same failure behavior. Prefer bracket indexing in ordinary code.

Source: `lib/typed-map.x:252`

MapLongDouble.iter

```
Iter MapLongDouble.iter(MapLongDouble map, Iter dest)
```

Initializes `dest` as a single-pass iterator over boxed values. The caller owns `dest`, and the iterator borrows `map`; both must remain live during traversal. A null `dest` returns NULL. Values follow bucket order, a null map is exhausted, and structural mutation invalidates the iterator.

Source: `lib/typed-map.x:261`

MapLongDouble.keys

```
Iter MapLongDouble.keys(MapLongDouble map, Iter dest)
```

Initializes `dest` as a single-pass iterator over boxed keys. The caller owns `dest`, and the iterator borrows `map`; both must remain live during traversal. A null `dest` returns NULL. Keys follow bucket order, a null map is exhausted, and structural mutation invalidates the iterator. MapIntInt and both String-keyed families box keys without allocation. Each MapLongDouble pull allocates its long key as a `Var` in the current `Scope`; the yielded `Var` and anything retaining it must not outlive that `Scope`.

Raises: MapLongDouble iteration may raise `<alloc-fail>` or `<bad-enc>` while boxing a key.

Source: `lib/typed-map.x:261`

MapLongDouble.len

```
unsigned MapLongDouble.len(MapLongDouble map)
```

Returns the number of entries, or zero for a null map.

Source: `lib/typed-map.x:252`

MapLongDouble.map

```
Map MapLongDouble.map(MapLongDouble map)
```

Returns a fresh ordinary Map containing boxed keys and values. A null input returns NULL. The typed Map and its storage are unchanged.

Raises: any cause from boxing or inserting an entry.

Source: `lib/typed-map.x:260`

MapLongDouble.merge

```
MapLongDouble MapLongDouble.merge(MapLongDouble map, MapLongDouble other)
```

Copies every entry of `other` into `map` and returns the destination. `other` wins conflicts. A null `other` is a no-op; a null destination is replaced by a fresh map. When a supplied destination fails partway, entries already copied remain; a newly created destination is discarded.

Raises: an allocation, size, or invariant cause while inserting.

Source: `lib/typed-map.x:252`

MapLongDouble.new

```
MapLongDouble MapLongDouble.new(void)
```

Returns a fresh empty map owned by the current `Scope`. The initial table has two buckets and grows automatically.

Raises: `<alloc-fail>` or `<size-limit>` when storage cannot be created.

Source: `lib/typed-map.x:252`

MapLongDouble.new_capacity

```
MapLongDouble MapLongDouble.new_capacity(unsigned capacity)
```

Returns a fresh empty map with exactly `capacity` buckets. The result and its backing arrays belong to the current `Scope`. `capacity` must be a power of two of at least two.

Raises: `<bad-arg>` for another capacity, or `<alloc-fail>` / `<size-limit>` when storage cannot be created.

Source: `lib/typed-map.x:252`

MapLongDouble.postfixindex

```
double MapLongDouble.postfixindex(MapLongDouble map, long key, Symbol op)
```

Applies postfix `++` or `--` and returns the previous value. The key must already exist, and only generated numeric families support these operations.

Raises: `<bad-arg>` for a null map or absent key, or `<bad-op>` for an unsupported operation. These failures leave the value unchanged.

Source: `lib/typed-map.x:252`

MapLongDouble.repr

```
String MapLongDouble.repr(MapLongDouble map)
```

Returns the readable typed-Map representation.

Source: `lib/typed-map.x:260`

MapLongDouble.set

```
void MapLongDouble.set(MapLongDouble map, long key, double val)
```

Copies `key` and `val` into the table, replacing an existing value. Insertion invalidates live cursors and iterators; replacement does not.

Raises: `<bad-arg>` for a null map, or an allocation, size, or invariant cause while inserting. Allocation and size failures leave the table unchanged.

Source: `lib/typed-map.x:252`

MapLongDouble.setdefault

```
double MapLongDouble.setdefault(MapLongDouble map, long key, double defval)
```

Returns the value for `key`, inserting `defval` when absent. Insertion copies the native key and value fields and invalidates live cursors and iterators; an existing-key read does not mutate the table.

Raises: `<bad-arg>` for a null map, or an allocation, size, or invariant cause while inserting. Allocation and size failures leave the table unchanged.

Source: `lib/typed-map.x:252`

MapLongDouble.setindex

```
double MapLongDouble.setindex(MapLongDouble map, long key, double val)
```

Stores `val` under `key` and returns `val`. This is the bracket-facing form of `MapLongDouble.set`, with the same cursor invalidation and failure behavior.

Source: `lib/typed-map.x:252`

MapLongDouble.str

```
String MapLongDouble.str(MapLongDouble map)
```

Returns the typed-Map display String.

Source: `lib/typed-map.x:260`

MapLongDouble.truth

```
int MapLongDouble.truth(MapLongDouble map)
```

Returns one when `map` contains an entry.

Source: `lib/typed-map.x:252`

MapLongDouble.try_del

```
int MapLongDouble.try_del(MapLongDouble map, long key, double &?out)
```

Removes `key`, writes its value to `out`, and returns one when present. A null map, null `out`, or absent key returns zero without writing or mutation. Successful removal invalidates live cursors and iterators.

Source: `lib/typed-map.x:252`

MapLongDouble.try_get

```
int MapLongDouble.try_get(MapLongDouble map, long key, double &?out)
```

Writes the value for `key` to `out` and returns one when present. A null map, null `out`, or absent key returns zero without writing.

Source: `lib/typed-map.x:252`

MapLongDouble.try_next

```
int MapLongDouble.try_next(MapLongDouble map, unsigned &?cursor, long &?key, double &?val)
```

Writes the next bucket's key and value, advances `cursor`, and returns one. Initialize the caller-owned cursor to zero. A null argument or exhaustion returns zero without changing `key` or `val`. Traversal follows bucket order, not insertion order, and structural mutation invalidates the cursor.

Source: `lib/typed-map.x:252`

MapLongDouble.updateindex

```
double MapLongDouble.updateindex(MapLongDouble map, long key, Symbol op, double rhs)
```

Applies `op` to the value for `key` and returns the stored result. `+` inserts `rhs` when the key is absent; other operations require an existing key. Supported operations depend on the generated value type.

Raises: `<bad-arg>` for a null map or missing required key, a cause from the generated value operation, or an allocation, size, or invariant cause while inserting. Invalid operations leave an existing value unchanged. Allocation and size failures leave the table unchanged.

Source: `lib/typed-map.x:252`

MapLongDouble.var

```
Var MapLongDouble.var(MapLongDouble map)
```

Boxes `map` with its registered typed-map tag without copying it. The typed pointer and `Var` share the same mutable map identity.

Source: `lib/typed-map.x:261`

MapLongDouble.write_repr

```
Buffer MapLongDouble.write_repr(MapLongDouble map, Buffer out)
```

Appends the readable typed-Map representation in bucket order.

Source: `lib/typed-map.x:260`

MapLongDouble.write_str

```
Buffer MapLongDouble.write_str(MapLongDouble map, Buffer out)
```

Appends the typed-Map display text in bucket order.

Source: `lib/typed-map.x:260`

MapStringInt

MapStringInt.cleanup

```
void MapStringInt.cleanup(MapStringInt map)
```

Releases the record and its backing Blocks, borrowing stored values.

Source: `lib/typed-map.x:336`

MapStringInt.compare

```
int MapStringInt.compare(MapStringInt a, MapStringInt b)
```

Compares typed Maps by size and then sorted native key/value contents. Identical handles compare equal and NULL sorts first. Neither Map is mutated. Raises: `<alloc-fail>` while creating temporary storage, or any cause from key or value comparison.

Source: `lib/typed-map.x:319`

MapStringInt.contains

```
int MapStringInt.contains(MapStringInt map, String key)
```

Returns one when `key` is present, or zero for a null map.

Source: `lib/typed-map.x:311`

MapStringInt.copy

```
MapStringInt MapStringInt.copy(MapStringInt map)
```

Returns a fresh shallow copy in the current `Scope`. The result has independent table storage but copies native key and value fields. A null input produces a fresh empty map.

Raises: an allocation, size, or invariant cause while copying.

Source: `lib/typed-map.x:311`

MapStringInt.del

```
int MapStringInt.del(MapStringInt map, String key)
```

Removes `key` and returns its value. Successful removal invalidates live cursors and iterators. Prefer `try_del` when absence is an ordinary outcome. Boxed families return `void` for absence; native families raise `<bad-arg>`.

Source: `lib/typed-map.x:311`

MapStringInt.enumerate

```
Iter MapStringInt.enumerate(MapStringInt map, Iter dest)
```

Initializes `dest` to yield canonical `(key value)` `Lists` in bucket order. The caller owns `dest`, and the iterator borrows `map`; both must remain live during traversal. A null `dest` returns NULL, and a null map is exhausted. Each step boxes both fields and interns two `List` cells. Each cell belongs to the pool owning its canonical match, which may be an ancestor pool, or to the current pool when newly allocated. Structural mutation invalidates the iterator. Each MapLongDouble pull allocates its long key as a `Var` in the current `Scope`; the yielded `List` and anything retaining it must not outlive that `Scope`.

Raises: `<alloc-fail>` or `<size-limit>` while interning the `List`, and for MapLongDouble, `<alloc-fail>` or `<bad-enc>` while boxing its key. Shared causes do not return from the pull.

Source: `lib/typed-map.x:336`

MapStringInt.equal

```
int MapStringInt.equal(MapStringInt a, MapStringInt b)
```

Reports key/value equality independent of bucket and insertion order. Two null maps compare equal; a null and a nonnull map compare unequal.

Source: `lib/typed-map.x:311`

MapStringInt.get

```
int MapStringInt.get(MapStringInt map, String key)
```

Returns the value for `key`. Prefer `try_get` when absence is an ordinary outcome. Boxed families return `void` for absence; native families raise `<bad-arg>`.

Source: `lib/typed-map.x:311`

MapStringInt.getdefault

```
int MapStringInt.getdefault(MapStringInt map, String key, int defval)
```

Returns the value for `key`, or `defval` without inserting it. A null map is treated as an empty one.

Source: `lib/typed-map.x:311`

MapStringInt.getindex

```
int MapStringInt.getindex(MapStringInt map, String key)
```

Returns the value selected by bracket indexing. This is the bracket-facing form of `MapStringInt.get`, with the same failure behavior. Prefer bracket indexing in ordinary code.

Source: `lib/typed-map.x:311`

MapStringInt.iter

```
Iter MapStringInt.iter(MapStringInt map, Iter dest)
```

Initializes `dest` as a single-pass iterator over boxed values. The caller owns `dest`, and the iterator borrows `map`; both must remain live during traversal. A null `dest` returns NULL. Values follow bucket order, a null map is exhausted, and structural mutation invalidates the iterator.

Source: `lib/typed-map.x:336`

MapStringInt.keys

```
Iter MapStringInt.keys(MapStringInt map, Iter dest)
```

Initializes `dest` as a single-pass iterator over boxed keys. The caller owns `dest`, and the iterator borrows `map`; both must remain live during traversal. A null `dest` returns NULL. Keys follow bucket order, a null map is exhausted, and structural mutation invalidates the iterator. MapIntInt and both String-keyed families box keys without allocation. Each MapLongDouble pull allocates its long key as a `Var` in the current `Scope`; the yielded `Var` and anything retaining it must not outlive that `Scope`.

Raises: MapLongDouble iteration may raise `<alloc-fail>` or `<bad-enc>` while boxing a key.

Source: `lib/typed-map.x:336`

MapStringInt.len

```
unsigned MapStringInt.len(MapStringInt map)
```

Returns the number of entries, or zero for a null map.

Source: `lib/typed-map.x:311`

MapStringInt.map

```
Map MapStringInt.map(MapStringInt map)
```

Returns a fresh ordinary Map containing boxed keys and values. A null input returns NULL. The typed Map and its storage are unchanged.

Raises: any cause from boxing or inserting an entry.

Source: `lib/typed-map.x:319`

MapStringInt.merge

```
MapStringInt MapStringInt.merge(MapStringInt map, MapStringInt other)
```

Copies every entry of `other` into `map` and returns the destination. `other` wins conflicts. A null `other` is a no-op; a null destination is replaced by a fresh map. When a supplied destination fails partway, entries already copied remain; a newly created destination is discarded.

Raises: an allocation, size, or invariant cause while inserting.

Source: `lib/typed-map.x:311`

MapStringInt.new

```
MapStringInt MapStringInt.new(void)
```

Returns a fresh empty map owned by the current `Scope`. The initial table has two buckets and grows automatically.

Raises: `<alloc-fail>` or `<size-limit>` when storage cannot be created.

Source: `lib/typed-map.x:311`

MapStringInt.new_capacity

```
MapStringInt MapStringInt.new_capacity(unsigned capacity)
```

Returns a fresh empty map with exactly `capacity` buckets. The result and its backing arrays belong to the current `Scope`. `capacity` must be a power of two of at least two.

Raises: `<bad-arg>` for another capacity, or `<alloc-fail>` / `<size-limit>` when storage cannot be created.

Source: `lib/typed-map.x:311`

MapStringInt.postfixindex

```
int MapStringInt.postfixindex(MapStringInt map, String key, Symbol op)
```

Applies postfix `++` or `--` and returns the previous value. The key must already exist, and only generated numeric families support these operations.

Raises: `<bad-arg>` for a null map or absent key, or `<bad-op>` for an unsupported operation. These failures leave the value unchanged.

Source: `lib/typed-map.x:311`

MapStringInt.repr

```
String MapStringInt.repr(MapStringInt map)
```

Returns the readable typed-Map representation.

Source: `lib/typed-map.x:319`

MapStringInt.set

```
void MapStringInt.set(MapStringInt map, String key, int val)
```

Copies `key` and `val` into the table, replacing an existing value. Insertion invalidates live cursors and iterators; replacement does not.

Raises: `<bad-arg>` for a null map, or an allocation, size, or invariant cause while inserting. Allocation and size failures leave the table unchanged.

Source: `lib/typed-map.x:311`

MapStringInt.setdefault

```
int MapStringInt.setdefault(MapStringInt map, String key, int defval)
```

Returns the value for `key`, inserting `defval` when absent. Insertion copies the native key and value fields and invalidates live cursors and iterators; an existing-key read does not mutate the table.

Raises: `<bad-arg>` for a null map, or an allocation, size, or invariant cause while inserting. Allocation and size failures leave the table unchanged.

Source: `lib/typed-map.x:311`

MapStringInt.setindex

```
int MapStringInt.setindex(MapStringInt map, String key, int val)
```

Stores `val` under `key` and returns `val`. This is the bracket-facing form of `MapStringInt.set`, with the same cursor invalidation and failure behavior.

Source: `lib/typed-map.x:311`

MapStringInt.str

```
String MapStringInt.str(MapStringInt map)
```

Returns the typed-Map display String.

Source: `lib/typed-map.x:319`

MapStringInt.truth

```
int MapStringInt.truth(MapStringInt map)
```

Returns one when `map` contains an entry.

Source: `lib/typed-map.x:311`

MapStringInt.try_del

```
int MapStringInt.try_del(MapStringInt map, String key, int &?out)
```

Removes `key`, writes its value to `out`, and returns one when present. A null map, null `out`, or absent key returns zero without writing or mutation. Successful removal invalidates live cursors and iterators.

Source: `lib/typed-map.x:311`

MapStringInt.try_get

```
int MapStringInt.try_get(MapStringInt map, String key, int &?out)
```

Writes the value for `key` to `out` and returns one when present. A null map, null `out`, or absent key returns zero without writing.

Source: `lib/typed-map.x:311`

MapStringInt.try_next

```
int MapStringInt.try_next(MapStringInt map, unsigned &?cursor, String &?key, int &?val)
```

Writes the next bucket's key and value, advances `cursor`, and returns one. Initialize the caller-owned cursor to zero. A null argument or exhaustion returns zero without changing `key` or `val`. Traversal follows bucket order, not insertion order, and structural mutation invalidates the cursor.

Source: `lib/typed-map.x:311`

MapStringInt.updateindex

```
int MapStringInt.updateindex(MapStringInt map, String key, Symbol op, int rhs)
```

Applies `op` to the value for `key` and returns the stored result. `+` inserts `rhs` when the key is absent; other operations require an existing key. Supported operations depend on the generated value type.

Raises: `<bad-arg>` for a null map or missing required key, a cause from the generated value operation, or an allocation, size, or invariant cause while inserting. Invalid operations leave an existing value unchanged. Allocation and size failures leave the table unchanged.

Source: `lib/typed-map.x:311`

MapStringInt.var

```
Var MapStringInt.var(MapStringInt map)
```

Boxes `map` with its registered typed-map tag without copying it. The typed pointer and `Var` share the same mutable map identity.

Source: `lib/typed-map.x:336`

MapStringInt.write_repr

```
Buffer MapStringInt.write_repr(MapStringInt map, Buffer out)
```

Appends the readable typed-Map representation in bucket order.

Source: `lib/typed-map.x:319`

MapStringInt.write_str

```
Buffer MapStringInt.write_str(MapStringInt map, Buffer out)
```

Appends the typed-Map display text in bucket order.

Source: `lib/typed-map.x:319`

MapStringString

MapStringString.cleanup

```
void MapStringString.cleanup(MapStringString map)
```

Releases the record and its backing Blocks, borrowing stored values.

Source: `lib/typed-map.x:299`

MapStringString.compare

```
int MapStringString.compare(MapStringString a, MapStringString b)
```

Compares typed Maps by size and then sorted native key/value contents. Identical handles compare equal and NULL sorts first. Neither Map is mutated. Raises: `<alloc-fail>` while creating temporary storage, or any cause from key or value comparison.

Source: `lib/typed-map.x:281`

MapStringString.contains

```
int MapStringString.contains(MapStringString map, String key)
```

Returns one when `key` is present, or zero for a null map.

Source: `lib/typed-map.x:273`

MapStringString.copy

```
MapStringString MapStringString.copy(MapStringString map)
```

Returns a fresh shallow copy in the current `Scope`. The result has independent table storage but copies native key and value fields. A null input produces a fresh empty map.

Raises: an allocation, size, or invariant cause while copying.

Source: `lib/typed-map.x:273`

MapStringString.del

```
String MapStringString.del(MapStringString map, String key)
```

Removes `key` and returns its value. Successful removal invalidates live cursors and iterators. Prefer `try_del` when absence is an ordinary outcome. Boxed families return `void` for absence; native families raise `<bad-arg>`.

Source: `lib/typed-map.x:273`

MapStringString.enumerate

```
Iter MapStringString.enumerate(MapStringString map, Iter dest)
```

Initializes `dest` to yield canonical `(key value)` `Lists` in bucket order. The caller owns `dest`, and the iterator borrows `map`; both must remain live during traversal. A null `dest` returns NULL, and a null map is exhausted. Each step boxes both fields and interns two `List` cells. Each cell belongs to the pool owning its canonical match, which may be an ancestor pool, or to the current pool when newly allocated. Structural mutation invalidates the iterator. Each MapLongDouble pull allocates its long key as a `Var` in the current `Scope`; the yielded `List` and anything retaining it must not outlive that `Scope`.

Raises: `<alloc-fail>` or `<size-limit>` while interning the `List`, and for MapLongDouble, `<alloc-fail>` or `<bad-enc>` while boxing its key. Shared causes do not return from the pull.

Source: `lib/typed-map.x:299`

MapStringString.equal

```
int MapStringString.equal(MapStringString a, MapStringString b)
```

Reports key/value equality independent of bucket and insertion order. Two null maps compare equal; a null and a nonnull map compare unequal.

Source: `lib/typed-map.x:273`

MapStringString.get

```
String MapStringString.get(MapStringString map, String key)
```

Returns the value for `key`. Prefer `try_get` when absence is an ordinary outcome. Boxed families return `void` for absence; native families raise `<bad-arg>`.

Source: `lib/typed-map.x:273`

MapStringString.getdefault

```
String MapStringString.getdefault(MapStringString map, String key, String defval)
```

Returns the value for `key`, or `defval` without inserting it. A null map is treated as an empty one.

Source: `lib/typed-map.x:273`

MapStringString.getindex

```
String MapStringString.getindex(MapStringString map, String key)
```

Returns the value selected by bracket indexing. This is the bracket-facing form of `MapStringString.get`, with the same failure behavior. Prefer bracket indexing in ordinary code.

Source: `lib/typed-map.x:273`

MapStringString.iter

```
Iter MapStringString.iter(MapStringString map, Iter dest)
```

Initializes `dest` as a single-pass iterator over boxed values. The caller owns `dest`, and the iterator borrows `map`; both must remain live during traversal. A null `dest` returns NULL. Values follow bucket order, a null map is exhausted, and structural mutation invalidates the iterator.

Source: `lib/typed-map.x:299`

MapStringString.keys

```
Iter MapStringString.keys(MapStringString map, Iter dest)
```

Initializes `dest` as a single-pass iterator over boxed keys. The caller owns `dest`, and the iterator borrows `map`; both must remain live during traversal. A null `dest` returns NULL. Keys follow bucket order, a null map is exhausted, and structural mutation invalidates the iterator. MapIntInt and both String-keyed families box keys without allocation. Each MapLongDouble pull allocates its long key as a `Var` in the current `Scope`; the yielded `Var` and anything retaining it must not outlive that `Scope`.

Raises: MapLongDouble iteration may raise `<alloc-fail>` or `<bad-enc>` while boxing a key.

Source: `lib/typed-map.x:299`

MapStringString.len

```
unsigned MapStringString.len(MapStringString map)
```

Returns the number of entries, or zero for a null map.

Source: `lib/typed-map.x:273`

MapStringString.map

```
Map MapStringString.map(MapStringString map)
```

Returns a fresh ordinary Map containing boxed keys and values. A null input returns NULL. The typed Map and its storage are unchanged.

Raises: any cause from boxing or inserting an entry.

Source: `lib/typed-map.x:281`

MapStringString.merge

```
MapStringString MapStringString.merge(MapStringString map, MapStringString other)
```

Copies every entry of `other` into `map` and returns the destination. `other` wins conflicts. A null `other` is a no-op; a null destination is replaced by a fresh map. When a supplied destination fails partway, entries already copied remain; a newly created destination is discarded.

Raises: an allocation, size, or invariant cause while inserting.

Source: `lib/typed-map.x:273`

MapStringString.new

```
MapStringString MapStringString.new(void)
```

Returns a fresh empty map owned by the current `Scope`. The initial table has two buckets and grows automatically.

Raises: `<alloc-fail>` or `<size-limit>` when storage cannot be created.

Source: `lib/typed-map.x:273`

MapStringString.new_capacity

```
MapStringString MapStringString.new_capacity(unsigned capacity)
```

Returns a fresh empty map with exactly `capacity` buckets. The result and its backing arrays belong to the current `Scope`. `capacity` must be a power of two of at least two.

Raises: <bad-arg> for another capacity, or <alloc-fail> / <size-limit> when storage cannot be created.

Source: lib/typed-map.x:273

MapStringString.postfixindex

```
String MapStringString.postfixindex(MapStringString map, String key, Symbol op)
```

Applies postfix ++ or -- and returns the previous value. The key must already exist, and only generated numeric families support these operations.

Raises: <bad-arg> for a null map or absent key, or <bad-op> for an unsupported operation. These failures leave the value unchanged.

Source: lib/typed-map.x:273

MapStringString.repr

```
String MapStringString.repr(MapStringString map)
```

Returns the readable typed-Map representation.

Source: lib/typed-map.x:281

MapStringString.set

```
void MapStringString.set(MapStringString map, String key, String val)
```

Copies key and val into the table, replacing an existing value. Insertion invalidates live cursors and iterators; replacement does not.

Raises: <bad-arg> for a null map, or an allocation, size, or invariant cause while inserting. Allocation and size failures leave the table unchanged.

Source: lib/typed-map.x:273

MapStringString.setdefault

```
String MapStringString.setdefault(MapStringString map, String key, String defval)
```

Returns the value for key, inserting defval when absent. Insertion copies the native key and value fields and invalidates live cursors and iterators; an existing-key read does not mutate the table.

Raises: <bad-arg> for a null map, or an allocation, size, or invariant cause while inserting. Allocation and size failures leave the table unchanged.

Source: lib/typed-map.x:273

MapStringString.setindex

```
String MapStringString.setindex(MapStringString map, String key, String val)
```

Stores val under key and returns val. This is the bracket-facing form of MapStringString.set, with the same cursor invalidation and failure behavior.

Source: lib/typed-map.x:273

MapStringString.str

```
String MapStringString.str(MapStringString map)
```

Returns the typed-Map display String.

Source: lib/typed-map.x:281

MapStringString.truth

```
int MapStringString.truth(MapStringString map)
```

Returns one when `map` contains an entry.

Source: `lib/typed-map.x:273`

MapStringString.try_del

```
int MapStringString.try_del(MapStringString map, String key, String &?out)
```

Removes `key`, writes its value to `out`, and returns one when present. A null map, null `out`, or absent key returns zero without writing or mutation. Successful removal invalidates live cursors and iterators.

Source: `lib/typed-map.x:273`

MapStringString.try_get

```
int MapStringString.try_get(MapStringString map, String key, String &?out)
```

Writes the value for `key` to `out` and returns one when present. A null map, null `out`, or absent key returns zero without writing.

Source: `lib/typed-map.x:273`

MapStringString.try_next

```
int MapStringString.try_next(MapStringString map, unsigned &?cursor, String &?key, String &?val)
```

Writes the next bucket's key and value, advances `cursor`, and returns one. Initialize the caller-owned cursor to zero. A null argument or exhaustion returns zero without changing `key` or `val`. Traversal follows bucket order, not insertion order, and structural mutation invalidates the cursor.

Source: `lib/typed-map.x:273`

MapStringString.updateindex

```
String MapStringString.updateindex(MapStringString map, String key, Symbol op, String rhs)
```

Applies `op` to the value for `key` and returns the stored result. `+` inserts `rhs` when the key is absent; other operations require an existing key. Supported operations depend on the generated value type.

Raises: `<bad-arg>` for a null map or missing required key, a cause from the generated value operation, or an allocation, size, or invariant cause while inserting. Invalid operations leave an existing value unchanged. Allocation and size failures leave the table unchanged.

Source: `lib/typed-map.x:273`

MapStringString.var

```
Var MapStringString.var(MapStringString map)
```

Boxes `map` with its registered typed-map tag without copying it. The typed pointer and `Var` share the same mutable map identity.

Source: `lib/typed-map.x:299`

MapStringString.write_repr

```
Buffer MapStringString.write_repr(MapStringString map, Buffer out)
```

Appends the readable typed-Map representation in bucket order.

Source: `lib/typed-map.x:281`

MapStringString.write_str

```
Buffer MapStringString.write_str(MapStringString map, Buffer out)
```

Appends the typed-Map display text in bucket order.

Source: `lib/typed-map.x:281`

`Var`

Var.mapintint

```
MapIntInt Var.mapintint(Var value)
```

Returns the typed-map pointer carried by `value`. `value` must carry this family's registered tag; the converter does not validate the tag before reading its pointer payload.

Source: `lib/typed-map.x:241`

Var.maplongdouble

```
MapLongDouble Var.maplongdouble(Var value)
```

Returns the typed-map pointer carried by `value`. `value` must carry this family's registered tag; the converter does not validate the tag before reading its pointer payload.

Source: `lib/typed-map.x:261`

Var.mapstringint

```
MapStringInt Var.mapstringint(Var value)
```

Returns the typed-map pointer carried by `value`. `value` must carry this family's registered tag; the converter does not validate the tag before reading its pointer payload.

Source: `lib/typed-map.x:336`

Var.mapstringstring

```
MapStringString Var.mapstringstring(Var value)
```

Returns the typed-map pointer carried by `value`. `value` must carry this family's registered tag; the converter does not validate the tag before reading its pointer payload.

Source: `lib/typed-map.x:299`

Public types

Type	Kind	Summary
`MapIntInt`	struct	A mutable, `Scope`-owned map from native `int` keys to `int` values.
`MapLongDouble`	struct	A mutable, `Scope`-owned map from native `long` keys to `double` values.
`MapStringInt`	struct	A mutable, `Scope`-owned map from canonical `String` keys to native `int` values.
`MapStringString`	struct	A mutable, `Scope`-owned map from canonical `String`s to canonical `String`s.

MapIntInt

```
typedef struct MapIntInt { Scope scope, Bytes hashes, entries; unsigned used; unsigned capacity; unsigned mask; } *MapIntInt
```

A mutable, `Scope`-owned map from native `int` keys to `int` values. Keys and values are copied into native entry fields; assignment shares the map, while `MapIntInt.copy` makes an independent table. Compound indexed updates support arithmetic, remainder, bitwise, and shift operators, plus postfix increment and decrement.

Source: `lib/typed-map.x:27`

MapLongDouble

```
typedef struct MapLongDouble { Scope scope, Bytes hashes, entries; unsigned used; unsigned capacity; unsigned mask; } *MapLongDouble
```

A mutable, `Scope`-owned map from native `long` keys to `double` values. Keys and values are copied into native entry fields; assignment shares the map, while `MapLongDouble.copy` makes an independent table. Compound indexed updates support `+`, `-`, `*`, and `/`, plus postfix increment and decrement.

Source: `lib/typed-map.x:40`

MapStringInt

```
typedef struct MapStringInt { Scope scope, Bytes hashes, entries; unsigned used; unsigned capacity; unsigned mask; } *MapStringInt
```

A mutable, `Scope`-owned map from canonical `String` keys to native `int` values. Exporting an owned map rebuilds it with destination-canonical keys before replacing its storage. Compound indexed updates support arithmetic, remainder, bitwise, and shift operators, plus postfix increment and decrement.

Source: `lib/typed-map.x:70`

MapStringString

```
typedef struct MapStringString { Scope scope, Bytes hashes, entries; unsigned used; unsigned capacity; unsigned mask; } *MapStringString
```

A mutable, `Scope`-owned map from canonical `Strings` to canonical `Strings`. The native entry fields borrow the supplied `String` pointers. Exporting a map owned by a `Context` recanonicalizes only `Strings` owned by that `Context` in the destination pool before moving the map. Borrowed or outer-owned `Strings` retain their existing pool lifetime, which must outlive the map. Compound indexed update supports only concatenation; postfix update is unsupported.

Source: `lib/typed-map.x:57`

Design notes

These optional families store keys and values in native record fields. Their static operations do not box through `Var`. Separate methods provide `Var` conversion and `Iter` traversal.

Tests and examples

`make verify` (`unittest/test-typed-map.x`).

lib/var/var.x

Variant type for dynamic typing.

Primary API

Function	Summary
`x2c_var_custom_descriptor`	Returns a boxed custom object's descriptor, or NULL for another value.
`x2c_var_declare`	Returns the process-lifetime descriptor declared for custom `tag`, declaring it on first use.
`x2c_var_descriptor_index`	Returns `value`'s dense built-in descriptor row, or `-1` when it has none.
`x2c_var_tag_descriptor_index`	Returns a built-in object or `Symbol` tag's descriptor row, or `-1`.
`Var.box_record`	Boxes a copy of the `size`-byte record at `record` as custom `tag`.
`Var.is`	Reports whether `value` has the requested runtime tag.
`Var.is_floating`	Reports whether `v` belongs to the floating runtime family.
`Var.is_integer`	Reports whether `v` belongs to the integer runtime family.
`Var.is_nil`	Reports whether `v` is the typed empty `List`.
`Var.is_null`	Reports whether `v` is the all-zero `Null` value.
`Var.is_object`	Reports whether `v` holds a registered boxed object.
`Var.is_pointer`	Reports whether `v` holds a native pointer.
`Var.is_reference`	Reports whether `v` holds a pointer to a boxed handle.

Function	Summary
Var.is_void	Reports whether v is the absence sentinel void.
Var.new	Constructs a Var with tag from its tag-directed variadic payload.
Var.null	Returns Null, the all-zero Var that stands for external nil.
Var.tag_bottom	Returns the bottom encoding field of built-in tag.
Var.tag_top	Returns the top encoding field of built-in tag.

Functions

x2c_var_custom_descriptor

```
VarDescriptor *x2c_var_custom_descriptor(Var value)
```

Returns a boxed custom object's descriptor, or NULL for another value.

Source: lib/var/classes.x:80

x2c_var_declare

```
VarDescriptor *x2c_var_declare(Symbol tag)
```

Returns the process-lifetime descriptor declared for custom tag, declaring it on first use. The caller holds the descriptor lock and has checked that registration is open and tag is not built in.

Source: lib/var/classes.x:222

x2c_var_descriptor_index

```
int x2c_var_descriptor_index(Var value)
```

Returns value's dense built-in descriptor row, or -1 when it has none. The row is the tag's offset from <array> and includes the Symbol row. Numbers, pointers, references, custom tags, and invalid encodings have no built-in row.

Source: lib/var/encoding.x:273

x2c_var_tag_descriptor_index

```
int x2c_var_tag_descriptor_index(Symbol tag)
```

Returns a built-in object or Symbol tag's descriptor row, or -1.

Source: lib/var/encoding.x:289

Var

Var.box_record

```
Var Var.box_record(Symbol tag, const void *record, size_t size)
```

Boxes a copy of the size-byte record at record as custom tag. The copy is allocated in the active Scope. Once the direct custom rows are taken, the copy carries its descriptor in front of it.

Raises: <bad-target> when tag names no declared class, or <alloc-fail> when the copy cannot be allocated.

Source: lib/var/classes.x:96

Var.is

```
int Var.is(Var var, Symbol tag)
```

Reports whether value has the requested runtime tag. This tests one family. A Var holding an unsigned char answers 0 for <i32> even though both are integers. When any integer will do, ask Var.is_integer, or compare Var.kind. As with Var.tag, validate externally constructed bits first: an invalid encoding uses the <f64> fallback.

Source: lib/var/encoding.x:165

Var.is_floating

```
meta native int Var.is_floating(Var v)
```

Reports whether v belongs to the floating runtime family. True for <f32>, <f64>, and <ldouble>, and for the discrete <nan>, <+inf>, and <-inf> values. Read the payload with Var.floating, or with Var.long_double_value when the tag is <ldouble> and the extra precision matters.

Source: lib/var/encoding.x:190

Var.is_integer

```
int Var.is_integer(Var v)
```

Reports whether v belongs to the integer runtime family. True for every signed and unsigned integer family from <u8> through <ullong>, including the scope-owned wide boxes. Symbols are a kind of their own and answer 0 here, even though Var.integer returns a Symbol's numeric value.

Source: lib/var/encoding.x:198

Var.is_nil

```
meta native int Var.is_nil(Var v)
```

Reports whether v is the typed empty List. The empty List is a native null pointer carrying the <list> tag. Var.is_null answers 0 and Var.tag answers <list>. Every %() and every exhausted List.cdr yields this one value, so an identity test on the bits is a valid emptiness test.

Source: lib/var/encoding.x:258

Var.is_null

```
meta native int Var.is_null(Var v)
```

Reports whether v is the all-zero Null value. Null is a value: the null pointer and the external nil. It is legal collection data, iterating a List can return it, and it is false in a condition. Its tag decodes as <p48>, so there is no dedicated null family for Var.is to match.

Write Null as (Var) { .u64 = 0 }. An unresolved C NULL macro also converts to this value when its x2c target is Var.

Source: lib/var/encoding.x:245

Var.is_object

```
meta native int Var.is_object(Var v)
```

Reports whether v holds a registered boxed object. True for the builtin classes such as String, List, Array, Map, File, and Iter, and for any tag registered with Var.register_object_tag. A pointer to a handle, such as <string*>, is <reference> instead and answers 0 here.

Source: lib/var/encoding.x:212

Var.is_pointer

```
meta native int Var.is_pointer(Var v)
```

Reports whether v holds a native pointer.

Source: lib/var/encoding.x:201

Var.is_reference

```
meta native int Var.is_reference(Var v)
```

Reports whether v holds a pointer to a boxed handle.

Source: lib/var/encoding.x:204

Var.is_void

```
int Var.is_void(Var v)
```

Reports whether `v` is the absence sentinel `void`. An API returns `void` to say there is nothing here. It is excluded from every collection and iterator value domain: it cannot be pushed into an `Array` or stored in a `Map`. Equality and identity still inspect it. Two sentinels compare equal and identical, while one sentinel and one ordinary value compare unequal. Truthiness, hashing, ordering, conversion, and iteration on `void` raise to a matching catch or terminate.

`Null` is the all-zero `Var`: legal collection data and false in a condition. A `void` result may mean missing, exhausted, or invalid; each API documents its meaning.

```
printf("null: void=%d null=%d\n", raw is void, raw.is_null());
printf("void: void=%d null=%d\n", void is void, void.is_null());
```

This is the test that accepts a `void` argument.

Source: `lib/var/encoding.x:234`

Var.new

```
Var Var.new(Symbol tag, ...)
```

Constructs a `Var` with `tag` from its tag-directed variadic payload. The tag chooses how the argument is read, so pass exactly the C type the tag names: an `int` for `<i32>`, an `unsigned long` for `<u48>`, a `double` for `<f64>`, a `long double` for `<ldouble>`, a pointer for any pointer, reference, or object family, and a `Symbol`'s numeric value for `<symbol>`. `<void>` consumes no payload. Variadic arguments are not converted for you. Assignment, `Var boxed = 42;`, is the usual way to box a value. Use this constructor when the tag is computed at run time.

Immediate values are stored inline. Wide numeric tags allocate a box in the active `Scope`. Pointer, reference, and object tags borrow the address and encode only its low 48 bits; they do not take ownership, and the address must satisfy the alignment implied by the tag. A declared custom tag is accepted too and requires an 8-byte-aligned pointer. Once the direct custom rows are taken, a custom object boxes through a process-lifetime cell, one per class and address, so boxing one object twice gives identical bits.

Raises: `<bad-target>` when `tag` is neither known nor registered, `<conv-range>` when a scalar does not fit the tag's payload width, `<bad-arg>` when an `<array>` or `<map>` pointer is null, `<alloc-fail>` when a wide box cannot be allocated, and `<bad-enc>` when a box address cannot be represented or a pointer does not satisfy its tag's alignment.

Source: `lib/var/construction.x:28`

Var.null

```
meta native Var Var.null(void)
```

Returns `Null`, the all-zero `Var` that stands for external `nil`. Generated call adapters return it for a `void` target.

Source: `lib/var/encoding.x:250`

Var.tag_bottom

```
meta native unsigned long Var.tag_bottom(Symbol tag)
```

Returns the bottom encoding field of built-in `tag`.

Source: `lib/var/encoding.x:70`

Var.tag_top

```
meta native unsigned long Var.tag_top(Symbol tag)
```

Returns the top encoding field of built-in `tag`.

Source: `lib/var/encoding.x:62`

Advanced and interop API

Function	Summary
`Var.box_long`	Boxes a `long` into a scope-owned `<long>` value.
`Var.box_long_double`	Boxes a `long double` into a scope-owned `<ldouble>` value.
`Var.box_long_long`	Boxes a native signed long-long value without losing precision.
`Var.box_ulong`	Boxes an `unsigned long` into a scope-owned `<ulong>` value.
`Var.box_ulong_long`	Boxes a native unsigned long-long value without losing precision.
`Var.encoding_valid`	Reports whether `value` has a valid structural `Var` encoding.
`Var.floating`	Returns `v`'s payload as a `double` when its tag is floating, or 0.0.
`Var.integer`	Returns `v`'s payload as a `long` when its tag is integral, or 0.
`Var.integer_compare`	Orders the integer payloads of `a` and `b`, returning −1, 0, or 1.
`Var.integer_floating_compare`	Orders an integer-kinded `integer` against a floating `floating`.
`Var.kind`	Returns the `Symbol` naming the coarse category of `v`'s payload.
`Var.known_tag`	Reports whether `tag` has a built-in encoding or declared custom class.
`Var.long_double_value`	Returns the payload of an `<ldouble>` box, or 0.0 if `v` has another tag.
`Var.long_long_value`	Returns an `<llong>` box's signed payload, or 0 for another tag.
`Var.long_value`	Returns the payload of a `<long>` box, or 0 if `v` has a different tag.
`Var.parse`	Parses `str` as source text of kind `kind` and returns the boxed value.
`Var.pointer`	Returns the raw address stored in `v`, or NULL if it holds no address.
`Var.register_object_tag`	Declares custom boxed-object `tag` and returns 0.
`Var.tag`	Returns the `Symbol` naming the exact family of `v`'s payload.
`Var.ulong_long_value`	Returns a `<ullong>` box's unsigned payload, or 0 for another tag.
`Var.ulong_value`	Returns the payload of a `<ulong>` box, or 0 if `v` has a different tag.
`Var.wide_compare`	Orders two wide boxes carrying the same tag, returning −1, 0, or 1.
`Var.wide_equal`	Reports content equality for supported wide scalar boxes.
`Var.wide_hash`	Returns a supported wide scalar box's content hash, or 0 otherwise.

`Var`

Var.box_long

```
Var Var.box_long(long value)
```

Boxes a `long` into a scope-owned `<long>` value. A `Var` is eight bytes, and the tag consumes some of them. The five widest native families, `long`, `unsigned long`, `long long`, `unsigned long long`, and `long double`, cannot be stored inline. Boxing one allocates a small immutable box in the active scope, and the result lives for that scope's lifetime like any other scope allocation.

Two boxes of the same number are equal and identical: == compares the values, and === compares the type and the value bits. `Var.same` compares the 64 `Var` bits, and therefore the box addresses.

```
printf("equal=%d identical=%d same=%d tag=%s\n", five == also,
       five === also, five.same(also), five.tag().str());
```

Raises: `<alloc-fail>` when the box cannot be allocated and `<bad-enc>` when its address cannot be represented in a `Var`. Before `Error` initialization, allocation failure uses the raw fatal floor.

Source: `lib/var/numbers.x:66`

Var.box_long_double

```
Var Var.box_long_double(long double value)
```

Boxes a `long double` into a scope-owned `<ldouble>` value. This is the only family that keeps a full `long double` payload. The tag names that C family; it does not promise a bit width. `Var.floating` narrows the value to `double`; `Var.long_double_value` preserves the extra precision.

Raises: `<alloc-fail>` when the box cannot be allocated and `<bad-enc>` when its address cannot be represented in a `Var`. Before `Error` initialization, allocation failure uses the raw fatal floor.

Source: `lib/var/numbers.x:119`

Var.box_long_long

```
Var Var.box_long_long(long long value)
```

Boxes a native signed long-long value without losing precision.

Raises: `<alloc-fail>` when the box cannot be allocated and `<bad-enc>` when its address cannot be represented in a `Var`. Before `Error` initialization, allocation failure uses the raw fatal floor.

Source: `lib/var/numbers.x:93`

Var.box_ulong

```
Var Var.box_ulong(unsigned long value)
```

Boxes an `unsigned long` into a scope-owned `<ulong>` value. `<ulong>` is a distinct family from `<long>`. A box made here is never `Var.equal` or identical to one made by `Var.box_long`, even when both hold the same bit pattern, because `Var.wide_equal` requires matching tags. == converts the two as C does. Read it back with `Var.ulong_value`, the one reader that returns the payload unsigned.

Raises: `<alloc-fail>` when the box cannot be allocated and `<bad-enc>` when its address cannot be represented in a `Var`. Before `Error` initialization, allocation failure uses the raw fatal floor.

Source: `lib/var/numbers.x:82`

Var.box_ulong_long

```
Var Var.box_ulong_long(unsigned long long value)
```

Boxes a native unsigned long-long value without losing precision.

Raises: `<alloc-fail>` when the box cannot be allocated and `<bad-enc>` when its address cannot be represented in a `Var`. Before `Error` initialization, allocation failure uses the raw fatal floor.

Source: `lib/var/numbers.x:104`

Var.encoding_valid

```
int Var.encoding_valid(Var value)
```

Reports whether `value` has a valid structural `Var` encoding. An address-bearing encoding must still refer to live storage of the right type; wide encodings in particular require a readable `Scope`-owned box.

Source: `lib/var/encoding.x:264`

Var.floating

```
meta native double Var.floating(Var v)
```

Returns `v`'s payload as a `double` when its tag is floating, or 0.0. Handles `<f32>`, `<f64>`, and `<ldouble>`, and reconstructs NaN, `+Inf`, and `-Inf` from their discrete tags. An `<ldouble>` payload is narrowed to `double`; call `Var.long_double_value` to keep the full precision.

Like the other raw payload readers this one is silent. An integer, a `String`, or `void` reads as 0.0, and nothing distinguishes that from a stored zero. When the tag is not already known, convert with `Var.convert` in `lib/varconvert.x`, which raises a conversion failure; the scalar-named readers such as `Var.double` do this for you.

An unhandled tag yields 0.0.

Source: lib/var/numbers.x:180

Var.integer

```
meta native long Var.integer(Var v)
```

Returns v's payload as a long when its tag is integral, or 0. This reads the payload; it does not convert. It decodes every integer family, from <u8> through <ullong>, including the scope-owned boxes, and returns 0 for a tag it does not handle. A double reads as 0, and so does a String; nothing reports the mismatch. A <ullong> or <llong> payload is truncated to long, and a <ulong> above LONG_MAX comes back negative. Var.ulong_value preserves the unsigned payload. A Symbol reads as its numeric Symbol value.

When the tag might not be what you expect, convert instead of reading. Var.convert in lib/varconvert.x raises a conversion failure. Assigning a Var to an int or a long, and the scalar-named readers such as Var.int, go through Var.convert too.

```
printf("small=%ld text=%ld\n", small.integer(), text.integer());
```

An unhandled tag yields 0.

Source: lib/var/numbers.x:224

Var.integer_compare

```
int Var.integer_compare(Var a, Var b)
```

Orders the integer payloads of a and b, returning −1, 0, or 1. Each value is decomposed into a sign and an unsigned magnitude first, so the whole integer range orders correctly, including a <ullong> above LONG_MAX against a negative <long>. Subtracting in a fixed-width integer type could overflow.

Both arguments are assumed to be integer-kinded. Another value is decoded by Var.integer, which reads it as 0. Confirm with Var.is_integer when the kinds are not known.

Source: lib/var/ordering.x:19

Var.integer_floating_compare

```
int Var.integer_floating_compare(Var integer, Var floating)
```

Orders an integer-kinded integer against a floating floating. Returns −1, 0, or 1 for less, equal, and greater. The integer is never converted to floating point, so a large <ullong> and a nearby double order by their true values. A floating value with a fractional part is never equal to an integer.

+Inf is greater than every integer and -Inf is less than every integer. NaN is not ordered and reports −1. Test the tag for <nan> first if that distinction matters.

Source: lib/var/ordering.x:60

Var.kind

```
meta native Symbol Var.kind(Var v)
```

Returns the Symbol naming the coarse category of v's payload. The categories are <integer>, <floating>, <symbol>, <object>, <pointer>, <reference>, and <void>, and every tag belongs to exactly one. Use kind when a group of families is treated alike. All twelve integer widths answer <integer>, and every builtin class handle such as String, List, Map, and File answers <object>.

<pointer> means a raw C pointer such as <u8*>, while <reference> means a pointer to an object handle such as <string*>. NaN and the infinities are <floating>. Only void is <void>.

Source: lib/var/encoding.x:178

Var.known_tag

```
int Var.known_tag(Symbol tag)
```

Reports whether `tag` has a built-in encoding or declared custom class.

Source: `lib/var/var.x:62`

Var.long_double_value

```
meta native long double Var.long_double_value(Var v)
```

Returns the payload of an `<ldouble>` box, or 0.0 if `v` has another tag. This is the only reader that preserves a `long double`. Every other floating tag, `<f64>` included, yields 0.0 here instead of being widened. `Var.floating` is the general floating reader.

A tag mismatch yields 0.0.

Source: `lib/var/numbers.x:306`

Var.long_long_value

```
meta native long long Var.long_long_value(Var v)
```

Returns an `<llong>` box's signed payload, or 0 for another tag.

Source: `lib/var/numbers.x:292`

Var.long_value

```
meta native long Var.long_value(Var v)
```

Returns the payload of a `<long>` box, or 0 if `v` has a different tag. The test is on the tag, so a `<ulong>`, `<llong>`, or `<i32>` value reads as 0 with no complaint. `Var.integer` accepts any integer family; use this one when the tag is known and the payload must survive intact.

A tag mismatch yields 0.

Source: `lib/var/numbers.x:276`

Var.parse

```
meta native Var Var.parse(String str, Symbol kind)
```

Parses `str` as source text of kind `kind` and returns the boxed value. The kinds understood are `<int>`, `<float>`, `<double>`, `<string>`, `<symbol>`, and `<char>`. For `<string>`, matching `%"..."` or `"..."` delimiters are removed; unquoted input is also accepted, and either form is unescaped. `<char>` expects `'a'` complete with its quotes and produces an `<i32>`. Both `<float>` and `<double>` produce an `<f64>`; there is no path here to `<f32>`.

Integer and floating kinds require the whole input to parse, apart from surrounding whitespace. Malformed input, trailing text, a value outside the native reader's range, or an `<int>` outside `int` range returns `void`, so a parsed numeric zero remains distinguishable from failure.

`<symbol>` accepts the leading atom or `<...>` `Symbol` literal and ignores trailing text; no recognized leading `Symbol` produces the zero `Symbol`. A kind this function does not handle returns `void`, and a bad character literal gives `<i32>` −1.

```
printf("%s=%s %s=%s refused=%d\n", count.tag().str(), count,
       broken.tag().str(), broken, refused is void);
```

Raises: `<alloc-fail>` while constructing `String` or quoted-`Symbol` output.

Source: `lib/var/construction.x:205`

Var.pointer

```
void *Var.pointer(Var v)
```

Returns the raw address stored in v, or NULL if it holds no address. Every pointer, reference, and object family shares one decoder: the payload is masked free of the subtype bits its family reserves, so the result is the stored low-48-bit address for a <u8*>, a <string> handle, and a registered custom object. The returned pointer is borrowed; this operation does not retain it or change its lifetime. A value that is not address-shaped, such as an integer, a double, a Symbol, a wide box, or void, reads as NULL, and nothing distinguishes that from a stored null pointer.

Nothing here proves that an accepted address is live or came from the right constructor; that remains the typed API's precondition. This raw decoder also accepts some reserved pointer-shaped bit patterns. Validate external bits with Var.encoding_valid, then confirm the family with Var.tag or Var.is before trusting the result.

Source: lib/var/classes.x:181

Var.register_object_tag

```
int Var.register_object_tag(Symbol tag)
```

Declares custom boxed-object tag and returns 0. Declaring spends no Var row; the first box of a value assigns one. Declaring a tag again returns 0; NULL or a built-in tag returns −1 without changing the registry. Declaration must finish before the first successful Thread.start; afterward it raises <bad-state>. Native registry-mutex failure aborts.

Source: lib/var/classes.x:207

Var.tag

```
meta native Symbol Var.tag(Var v)
```

Returns the Symbol naming the exact family of v's payload. The tag names one family: <i32>, <string>, <f64>, <symbol>, or a registered custom object tag. Built-in families have one row in the encoding table in this file. Use Var.kind for a group of families and Var.is to test one family.

Raw Null, the all-zero Var written (Var) { .u64 = 0 }, is a native null pointer. It decodes as <p48>; there is no dedicated null family. Test it with Var.is_null. An encoding matching no reserved row decodes as <f64>, since the shifted double range covers everything left over.

```
printf("%s %s\n", raw.tag().str(), void.tag().str());
```

Source: lib/var/encoding.x:151

Var.ulong_long_value

```
meta native unsigned long long Var.ulong_long_value(Var v)
```

Returns a <ullong> box's unsigned payload, or 0 for another tag.

Source: lib/var/numbers.x:296

Var.ulong_value

```
meta native unsigned long Var.ulong_value(Var v)
```

Returns the payload of a <ulong> box, or 0 if v has a different tag. Prefer this to Var.integer for a <ulong>: the general reader casts the payload to long, which reinterprets anything above LONG_MAX as negative, while this reader returns the unsigned value. A tag that is not <ulong> yields 0 silently, so test with Var.is when the tag is in doubt.

A tag mismatch yields 0.

Source: lib/var/numbers.x:288

Var.wide_compare

```
int Var.wide_compare(Var a, Var b)
```

Orders two wide boxes carrying the same tag, returning −1, 0, or 1. Integer families delegate to `Var.integer_compare`. An `<ldouble>` pair compares as `long double` and falls back to a byte comparison when neither operand is less than the other, so distinct NaN representations still order deterministically.

0 means equal, and it is also the answer for mismatched tags and for an argument that is not a wide box. Check the tags first, or use `Var.compare`, the runtime's full ordering.

Source: `lib/var/numbers.x:377`

Var.wide_equal

```
int Var.wide_equal(Var a, Var b)
```

Reports content equality for supported wide scalar boxes. Boxed `<long>`, `<ulong>`, `<llong>`, `<ullong>`, and `<ldouble>` values are separate allocations, so bit identity says nothing about their contents. This is the payload comparison that `Var.equal` and `===` use for those tags. An `<ldouble>` pair is compared byte for byte, so two NaNs sharing one representation compare equal here.

A 0 result means "not equal as wide boxes". It also covers an argument that is not a wide box and a pair whose tags differ. For a general equality test use `Var.equal`, which covers every family, or `==`, which compares two numbers as C does.

Source: `lib/var/numbers.x:348`

Var.wide_hash

```
unsigned Var.wide_hash(Var v)
```

Returns a supported wide scalar box's content hash, or 0 otherwise.

Source: `lib/var/numbers.x:316`

Runtime-internal callables

These callables connect runtime translation units. They are documented for source readers but are not supported as user API.

Function	Summary
`Var.clone_wide`	Clones the wide numeric `value` into a new box in the active `Scope`.
`Var.custom_descriptor_index`	Returns a boxed custom object's row, or `-1` for another value.
`Var.move_wide_to`	Moves a wide numeric box into the destination `Scope` held by `scope`.
`Var.wide_owner`	Returns the `Scope` owning a live wide numeric box, or NULL otherwise.

`Var`

Var.clone_wide

```
meta native Var Var.clone_wide(Var value)
```

Clones the wide numeric `value` into a new box in the active `Scope`. The clone is `==` and `===` to `value` but has its own box, so `Var.same` tells the two apart. Returns `void` when `value` is not wide.

Raises: `<alloc-fail>` when the clone cannot be allocated, or `<bad-enc>` if its address cannot be represented in a `Var`.

Source: `lib/var/numbers.x:131`

Var.custom_descriptor_index

```
int Var.custom_descriptor_index(Var value)
```

Returns a boxed custom object's row, or `-1` for another value. Rows below 30 belong to one class each; row 30 holds every overflow heap class and row 31 every overflow record.

Source: `lib/var/encoding.x:283`

Var.move_wide_to

```
Self Var.move_wide_to(Self value, Scope *scope)
```

Moves a wide numeric box into the destination `Scope` held by `scope`. The box is not copied and `value` keeps its identity. If the destination slot is NULL, a new `Scope` is created there. Returns `value` unchanged for another family. For a wide value, raises `<bad-arg>` when `scope` is NULL, or `<alloc-fail>` when a new destination `Scope` cannot be allocated. Ownership is unchanged on failure.

Source: `lib/var/numbers.x:155`

Var.wide_owner

```
Scope Var.wide_owner(Var v)
```

Returns the `Scope` owning a live wide numeric box, or NULL otherwise.

Source: `lib/var/numbers.x:162`

Public types

Type	Kind	Summary
TagId	enum	Names the built-in `Var` tags in `var-tags.x` ledger order, which is the order of `x2c_var_taginfo`; `lib/var-ledger.x` checks the two agree.
VarDecodeGroup	struct	Resolves one encoding top: the selector is the middle field when `by_middle` is set and the bottom field under `mask` otherwise, and `ids` maps it to a `TagId`, or `_invalid_` where no tag has it.
VarTagInfo	struct	Describes one built-in `Var` tag's kind and encoding fields.

TagId

```
typedef enum TagId { _invalid_ = -1, _u8_, _i8_, _u16_, _i16_, _u32_, _i32_, _f32_, _u48_, _i48_,
_p48_, _f64_, _long_, _ulong_, _llong_, _ullong_, _ldouble_, _u8_p_, _i8_p_, _u16_p_, _i16_p_,
_u32_p_, _i32_p_, _f32_p_, _ulong_p_, _long_p_, _f64_p_, _ullong_p_, _llong_p_, _ldouble_p_,
_p48_p_, _u8_pp_, _i8_pp_, _u16_pp_, _i16_pp_, _u32_pp_, _i32_pp_, _f32_pp_, _ulong_pp_, _long_pp_,
_f64_pp_, _ullong_pp_, _llong_pp_, _ldouble_pp_, _array_, _block_, _buffer_, _bytes_, _context_,
_error_, _file_, _func_, _iter_, _lambda_, _list_, _logger_, _map_, _mutex_, _pipe_, _proc_,
_regexp_, _rope_, _scope_, _slice_, _socket_, _stream_, _string_, _symbol_, _tensor_, _thread_,
_token_, _var_, _array_p_, _block_p_, _buffer_p_, _bytes_p_, _context_p_, _error_p_, _file_p_,
_func_p_, _iter_p_, _lambda_p_, _list_p_, _logger_p_, _map_p_, _mutex_p_, _pipe_p_, _proc_p_,
_regexp_p_, _rope_p_, _scope_p_, _slice_p_, _socket_p_, _stream_p_, _string_p_, _symbol_p_,
_tensor_p_, _thread_p_, _token_p_, _var_p_, _nan_, _neginf_, _posinf_, _void_, _tag_count_ } TagId
```

Names the built-in `Var` tags in `var-tags.x` ledger order, which is the order of `x2c_var_taginfo`; `lib/var-ledger.x` checks the two agree.

Source: `lib/var/var.x:28`

VarDecodeGroup

```
typedef struct VarDecodeGroup { unsigned char mask, by_middle; signed char ids[8]; } VarDecodeGroup
```

Resolves one encoding top: the selector is the middle field when `by_middle` is set and the bottom field under `mask` otherwise, and `ids` maps it to a `TagId`, or `_invalid_` where no tag has it.

Source: `lib/var/var.x:56`

VarTagInfo

```
typedef struct VarTagInfo { Symbol tag, kind; unsigned long top, middle, bottom; } VarTagInfo
```

Describes one built-in `Var` tag's kind and encoding fields.

Source: `lib/var/var.x:48`

Design notes

Var owns the one-word encoding of dynamic values. Doubles keep their IEEE 754 bits, and the special-value ranges carry tagged payloads, native pointers, `Symbol`s, and boxes for wide scalars and custom classes. A `Var` preserves its runtime kind, and `void` is excluded from value-bearing protocols.

Tests and examples

`make verify` (`unittest/test-var.x`) and `make examples` (`type-conversion-matrix`).

`lib/varconvert.x`

`Var` numeric conversion policy.

Primary API

Function	Summary
`Var.convert`	Converts `value` to a numeric `target`, or returns exact-tag identity.
`X2CVarNumeric.f32`	Converts a decoded numeric value to native `float` without boxing.
`X2CVarNumeric.f64`	Converts a decoded numeric value to native `double` without boxing.
`X2CVarNumeric.ldouble`	Converts a decoded numeric value to native `long double` without boxing.

`Var`

Var.convert

```
meta native Var Var.convert(Var value, Symbol target)
```

Converts `value` to a numeric `target`, or returns exact-tag identity. All fifteen numeric families cross through the rules in this module. Integer narrowing keeps low bits, floating-to-integer truncates toward zero with a range check, and floating results follow host conversion. Identity returns the original `Var` and preserves its ownership; a newly boxed wide numeric result belongs to the active `Scope`. Converting a discrete `<nan>`, `<-inf>`, or `<+inf>` value to `<f64>` also returns the original `Var`, preserving its discrete tag.

Raises: `<bad-enc>` for invalid `Var` bits, `<void-op>` for `void`, `<bad-target>` for an unsupported target, `<no-convert>` for a nonnumeric source, `<conv-range>` when the result does not fit, or `<alloc-fail>` while boxing a wide result. A nonnumeric source is rejected before decoding, with the decoder's `<bad-types>` detail nested under `<no-convert>`.

Source: `lib/varconvert.x:149`

`X2CVarNumeric`

X2CVarNumeric.f32

```
float X2CVarNumeric.f32(X2CVarNumeric &value)
```

Converts a decoded numeric value to native `float` without boxing. Uses the target host cast, preserving its rounding behavior.

Source: `lib/varconvert.x:353`

X2CVarNumeric.f64

```
double X2CVarNumeric.f64(X2CVarNumeric &value)
```

Converts a decoded numeric value to native `double` without boxing. Uses the target host cast, preserving its rounding behavior.

Source: `lib/varconvert.x:362`

X2CVarNumeric.ldouble

```
long double X2CVarNumeric.ldouble(X2CVarNumeric &value)
```

Converts a decoded numeric value to native `long double` without boxing. Uses the target host cast, preserving its rounding behavior.

Source: `lib/varconvert.x:371`

Advanced and interop API

Function	Summary
`Var.array`	Extracts the `Array` payload of `value`.
`Var.as_iter`	Extracts the `Iter` payload of `value`.
`Var.block`	Extracts the `Block` payload of `value`.
`Var.buffer`	Extracts the `Buffer` payload of `value`.
`Var.bytes`	Extracts the `Bytes` payload of `value`.
`Var.file`	Extracts the `File` payload of `value`.
`Var.integer_box`	Boxes the low target-width bits of `raw` using integer `target`.
`Var.integer_tag`	Returns the integer tag selected by `rank` and signedness.
`Var.list`	Extracts the `List` payload of `value`.
`Var.map`	Extracts the `Map` payload of `value`.
`Var.numeric_decode`	Decodes a numeric `value` into caller-owned `out` storage.
`Var.numeric_info`	Writes numeric-family metadata for `tag` and returns nonzero.
`Var.signed_from_bits`	Interprets the low `bits` of `raw` as a two's-complement signed value.
`Var.string`	Extracts the `String` payload of `value`.
`Var.width_mask`	Returns a mask containing the low `bits` bits.

`Var`

Var.array

```
Array Var.array(Var value)
```

Extracts the `Array` payload of `value`.

Raises: `<no-convert>` for any other tag. An implicit conversion from `Var` to `Array` uses `Var.array_payload`, which reads the payload or NULL without raising.

Source: `lib/varconvert.x:90`

Var.as_iter

```
Iter Var.as_iter(Var value)
```

Extracts the `Iter` payload of `value`.

Raises: `<no-convert>` for any other tag. An implicit conversion from `Var` to `Iter` uses `Var.iter_payload`, which reads the payload or NULL without raising.

Source: `lib/varconvert.x:95`

Var.block

```
Block Var.block(Var value)
```

Extracts the `Block` payload of `value`.

Raises: `<no-convert>` for any other tag. An implicit conversion from `Var` to `Block` uses `Var.block_payload`, which reads the payload or NULL without raising.

Source: `lib/varconvert.x:91`

Var.buffer

```
Buffer Var.buffer(Var value)
```

Extracts the `Buffer` payload of `value`.

Raises: `<no-convert>` for any other tag. An implicit conversion from `Var` to `Buffer` uses `Var.buffer_payload`, which reads the payload or NULL without raising.

Source: `lib/varconvert.x:92`

Var.bytes

```
Bytes Var.bytes(Var value)
```

Extracts the `Bytes` payload of `value`.

Raises: `<no-convert>` for any other tag. An implicit conversion from `Var` to `Bytes` uses `Var.bytes_payload`, which reads the payload or NULL without raising.

Source: `lib/varconvert.x:93`

Var.file

```
File Var.file(Var value)
```

Extracts the `File` payload of `value`.

Raises: `<no-convert>` for any other tag. An implicit conversion from `Var` to `File` uses `Var.file_payload`, which reads the payload or NULL without raising.

Source: `lib/varconvert.x:94`

Var.integer_box

```
Var Var.integer_box(Symbol target, unsigned long long raw)
```

Boxes the low target-width bits of `raw` using integer `target`. Signed targets interpret those bits as two's-complement. Immediate targets return self-contained values; wide targets allocate their boxes in the active `Scope`.

Raises: `<bad-target>` for a noninteger target, or `<alloc-fail>` or `<bad-enc>` while boxing a wide result.

Source: `lib/varconvert.x:278`

Var.integer_tag

```
Symbol Var.integer_tag(int rank, int unsigned_value)
```

Returns the integer tag selected by `rank` and signedness. `rank` must be positive. Ranks one through three map to `<i32>` or `<u32>`; callers applying integer promotion must first select its resulting signedness. Ranks four through six select the 48-bit, `long`, and `long long` families. A rank above six returns the null `Symbol`.

Source: `lib/varconvert.x:307`

Var.list

```
List Var.list(Var value)
```

Extracts the `List` payload of `value`.

Raises: `<no-convert>` for any other tag. An implicit conversion from `Var` to `List` uses `Var.list_payload`, which reads the payload or NULL without raising.

Source: `lib/varconvert.x:96`

Var.map

```
Map Var.map(Var value)
```

Extracts the `Map` payload of `value`.

Raises: `<no-convert>` for any other tag. An implicit conversion from `Var` to `Map` uses `Var.map_payload`, which reads the payload or NULL without raising.

Source: `lib/varconvert.x:97`

Var.numeric_decode

```
void Var.numeric_decode(Var value, X2CVarNumeric &?out)
```

Decodes a numeric `value` into caller-owned `out` storage. The result retains no pointer into `value`; integer and floating payloads use the members described by `X2CVarNumeric`.

Raises: `<bad-arg>` for a null output, `<bad-enc>` for invalid `Var` bits, `<void-op>` for `void`, or `<bad-types>` for a nonnumeric tag. These failures leave `out` unchanged.

Source: `lib/varconvert.x:195`

Var.numeric_info

```
int Var.numeric_info(Symbol tag, X2CVarNumericInfo &?out)
```

Writes numeric-family metadata for `tag` and returns nonzero. The special `<nan>`, `<-inf>`, and `<+inf>` tags report the `<f64>` family. A null `out` or nonnumeric tag returns zero and leaves storage untouched.

Source: `lib/varconvert.x:179`

Var.signed_from_bits

```
long long Var.signed_from_bits(unsigned long long raw, int bits)
```

Interprets the low `bits` of `raw` as a two's-complement signed value. `bits` must be between one and the width of `unsigned long long`.

Source: `lib/varconvert.x:326`

Var.string

```
String Var.string(Var value)
```

Extracts the `String` payload of `value`.

Raises: `<no-convert>` for any other tag. A `String` destination converts a `Var` implicitly through `Var.as_string` instead, and `Var.string_payload` reads the payload or NULL without raising.

Source: `lib/varconvert.x:72`

Var.width_mask

```
unsigned long long Var.width_mask(int bits)
```

Returns a mask containing the low `bits` bits. Zero yields zero and a width at least `unsigned long long` yields `ULLONG_MAX`; `bits` must not be negative.

Source: `lib/varconvert.x:319`

Public types

Type	Kind	Summary
X2CVarNumeric	struct	Holds a decoded numeric `Var` and its promotion metadata.
X2CVarNumericInfo	struct	Describes one numeric `Var` family without holding a value.

X2CVarNumeric

```
typedef struct X2CVarNumeric { Symbol tag, int floating, unsigned_value, bits, rank; unsigned long long raw; long double floating_value; } X2CVarNumeric
```

Holds a decoded numeric `Var` and its promotion metadata. Integer values use the low `bits` of `raw`, with signed values represented in two's-complement; floating values use `floating_value`. The record holds no storage and retains nothing from the source `Var`.

Source: `lib/varconvert.x:28`

X2CVarNumericInfo

```
typedef struct X2CVarNumericInfo { Symbol tag, int floating, unsigned_value, bits, rank; } X2CVarNumericInfo
```

Describes one numeric `Var` family without holding a value. `tag` is canonical, `floating` and `unsigned_value` classify it, `bits` is its native payload width, and `rank` orders promotion. Callers normally obtain a valid record from `Var.numeric_info`.

Source: `lib/varconvert.x:19`

Design notes

Converts among `Var`'s fifteen numeric families. Integer conversions keep the target width's low bits; floating-to-integer conversions truncate toward zero and require a representable result. A nonnumeric value converts only to its own tag.

Tests and examples

`make verify` (`unittest/test-var.x`).

lib/varops.x

Boxed `Var` operators, updates, and truthiness.

Primary API

Function	Summary
`x2c_var_equal`	Evaluates == on two `Var` operands.
`x2c_var_identical`	Evaluates === on two `Var` operands.
`x2c_var_order`	Orders two `Var` operands for <, <=, >, and >=.
`x2c_var_postfix_volatile`	Applies `Var.postfix` semantics to a volatile destination in generated code.
`x2c_var_update_volatile`	Applies `Var.update` semantics to a volatile destination in generated code.
`Var.add`	Adds dynamic values through numeric, `String`, or registered `add` behavior.
`Var.binary`	Applies a dynamic arithmetic, bitwise, comparison, or logical operator.
`Var.div`	Divides dynamic values through numeric or registered `div` behavior.
`Var.fallback_truth`	Returns built-in truthiness without consulting a registered descriptor.
`Var.matmul`	Multiplies matrices through a registered `matmul` behavior.
`Var.mod`	Computes dynamic remainder through integer or registered `mod` behavior.
`Var.mul`	Multiplies dynamic values through numeric or registered `mul` behavior.
`Var.neg`	Negates a dynamic value through registered `neg` or numeric subtraction.
`Var.postfix`	Applies dynamic postfix ++ or -- and returns the prior value.
`Var.sub`	Subtracts dynamic values through numeric or registered `sub` behavior.
`Var.truth`	Returns dynamic truthiness through registered dispatch or built-in rules.
`Var.update`	Applies a failure-atomic dynamic compound update and returns the new value.

Functions

x2c_var_equal

```
int x2c_var_equal(Var a, Var b)
```

Evaluates == on two Var operands. Two numbers compare as their native values do in C after the usual arithmetic conversions, so 3 == 3L and -1 == 4294967295u hold and a NaN equals nothing. Other values compare through Var.equal.

Raises: <bad-enc> when a numeric comparison meets invalid Var bits.

Source: lib/varops.x:475

x2c_var_identical

```
int x2c_var_identical(Var a, Var b)
```

Evaluates === on two Var operands. Two numbers are identical when they have the same tag and the same bits, so two boxes of one long are identical, NaN === NaN holds, and +0.0 === -0.0 does not. Other values are identical when their Var bits are, as Var.same reports.

Source: lib/varops.x:487

x2c_var_order

```
double x2c_var_order(Var a, Var b)
```

Orders two Var operands for <, <=, >, and >=. Two numbers order as their native values do in C after the usual arithmetic conversions. The result is −1, 0, or 1, or NaN when a NaN leaves the pair unordered, so each relation to zero is false. Other values return the order Var.compare gives.

Raises: <void-op> when either operand is void, and <bad-enc> when a numeric comparison meets invalid Var bits.

Source: lib/varops.x:497

x2c_var_postfix_volatile

```
Var x2c_var_postfix_volatile(volatile Var &?lhs, Symbol op)
```

Applies Var.postfix semantics to a volatile destination in generated code. Volatile preserves accesses across exception transfer, without providing thread synchronization.

Source: lib/varops.x:667

x2c_var_update_volatile

```
Var x2c_var_update_volatile(volatile Var &?lhs, Symbol op, Var rhs)
```

Applies Var.update semantics to a volatile destination in generated code. Volatile preserves accesses across exception transfer, without providing thread synchronization.

Source: lib/varops.x:623

Var

Var.add

```
meta native Var Var.add(Var lhs, Var rhs)
```

Adds dynamic values through numeric, String, or registered add behavior. Numeric promotion, failure, and result ownership follow Var.binary; String addition returns a canonical concatenation, and a protocol result keeps the ownership chosen by its callback.

Source: lib/varops.x:216

Var.binary

```
Var Var.binary(Var lhs, Symbol op, Var rhs)
```

Applies a dynamic arithmetic, bitwise, comparison, or logical operator. Integers use C-style promotion and wrap to the result type's width; shifts use the promoted left operand's type and sign-fill signed right shifts. Floating arithmetic uses the widest floating-point operand type. Comparisons and identity tests give the operators' results on Var operands, so two numbers compare as their native values do in C. The logical operators are eager in this direct API. Equality and identity are the only operations that accept void and return an <i32> predicate.

Immediate results are self-contained, canonical `String`s keep their pool lifetime, and newly boxed wide results belong to the active `Scope`.

Raises: `<bad-enc>`, `<void-op>`, or `<bad-op>` at the dynamic boundary; numeric operations may additionally raise `<bad-types>`, `<div-zero>`, or `<bad-shift>`, and `String` concatenation or wide boxing may raise `<size-limit>`, `<alloc-fail>`, or `<bad-enc>`. A selected protocol, truth, or comparison callback may raise its own cause.

Source: `lib/varops.x:172`

Var.div

```
meta native Var Var.div(Var lhs, Var rhs)
```

Divides dynamic values through numeric or registered `div` behavior. Numeric integer zero divisors raise; floating division uses host infinity and NaN behavior. Other promotion, failure, and ownership follow `Var.binary`.

Source: `lib/varops.x:245`

Var.fallback_truth

```
int Var.fallback_truth(Var value)
```

Returns built-in truthiness without consulting a registered descriptor. Numeric and `Symbol` zero and null pointer-bearing values are false; other supported built-ins are true. Container-specific truth comes from dispatch.

Raises: `<bad-enc>` for invalid `Var` bits, `<void-op>` for `void`, or `<bad-types>` when no truthiness rule exists.

Source: `lib/varops.x:585`

Var.matmul

```
Var Var.matmul(Var lhs, Var rhs)
```

Multiplies matrices through a registered `matmul` behavior. `@` has no numeric meaning, so numeric operands raise `<bad-op>`; a protocol result keeps the ownership chosen by its callback.

Source: `lib/varops.x:237`

Var.mod

```
meta native Var Var.mod(Var lhs, Var rhs)
```

Computes dynamic remainder through integer or registered `mod` behavior. Numeric operands use the common promoted integer type and reject a zero divisor; floating operands are not accepted. Other failure and ownership follow `Var.binary`.

Source: `lib/varops.x:253`

Var.mul

```
meta native Var Var.mul(Var lhs, Var rhs)
```

Multiplies dynamic values through numeric or registered `mul` behavior. Numeric promotion, failure, and result ownership follow `Var.binary`; a protocol result keeps the ownership chosen by its callback.

Source: `lib/varops.x:230`

Var.neg

```
meta native Var Var.neg(Var value)
```

Negates a dynamic value through registered `neg` or numeric subtraction. Without a selected protocol, this computes `0 - value` with ordinary `Var` promotion and wrapping, so a narrow integer promotes before negation.

Raises: `<bad-enc>` for invalid bits, `<void-op>` for `void`, `<no-member>` for an object without `neg`, or any cause from protocol or numeric subtraction.

Source: `lib/varops.x:282`

Var.postfix

```
Var Var.postfix(Var &?lhs, Symbol op)
```

Applies dynamic postfix `++` or `--` and returns the prior value. The update adds or subtracts an `<i32>` one through `Var.update`, preserving the destination tag. If a delegated protocol operation returns `void`, the function leaves the destination unchanged and returns `void`.

Raises: `<bad-arg>` for a null destination, `<bad-enc>` for invalid `Var` bits, `<void-op>` for `void`, `<bad-op>` for an operator other than `++` or `--`, or any cause from `Var.update`. These failures leave the stored value unchanged.

Source: `lib/varops.x:661`

Var.sub

```
meta native Var Var.sub(Var lhs, Var rhs)
```

Subtracts dynamic values through numeric or registered `sub` behavior. Numeric promotion, failure, and result ownership follow `Var.binary`; a protocol result keeps the ownership chosen by its callback.

Source: `lib/varops.x:223`

Var.truth

```
meta native int Var.truth(Var value)
```

Returns dynamic truthiness through registered dispatch or built-in rules. Numeric and `Symbol` zero and null unhandled pointer-bearing values are false; their nonzero or nonnull counterparts are true. A registered truth callback supplies its own result, including for a custom object, so it may return false independently of object state. The call does not retain `value` or inspect the contents of an iterator itself.

Raises: `<bad-enc>` for invalid `Var` bits, `<void-op>` for `void`, or `<bad-types>` when no truthiness rule exists, plus any cause raised by a selected descriptor callback.

Source: `lib/varops.x:573`

Var.update

```
Var Var.update(Var &?lhs, Symbol op, Var rhs)
```

Applies a failure-atomic dynamic compound update and returns the new value. The operation is limited to arithmetic, remainder, bitwise, and shift operators. The result is converted back to the destination's original tag and stored only after both computation and conversion complete; this does not provide thread synchronization. If a delegated protocol operation returns `void`, the function leaves the destination unchanged.

Raises: `<bad-arg>` for a null destination, or any cause from `Var.binary` and `Var.convert`. These failures leave the stored value unchanged.

Source: `lib/varops.x:616`

Advanced and interop API

Function	Summary
`x2c_var_update_f32`	Applies a dynamic compound `op` to a native `float` lvalue.
`x2c_var_update_f64`	Applies a dynamic compound `op` to a native `double` lvalue.
`x2c_var_update_i16`	Applies a dynamic compound `op` to a native `short` lvalue.
`x2c_var_update_i32`	Applies a dynamic compound `op` to a native `int` lvalue.
`x2c_var_update_i8`	Applies a dynamic compound `op` to a native `char` lvalue.
`x2c_var_update_long`	Applies a dynamic compound `op` to a native `long` lvalue.
`x2c_var_update_long_double`	Applies a dynamic compound `op` to a native `long double` lvalue.

Function	Summary
x2c_var_update_long_long	Applies a dynamic compound op to a native long long lvalue.
x2c_var_update_schar	Applies a dynamic compound op to a native signed char lvalue.
x2c_var_update_u16	Applies a dynamic compound op to a native ushort lvalue.
x2c_var_update_u32	Applies a dynamic compound op to a native uint lvalue.
x2c_var_update_u8	Applies a dynamic compound op to a native uchar lvalue.
x2c_var_update_ulong	Applies a dynamic compound op to a native ulong lvalue.
x2c_var_update_ulong_long	Applies a dynamic compound op to a native unsigned long long lvalue.

Functions

x2c_var_update_f32

```
float x2c_var_update_f32(volatile float *lhs, Symbol op, Var rhs)
```

Applies a dynamic compound op to a native float lvalue. The current value is boxed in its declared family, combined with rhs, converted back to that family, and stored only after all steps succeed. Returns the stored native value. This is failure-atomic but provides no thread synchronization despite accepting a volatile pointer.

Raises: <bad-arg> for a null lvalue, or any cause from Var.binary, Var.convert, or wide boxing. A transferring failure leaves the lvalue unchanged. If a delegated protocol operation returns void, the helper also leaves it unchanged and returns the native zero for float.

Source: lib/varops.x:124

x2c_var_update_f64

```
double x2c_var_update_f64(volatile double *lhs, Symbol op, Var rhs)
```

Applies a dynamic compound op to a native double lvalue. The current value is boxed in its declared family, combined with rhs, converted back to that family, and stored only after all steps succeed. Returns the stored native value. This is failure-atomic but provides no thread synchronization despite accepting a volatile pointer.

Raises: <bad-arg> for a null lvalue, or any cause from Var.binary, Var.convert, or wide boxing. A transferring failure leaves the lvalue unchanged. If a delegated protocol operation returns void, the helper also leaves it unchanged and returns the native zero for double.

Source: lib/varops.x:125

x2c_var_update_i16

```
short x2c_var_update_i16(volatile short *lhs, Symbol op, Var rhs)
```

Applies a dynamic compound op to a native short lvalue. The current value is boxed in its declared family, combined with rhs, converted back to that family, and stored only after all steps succeed. Returns the stored native value. This is failure-atomic but provides no thread synchronization despite accepting a volatile pointer.

Raises: <bad-arg> for a null lvalue, or any cause from Var.binary, Var.convert, or wide boxing. A transferring failure leaves the lvalue unchanged. If a delegated protocol operation returns void, the helper also leaves it unchanged and returns the native zero for short.

Source: lib/varops.x:116

x2c_var_update_i32

```
int x2c_var_update_i32(volatile int *lhs, Symbol op, Var rhs)
```

Applies a dynamic compound op to a native int lvalue. The current value is boxed in its declared family, combined with rhs, converted back to that family, and stored only after all steps succeed. Returns the stored native value. This is failure-atomic but provides no thread synchronization despite accepting a volatile pointer.

Raises: <bad-arg> for a null lvalue, or any cause from Var.binary, Var.convert, or wide boxing. A transferring failure leaves the lvalue unchanged. If a delegated protocol operation returns void, the helper also leaves it unchanged and returns the native zero for int.

Source: lib/varops.x:118

x2c_var_update_i8

```
char x2c_var_update_i8(volatile char *lhs, Symbol op, Var rhs)
```

Applies a dynamic compound `op` to a native `char` lvalue. The current value is boxed in its declared family, combined with `rhs`, converted back to that family, and stored only after all steps succeed. Returns the stored native value. This is failure-atomic but provides no thread synchronization despite accepting a volatile pointer.

Raises: `<bad-arg>` for a null lvalue, or any cause from `Var.binary`, `Var.convert`, or wide boxing. A transferring failure leaves the lvalue unchanged. If a delegated protocol operation returns `void`, the helper also leaves it unchanged and returns the native zero for `char`.

Source: `lib/varops.x:113`

x2c_var_update_long

```
long x2c_var_update_long(volatile long *lhs, Symbol op, Var rhs)
```

Applies a dynamic compound `op` to a native `long` lvalue. The current value is boxed in its declared family, combined with `rhs`, converted back to that family, and stored only after all steps succeed. Returns the stored native value. This is failure-atomic but provides no thread synchronization despite accepting a volatile pointer.

Raises: `<bad-arg>` for a null lvalue, or any cause from `Var.binary`, `Var.convert`, or wide boxing. A transferring failure leaves the lvalue unchanged. If a delegated protocol operation returns `void`, the helper also leaves it unchanged and returns the native zero for `long`.

Source: `lib/varops.x:120`

x2c_var_update_long_double

```
long double x2c_var_update_long_double(volatile long double *lhs, Symbol op, Var rhs)
```

Applies a dynamic compound `op` to a native `long double` lvalue. The current value is boxed in its declared family, combined with `rhs`, converted back to that family, and stored only after all steps succeed. Returns the stored native value. This is failure-atomic but provides no thread synchronization despite accepting a volatile pointer.

Raises: `<bad-arg>` for a null lvalue, or any cause from `Var.binary`, `Var.convert`, or wide boxing. A transferring failure leaves the lvalue unchanged. If a delegated protocol operation returns `void`, the helper also leaves it unchanged and returns the native zero for `long double`.

Source: `lib/varops.x:126`

x2c_var_update_long_long

```
long long x2c_var_update_long_long(volatile long long *lhs, Symbol op, Var rhs)
```

Applies a dynamic compound `op` to a native `long long` lvalue. The current value is boxed in its declared family, combined with `rhs`, converted back to that family, and stored only after all steps succeed. Returns the stored native value. This is failure-atomic but provides no thread synchronization despite accepting a volatile pointer.

Raises: `<bad-arg>` for a null lvalue, or any cause from `Var.binary`, `Var.convert`, or wide boxing. A transferring failure leaves the lvalue unchanged. If a delegated protocol operation returns `void`, the helper also leaves it unchanged and returns the native zero for `long long`.

Source: `lib/varops.x:122`

x2c_var_update_schar

```
signed char x2c_var_update_schar(volatile signed char *lhs, Symbol op, Var rhs)
```

Applies a dynamic compound `op` to a native `signed char` lvalue. The current value is boxed in its declared family, combined with `rhs`, converted back to that family, and stored only after all steps succeed. Returns the stored native value. This is failure-atomic but provides no thread synchronization despite accepting a volatile pointer.

Raises: `<bad-arg>` for a null lvalue, or any cause from `Var.binary`, `Var.convert`, or wide boxing. A transferring failure leaves the lvalue unchanged. If a delegated protocol operation returns `void`, the helper also leaves it unchanged and returns the native zero for `signed char`.

Source: `lib/varops.x:114`

x2c_var_update_u16

```
ushort x2c_var_update_u16(volatile ushort *lhs, Symbol op, Var rhs)
```

Applies a dynamic compound `op` to a native `ushort` lvalue. The current value is boxed in its declared family, combined with `rhs`, converted back to that family, and stored only after all steps succeed. Returns the stored native value. This is failure-atomic but provides no thread synchronization despite accepting a volatile pointer.

Raises: `<bad-arg>` for a null lvalue, or any cause from `Var.binary`, `Var.convert`, or wide boxing. A transferring failure leaves the lvalue unchanged. If a delegated protocol operation returns `void`, the helper also leaves it unchanged and returns the native zero for `ushort`.

Source: `lib/varops.x:117`

x2c_var_update_u32

```
uint x2c_var_update_u32(volatile uint *lhs, Symbol op, Var rhs)
```

Applies a dynamic compound `op` to a native `uint` lvalue. The current value is boxed in its declared family, combined with `rhs`, converted back to that family, and stored only after all steps succeed. Returns the stored native value. This is failure-atomic but provides no thread synchronization despite accepting a volatile pointer.

Raises: `<bad-arg>` for a null lvalue, or any cause from `Var.binary`, `Var.convert`, or wide boxing. A transferring failure leaves the lvalue unchanged. If a delegated protocol operation returns `void`, the helper also leaves it unchanged and returns the native zero for `uint`.

Source: `lib/varops.x:119`

x2c_var_update_u8

```
uchar x2c_var_update_u8(volatile uchar *lhs, Symbol op, Var rhs)
```

Applies a dynamic compound `op` to a native `uchar` lvalue. The current value is boxed in its declared family, combined with `rhs`, converted back to that family, and stored only after all steps succeed. Returns the stored native value. This is failure-atomic but provides no thread synchronization despite accepting a volatile pointer.

Raises: `<bad-arg>` for a null lvalue, or any cause from `Var.binary`, `Var.convert`, or wide boxing. A transferring failure leaves the lvalue unchanged. If a delegated protocol operation returns `void`, the helper also leaves it unchanged and returns the native zero for `uchar`.

Source: `lib/varops.x:115`

x2c_var_update_ulong

```
ulong x2c_var_update_ulong(volatile ulong *lhs, Symbol op, Var rhs)
```

Applies a dynamic compound `op` to a native `ulong` lvalue. The current value is boxed in its declared family, combined with `rhs`, converted back to that family, and stored only after all steps succeed. Returns the stored native value. This is failure-atomic but provides no thread synchronization despite accepting a volatile pointer.

Raises: `<bad-arg>` for a null lvalue, or any cause from `Var.binary`, `Var.convert`, or wide boxing. A transferring failure leaves the lvalue unchanged. If a delegated protocol operation returns `void`, the helper also leaves it unchanged and returns the native zero for `ulong`.

Source: `lib/varops.x:121`

x2c_var_update_ulong_long

```
unsigned long long x2c_var_update_ulong_long(volatile unsigned long long *lhs, Symbol op, Var rhs)
```

Applies a dynamic compound `op` to a native `unsigned long long` lvalue. The current value is boxed in its declared family, combined with `rhs`, converted back to that family, and stored only after all steps succeed. Returns the stored native value. This is failure-atomic but provides no thread synchronization despite accepting a volatile pointer.

Raises: `<bad-arg>` for a null lvalue, or any cause from `Var.binary`, `Var.convert`, or wide boxing. A transferring failure leaves the lvalue unchanged. If a delegated protocol operation returns `void`, the helper also leaves it unchanged and returns the native zero for `unsigned long long`.

Source: `lib/varops.x:123`

Design notes

`Var` arithmetic applies C-style numeric promotion to runtime tags, wraps integer results to the selected type's width, and delegates eligible object operations through registered protocols. Comparisons of two numbers use the same promotion. Compound updates compute and convert completely before changing their destination.

Tests and examples

`make verify` (`unittest/test-varops.x`).

Advanced Topics

The first two chapters apply x2c to one specialized task each, with the setup each one needs. The third states the invariant behind the region warnings:

- Automatic Differentiation uses the optional autodiff package and its compile-time macros to calculate derivatives.
- Training and Inference with torch uses the optional Torch package and libtorch for machine learning.
- The Region Model states the invariant behind the region warnings and the limits of the check.

The autodiff and Torch packages are separate from the standard library. For the package mechanism itself, see Packages.

Automatic Differentiation

`autodiff-macros.x` differentiates ordinary `double` code four ways:

- Dual numbers carry a derivative through operators at runtime.
- `$ad.forward()` generates a tangent function beside a decorated function, at compile time, from the typed AST.
- `$ad.reverse()` generates a gradient function the same way, recording a tape during the forward sweep. `$ad.checkpoint(K)` computes the same gradient with a tape about `K` times smaller.
- `autodiff.x` records a runtime tape for code that the decorators cannot transform.

All four are ordinary x2c: a macro file, a runtime unit, and the existing protocol and decorator machinery. The compiler has no derivative support of its own. Both files belong to the optional `autodiff` package in `packages/autodiff`, which has no native dependency. `import "autodiff";` gives a unit the macros and the runtime tape, because the package's `autodiff.x` includes its macro module. A unit can also include that module directly by its path, as `#include "../src/autodiff-macros.x"` does in the package's fixtures; the listings below write the short name. The package's README lists its files and checks. For other specialized capabilities, see Advanced Topics.

Dual numbers

Declare a struct with `value` and `tangent` fields and let `$ad.dual` generate its arithmetic. The scalar type and the primitives it lifts are holes, so the same family nests:

```
#include "autodiff-macros.x"

typedef struct Dual { double value; double tangent; } Dual;
typedef struct Dual2 { Dual value; Dual tangent; } Dual2;

$ad.dual(Dual, dual, <dual>, double, sin, cos, exp, log, sqrt, tanh);
$ad.dual(Dual2, dual2, <dual2>, Dual,
         Dual.sin, Dual.cos, Dual.exp, Dual.log, Dual.sqrt, Dual.tanh);

static Dual2 cube(Dual2 x) => x * x * x;

int main(void) {
  Dual x = { 2.0, 1.0 };
  Dual y = x * x * x - 2.0 * x;
  Dual2 z = cube((Dual2) { { 2.0, 1.0 }, { 1.0, 0.0 } });
  return y.tangent == 10.0 && z.tangent.tangent == 12.0 ? 0 : 1;
}
```

The holes are the struct type, the converter name, a fresh `Var` tag, the scalar type, and the scalar spellings of `sin`, `cos`, `exp`, `log`, `sqrt`, and `tanh`. Declare every struct of the family before the first invocation. Each instantiation adopts `protocol Var`, so `+ - * /`, unary `-`, comparisons, and dotted calls such as `x.sin()`, `x.fabs()`, and `x.pow(y)` resolve to the generated methods. A `double` beside a dual operand converts through the generated converter, so `2.0 * x` and `x > 1.0` read as they would on scalars.

Each depth of nesting is a distinct C type. A derivative taken inside a function that is itself being differentiated must use the next type in the family; mixing a `Dual` into a `Dual2` expression is a type error. That is a weaker guarantee than tagged perturbations give a dynamically typed implementation: the types make an inner and

an outer perturbation distinct only when the author instantiates one type per level, and a nested derivative written with a single `Dual` type exhibits the classic perturbation confusion silently.

Forward mode by transformation

`$ad.forward()` decorates a file-scope function that returns `double` and emits `NAME_dot` beside it. Every `double` parameter `p` is followed by a tangent parameter `p_dot`, and the result is the directional derivative:

```
#include "autodiff-macros.x"

$ad.forward()
static double scale(double a, int k) => a * (double) k;

$ad.forward()
static double model(double x, double y, int n) {
  double s = 0.0;
  for (int i = 1; i <= n; i++) s += scale(x, i) / y;
  double t = x * y + 3.0;
  if (t > 1.0) t *= t;
  else t = -t;
  return sin(t) / x - 2.0 * y + s;
}

int main(void) {
  double df_dx = model_dot(1.0, 1.0, 2.0, 0.0, 3);
  double df_dy = model_dot(1.0, 0.0, 2.0, 1.0, 3);
  return df_dx > 22.9 && df_dx < 23.0 && df_dy > 6.4 && df_dy < 6.5 ? 0 : 1;
}
```

The transformation reads the typed AST. `double` parameters and plain `double` locals carry tangents; other `double` expressions are constants, and `int` control flow is copied unchanged. Supported statements are declarations, assignment including compound assignment and increments, `if`, `while`, `do`, `for`, `break`, `continue`, and `return`. Supported expressions are literals, identifiers, parentheses, casts, `+ - * /`, unary `-`, the conditional operator, the primitives listed below, and calls to functions decorated with `$ad.forward()` earlier in the same unit, which become calls to their `_dot` siblings. Anything else is a diagnostic at the invocation that names the statement or expression.

The generated function is plain scalar C. Zero and unit factors are folded at generation time, so `t_dot = x_dot * y + x * y_dot` is what appears in the output. A primitive term tests its tangent against zero before multiplying, so an infinite partial such as `sqrt` at 0 does not turn a zero tangent into NaN.

Reverse mode by transformation

`$ad.reverse()` emits `NAME_grad`. The original parameters are followed by one `double *p_grad` per `double` parameter; the result is the primal value, and each slot receives the partial derivative of that result. The unit must include `typed-array.x` because the generated function records a tape on an `ArrayDbl`:

```
#include "autodiff-macros.x"

$ad.reverse()
static double model(double x, double y, int n) {
  double s = 0.0;
  for (int i = 0; i < n; i++) {
    if (i == 1) continue;
    if (s > 40.0) break;
    s += x * x * (double) i;
  }
  double t = x * y + 3.0;
  if (t > 100.0) return t * s;
  return sin(t) / x - 2.0 * y + s;
}

int main(void) {
  double x_grad, y_grad;
  double value = model_grad(1.0, 2.0, 6, &x_grad, &y_grad);
  return fabs(value - 9.041076) < 1e-6 && fabs(x_grad - 29.526249) < 1e-6
      && fabs(y_grad + 1.716338) < 1e-6 ? 0 : 1;
}
```

The forward sweep runs the original statements. Every assignment first pushes the value it overwrites, an `if` pushes which branch ran, and each loop iteration ends by pushing an exit code: zero when the body completed, or the code of the `break` or `continue` that left it. A `return` pushes its own code and jumps to the reverse sweep. The reverse sweep pops in mirror order, restoring each overwritten value before accumulating the adjoints of the assignment that produced it, so every partial derivative is evaluated at the values the forward sweep saw. There is no renaming into single-assignment form; restoring values makes it unnecessary.

For every exit, the statements that ran before it are known at compile time, and the transformation generates the reverse of exactly that prefix. A loop's reverse pops the trip count and then, per iteration, the exit code that selects among those prefixes. Loops nest, `do` becomes a `while` with a first-iteration flag, and a `return` inside a loop reverses the partial iteration and then the completed ones. Every local is hoisted to the function head, so names must be unique within the function. `goto`, `switch`, and assignment inside a larger expression are diagnostics.

Calls to functions decorated with `$ad.reverse()` earlier in the unit call their `_grad` siblings, which recompute the callee's primal and return its partials. `$ad.both()` emits both siblings; two decorators cannot stack when each produces several items.

Checkpointing

A recorded loop stores every overwritten value for every iteration, so tape memory grows with the trip count. `$ad.checkpoint(K)` is `$ad.reverse()` with every loop run twice instead: the forward sweep runs the loop without recording and pushes a snapshot of the variables the loop assigns once every `K` iterations; the reverse sweep restores each block from its snapshot, replays it with recording, and reverses the replay. Tape memory is one block's tape plus one snapshot per block, so it still grows with the trip count, divided by `K`. `break` and `continue` replay exactly as before; a checkpointed loop cannot contain `return`.

```
#include "autodiff-macros.x"

$ad.checkpoint(64)
static double relax(double x, double y, int steps) {
  double s = x;
  for (int i = 0; i < steps; i++) s += 0.001 * (y - s) * cos(s * 0.01);
  return s;
}

int main(void) {
  double x_grad, y_grad;
  double value = relax_grad(1.0, 3.0, 100000, &x_grad, &y_grad);
  return fabs(value - 3.0) < 1e-9 && fabs(y_grad - 1.0) < 1e-9 ? 0 : 1;
}
```

Measured on one 2,000,000-step relaxation loop (the benchmark `packages/autodiff/benchmarks/autodiff-checkpoint.x`, one process per variant, Apple M-series, `-O2`):

Variant	Peak memory	Time
primal only	1.6 MB	0.033 s
`$ad.reverse()`	51.5 MB	0.053 s
`$ad.checkpoint(16)`	3.6 MB	0.068 s
`$ad.checkpoint(64)`	2.1 MB	0.065 s
`$ad.checkpoint(256)`	1.8 MB	0.066 s
`$ad.checkpoint(1024)`	1.7 MB	0.065 s

Full recording stores three doubles per iteration; checkpointing pays about a quarter more time for running the loop twice, and its memory is dominated by the snapshots at small `K` and by the block tape at large `K`.

This is two-level checkpointing with a fixed block size. The divide-and-conquer schedule of Siskind and Pearlmutter (2018), which needs no block size and achieves logarithmic growth, is not implemented; a program whose loops outgrow a fixed block can nest the decorated function in a caller that is itself decorated.

A worked example

`packages/autodiff/examples/autodiff-fit.x` fits the rate and capacity of a logistic growth model to observations. The loss integrates the model with 4,000 Euler steps and accumulates squared residuals at ten sample times inside the loop, so the gradient runs through a long loop with a branch in it. The example prints the gradient beside a central finite difference, then takes gradient-descent steps:

```
loss 4702.386283
d/drate     -15650.816053  finite difference -15650.816049
d/dcapacity -118.848124  finite difference -118.848118
step 50  loss   2.373136  rate 0.9140  capacity 49.1688
step 100  loss   0.014606  rate 0.9016  capacity 49.9274
...
```

The parameters recover the true values 0.9 and 50 to three digits; the remaining loss is the Euler discretization error of the model itself.

Primitives

Both transformations differentiate calls to these `<math.h>` functions, whose derivatives are closed forms in the same primitives:

Family	Functions
Trigonometric	`sin cos tan asin acos atan atan2`
Hyperbolic	`sinh cosh tanh asinh acosh atanh`
Exponential	`exp exp2 expm1 log log2 log10 log1p pow`
Roots and norms	`sqrt cbrt hypot`
Piecewise	`fabs fmin fmax`

`fabs`, `fmin`, and `fmax` differentiate as the branch that was taken; at a tie the derivative follows the first argument. The dual family provides the subset its holes name, plus `fabs` and a `pow` computed as `exp(b log a)`, which unlike C `pow` needs a positive base.

Runtime tape

When the shape of the computation depends on data in ways the decorators reject, import the package's runtime tape and record it:

```
import "autodiff" with AdTape, AdNode;
int main(void) {
  AdTape tape = AdTape.new();
  AdNode x = tape.input(1.5), y = tape.input(2.0);
  AdNode t = x * y, limit = tape.input(100.0);
  while (t < limit) t = t * t;
  AdNode r = t.log() - y.exp();
  tape.backward(r);
  return x.adjoint > 0.0 && y.adjoint < 0.0 ? 0 : 1;
}
```

Every operation records a closure that propagates its adjoint to its operands; `AdTape.backward` seeds the result and replays the tape in reverse, skipping callbacks whose adjoints are zero. An unused singular operation therefore does not contaminate the requested gradient. Repeating `backward` clears the previous adjoints before seeding the new result. Values box through `Var` and each operation allocates a node in the active `Scope`, so this is the slow path.

Use one tape for a computation, creating constants with that tape's `input` as well. Binary arithmetic operations reject nodes from different tapes with `bad-arg` before recording anything. `backward` rejects a result from another tape before clearing or seeding adjoints. Keep the tape and its nodes alive together; do not change a recorded node's `tape` field.

Choosing

Use dual numbers for a few derivatives of small functions, or for higher-order derivatives through nesting. Use `$ad.forward()` when the generated scalar C should be readable or when the function has few inputs. Use `$ad.reverse()` for gradients with many inputs, and `$ad.checkpoint(K)` when its loops run long. Use the tape when the decorators reject the code. For gradients over tensors rather than scalars, the torch package reaches libtorch's autograd through the same operator protocols.

Background

Forward-mode accumulation over a program's elementary operations goes back to Wengert (1964); Baydin et al. (2018) survey the field and its vocabulary. The transformation route follows the standard formulation of forward and reverse mode over a program's statements (Griewank and Walther, 2008): recording overwritten values rather than renaming into single-assignment form is the classic tape discipline, and the exit-code treatment of control flow makes the replay exact for `break`, `continue`, and early `return`. Two-level checkpointing is the simplest member of the family that Griewank (1992) made logarithmic and that Siskind and Pearlmutter (2018) freed from user annotation.

Perturbation confusion is the failure analyzed by Siskind and Pearlmutter (2005) and, for higher-order functions, by Manzyuk et al. (2019). The dual family here does not tag perturbations; it offers one distinct type per nesting level and relies on the author to use them, as described above. The runtime tape records closures that push adjoints to their operands, which is a Wengert tape with a closure in place of an opcode; it is in the spirit of, but not the same construction as, the backpropagators of Pearlmutter and Siskind (2008), which are returned functions composed without any tape. What that line of work adds beyond this chapter, first-class derivative operators applied to arbitrary closures with the overhead removed by program analysis, is a compiler feature rather than a macro and is not attempted here.

- R. E. Wengert. A simple automatic derivative evaluation program. *Communications of the ACM* 7(8), 1964.
- A. Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software* 1(1), 1992.
- J. M. Siskind and B. A. Pearlmutter. Perturbation confusion and referential transparency: correct functional implementation of forward-mode AD. *Implementation and Application of Functional Languages*, 2005.
- B. A. Pearlmutter and J. M. Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems* 30(2), 2008.
- A. Griewank and A. Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed. SIAM, 2008.
- A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *Journal of Machine Learning Research* 18, 2018.
- J. M. Siskind and B. A. Pearlmutter. Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software* 33(4-6), 2018.
- O. Manzyuk, B. A. Pearlmutter, A. A. Radul, D. R. Rush, and J. M. Siskind. Perturbation confusion in forward automatic differentiation of higher-order functions. *Journal of Functional Programming* 29, 2019.

Training and Inference with torch

For package setup and supported operations, see the Torch package README. This chapter belongs to Advanced Topics.

The `torch` package reaches PyTorch's C++ library, libtorch, from x2c. Tensors, autograd, layers, optimizers, schedules, checkpoints, TorchScript inference, and the MNIST reader are all libtorch's; the package supplies x2c types and methods over them. The entry unit is `packages/torch/src/torch.x`, the C ABI it compiles over is `src/torch-2.10.h`, and `src/torch-shim.cpp` implements that ABI. libtorch has no C API and no stable C++ ABI, so the shim is the package's only C++ and it is tied to exactly the pinned version, 2.10.0.

A program links the pinned libtorch shared libraries dynamically through an rpath into the prepared prefix, so it depends on that prefix at run time. `torch.compile` and TorchScript capture work on Python code and are not available here; x2c runs a TorchScript model but cannot produce one.

```
make -C packages/torch prepare build test run
```

`prepare` downloads the pinned platform's libtorch archive into the shared dependency cache; `run-mnist` and `verify-jit` are the commands for the two examples that need the MNIST files and a Python-scripted model.

Tensors

`Tensor.of` copies a List of numbers, or of nested Lists of rows, into a tensor of a given shape and dtype. `zeros`, `ones`, `full`, `rand`, `randn`, and `arange` build one without a value list, and `Tensor.scalar` and `Tensor.scalar_integer` build a zero-dimensional one. The dtype constants are `XT_UINT8`, `XT_INT8`, `XT_INT16`, `XT_INT32`, `XT_INT64`, `XT_FLOAT16`, `XT_FLOAT32`, `XT_FLOAT64`, and `XT_BOOL`. `rank`, `size`, `shape`, `numel`, and `dtype` report a tensor's structure, and `Tensor.str` is libtorch's own printed form.

`Tensor.item` returns the one value of a single-element tensor as a `Var` tagged by the tensor's dtype, `<long>` for an integer or bool tensor and `<f64>` for a floating one, so `t.item().double()` and `t.item().integer()` are the usual spellings. `Tensor.to_values` returns every element in row-major order as an Array of the same Vars.

An integer dtype never passes through a double, so `Tensor.of(%(9007199254740993), %(1), XT_INT64)` round-trips exactly and so does doubling it. A shape or index List holds numbers. A bare name inside `%()` is an `Atom`, stored as a compact `Symbol` when it fits, so `%(rows 2)` raises `<no-convert>`.

Operators

`Tensor` adopts a protocol carrying `add`, `sub`, `mul`, `div`, `neg`, and `matmul`, so `+ - * /` are elementwise, unary `-` negates, and `left.matmul(right)` multiplies matrices. A `double` beside a `Tensor` converts to a float64 scalar tensor and an `int` or `long` to an int64 one, which is why `2 * t` stays exact on an integer tensor. A `Tensor` also boxes into a `Var`, and the operators work on the boxed form.

```
Tensor a = Tensor.of(%(1 2 3 4), %(2 2), XT_FLOAT64);
Tensor b = Tensor.of(%(1 0 0 1), %(2 2), XT_FLOAT64);
EXPECT_TRUE((a * b).equal(Tensor.of(%(1 0 0 4), %(2 2), XT_FLOAT64)));
EXPECT_TRUE((a.matmul(b)).equal(a));
EXPECT_NEAR((2.0 * a).sum().item().double(), 20.0, 1e-12);
```

Indexing

`t[key]` reads three ways: an integer selects along the first dimension, a List of integers selects each dimension in turn, so `t[%(1 2)]` is `t.select(0, 1).select(0, 2)`, and a bool `Tensor` is a mask. `select`, `narrow`, `slice`, `reshape`, `transpose`, `t`, `squeeze`, `unsqueeze`, `flatten`, `index_select`, `masked_select`, `cat`, and `stack` are the shape operations `torch.x` spells; results view the same storage.

```
Tensor m = Tensor.of(%((1 2) (3 4)), %(2 2), XT_FLOAT64);
EXPECT_NEAR(m[%(1 0)].item().double(), 3.0, 1e-12);
Tensor mask = m.gt(Tensor.scalar(2.0, XT_FLOAT64));
EXPECT_INT_EQ(mask.dtype(), XT_BOOL);
EXPECT_NEAR(m[mask].sum().item().double(), 7.0, 1e-12);
```

Lifetimes

Every record here, `Tensor`, `Module`, `Optimizer`, `Scheduler`, and `JitModule`, owns one libtorch handle and is allocated with a `Scope` finalizer, so a value is released with the scope that created it. A training step is one `Scope.retain` and `Scope.release` pair around the forward, backward, and update, with the model and optimizer in the enclosing scope. `free` releases a handle early and returns NULL.

List results such as `parameters`, `named_parameters`, and generated tuples intern their wrapper references in the current List pool. Scope release does not reclaim those canonical cells. Long-running loops can bracket each request with `Pool.open` and `Pool.close`, releasing the request Scope before its List pool. Keep stable parameter handles outside that bracket when useful. Surviving values still need their ordinary Scope and pool ownership; a pool bracket does not extend a Tensor wrapper's lifetime.

An unnamed operator temporary is released sooner. `Tensor` declares the `discard` member of the protocol chapter, so the compiler releases the product in `a * b + c` right after the addition has used it, and a long chain of operators keeps only its inputs and its result alive. A value bound to a name is never discarded, so a loop

variable reassigned each step keeps its previous value until the scope ends; free it before the assignment, or give each step its own scope, when such a chain is long.

Autograd

`requires_grad_` marks a leaf that accumulates a gradient, `backward` runs the backward pass, `grad` reads the accumulated gradient and raises when none has been computed, and `zero_grad` clears it. `add_(b, alpha)` is the in-place `a += alpha * b` of a hand-written update. `Torch.no_grad` and `Torch.inference_mode` install a guard in the active scope, so releasing that scope restores the previous mode, including the release a `defer` runs while an Error transfers out. `inference_mode` is the stronger of the two: its results carry no autograd metadata at all. `Torch.grad_enabled` reports the current mode. From `examples/fit-line.x`:

```
Tensor x = Tensor.arange(0.0, 8.0, 1.0, XT_FLOAT64).reshape(%(8 1));
Tensor y = 3.0 * x - 1.0;
Tensor w = Tensor.randn(%(1 1), XT_FLOAT64).requires_grad_(1);
for (int step = 0; step < 200; step++) $scope() {
  Tensor error = Tensor.mse_loss(x.matmul(w), y);
  error.backward();
  Torch.no_grad();
  w.add_(w.grad(), -0.02);
  w.zero_grad();
}
```

Composing a model

A model is a `Module.composed()` root holding children registered under names, and its forward is ordinary x2c. That forward never enters C++, but libtorch still holds the module tree, so parameter enumeration, `zero_grad`, the optimizers, and serialization all work and the names match Python's `state_dict()`.

```
static Module _mlp(void) {
  Module model = Module.composed();
  model.register("l1", Module.linear(4, 8));
  model.register("l2", Module.linear(8, 1));
  return model;
}

static Tensor _forward(Module model, Tensor x) {
  Tensor hidden = model.child("l1").forward(x).tanh();
  return model.child("l2").forward(hidden);
}
```

A child's parameters are tensors, so `x.matmul(layer.parameters()[0].tensor().t()) + layer.parameters()[1].tensor()` is that affine step written out, as `examples/mlp.x` writes it. `Module.forward` runs a native forward where libtorch has one; a composed root has none and raises. `parameters`, `buffers`, `named_parameters`, and `named_buffers` are recursive, and a child's names are qualified by its registered name, so the model above enumerates `l1.weight l1.bias l2.weight l2.bias`. `train`, `eval`, and `is_training` carry the mode down the tree, so dropout and the normalizations read it; `to_dtype` converts parameters and buffers together.

The native layers are `linear`, `conv1d`, `conv2d`, `batch_norm1d`, `batch_norm2d`, `layer_norm`, `dropout`, `embedding`, `lstm`, `gru`, `max_pool2d`, `avg_pool2d`, `flatten`, `relu`, `tanh`, `sigmoid`, and `sequential`. Each takes PyTorch's arguments in PyTorch's order and its defaults, and produces the names Python's `state_dict()` uses; beside each one with more options is a `_with` form taking every one of them, such as `conv2d_with(in, out, kernel, stride, padding, dilation, groups, bias)`. `packages/torch/README.md` carries the full table.

`Module.sequential()` forwards its children in the order `Module.push` appended them, naming each by its position, so `model.forward(images)` is one call. The recurrent layers produce a state as well as a sequence and run through `Module.forward_state`: `(output hidden)` for a GRU and `(output hidden cell)` for an LSTM. `Tensor.mse_loss`, `.cross_entropy` (int64 class targets), `.nll_loss`, and `.bce_with_logits` are the losses `torch.x` spells; `.l1_loss` and `.huber_loss` come from the generated tier with the schema's own reduction argument, where 1 is the mean.

Training

`Optimizer.sgd`, `.sgd_momentum`, `.adam`, `.adamw`, `.rmsprop`, and `.adagrad` build over a module's parameters, and `Optimizer.over` takes a List of loose tensors with one of `XT_SGD .. XT_ADAGRAD`. `lr` and `set_lr` read and write every parameter group. libtorch's `DataLoader` is a template over a compile-time dataset and cannot cross a C ABI, so mini-batching is x2c: a permutation and an `index_select`.

```
Optimizer adam = Optimizer.adam(model, 0.05);
for (int step = 0; step < 200; step++) $scope() {
  Tensor pick = Torch.randperm(64).narrow(0, 0, 16);
  adam.zero_grad();
  Tensor error = Tensor.mse_loss(_forward(model, x.index_select(0, pick)),
                                 y.index_select(0, pick));
  error.backward();
  adam.step();
}
```

libtorch ships exactly two learning-rate schedules, `Scheduler.step_lr` and `Scheduler.reduce_on_plateau`; the first advances with `step`, the second with `step_metric`. The other three are x2c arithmetic over `Optimizer.lr` and `set_lr`, each writing the rate for the number of completed `step` calls, starting at construction with none taken:

- `Scheduler.cosine(o, t_max, eta_min)` anneals along a half cosine: after `t` steps the rate is `eta_min + (lr - eta_min) * (1 + cos(pi * t / t_max)) / 2`, and it stays at `eta_min` past `t_max`.
- `Scheduler.linear_warmup(o, steps, base_lr)` raises the rate from `base_lr / steps` to `base_lr`: `base_lr * min(1, (t + 1) / steps)`.
- `Scheduler.multistep(o, milestones, gamma)` multiplies the rate by `gamma` once for each milestone step count reached, as PyTorch's `MultiStepLR` does.

`Scheduler.steps` reports how many steps an x2c schedule has taken, and `step_metric` on one raises: it advances without a metric.

Checkpoints and Python

`Module.save` writes a pickled dict of name to tensor and `Module.load` reads one, requiring every parameter and buffer name to be present. `Checkpoint.save` and `Checkpoint.load` are the same format over a Map. Python code must follow two rules:

- Python must save a plain dict: `torch.save(dict(model.state_dict()), path)`. The `OrderedDict` that `state_dict()` returns does not unpickle in C++.
- Python must read with `weights_only=False`, or allow the tag the C++ pickler writes: `torch.serialization.add_safe_globals([torch.jit._pickle.restore_type_tag])`.

`Module.save_archive` and `load_archive` use libtorch's own archive instead; Python reads that only through `torch.jit.load`. Optimizer state saves and loads through the C++ archive for resuming in x2c or C++. Adam additionally supports explicit Python state-dict exchange through `Optimizer.save_python` and `load_python`, described below.

TorchScript inference

`JitModule.load` reads a module Python scripted or traced, `forward` takes a List of Tensor and returns a List of Tensor, one entry for a tensor result and one per element for a tuple of tensors, and `train` and `eval` set the mode. A file that is not TorchScript raises `<bad-state>`. From `examples/jit-infer.x`:

```
JitModule model = JitModule.load(path);
model.eval();
Torch.inference_mode();
List results = model.forward(%($x));
Tensor logits = results[0].tensor();
```

`make -C packages/torch verify-jit` scripts a 2-layer MLP in the pinned Python torch, runs the example over a fixed batch, and compares every number with Python's: `verify-jit: agreed on 8 values`.

MNIST

`Torch.mnist(root, train)` reads the four IDX files under `root` through libtorch's own reader and returns `(images targets)`: an N x 1 x 28 x 28 float32 tensor scaled to [0, 1] and N int64 classes. The reader checks the published row counts, 60,000 and 10,000. `examples/mnist.x` trains a sequential convolutional model for one epoch with Adam and a cosine anneal, then reports accuracy on the test set. It takes the data directory as an argument or from `TORCH_MNIST`, and prints where to obtain the files when they are absent:

```
make -C packages/torch run-mnist TORCH_MNIST=/path/to/mnist

mnist /tmp/mnist-real  train 60000  test 10000
batch    0  loss 2.335118  lr 0.001000
batch  900  loss 0.337217  lr 0.000103
test accuracy 0.9323
```

That run used the published files, in 3.8 seconds. The tests build a 10,000-row synthetic IDX set instead, so `make test` exercises the reader without the download.

The generated operator tier

`src/torch-ops.x` is generated by `tools/gen-ops.py` from the pinned operator schema in `schema/`, beside `src/torch.x`. It binds 1182 operators as `Tensor` methods with the schema's own argument structure: `int` is `long`, `float` is `double`, a `Scalar` is a `Var` that keeps integer and floating values distinct, an absent optional is `Var.null()` or a NULL `Tensor`, `int[]` is a `List`, `Tensor[]` is a `List` of tensors, and tuple results come back as a `List`. Where the two units share a name, `torch.x` takes precedence and the generated overload takes a suffix, so `sum.dim_IntList` is `Tensor.sum_dim`.

`packages/torch/schema/README.md` records the pinned schema file and its checksum, then a table of counts measured by `make gen-ops`: 2666 entries parsed, 1193 selected, and each rejection reason with its count. Below the table it names what this tier leaves out, including every `out=` form, named tensors, `Dimname` overloads, the `_foreach_*` family, custom autograd functions, and everything the private _-prefixed operators reach.

Errors

A failure inside libtorch raises `<bad-state>` with `(library "torch")`, the operation name, and the first line of libtorch's text under `(message ...)`; the full text stays available through `xt_last_error_full` in the raw API. Every entry point in `torch-2.10.h` catches, so no C++ exception crosses the ABI. This arm in `examples/mlp.x` prints `caught matmul: mat1 and mat2 shapes cannot be multiplied (3x5 and 4x8)`.

```
try { (void) _forward(model, Tensor.randn(%(3 5), XT_FLOAT32)); }
catch %(bad-state (library "torch") *detail): {
  String operation = detail.assoc(<operation>).string();
  String message = detail.assoc(<message>).string();
  printf("%s", %"caught $operation: $message\n");
}
```

Devices, optimizer exchange, custom gradients, and Lisp

The macOS arm64 profile supports CPU and MPS. The Linux x86_64 profile uses the pinned CPU archive and the system C++ runtime. GPU support on Linux and CUDA are outside these profiles. Device names are ordinary strings; `Tensor.to_device(device, dtype, non_blocking, copy)` already belongs to the generated operator surface. `Module.to_device(device)` moves parameters and buffers; call it before constructing an optimizer. Existing constructors keep their CPU behavior. MPS tensors use float32 or a supported integer dtype; MPS cannot represent float64. `to_values` explicitly copies to CPU before reading values. `Torch.mps_available()` reports availability, and `Torch.mps_synchronize()` waits for queued kernels when measuring execution.

```
TORCH_MNIST=/path/to/mnist TORCH_EPOCHS=2 TORCH_DEVICE=mps \
  make -C packages/torch run-mnist
```

The MNIST example defaults to one epoch. It writes a model checkpoint and compares held-out accuracy after reload. On the development MPS device, two epochs reached 95.23% and the reloaded model reached the same accuracy. Generated operators still depend on MPS kernel coverage. With fallback disabled, `make -C`

`packages/torch verify-mps` records that `Tensor.linalg_eig` raises the native error because `aten::linalg_eig` is unavailable on MPS in 2.10.0. Other generated operators can also lack MPS kernels.

`Optimizer.save_python(path)` and `load_python(path)` exchange Adam's `state_dict()` layout, including parameter groups, moments, exact step counts, and AMSGrad state. Parameter IDs match the destination's parameters by order; use the same parameter order when constructing both optimizers. Import parses all groups and states before replacing the optimizer. Numeric options import by value, including integers and scalar tensors. Unsupported semantic options such as `maximize`, `capturable`, `differentiable`, and decoupled weight decay are rejected. Backend execution flags become the ordinary scalar implementation. Existing `save`/`load` retain their C++ archive format. Other optimizer algorithms still use that archive format.

Python reads the file with `torch.load(path, weights_only=False)` and passes its dictionary to `optimizer.load_state_dict`. Python model checkpoints must use `dict(model.state_dict())`, as described in the checkpoint section. `make -C packages/torch verify-interchange` checks every parameter, moment and option through both resume directions, multiple groups and an uninitialized optimizer. Its `LibtorchAdam` reference uses libtorch's operation order. Stock Python Adam remains a separately measured comparison; the package does not promise bit-identical long training against its different floating-point order.

`Tensor.custom(forward, backward, inputs)` creates one differentiable output. The forward Func receives an `AutogradContext` and a List of inputs; backward receives that context and the output gradient, and returns one Tensor or Null per input. `save_for_backward`, `saved_tensors`, and `needs_input_grad` expose native autograd's saved values and input-gradient requirements. Saved native tensors outlive callback wrappers. Funcs and their captured referents remain borrowed and must outlive the graph on its creating thread.

Use `output.backward_callbacks()` for custom graphs. It temporarily disables autograd's worker scheduling, executes CPU or MPS callbacks on the invoking thread and restores the prior scheduling state. It does not change native kernel thread counts. Callback Errors are caught before returning to C++ and re-raised after the native call returns. Initial custom functions have one output and first-order gradients; in-place input changes, nested custom callbacks, reentrant backward and higher-order differentiation are unsupported. Callback contexts cannot escape. Custom graphs are not serializable. Custom forward also runs under `Torch.inference_mode`. Inputs must still not be mutated; inference tensors have no version counters to diagnose mutation.

`packages/torch/examples/custom-activation.x` trains a network through an x2c swish derivative:

```
make -C packages/torch run-custom
TORCH_DEVICE=mps make -C packages/torch run-custom
make -C packages/torch verify-custom
```

`TorchLisp.install(lisp)` installs tensor construction, arithmetic, readers, linear models, optimizers, training, model checkpoints and `torch-free`. `torch-values` returns an ordinary Lisp List; `torch-item` preserves integer values. Native values created during evaluation belong to the existing Lisp session. Caller-injected values retain their existing ownership. `packages/torch/examples/inline-lisp.x` builds and trains a persistent model through these ordinary operations, with no native training-loop binding.

```
make -C packages/torch run-lisp
make -C packages/torch verify-lifetimes
```

`torch-free` releases native resources early and invalidates every alias of that object. Wrappers remain in the session Scope until `Lisp.destroy`; repeated evaluation therefore has measurable wrapper growth even when native handle counts stay flat. Release prediction and loss tensors after each training step. Bare no-grad or inference guards are not Lisp bindings because their lifetime would otherwise extend to session destruction. The optional lifetime check builds isolated instrumented objects using the existing handle hooks. In 128 measured Lisp steps, native handles stayed at four while Scope allocations grew from 355 to 2159. Session destruction returned native handles to baseline; 64 custom graphs also returned to baseline after each graph.

Performance

x2c and PyTorch use the same libtorch backend. In the measured CPU workloads, x2c took less time for small-model training and prediction; convolutional training and tensor chains took roughly the same time. The chain results use explicit releases or a scope per iteration, as described under Lifetimes.

The table gives median seconds from five fresh-process pairs on an Apple M4 Max, using PyTorch 2.10 and the same native libraries. The one- and four-thread columns set the native intra-op workers; inter-op threads stay at one. Each row measures a different amount of work, so compare languages within a row.

Workload	x2c, 1 thread	Python, 1	x2c, 4 threads	Python, 4
Tabular training, native model	12.86	15.62	15.00	18.78
Tabular training, explicit operations*	13.57	16.58	15.69	19.40
Prediction, batch 1	8.46	15.01	8.46	14.94
Prediction, batch 32	9.39	13.27	9.44	13.19
Prediction, batch 256	12.41	13.11	17.66	18.03
MNIST convolutional training	14.15	14.36	9.98	10.28
Sequence training, window 32	12.09	15.20	12.10	15.12
Tensor chain, explicit release	2.60	2.65	5.97	6.19
Tensor chain, scope per iteration	2.70	2.67	6.10	6.24

Small-model training took about 18-20% less time, batch-one prediction about 44% less, and batch-32 prediction about 29% less. This is consistent with compiled control code reducing the overhead of frequent calls. Differences were small for larger prediction batches, MNIST, and the tensor chains; sample variation limits conclusions about those differences. More threads did not improve every workload. These results do not measure preprocessing or establish a general speedup.

The application rows come from the original comparison, with the 50-warmup-batch correction used for MNIST. The chains come from the separate fresh remedy comparison. These reports preserve the workload counts, individual samples, and methods; the applications were not all rerun with the remedies.

*The stock-PyTorch comparison of explicit tabular training remains a failed numerical check: its validation-loss difference was 0.223%, above the 0.1% tolerance. The other training configurations passed their tolerances. All three applications passed a separate Adam control that follows libtorch's operation order; that control measured correctness only. The chain checks and accumulated timing results agree exactly as reported to 12 significant digits, without claiming bitwise equality of every tensor element.

Separate startup and checkpoint diagnostics measured these ranges across the applications, in milliseconds:

Operation	x2c	Python
Fresh-process startup	108-113	488-504
Save checkpoint	0.7-2.6	0.8-2.6
Reload checkpoint	6.4-8.3	0.5-1.2

Startup includes imports and native initialization. Checkpoint formats differ between languages, so these compare application paths rather than identical serialization kernels.

Memory diagnostics measured about 151 MB for x2c versus 255 MB for Python after 400,000 training steps. The corrected sequence-window sweep reached a cumulative process peak of 131 MB versus 257 MB. Repeated temporary creation with a per-request List pool ended at 143 MB versus 188 MB after 400,000 requests. Despite that lower total, the pooled run's peak growth above its initial footprint was 2.24 times Python's. Native handles and canonical pool storage stayed bounded. These historical observations do not establish generally lower memory use.

A later memory investigation of the pooled request profile on macOS arm64 CPU identified substantial allocator-reclaimable residency and ruled out its 2 KB Pool depot as the source of the multi-MB residual. It did not identify individual allocation stacks or separate every native library cache from allocator fragmentation. Python was not rerun, so this does not revise the earlier peak ratio or establish an indefinite memory bound or a GPU/Linux result.

All measurements used one active desktop CPU system. They do not cover GPU performance, `torch.compile`, or workloads beyond these comparisons.

Limits

- macOS arm64 CPU/MPS and Linux x86_64 CPU; no CUDA profile.
- No distributed training.
- Python optimizer interchange covers Adam; other optimizers retain the C++ archive format. Module state works in both directions.
- The generated tier binds 1182 operators; `packages/torch/schema/README.md` lists the families it leaves out.
- Custom autograd currently supports one output and first-order gradients.
- `torch.compile` and TorchScript capture of x2c code are not possible: they capture Python. x2c runs a TorchScript model but cannot produce one. Results agree with Python within float32 tolerance.

The Region Model

The invariant

A value allocated inside a region must not be reachable after that region ends. Four kinds of place outlive a region: the function's result, storage declared outside the region (an outer local, an object a parameter points at, or a static), an object owned by an outer or sibling region, and any pointer whose target the compiler cannot identify, including a pointer that a called function stores into for the caller.

A region is a `$scope()` block, a `Scope.retain` and `Scope.release` pair, a `$scope(&slot)` push, a `Pool.open` bracket, an `$auto` local, the storage of a `Scope` local that `Scope.destroy` ends, or the function's own locals and parameters, whose storage ends when the function returns. `src/regions/regions.x` analyzes these forms before the transform driver lowers them, so a region is still the call that opens it and the `defer` beside it that closes it.

Exemptions

The pass exempts these ways of leaving a region.

A canonical-value pool owns each canonical `String` and `List`, so ending a `Scope` region does not free it. A value made inside a `Pool.open` bracket belongs to that bracket's pool and is tracked like scope-born storage. This includes `String.new`, `String.malloc`, and List cells returned by a helper. `cons(a, rest)` and `%($a)` retain their arguments in the cell's pool: a value from that same pool stays there, while a value from a shorter-lived region reports at the store.

`Scope.move` and `Context.export` change the storage's owner. After the move, the region the pass was tracking no longer holds the allocation, so the allocation survives the end of that region. The promote family - `String.promote`, `List.promote`, and `Atom.promote` - hands the value to an ancestor pool the same way and is exempt for the same reason. Moving into a caller's slot is the ordinary way to return mutable storage, and the pass stops tracking the value there. Moving into a `Scope` local of the same function ties the value to that local's storage, so a read after `Scope.destroy` reports.

A typed conversion copies. Assigning a `Block`, `Buffer`, or `Array` into a `String` or `List` declaration produces a new canonical value with its own owner, so the target does not inherit the region. The pass classifies the assignment by the declared type through the compiler's `Var` tag for that type, as the converter does. A `String *` or `List *` points at storage and gets no exemption.

Every region a function opens ends before the function's own storage. The address of a local can therefore be stored into another local, or into an object of a region the function opens, without a warning. A reference capture, `using &name`, moves the local into a cell of the active region, so the closure holds that cell, which the active region owns.

A store into a place that an earlier `defer` in the same block writes back, as `$let` does, is undone before the block ends and is not reported. A store made before that `defer`, or after a `defer` that runs on only one branch, is reported, because a `return` between them leaves the place holding the value.

A destination that converts its value by copying does not keep it: a `List` parameter given an `Array`, or a `Var` given a C string, which it boxes as a fresh `String`. A local C array passed on, returned, or stored counts as the address of its first element.

A number holds no address. A value of integer, floating-point, or enum type, or a destination of such a type, carries nothing the pass tracks. An address cast to `intptr_t` and then returned does not report, and a pointer cast back from that number is not followed.

What the warnings do not cover

The check covers lexical regions and the storage it tracks. It does not cover:

- storage from plain `malloc` or a C library;
- pointer arithmetic and casts through raw C types;
- values reached through a field of a stack `struct`, which the pass treats as one storage, so an address kept in a field of a struct that is copied out is not followed;
- callbacks and function pointers, including entry points a Lisp binding calls;
- calls into functions defined in other units, except the runtime operations the pass knows; such a call is assumed to keep none of its arguments and to return no tracked storage;
- other aliases of storage that `Scope.free` or `Scope.realloc` ended; the pass ends only the local passed to the call, and it reports a free only of storage it can see is not from a Scope allocator: a literal, the function's own storage, or a pooled value;
- `Context` regions, which are not modeled;
- `$auto` and `defer` cleanups other than the runtime's own `Array`, `Map`, `Block`, `Bytes`, `Buffer`, `Context`, and `Scope` cleanups;
- a release inside a branch, which leaves the region open for the statements after that branch.

Each of these can produce a dangling pointer with no warning. A program without warnings is not proven memory safe.

Optional lifetime proof

`x2c-graph certify` checks selected entry points and the functions they can call across the supplied source units. It applies the same region ownership and parameter-effect rules as the compiler's local pass, then asks whether every reachable memory effect is covered. A `proved` result establishes the scoped lifetime invariant for those paths under the native effect assumptions listed in the report. If a root hands storage to its caller or another owner, the report lists the remaining release obligation.

The audit distinguishes a proven lifetime violation from an incomplete proof. An unknown call, pointer cast or arithmetic, indirect storage access, aggregate copy, or unmodeled scope ending is an obstacle, not a compiler warning or error. A project may provide explicit native effect contracts to the audit; they are assumptions of its result and do not change compilation. The optional command and its contract format are described in the source graph reference. The proof concerns scoped ownership and lifetime. It does not establish general C memory safety or correctness of emitted code.

How the check works

Each function gets one summary of two facts: which of Scope and Pool storage its result may use (possibly both), and where each parameter is sunk (returned by identity, retained in the fresh result, placed into another parameter's object or a static, or stored through an unknown pointer). The summaries of a unit's functions are computed to a fixpoint because a caller's facts depend on its callees' facts. Each round walks every body with a stack of open regions and a map from each binding to its known facts: its parameter index, the region it was born in, the local a pointer was taken from, and whether a free has already ended it. The first round that changes no summary walked every body against final summaries, so its warnings are the ones reported.

An address taken with `&` belongs to the storage it names. The address of a local, a parameter, or a compound literal belongs to the function's own storage. An address reached through a pointer, such as `&box->value`, belongs to whatever that pointer holds, so an address inside a parameter's object counts as that parameter in the summary. A caller of a function that returns `&box->value` therefore receives its own borrowed address as the result. An argument to a reference parameter is the address of the object it names, so `bind(object)` with `T &` is checked as `bind(&object)` with `T *`.

A `meta` function is walked the same way when it is defined, against the summaries of the `meta` functions defined before it. A compile-time call frees its locals when it returns, so a finding in a `meta` body is an error and the definition is rejected; see Meta Functions. A native function outside the runtime table described below takes a summary its signature implies; see Native modules.

Summaries stay inside the unit. For a call into another unit, the pass uses only a fixed table of runtime operations, such as a `Scope` allocator, `cons`, `Array.push`, `Scope.move`, and `Array.list_free`. Summaries are not written to the `.xi` interface. A dependent unit collects its includes before those units are transformed, and parallel workers do not share transformed units, so an interface summary would make a unit's warnings depend on input order, the job count, and interfaces left by an earlier translation. As a result, the pass reports no warning when a caller passes a region-born value to a function in another unit that stores its argument into a static.

Where the design sits

The region check is based on the ML Kit and Cyclone. The ML Kit inferred regions and a region-annotated program from an unannotated one. Cyclone made regions part of the type system of a C-like language and checked pointer lifetimes against them. From the ML Kit, x2c takes the idea that regions are the unit of deallocation; from Cyclone, the idea that a C program can be checked against them. The regions are the ones the author wrote with `$scope` and the retain and release calls. The check adds nothing to the program text and produces only warnings, except in compile-time code.

Rust and Swift put ownership and lifetime in the signature. Every call is checked locally against a declared contract, and a violation is an error. x2c has no lifetime annotations, so the check derives the same information, and derived facts are weaker than a declared contract. As a result, existing C-shaped code compiles unchanged, and ordinary code gets only warnings.

Go's escape analysis and Infer use the most similar methods. Go computes per-function escape facts and uses them to choose between stack and heap allocation. Infer computes per-procedure summaries and propagates them across a program to report memory and lifetime defects. x2c's pass applies Infer's method to Go's kind of fact: it computes summaries per function within one unit and uses them for warnings.

Compiler API (provisional)

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

Compiler API (provisional)

Functions and types exposed by each compiler module.

Module	Description
src/adapter-memo.x	memoized compiler adapter construction.
src/ast-rewrite.x	rewrite immediate AST children through canonical Lists.
src/ast.x	shared helpers for x2c compiler AST nodes.
src/builtins.x	the built-in macros' compile-time algorithms.
src/cache.x	cached literal slots and deferred file-static initializers.
src/deps.x	Make dependency output for x2c translation units.
src/diagnostics.x	compiler diagnostic storage and rendering.
src/expressions-reports.x	expression typing diagnostics.
src/fields.x	statements that copy or set named fields.
src/format.x	code formatting helpers for the x2c compiler.
src/frontend.x	configured compiler sessions and source units.
src/grammar.x	the source forms that lowering recognizes.
src/lambdas.x	lambda parsing and capture resolution.
src/linked-meta.x	shipped `meta` code compiled into the compiler.
src/main.x	x2c command dispatch.
src/meta-helper-client.x	the compiler's side of the project meta helper.
src/meta-project.x	the project meta build.
src/operator-ledger.x	binary operator facts and case projections.
src/parse-reports.x	package, binding, and alias diagnostics.
src/preprocess.x	C preprocessor directives in x2c source.
src/report.x	Command progress and completion receipts.
src/sourceview.x	request-owned source overlays.
src/stage.x	the values crossing between meta code and the program.
src/statements.x	x2c statement parsing.
src/toolchain.x	Host preprocessing, compilation, archive, and link actions.
src/translation.x	synchronous compiler translation requests.
src/type-ledger.x	the compiler's Var tag tables, projected from the ledger.
src/type.x	x2c semantic types.
src/utils.x	System utilities for environment discovery and workers.
src/callables/callables.x	callable values lowered to C helpers.
src/cleanup/cleanup.x	cleanup regions and the transfers that leave them.
src/cli/cli.x	x2c command-line parsing and presentation.
src/collect/collect.x	source-ordered shallow symbol collection and replay.
src/compiler/compiler.x	one x2c unit's translation state and its two parses.
src/components/bracket-access.x	Bracket access policy: Array and Map mutation through their typed helpers, and any other participant's read, store, and update through its indexing members, with each operand evaluated once, in order. Getter resolution remains with admission.
src/components/collection-literals.x	Array and Map literals.
src/components/delegate-members.x	delegate fields.
src/components/destructuring.x	flat List destructuring.

Module	Description
src/components/dynamic-operators.x	dynamic operators.
src/components/interpolation.x	interpolated Strings.
src/components/lambda-functions.x	lambda lowering.
src/components/pattern-match.x	match statements.
src/components/printf-format.x	Var values in printf-family formats.
src/components/protocol-adapters.x	protocol adapters and Var descriptors.
src/components/raise.x	raise.
src/components/string-switch.x	switches on a String.
src/components/try-catch.x	try, catch, and finally.
src/emit/emit.x	C tokens from normalized x2c ASTs.
src/expressions/expressions.x	expression syntax and its resolution.
src/generate/generate.x	the generated header and source of one unit.
src/initializers/initializers.x	brace initializer conversion.
src/literals/literals.x	x2c literal parsing.
src/macros/macros.x	source macros and the compile-time Lisp they run.
src/meta-group/meta-group.x	a unit's meta group, emitted as C.
src/meta-native/meta-native.x	meta functions and the native code they call.
src/meta-sdk/meta-sdk.x	the compiler's answers to `lib/meta.x` operations.
src/parse/parse.x	x2c declarations, parsed from source or constructed.
src/project/project.x	x2c project manifests.
src/protocol/protocol.x	protocols from declaration to conform forms.
src/regions/regions.x	values that can outlive the region that allocated them.
src/symtab/symtab.x	the compiler's semantic symbol table.
src/transform/transform.x	x2c AST transformation pipeline.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/adapter-memo.x`

Memoized compiler adapter construction.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/ast-rewrite.x`

Rewrite immediate AST children through canonical Lists.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/ast.x`

Shared helpers for x2c compiler AST nodes.

Functions

Function	Summary
Ast.addressed_identifier	Returns the name whose address an expression takes, or NULL.

Function	Summary
Ast.append_modifiers	Returns declarator with modifiers after its own, as a declaration whose base contributes modifiers declares each name.
Ast.binding	Constructs a (binding identity spelling) node.
Ast.binding_spelling	Returns a valid binding node's source spelling, or NULL.
Ast.collect_binding_references	Records in referenced the identity of every binding node names.
Ast.contains_head	Returns whether any list under value has kind as its head.
Ast.designated	Returns the innermost node value designates, past the forms that still name the same object: an expr wrapper, parentheses, a member, and an index into an array.
Ast.direct_identifier	Returns the name an expression designates directly, following the forms that still name the same object - parentheses, a member, an array index, a dereference - or NULL when the expression designates no single name.
Ast.entry_call	Returns the statement calling the void (void) function entry.
Ast.indirect_identifier	Returns the name of the pointer an expression designates through, or NULL when it designates no object through a single name.
Ast.initializer_cases	Returns initializer alternatives and their optional native macro input.
Ast.initializer_functions	Returns function alternatives when every initializer arm calls one shared input, and stores that input expression in source.
Ast.lvalue_binding	Returns the binding whose stored object the lvalue ast names, or NULL.
Ast.never_returns	Returns whether control cannot flow out the bottom of ast.
Ast.prototype_declarator	Returns declarator with the outermost volatile removed from each of its parameters.
Ast.reference_guard_exits	Returns whether arm ends with a return or non-returning raise.
Ast.replace_nodes	Returns root with each list beneath it that replaced maps, keyed by its address, swapped for its value, and adds each rebuilt ancestor to replaced.
Ast.rewrap_origin	Returns replacement under the anchors of original, the statement it replaces, so a rewrite does not lose the node's source position.
Ast.rewrite_children	Applies per_child to each List child of ast and returns the node rebuilt from the results; non-list children pass through.
Ast.try_binding	Extracts a valid (binding positive-integer string) node.
Ast.without_origin	Returns ast without the (at ORIGIN ...) anchors it arrived wrapped in.
Ast.written_definitions	Returns decl with each function definition marked kernel as written.
Ast.written_operand	Returns the operand a write node changes: the left operand of an assignment or prefix update, or the operand of a postfix update; NULL when node writes no operand.
Symbol.changes_left_operand	Returns whether op writes its left operand.
Symbol.is_assignment_op	Returns whether op is plain or compound assignment.

Ast

Ast.addressed_identifier

```
String Ast.addressed_identifier(Var value)
```

Returns the name whose address an expression takes, or NULL.

Source: src/ast.x:141

Ast.append_modifiers

```
List Ast.append_modifiers(List declarator, List modifiers)
```

Returns declarator with modifiers after its own, as a declaration whose base contributes modifiers declares each name.

Source: `src/ast.x:59`

Ast.binding

`List Ast.binding(int identity, String spelling)`

Constructs a `(binding identity spelling)` node. The caller must supply a positive compiler-issued identity.

Source: `src/ast.x:30`

Ast.binding_spelling

`String Ast.binding_spelling(List binding)`

Returns a valid binding node's source spelling, or `NULL`.

Source: `src/ast.x:50`

Ast.collect_binding_references

`void Ast.collect_binding_references(Var node, Map referenced)`

Records in `referenced` the identity of every binding `node` names.

Source: `src/ast.x:256`

Ast.contains_head

`int Ast.contains_head(Var value, Symbol kind)`

Returns whether any list under `value` has `kind` as its head.

Source: `src/ast.x:218`

Ast.designated

`List Ast.designated(Var value)`

Returns the innermost node `value` designates, past the forms that still name the same object: an `expr` wrapper, parentheses, a member, and an index into an array. What remains is a name, a designation through a pointer, or another expression; a non-list designates nothing.

Source: `src/ast.x:114`

Ast.direct_identifier

`String Ast.direct_identifier(Var value)`

Returns the name an expression designates directly, following the forms that still name the same object - parentheses, a member, an array index, a dereference - or `NULL` when the expression designates no single name. A declaration qualifier that must reach one object, such as the `volatile` an error transfer requires, applies to this name.

Source: `src/ast.x:160`

Ast.entry_call

`List Ast.entry_call(String entry)`

Returns the statement calling the `void (void)` function `entry`.

Source: `src/ast.x:310`

Ast.indirect_identifier

`String Ast.indirect_identifier(Var value)`

Returns the name of the pointer an expression designates through, or `NULL` when it designates no object through a single name. `*pointer`, `pointer[index]`, and `pointer->member` all change the object the pointer holds, which `Ast.direct_identifier` reports as no name at all.

Source: `src/ast.x:178`

Ast.initializer_cases

```
List Ast.initializer_cases(Ast ast, List &input)
```

Returns initializer alternatives and their optional native macro input.

Source: `src/ast.x:388`

Ast.initializer_functions

```
List Ast.initializer_functions(Ast ast, List &source)
```

Returns function alternatives when every initializer arm calls one shared input, and stores that input expression in `source`. Other forms return NULL.

Source: `src/ast.x:402`

Ast.lvalue_binding

```
List Ast.lvalue_binding(Ast ast)
```

Returns the binding whose stored object the lvalue `ast` names, or `NULL`. Pointer dereferences and pointer indexes name another object; an array field remains part of its containing aggregate.

Source: `src/ast.x:133`

Ast.never_returns

```
int Ast.never_returns(Ast ast)
```

Returns whether control cannot flow out the bottom of `ast`. Recognized terminals are shared non-returning raises, native termination calls, and blocks ending in either one when the block contains no `return`. Generation uses this fact to mark the enclosing function `_Noreturn`.

Source: `src/ast.x:335`

Ast.prototype_declarator

```
List Ast.prototype_declarator(List declarator)
```

Returns `declarator` with the outermost `volatile` removed from each of its parameters. C ignores a parameter's top-level qualifier when it compares a prototype with its definition, and the qualifier the error transfer requires belongs to the definition that writes the parameter, not to the declaration its callers read.

Source: `src/ast.x:77`

Ast.reference_guard_exits

```
int Ast.reference_guard_exits(List arm)
```

Returns whether `arm` ends with a return or non-returning raise.

Source: `src/ast.x:316`

Ast.replace_nodes

```
Var Ast.replace_nodes(Var root, Map replaced)
```

Returns `root` with each list beneath it that `replaced` maps, keyed by its address, swapped for its value, and adds each rebuilt ancestor to `replaced`. Only the ancestors of a swapped list are new, so `root` returns itself when nothing beneath it is swapped. The walk keeps deep expression chains off the C stack.

Source: `src/ast.x:229`

Ast.rewrap_origin

```
Ast Ast.rewrap_origin(Ast original, Ast replacement)
```

Returns `replacement` under the anchors of `original`, the statement it replaces, so a rewrite does not lose the node's source position.

Source: `src/ast.x:288`

Ast.rewrite_children

```
Ast Ast.rewrite_children(Ast ast, Func per_child)
```

Applies `per_child` to each `List` child of `ast` and returns the node rebuilt from the results; non-list children pass through. When no child changed, no scratch storage is allocated and `ast` itself returns, so the fixed-point transform driver can compare unchanged-node identity.

Source: `src/ast.x:212`

Ast.try_binding

```
int Ast.try_binding( List binding, int &?identity, String &?spelling)
```

Extracts a valid `(binding positive-integer string)` node. Returns one on success and writes only non-`NULL` outputs; failure returns zero without changing either output.

Source: `src/ast.x:37`

Ast.without_origin

```
Ast Ast.without_origin(Ast ast)
```

Returns `ast` without the `(at ORIGIN ...)` anchors it arrived wrapped in. Every block statement carries one so that a transform-phase diagnostic can name its line; a pass that dispatches on a statement tag has to see the statement, not the anchor.

Source: `src/ast.x:276`

Ast.written_definitions

```
List Ast.written_definitions(List decl)
```

Returns `decl` with each function definition marked `kernel` as written. The transform strips the marker before it offers the definition only to shipped function rules.

Source: `src/ast.x:300`

Ast.written_operand

```
List Ast.written_operand(Ast node)
```

Returns the operand a write `node` changes: the left operand of an assignment or prefix update, or the operand of a postfix update; `NULL` when `node` writes no operand.

Source: `src/ast.x:195`

Symbol

Symbol.changes_left_operand

```
int Symbol.changes_left_operand(Symbol op)
```

Returns whether `op` writes its left operand.

Source: `src/ast.x:382`

Symbol.is_assignment_op

```
int Symbol.is_assignment_op(Symbol op)
```

Returns whether `op` is plain or compound assignment.

Source: `src/ast.x:378`

Public types

Type	Kind	Summary
`Ast`	alias	Represents one compiler AST node as a canonical immutable `List`.

Ast

```
typedef List Ast
```

Represents one compiler AST node as a canonical immutable `List`. `NULL` denotes no node, and nonempty nodes have the canonical `List`-pool lifetime.

Source: `src/ast.x:13`

Design notes

Simple fixed-shape nodes remain direct immutable `Lists`. This module holds the node operations that several compiler phases share.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/builtins.x`

The built-in macros' compile-time algorithms.

Functions

Function	Summary
`Compiler.builtin_targets`	Returns each built-in algorithm by the name compile-time code calls it with.

`Compiler`

Compiler.builtin_targets

```
Map Compiler.builtin_targets(void)
```

Returns each built-in algorithm by the name compile-time code calls it with.

Source: `src/builtins.x:924`

Design notes

The built-in macros expand through these functions, which run as native code inside the compiler. `Compiler.builtin_targets` binds each into the shared session under the name its callers use, such as `etc/builtin-macros.x` and `etc/lisp-bindings.x`.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

src/cache.x

Cached literal slots and deferred file-static initializers.

Functions

Function	Summary
Compiler.initialization_guard	Returns the declaration of the file's initialization guard, which starts at zero.
Compiler.setup_cache_init	Materializes cached literals and deferred file-static initialization.
Compiler.zero_initializer	Preserves native initializer shape so C infers dimensions and checks designators before cached values are assigned during initialization.

Compiler

Compiler.initialization_guard

```
List Compiler.initialization_guard(Compiler c, List guard)
```

Returns the declaration of the file's initialization guard, which starts at zero.

Source: src/cache.x:516

Compiler.setup_cache_init

```
List Compiler.setup_cache_init( Compiler c, List header, List source, String prefix, String guard_name, String initializer_name)
```

Materializes cached literals and deferred file-static initialization. header and source must be partitioned lowered AST regions from this compiler. Every (cache id) must index c.id_keys, and file-static dependency state from the full parse must be complete. prefix, guard_name, and initializer_name name the header's private slots, guard, and initializer. Returns (header source bindings); bindings maps source cache ids to emitted slots. Appends initialization work to the <prepare>, <statics>, and <finish> initialization areas; the operation is not idempotent. Header cache storage remains private to each C translation unit that includes it.

Source: src/cache.x:64

Compiler.zero_initializer

```
List Compiler.zero_initializer(Compiler c, List value)
```

Preserves native initializer shape so C infers dimensions and checks designators before cached values are assigned during initialization.

Source: src/cache.x:279

Design notes

This module materializes a unit's cached literals between lowering and C emission: it declares their static slots, initializes them in the unit's initialization phases, and moves file-static initializers C cannot evaluate into generated helpers. Immutable values share generated storage; mutable collections are copied at use sites, so caching does not change identity.

Compiler.cache in compiler/literal-cache.x decides cache-key equality and assigns the (cache id) nodes whose ids index Compiler.id_keys; this module trusts that mapping. Header and source regions may materialize the same immutable value in separate static slots; String and List canonicalization preserves value identity across them.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/deps.x

Make dependency output for x2c translation units.

Functions

Function	Summary
translation_depfile_parse	Parses prerequisite words after the first unescaped colon in text.
Compiler.write_depfile	Publishes the Make depfile for one completed translation.

Functions

translation_depfile_parse

```
List translation_depfile_parse(String text)
```

Parses prerequisite words after the first unescaped colon in text. Backslash escapes and doubled dollars are decoded in that region. A backslash-newline continues the rule; an ordinary newline ends it before any phony rules. Escaped target colons remain part of their words. A NULL input or no parsed prerequisites returns NULL. Returned cells and path Strings follow the current canonical List and String pool lifetimes.

Source: src/deps.x:132

Compiler

Compiler.write_depfile

```
int Compiler.write_depfile( Compiler compiler, CliRequest request, String input, String output_dir)
```

Publishes the Make depfile for one completed translation. compiler.deps must reflect the translated input, and the selected depfile's parent directory must exist. Disabled dependency output and inspection requests return one without writing. Otherwise prerequisites are unique and sorted; the target is request.dep_target or the generated C and header pair, followed by optional phony rules. A process-specific sibling is written and closed before rename, so handled open, write, close, or rename failures preserve any existing depfile, report to stderr, and return zero. Cleanup attempts to unlink an opened sibling but does not report an unlink failure. Allocation failure transfers through ordinary runtime Error handling instead of returning zero.

Source: src/deps.x:38

Design notes

Translation collects the prerequisites. This module writes them in a stable Make spelling and publishes the file atomically.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/diagnostics.x

Compiler diagnostic storage and rendering.

Functions

Function	Summary
diagnostics_write_json	Sends every later printed diagnostic to path as JSON Lines.
Compiler.diagnostics	Returns a report-order snapshot of all collected diagnostics.
Compiler.display_path	Returns a physical source path for semantic facts, otherwise a path relative to the compiler root.
Compiler.dump_cache	Prints each cached numeric identifier and its key to stdout.
Compiler.dump_symbol_table	Prints every entry in map to stdout in Map iteration order.
Compiler.dump_tokens	Prints every non-EOF token with its position and visible content.
Compiler.error_count	Returns the number of counted diagnostics accepted since the last reset.
Compiler.file_prefix	Returns the prefix that the statics of the module file holding the current token take, or NULL outside a module unit.

Function	Summary
`Compiler.origin_location`	Resolves a recorded occurrence through generated ancestry to its source.
`Compiler.print_diagnostic`	Writes one structured diagnostic entry and source context to stderr, or one JSON line after `diagnostics_write_json`.
`Compiler.report_error`	Submits a located compiler error, then transfers or exits.
`Compiler.report_problem`	Records and emits a located compiler error, then returns so the translation continues; it fails when it finishes.
`Compiler.report_warning`	Records and emits a located warning without consuming the error limit.
`Compiler.report_warning_at`	Records and emits a warning at a location built earlier by `Compiler.token_location`, for a report raised after its token has been consumed.
`Compiler.source_token`	Returns a copy of `token` at its line, column, and byte offset in the own text of the file that holds it, and sets `file` to that file.
`Compiler.take_diagnostics`	Moves collected child reports into the caller's store without re-emitting.
`Compiler.text_source`	Returns the file that a module's line markers give byte `position` of the compiler's text, and sets `offset` to that byte's offset in the file.
`Compiler.token_column`	Returns `token`'s column in the file that holds it.
`Compiler.token_location`	Builds the diagnostic location for `token` or the current token.
`Compiler.token_source`	Returns the file that the line markers of a merged text give `token`, and sets `line` to the token's line in that file.
`Diagnostics.entries`	Returns an immutable `List` snapshot in publication order.
`Diagnostics.hold`	Stops streaming until `Diagnostics.release` and records the current entries, count, and limit state.
`Diagnostics.new`	Creates an empty diagnostic store that streams through `printer`.
`Diagnostics.reached_limit`	Returns whether counted reports have reached the positive limit.
`Diagnostics.release`	Restores the streaming saved by `hold`.
`Diagnostics.report`	Records and synchronously emits one diagnostic unless already limited.
`Diagnostics.reset`	Clears stored entries and limit state while preserving configuration.

Functions

diagnostics_write_json

```
int diagnostics_write_json(String path)
```

Sends every later printed diagnostic to `path` as JSON Lines. The file is created or truncated. Each diagnostic is one append write, so forked translation workers sharing the descriptor never interleave lines, and a line is complete before any exit. Returns zero when `path` cannot be opened.

Source: `src/diagnostics.x:223`

`Compiler`

Compiler.diagnostics

```
List Compiler.diagnostics(Compiler compiler)
```

Returns a report-order snapshot of all collected diagnostics. Snapshot cells are canonicalized through the active pool hierarchy and share entry values; each retains its actual producing-pool lifetime.

Source: `src/diagnostics.x:647`

Compiler.display_path

```
String Compiler.display_path(Compiler compiler, String path)
```

Returns a physical source path for semantic facts, otherwise a path relative to the compiler root. Pseudo paths and NULL stay unchanged.

Source: `src/diagnostics.x:338`

Compiler.dump_cache

`void Compiler.dump_cache(Compiler compiler)`

Prints each cached numeric identifier and its key to stdout.

Source: `src/diagnostics.x:683`

Compiler.dump_symbol_table

`void Compiler.dump_symbol_table(Compiler compiler, Map map)`

Prints every entry in `map` to stdout in `Map` iteration order.

Source: `src/diagnostics.x:678`

Compiler.dump_tokens

`void Compiler.dump_tokens(Compiler compiler)`

Prints every non-EOF token with its position and visible content. `Compiler.tokenize` must have populated the compiler's tokenizer.

Source: `src/diagnostics.x:668`

Compiler.error_count

`int Compiler.error_count(Compiler compiler)`

Returns the number of counted diagnostics accepted since the last reset. Warnings and the generated limit notice are excluded.

Source: `src/diagnostics.x:641`

Compiler.file_prefix

`String Compiler.file_prefix(Compiler c)`

Returns the prefix that the statics of the module file holding the current token take, or NULL outside a module unit. A module's first row marks the start of its text, so other units return before any scan.

Source: `src/diagnostics.x:388`

Compiler.origin_location

`List Compiler.origin_location(Compiler compiler, int occurrence)`

Resolves a recorded occurrence through generated ancestry to its source. Returns NULL when `occurrence` is outside the origin table. The occurrence must come from this compiler's current parse and transform state. Returned location cells are canonicalized through the active pool hierarchy and retain their actual producing-pool lifetime. They share the recorded filename, which retains its own producing-pool lifetime.

Source: `src/diagnostics.x:313`

Compiler.print_diagnostic

`void Compiler.print_diagnostic(Compiler compiler, List entry)`

Writes one structured diagnostic entry and source context to stderr, or one JSON line after `diagnostics_write_json`. NULL is ignored. A present location supplies `file`, one-based `line` and `column`, and token `length`; `String` notes are joined into one note line.

Source: `src/diagnostics.x:276`

Compiler.report_error

```
void Compiler.report_error( Compiler compiler, Symbol code, String message, Token token, List notes)
```

Submits a located compiler error, then transfers or exits. An explicit `token` wins; otherwise an active recorded origin is resolved before the current token. NULL message defaults to `"compiler error"`.

Raises: `<malformed>` with the supplied category while a recovery boundary is active. Without one, exits the process with status 1.

Source: `src/diagnostics.x:496`

Compiler.report_problem

```
void Compiler.report_problem( Compiler c, Symbol code, String message, Token token, List notes)
```

Records and emits a located compiler error, then returns so the translation continues; it fails when it finishes. Location and the message default match `Compiler.report_error`.

Source: `src/diagnostics.x:513`

Compiler.report_warning

```
void Compiler.report_warning( Compiler c, Symbol code, String message, Token token, List notes)
```

Records and emits a located warning without consuming the error limit. Location selection matches `Compiler.report_error`; NULL code becomes `<warning>` and NULL message becomes `"compiler warning"`. This operation returns without raising or changing the process exit status.

Source: `src/diagnostics.x:524`

Compiler.report_warning_at

```
void Compiler.report_warning_at( Compiler compiler, Symbol code, String message, List location,
List notes)
```

Records and emits a warning at a location built earlier by `Compiler.token_location`, for a report raised after its token has been consumed. Defaults match `Compiler.report_warning`.

Source: `src/diagnostics.x:533`

Compiler.source_token

```
Token Compiler.source_token(Compiler c, Token token, String &file)
```

Returns a copy of `token` at its line, column, and byte offset in the own text of the file that holds it, and sets `file` to that file. The host preprocessor's rows record no offsets, so a token of its output keeps its merged offset.

Source: `src/diagnostics.x:448`

Compiler.take_diagnostics

```
void Compiler.take_diagnostics(Compiler c, Compiler child)
```

Moves collected child reports into the caller's store without re-emitting. Shared stores already contain their entries. The child's separate store remains configured and empty after its reports have been transferred.

Source: `src/diagnostics.x:629`

Compiler.text_source

```
String Compiler.text_source(Compiler c, int position, int &offset)
```

Returns the file that a module's line markers give byte `position` of the compiler's text, and sets `offset` to that byte's offset in the file. Without such a marker, the file is the compiler's own and `offset` is `position`.

Source: `src/diagnostics.x:434`

Compiler.token_column

```
int Compiler.token_column(Compiler c, Token token)
```

Returns `token`'s column in the file that holds it. A module region that begins inside a line of its file continues that line's columns.

Source: `src/diagnostics.x:423`

Compiler.token_location

```
List Compiler.token_location(Compiler compiler, Token token)
```

Builds the diagnostic location for `token` or the current token. If neither exists, returns the current file at line 1, column 1, and byte position 0 with zero length. Semantic facts use physical paths; ordinary token locations use `Compiler.display_path`. Location cells and a derived path are canonicalized through the active pool hierarchy and retain their actual producing-pool lifetimes; an unchanged filename retains the compiler's producing-pool lifetime.

Source: `src/diagnostics.x:357`

Compiler.token_source

```
String Compiler.token_source(Compiler c, Token token, int &?line)
```

Returns the file that the line markers of a merged text give `token`, and sets `line` to the token's line in that file. A token outside the marked regions keeps the compiler's own file and its line.

Source: `src/diagnostics.x:376`

Diagnostics

Diagnostics.entries

```
List Diagnostics.entries(Diagnostics diag)
```

Returns an immutable `List` snapshot in publication order. Snapshot cells are canonicalized through the active pool hierarchy and retain their actual producing-pool lifetime. They share the stored entry `Lists`. The snapshot includes warnings and the limit notice; changing or resetting `diag` does not change it.

Source: `src/diagnostics.x:105`

Diagnostics.hold

```
DiagnosticsHold Diagnostics.hold(Diagnostics diag)
```

Stops streaming until `Diagnostics.release` and records the current entries, count, and limit state.

Source: `src/diagnostics.x:72`

Diagnostics.new

```
Diagnostics Diagnostics.new(Compiler printer, int limit)
```

Creates an empty diagnostic store that streams through `printer`. A NULL `printer` does not stream. A negative `limit` is treated as zero; zero collects without a stopping threshold. The printer is borrowed.

Source: `src/diagnostics.x:53`

Diagnostics.reached_limit

```
int Diagnostics.reached_limit(Diagnostics diag)
```

Returns whether counted reports have reached the positive limit. A zero limit never reports that it has been reached.

Source: `src/diagnostics.x:113`

Diagnostics.release

```
void Diagnostics.release(Diagnostics diag, DiagnosticsHold hold, int keep)
```

Restores the streaming saved by `hold`. When `keep` is set, entries published since the hold remain and stream now; otherwise they are discarded with the count and limit state they changed.

Source: `src/diagnostics.x:84`

Diagnostics.report

```
void Diagnostics.report( Diagnostics diag, Symbol code, String message, List location, List notes)
```

Records and synchronously emits one diagnostic unless already limited. Entries retain publication order. NULL `code` becomes `<driver>`. A report equal to a stored entry is ignored. Reaching a limit greater than one publishes a following `<limit>` notice; a limit of one stops after the first error. Later reports are ignored. Supplied message, location, and notes are shared; their canonical-value pools must outlive the store and its snapshots.

Source: `src/diagnostics.x:182`

Diagnostics.reset

```
void Diagnostics.reset(Diagnostics diag)
```

Clears stored entries and limit state while preserving configuration.

Source: `src/diagnostics.x:62`

Public types

Type	Kind	Summary
`Diagnostics`	struct	Collects diagnostic entries and optionally streams them.
`DiagnosticsHold`	struct	Records where `Diagnostics.hold` stopped streaming.

Diagnostics

```
typedef struct Diagnostics { Array entries; Map published; Compiler printer; int limit; int count, limit_notified; } *Diagnostics
```

Collects diagnostic entries and optionally streams them. A value is valid after `Diagnostics.new`. The store and backing `Array` are `Scope`-owned; entry `Lists` and immutable children retain the lifetime of their producing canonical-value pools. A borrowed `printer` must remain valid while it is installed.

Source: `src/diagnostics.x:17`

DiagnosticsHold

```
typedef struct DiagnosticsHold { Compiler printer; int entries, count, limit_notified; } DiagnosticsHold
```

Records where `Diagnostics.hold` stopped streaming.

Source: `src/diagnostics.x:26`

Design notes

Each report has one ordered entry. The store borrows its printer, while canonical entry values retain their producing-pool lifetimes.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/expressions-reports.x

Expression typing diagnostics.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/fields.x

Statements that copy or set named fields.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/format.x

Code formatting helpers for the x2c compiler.

Functions

Function	Summary
`Compiler.code_pretty_string`	Returns a canonical formatted C `String` for an emitted token `List`.

Compiler

Compiler.code_pretty_string

```
char *Compiler.code_pretty_string(Compiler c, List code, String output_file)
```

Returns a canonical formatted C `String` for an emitted token `List`. Token order and `code` are unchanged. Braces indent by two spaces, semicolons break lines only outside parentheses, and preprocessor tokens occupy their own lines. An empty `List` returns the empty `String`. Emitted `src-at` markers carry existing compiler origin IDs; zero restores `output_file` at its physical line.

Raises: `<size-limit>` or `<alloc-fail>` while materializing the result.

Source: `src/format.x:34`

Design notes

Converts emitted token `Lists` to pretty or compact C text without changing token order. Parenthesis depth suppresses statement breaks within expressions, and `Buffer` materializes the final `String`.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/frontend.x

Configured compiler sessions and source units.

Functions

Function	Summary
`Frontend.load_support`	Loads process-owned collection support and the native modules `request` names before units, and selects those modules, in order, for its compile-time calls.
`Frontend.new`	Borrows a configured request for sequential units.
`Frontend.open`	Runs the source stages.
`Frontend.open_reporting`	Runs the source stages for a command that reports to stderr.
`Frontend.open_session`	Opens an empty submission unit with the ordinary runtime prelude.
`Frontend.preload_macro_libraries`	Evaluates the compile-time libraries and installs the compiler surface's own definitions, once for the active build or translation target.

Function	Summary
`Frontend.start`	Tokenizes one input into a fresh unit with its own isolated `Context`.
`ParsedUnit.close`	Releases the unit after its caller has inspected or exported its results.
`ParsedUnit.collect`	Collects symbols and retains preprocessor outputs for adapter inspection.
`ParsedUnit.parse`	Parses a collected unit, retaining both its AST and unsuccessful reports.

`Frontend`

Frontend.load_support

```
void Frontend.load_support(CliRequest request)
```

Loads process-owned collection support and the native modules `request` names before units, and selects those modules, in order, for its compile-time calls.

Source: `src/frontend.x:532`

Frontend.new

```
Frontend Frontend.new(CliRequest request)
```

Borrows a configured request for sequential units. The request and this session must outlive its units. Initialize process support above any temporary command Context before creating a session inside that Context.

Source: `src/frontend.x:543`

Frontend.open

```
int Frontend.open(Frontend f, String filename, ParsedUnit &unit)
```

Runs the source stages. On either result, the caller must close the unit.

Source: `src/frontend.x:94`

Frontend.open_reporting

```
int Frontend.open_reporting(Frontend f, String filename, ParsedUnit &unit)
```

Runs the source stages for a command that reports to stderr. Collection and parsing print each diagnostic as it is reported; a tokenizing failure prints its diagnostics afterward. A failed unit is closed; the caller must close a successful one.

Source: `src/frontend.x:102`

Frontend.open_session

```
int Frontend.open_session(Frontend frontend, ParsedUnit &unit)
```

Opens an empty submission unit with the ordinary runtime prelude. Preload macro libraries first. The caller must close the unit on either result; submissions and inspection results borrow its Context.

Source: `src/frontend.x:116`

Frontend.preload_macro_libraries

```
int Frontend.preload_macro_libraries(Frontend frontend)
```

Evaluates the compile-time libraries and installs the compiler surface's own definitions, once for the active build or translation target. Returns zero after reporting a failed preload, without publishing a partial session.

Source: `src/frontend.x:479`

Frontend.start

```
int Frontend.start(Frontend frontend, String filename, ParsedUnit &unit)
```

Tokenizes one input into a fresh unit with its own isolated `Context`. The caller must close the unit on either result.

Source: `src/frontend.x:126`

`ParsedUnit`

ParsedUnit.close

```
void ParsedUnit.close(ParsedUnit &unit)
```

Releases the unit after its caller has inspected or exported its results.

Source: `src/frontend.x:166`

ParsedUnit.collect

```
int ParsedUnit.collect(ParsedUnit &unit, Frontend frontend)
```

Collects symbols and retains preprocessor outputs for adapter inspection.

Source: `src/frontend.x:132`

ParsedUnit.parse

```
int ParsedUnit.parse(ParsedUnit &p)
```

Parses a collected unit, retaining both its AST and unsuccessful reports.

Source: `src/frontend.x:155`

Public types

Type	Kind	Summary
`Frontend`	struct	Borrows a configured request and owns shared native-preprocessor setup.
`FrontendErrorSink`	callback	Receives borrowed native-preprocessor stderr synchronously during collect.
`ParsedUnit`	struct	Owns one isolated source lifetime, including unsuccessful diagnostics.

Frontend

```
typedef struct Frontend { CliRequest request; List include_dirs; Toolchain toolchain; FrontendErrorSink preprocessor_errors; } *Frontend
```

Borrows a configured request and owns shared native-preprocessor setup. Nested units close in reverse order; process caches outlive the session. The optional stderr sink runs synchronously; units also retain that text.

Source: `src/frontend.x:23`

FrontendErrorSink

```
typedef void (*FrontendErrorSink)(String text)
```

Receives borrowed native-preprocessor stderr synchronously during collect.

Source: `src/frontend.x:17`

ParsedUnit

```
typedef struct ParsedUnit { Context context; Compiler compiler, preprocessor; Map globals, enclosing_var_tags; List ast; String preprocessor_output, preprocessor_errors; int source_lines, generated_symbols; } ParsedUnit
```

Owns one isolated source lifetime, including unsuccessful diagnostics. Results remain borrowed until close; export values that must survive it.

Source: `src/frontend.x:33`

Design notes

Shares source setup, collection, parsing, and unit lifetimes between the command-line compiler and internal tools. Adapters own printing and exit.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/grammar.x`

The source forms that lowering recognizes.

Functions

Function	Summary
`source_any_lambda`	Returns a pattern for either lambda form, with or without captures.
`source_block_content`	Builds block content with its complete source-ordered statements.
`source_call_content`	Returns bare call content with the supplied callee and argument patterns.
`source_cast_content`	Builds cast content with its children unchanged.
`source_catch_handle`	Returns the error handler the parser introduced for the catch arms of the parsed try `node`, or NULL for a try without catches.
`source_commas_content`	Builds comma sequence content from its ordered rows.
`source_composite_content`	Builds braced initializer content from its ordered comma rows.
`source_conditional_statement`	Returns the canonical pattern for conditional statements.
`source_content_pattern`	Returns a source form's content pattern without its typed shell.
`source_declarator_row`	Returns a pattern for a declarator row, with or without its initializer.
`source_expression`	Returns the source expression beneath casts and parentheses with its type.
`source_generic_content`	Builds generic selection content with its declaration and ordered rows.
`source_identifier_content`	Builds identifier content with its bound child records.
`source_literal_content`	Builds literal content with its typed child records.
`source_operator_content`	Builds operator content with its token and source-ordered operands.
`source_operator_expression`	Builds a typed operator expression from its content children.
`source_postfix_content`	Builds postfix update content with its token and operand.
`source_postfix_expression`	Builds a typed postfix update expression from its content children.
`source_retain_catch_handle`	Restores the binder's handler after a caught template rebuild.
`source_return_content`	Builds return content with all bound and normalized fields.
`source_return_none`	Returns the return of no value that ends a function returning `Var`.
`source_return_type`	Returns a return node's target type, or NULL for another form.
`source_slice_content`	Builds slice content while retaining omitted bounds.
`source_string_content`	Builds string content from its ordered text, expression, and cache rows.
`source_va_arg_content`	Builds va-arg content with its declaration and operand.

Functions

source_any_lambda

```
meta List source_any_lambda(void)
```

Returns a pattern for either lambda form, with or without captures.

Source: src/grammar.x:98

source_block_content

```
meta List source_block_content(List fields)
```

Builds block content with its complete source-ordered statements.

Source: src/grammar.x:194

source_call_content

```
meta List source_call_content( Macro call, List callee, List arguments)
```

Returns bare call content with the supplied callee and argument patterns.

Source: src/grammar.x:182

source_cast_content

```
meta List source_cast_content(List parts)
```

Builds cast content with its children unchanged. The first child is a declaration or a semantic type at different stages.

Source: src/grammar.x:140

source_catch_handle

```
meta List source_catch_handle(List node)
```

Returns the error handler the parser introduced for the catch arms of the parsed try `node`, or NULL for a try without catches. The arms read their captures through it, and no source form writes it.

Source: src/grammar.x:226

source_commas_content

```
meta List source_commas_content(List rows)
```

Builds comma sequence content from its ordered rows.

Source: src/grammar.x:147

source_composite_content

```
meta List source_composite_content(List rows)
```

Builds braced initializer content from its ordered comma rows.

Source: src/grammar.x:152

source_conditional_statement

```
meta List source_conditional_statement(void)
```

Returns the canonical pattern for conditional statements. With is normally parsed away; foreach and finally remain accepted canonical compatibility heads.

Source: src/grammar.x:91

source_content_pattern

```
meta List source_content_pattern(Macro shape, List names)
```

Returns a source form's content pattern without its typed shell. Bare-content dispatch arms retain the fixed head the match emitter labels. Inside a shell, write `%(expr ? ${$shape(...)})` instead.

Source: src/grammar.x:177

source_declarator_row

```
meta List source_declarator_row(List fields)
```

Returns a pattern for a declarator row, with or without its initializer.

Source: src/grammar.x:206

source_expression

```
meta Var source_expression(Var value)
```

Returns the source expression beneath casts and parentheses with its type.

Source: src/grammar.x:211

source_generic_content

```
meta List source_generic_content(List parts)
```

Builds generic selection content with its declaration and ordered rows.

Source: src/grammar.x:143

source_identifier_content

```
meta List source_identifier_content(List fields)
```

Builds identifier content with its bound child records.

Source: src/grammar.x:198

source_literal_content

```
meta List source_literal_content(List fields)
```

Builds literal content with its typed child records.

Source: src/grammar.x:202

source_operator_content

```
meta List source_operator_content(List parts)
```

Builds operator content with its token and source-ordered operands. Constructed nonstandard operators retain the same form.

Source: src/grammar.x:161

source_operator_expression

```
meta List source_operator_expression(List type, List parts)
```

Builds a typed operator expression from its content children.

Source: src/grammar.x:164

source_postfix_content

```
meta List source_postfix_content(List parts)
```

Builds postfix update content with its token and operand.

Source: src/grammar.x:168

source_postfix_expression

```
meta List source_postfix_expression(List type, List parts)
```

Builds a typed postfix update expression from its content children.

Source: src/grammar.x:171

source_retain_catch_handle

```
meta List source_retain_catch_handle(List rebuilt, List handle)
```

Restores the binder's handler after a caught template rebuild. The source form has no hole for this derived identity.

Source: src/grammar.x:233

source_return_content

```
meta List source_return_content(List fields)
```

Builds return content with all bound and normalized fields.

Source: src/grammar.x:190

source_return_none

```
meta List source_return_none(void)
```

Returns the return of no value that ends a function returning `Var`.

Source: src/grammar.x:85

source_return_type

```
meta List source_return_type(List node)
```

Returns a return node's target type, or NULL for another form. The binder supplies the type; the source form does not write it.

Source: `src/grammar.x:79`

source_slice_content

```
meta List source_slice_content(List parts)
```

Builds slice content while retaining omitted bounds.

Source: `src/grammar.x:150`

source_string_content

```
meta List source_string_content(List rows)
```

Builds string content from its ordered text, expression, and cache rows. The parser and resolver share this shape.

Source: `src/grammar.x:157`

source_va_arg_content

```
meta List source_va_arg_content(List parts)
```

Builds va-arg content with its declaration and operand.

Source: `src/grammar.x:145`

Design notes

Each macro's body is the source form a lowering receives, so a lowering recognizes its input with `case` on the macro instead of spelling the parsed node. The accessors read the parts of a parsed node that no source form writes, such as a handler the parser introduced.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/lambdas.x`

Lambda parsing and capture resolution.

Functions

Function	Summary
`Compiler.begin_lambda_captures`	Opens lexical capture resolution while a lambda body is parsed or bound.
`Compiler.bind_lambda_expression`	Binds a constructed lambda through the lexical capture operations used by source literals.
`Compiler.capture_lambda_identifier`	Resolves an automatic identifier through each enclosing lambda's captures.
`Compiler.check_lambda_captures`	Rejects writes and reference access to read-only snapshot bindings.
`Compiler.end_lambda_captures`	Finishes the active lambda's captures in first-use order.
`Compiler.lambda_capture_required`	Reports whether the active lambda still needs to capture a binding.
`Compiler.lambda_param_types`	The parameter types of a lambda's function signature, keeping typed declarators; a bare parameter is a `Var`.
`Compiler.lift_lambda_types`	Moves a function's local type definitions reached by a lambda before that function, before cells and functions lower.
`Compiler.parse_lambda_literal`	Parses a `%!(...) => ...` literal and returns its typed lambda expression.
`Compiler.respell_captures`	Gives the captures of `binding` that the enclosing lambdas already recorded the C spelling `binding` emits under.

`Compiler`

Compiler.begin_lambda_captures

```
void Compiler.begin_lambda_captures( Compiler c, List references, List supplied)
```

Opens lexical capture resolution while a lambda body is parsed or bound. `references` names explicitly shared surrounding bindings; `supplied` contains canonical capture rows supplied by constructed syntax. Evolving rows live in semantic binding facts so macro transactions restore them.

Source: `src/lambdas.x:454`

Compiler.bind_lambda_expression

```
List Compiler.bind_lambda_expression( Compiler c, Type type, List parameters, List supplied, List body)
```

Binds a constructed lambda through the lexical capture operations used by source literals. Parameter declarations keep their existing declarators; supplied canonical capture rows retain their value or reference mode.

Source: `src/lambdas.x:332`

Compiler.capture_lambda_identifier

```
List Compiler.capture_lambda_identifier(Compiler c, List binding, Type type)
```

Resolves an automatic identifier through each enclosing lambda's captures. Fresh captured bindings keep sibling snapshots independent of shared-cell rewriting. Reference captures preserve qualifiers; snapshots of reference parameters copy their current referents.

Source: `src/lambdas.x:485`

Compiler.check_lambda_captures

```
void Compiler.check_lambda_captures(Compiler c, List ast)
```

Rejects writes and reference access to read-only snapshot bindings. The body has already resolved identifiers and call arguments. Templates defer this check until expansion; nested lambdas check their own bodies.

Source: `src/lambdas.x:639`

Compiler.end_lambda_captures

```
List Compiler.end_lambda_captures(Compiler c)
```

Finishes the active lambda's captures in first-use order.

Source: `src/lambdas.x:463`

Compiler.lambda_capture_required

```
int Compiler.lambda_capture_required(Compiler c, List binding)
```

Reports whether the active lambda still needs to capture a binding.

Source: `src/lambdas.x:472`

Compiler.lambda_param_types

```
List Compiler.lambda_param_types(Compiler c, List entries)
```

The parameter types of a lambda's function signature, keeping typed declarators; a bare parameter is a `Var`.

Source: `src/lambdas.x:291`

Compiler.lift_lambda_types

```
List Compiler.lift_lambda_types(Compiler c, List function)
```

Moves a function's local type definitions reached by a lambda before that function, before cells and functions lower. Their semantic tags stay the same; emitted names keep lexical scopes distinct.

Source: `src/lambdas.x:24`

Compiler.parse_lambda_literal

```
List Compiler.parse_lambda_literal(Compiler c)
```

Parses a `%!(...) => ...` literal and returns its typed lambda expression. Parameter bindings are in a new `Sym` scope, bodies use `Var` as the active return type, and capture rows come from `semantic_binding_facts`. Capturing lambdas have type `Func`; noncapturing lambdas retain a native function type.

Source: `src/lambdas.x:155`

Compiler.respell_captures

```
void Compiler.respell_captures(Compiler c, List binding)
```

Gives the captures of `binding` that the enclosing lambdas already recorded the C spelling `binding` emits under.

Source: `src/lambdas.x:621`

Design notes

A source lambda and a lambda that a macro or transform constructs bind through the same capture operations, which record each captured binding and its value or reference mode in semantic binding facts. The rows leave here resolved and in first-use order; `lambda-functions.x` lowers them.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/linked-meta.x

Shipped `meta` code compiled into the compiler.

Functions

Function	Summary
access_postfix	Stores the updated element and returns its previous value.
access_prefix	Stores and returns the incremented or decremented element.
access_read	Calls the getter bracket admission selected.
access_store	Selects the adopted collection setter for indexed assignment, or a participant's `setindex` member.
access_update	Selects collection compound updates and checks accepted operands, or a participant's `updateindex` member.
delegate_member	Rebuilds a call that finds no member on its receiver through the one delegate field path that provides the method.
destructure_declaration	Declares the names of `T (a, b) = source;` and then reads the source into them, or declares each target of `(T1 a, T2 b) = source;` initialized from its element.
destructure_statement	Reads the source of `(a, b) = source;` into its targets in a block.
destructure_value	Reads the source of `(a, b) = source` into its targets, and keeps the source in a result local as the expression's value.
dynamic_binary	Boxes both operands of an arithmetic, shift, or bitwise operator for `Var.binary`, after rejecting an operand that is neither a number nor, for `+` beside text, text.
dynamic_change	Adds or subtracts one through the `Var` update helpers; the postfix forms return the value before the change.
dynamic_compound	Updates a `Var` or numeric lvalue with a `Var` or numeric operand through the update helper of its storage.
dynamic_unary	Rejects a unary numeric operator on a `Var`.

Function	Summary
lambda_cells	Moves each local and parameter of the function node that a lambda in its body shares by reference into a Scope cell, so the binding outlives the call: a parameter is copied into its cell at entry, a local is declared as its cell, and every read goes through the cell.
lambda_closure	Lifts the capturing lambda node to a static function of its parameters and captures, and answers a Func of a record of the captured values and addresses.
lambda_lift	Lifts the noncapturing lambda node to a static function, whose name is its value.
linked_meta_hashes	Returns linked definition hashes and provider source dependencies.
linked_meta_targets	Returns the linked copies, and the runtime builders, by name.
match_lowering	Lowers the bound match node to C text around its subject, patterns, and bodies.
printf_values	Reads the Var values a printf-family call's static format consumes.
protocol_conformance	Places the forwarding members, descriptor thunks, method table, and descriptor registration of one conformance.
raise_lowering	Converts a raise's code and details, or declines a raise whose parts already have their types.
register_rewrite	Registers the decorated translator with its pattern and hole patterns.
string_switch_lowering	Lowers a switch on a String to an integer switch on its selector's index among the labels it owns, or declines another switch.
try_lowering	Lowers the parsed try node to its landing form.

Functions

access_postfix

```
Code access_postfix(Code code)
```

Stores the updated element and returns its previous value.

Source: src/linked-meta.x:325

access_prefix

```
Code access_prefix(Code code)
```

Stores and returns the incremented or decremented element.

Source: src/linked-meta.x:314

access_read

```
Code access_read(Code code)
```

Calls the getter bracket admission selected.

Source: src/linked-meta.x:218

access_store

```
Code access_store(Code code)
```

Selects the adopted collection setter for indexed assignment, or a participant's setindex member.

Source: src/linked-meta.x:254

access_update

```
Code access_update(Code code)
```

Selects collection compound updates and checks accepted operands, or a participant's updateindex member.

Source: src/linked-meta.x:281

delegate_member

```
Code delegate_member(Code code)
```

Rebuilds a call that finds no member on its receiver through the one delegate field path that provides the method.

Source: src/linked-meta.x:430

destructure_declaration

`Code destructure_declaration(Code node)`

Declares the names of `T (a, b) = source;` and then reads the source into them, or declares each target of `(T1 a, T2 b) = source;` initialized from its element.

Source: `src/linked-meta.x:484`

destructure_statement

`Code destructure_statement(Code node)`

Reads the source of `(a, b) = source;` into its targets in a block.

Source: `src/linked-meta.x:514`

destructure_value

`Code destructure_value(Code node)`

Reads the source of `(a, b) = source` into its targets, and keeps the source in a result local as the expression's value.

Source: `src/linked-meta.x:528`

dynamic_binary

`Code dynamic_binary(Code code)`

Boxes both operands of an arithmetic, shift, or bitwise operator for `Var.binary`, after rejecting an operand that is neither a number nor, for + beside text, text. The call is returned lowered, because an operator chain applies this rule once for each term and binding would search the whole remaining chain each time.

Source: `src/linked-meta.x:556`

dynamic_change

`Code dynamic_change(Code code)`

Adds or subtracts one through the `Var` update helpers; the postfix forms return the value before the change.

Source: `src/linked-meta.x:625`

dynamic_compound

`Code dynamic_compound(Code code)`

Updates a `Var` or numeric lvalue with a `Var` or numeric operand through the update helper of its storage.

Source: `src/linked-meta.x:605`

dynamic_unary

`Code dynamic_unary(Code code)`

Rejects a unary numeric operator on a `Var`.

Source: `src/linked-meta.x:641`

lambda_cells

`Code lambda_cells(Code node)`

Moves each local and parameter of the function `node` that a lambda in its body shares by reference into a Scope cell, so the binding outlives the call: a parameter is copied into its cell at entry, a local is declared as its cell, and every read goes through the cell. A lifted lambda's function moves its own. The function binds again only its cells and the statements that hold them; its other statements are kept as they are bound.

Source: `src/linked-meta.x:1120`

lambda_closure

`Code lambda_closure(Code node)`

Lifts the capturing lambda `node` to a static function of its parameters and captures, and answers a `Func` of a record of the captured values and addresses.

Source: `src/linked-meta.x:1042`

lambda_lift

```
Code lambda_lift(Code node)
```

Lifts the noncapturing lambda `node` to a static function, whose name is its value.

Source: `src/linked-meta.x:887`

linked_meta_hashes

```
Map linked_meta_hashes(void)
```

Returns linked definition hashes and provider source dependencies.

Source: `src/linked-meta.x:2392`

linked_meta_targets

```
Map linked_meta_targets(void)
```

Returns the linked copies, and the runtime builders, by name.

Source: `src/linked-meta.x:2156`

match_lowering

```
Code match_lowering(Code node)
```

Lowers the bound match `node` to C text around its subject, patterns, and bodies.

Source: `src/linked-meta.x:1536`

printf_values

```
Code printf_values(Code code)
```

Reads the Var values a printf-family call's static format consumes.

Source: `src/linked-meta.x:1672`

protocol_conformance

```
Code protocol_conformance(Code node)
```

Places the forwarding members, descriptor thunks, method table, and descriptor registration of one conformance.

Source: `src/linked-meta.x:1784`

raise_lowering

```
Code raise_lowering(Code node)
```

Converts a raise's code and details, or declines a raise whose parts already have their types.

Source: `src/linked-meta.x:1847`

register_rewrite

```
Code register_rewrite(Code function, Code pattern, List holes)
```

Registers the decorated translator with its pattern and hole patterns.

Source: `src/linked-meta.x:87`

string_switch_lowering

```
Code string_switch_lowering(Code node)
```

Lowers a switch on a String to an integer switch on its selector's index among the labels it owns, or declines another switch.

Source: `src/linked-meta.x:1945`

try_lowering

```
Code try_lowering(Code node)
```

Lowers the parsed try `node` to its landing form.

Source: `src/linked-meta.x:2100`

Design notes

Generated by tools/gen-linked-meta.sh; do not edit. Bodied `meta` functions and initialized static values from shipped source modules appear without their `meta` markers, so their functions and initializers execute as native code. `Compiler.bind_linked_meta` binds an included definition to its copy when its definition hashes agree, and a prelude component's definition always. Public functions with a run-time form bind to their ordinary provider's definitions and have no copy here.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/main.x`

X2c command dispatch.

Functions

Function	Summary
`main`	Initializes x2c and dispatches one command from `argv`.

Functions

main

```
int main(int argc, char **argv)
```

Initializes x2c and dispatches one command from `argv`. `argv[0]` locates the installation. The process status is zero for a successful translation or build, one for compiler or tool failure, and the executed program's status for `run`. Help and version exit with zero, while invalid CLI and preflight input exit with status two. An external command replaces this process and returns its own status.

Source: `src/main.x:140`

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/meta-helper-client.x`

The compiler's side of the project meta helper.

Functions

Function	Summary
`Compiler.begin_meta_unit`	Selects the table of the unit at `filename` for the calls that follow, and resets the unit's `meta static` values before the first one.
`Compiler.loaded_helper_call`	Calls a loaded function with data values and helper-owned selections.
`Compiler.meta_helper_call`	Calls the project `meta` function `name` in the helper with the values `arguments`, for the call at `site`, and returns its result.
`Compiler.report_meta_error`	Reports a meta function's error `message` with `notes` under `category` at the position recorded first in or around `node`, or at `site` when it has none.
`Compiler.stop_meta_helper`	Stops the helper this process runs, which a translation worker does when its units are done and every process does as it ends.
`Compiler.use_meta_helper`	Uses `helper` for ordinary translation, retaining its configuration through process shutdown.
`MetaHelper.check`	Checks that this helper built the provider table needed by `name`.
`MetaHelper.close`	Stops this helper and releases its pipe buffer and configuration maps.
`MetaHelper.new`	Creates an independent helper at `path`, with failures by table index and source paths mapped to their table indices.
`MetaHelper.restart`	Stops the process while retaining its executable and configuration.

Function	Summary
MetaHelper.retain_executable	Retains this helper's executable independently of later cache rebuilds.

Compiler

Compiler.begin_meta_unit

```
void Compiler.begin_meta_unit(Compiler c, String filename)
```

Selects the table of the unit at `filename` for the calls that follow, and resets the unit's `meta static` values before the first one.

Source: `src/meta-helper-client.x:491`

Compiler.loaded_helper_call

```
Var Compiler.loaded_helper_call( Compiler c, String name, Token site, List arguments, String provider)
```

Calls a loaded function with data values and helper-owned selections. Selectors resolve inside the helper. No address crosses its pipes.

Source: `src/meta-helper-client.x:109`

Compiler.meta_helper_call

```
Var Compiler.meta_helper_call( Compiler c, String name, Token site, List arguments, String provider)
```

Calls the project `meta` function `name` in the helper with the values `arguments`, for the call at `site`, and returns its result. A warning the body makes is reported at `site` and a failure it reports is the call's failure. A body that crashes, exits, or passes the deadline ends the helper, which is reported at `site` and started again for the next call. The call runs in the table of the file that wrote it: an included file's own, which runs the constants it computes as its own translation does, or else the unit's.

Source: `src/meta-helper-client.x:103`

Compiler.report_meta_error

```
void Compiler.report_meta_error( Compiler c, Var node, Symbol category, String message, Token site, List notes, int fatal)
```

Reports a meta function's error `message` with `notes` under `category` at the position recorded first in or around `node`, or at `site` when it has none. A `fatal` report does not return.

Source: `src/meta-helper-client.x:47`

Compiler.stop_meta_helper

```
void Compiler.stop_meta_helper(void)
```

Stops the helper this process runs, which a translation worker does when its units are done and every process does as it ends.

Source: `src/meta-helper-client.x:509`

Compiler.use_meta_helper

```
void Compiler.use_meta_helper(MetaHelper helper)
```

Uses `helper` for ordinary translation, retaining its configuration through process shutdown. Replaces and closes the preceding helper.

Source: `src/meta-helper-client.x:466`

MetaHelper

MetaHelper.check

```
void MetaHelper.check(MetaHelper m, Compiler c, String name, String provider)
```

Checks that this helper built the provider table needed by `name`. Does not invoke the function or reset native state.

Source: `src/meta-helper-client.x:479`

MetaHelper.close

```
void MetaHelper.close(MetaHelper helper)
```

Stops this helper and releases its pipe buffer and configuration maps. A later close is harmless.

Source: `src/meta-helper-client.x:528`

MetaHelper.new

```
MetaHelper MetaHelper.new(String path, Map failures, Map units)
```

Creates an independent helper at `path`, with failures by table index and source paths mapped to their table indices. No process starts until a call needs it. The caller keeps the canonical configuration values.

Source: `src/meta-helper-client.x:443`

MetaHelper.restart

```
void MetaHelper.restart(MetaHelper helper)
```

Stops the process while retaining its executable and configuration. Its next call starts with fresh native state.

Source: `src/meta-helper-client.x:519`

MetaHelper.retain_executable

```
void MetaHelper.retain_executable(MetaHelper helper)
```

Retains this helper's executable independently of later cache rebuilds. The copy lasts until close, including across helper restarts.

Source: `src/meta-helper-client.x:457`

Public types

Type	Kind	Summary
`MetaHelper`	struct	Owns one compiled helper's configuration and process.

MetaHelper

```
typedef struct MetaHelper { String path, snapshot; Map failures, units; Scope scope; pid_t pid, owner; int to, from, table, reset, status; Buffer input; } *MetaHelper
```

Owns one compiled helper's configuration and process. Callers close it after its last call; closing again is harmless.

Source: `src/meta-helper-client.x:71`

Design notes

The project meta build (`meta-project.x`) compiles each unit's `meta` group (`meta-group.x`) into one helper program. This file owns the compiler's side of that helper: starting it, one request and its replies per call, answers to the queries a body asks, and reporting a body that fails, crashes, or runs too long.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/meta-project.x

The project meta build.

Functions

Function	Summary
`Frontend.build_loaded`	Builds one helper exposing the loaded files' public functions.
`Frontend.build_meta`	Builds an independent helper for the meta functions `inputs` reach.
`Frontend.prepare_meta`	Builds the helper that runs the project `meta` functions `inputs` reach, or finds it in the cache, and makes it the one their translation calls.

Frontend

Frontend.build_loaded

```
MetaHelper Frontend.build_loaded(Frontend f, List filenames)
```

Builds one helper exposing the loaded files' public functions. Main's native signature stays inside the helper; argc/argv calls accept an Array of argument Strings through the data protocol.

Source: `src/meta-project.x:92`

Frontend.build_meta

```
MetaHelper Frontend.build_meta(Frontend f, List inputs)
```

Builds an independent helper for the meta functions `inputs` reach. The caller owns its lifetime. Uses the ordinary source, include, package, native object, and cache dependency handling.

Source: `src/meta-project.x:85`

Frontend.prepare_meta

```
void Frontend.prepare_meta(Frontend f, List inputs)
```

Builds the helper that runs the project `meta` functions `inputs` reach, or finds it in the cache, and makes it the one their translation calls. Inputs that reach none use no helper. A group or helper that does not build is reported at the first call that needs it.

Source: `src/meta-project.x:78`

Design notes

Before a translation starts, the bodied `meta` functions its inputs reach are compiled into one helper program linked against the runtime, which the translation then calls (`src/meta-helper-client.x`). They come from ordinary source units reached directly, by includes, or by packages. Meta code under the x2c root's `lib`, `src`, and `etc` is the compiler's own, linked into it.

Each reached source unit gets its own table and native object. Public functions link between units; static helpers stay with their provider. The helper is cached under the x2c cache root, keyed by the SHA-256 of those sources, the compiler stamp, the C compiler's identity, and the flags, and is built again when a file its build read, x2c source or C header, changes.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/operator-ledger.x

Binary operator facts and case projections.

Functions

Function	Summary
`Compiler.derived_member`	Returns the protocol member that derives a comparison operator.
`Compiler.operator_member`	Returns the protocol member corresponding to a direct binary operator.
`Symbol.binary_precedence`	Returns a binary operator's precedence level, or zero for any other `Symbol`.
`Symbol.compound_assignment`	Returns the compound assignment for a binary operator, or zero.
`Symbol.compound_operator`	Returns the binary operator computed by a compound assignment, or zero.

`Compiler`

Compiler.derived_member

```
Symbol Compiler.derived_member(Compiler c, Symbol op)
```

Returns the protocol member that derives a comparison operator. Inequality derives from `equal`, ordered comparisons derive from `compare`, and unsupported operators return zero.

Source: `src/operator-ledger.x:89`

Compiler.operator_member

```
Symbol Compiler.operator_member(Compiler c, Symbol op)
```

Returns the protocol member corresponding to a direct binary operator. Returns zero when the operator has no direct protocol mapping.

Source: `src/operator-ledger.x:79`

`Symbol`

Symbol.binary_precedence

```
int Symbol.binary_precedence(Symbol op)
```

Returns a binary operator's precedence level, or zero for any other `Symbol`. Levels run from 1 for `||` to 10 for the multiplicative operators, so a larger level binds more tightly. `===` and `!==` share the equality level, and `in` the relational level.

Source: `src/operator-ledger.x:59`

Symbol.compound_assignment

```
Symbol Symbol.compound_assignment(Symbol op)
```

Returns the compound assignment for a binary operator, or zero.

Source: `src/operator-ledger.x:71`

Symbol.compound_operator

```
Symbol Symbol.compound_operator(Symbol op)
```

Returns the binary operator computed by a compound assignment, or zero.

Source: `src/operator-ledger.x:65`

Design notes

Each row gives the operator, precedence, compound assignment, protocol member, and whether that member derives a comparison. Zero denotes an absent compound assignment or member. Private case projections generate the public operator lookups below.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/parse-reports.x

Package, binding, and alias diagnostics.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/preprocess.x

C preprocessor directives in x2c source.

Functions

Function	Summary
Compiler.leading_preproc	Returns source-ordered preprocessor nodes in the preceding trivia.
Compiler.note_object_macro	Records the name of the #define line content so a bare atom spelled the same way inside a literal can be flagged and a declaration prefix can be read.
Compiler.note_preprocessor_macros	Records object-like #define names and removes names after #undef.
Compiler.preproc_branch_state	Returns a group's hidden-arm state after its #elif or #else: 2 when the group's state was 1, and 0 otherwise.
Compiler.preproc_conditional_kind	Classifies the preprocessor line text as a conditional directive: <open> for #if, #ifdef, and #ifndef, <branch> for #elif and #else forms, <close> for #endif, or 0 for any other line.
Compiler.preproc_directive	Returns the preprocessor line text without its # and the blanks around the directive.
Compiler.preproc_include_target	Returns the file named by the #include line text, or NULL for any other line.
Compiler.preproc_marker_file	Returns the file that the host preprocessor's line marker text, # N "file" flags, names and sets line to N, or returns NULL for any other line.
Compiler.preproc_open_state	Returns the hidden-arm state of the conditional group that text opens: 2 when C never takes its first arm, because the condition requires a never-defined name or is 0; 1 when C never takes the arms after its first #else, because the condition is exactly ! defined(NAME); and 0 otherwise.
Compiler.preproc_track_arms	Follows the conditional groups open after the preprocessor line text.
Compiler.preproc_within_arms	Returns items inside the conditional arms arms tracked by Compiler.preproc_track_arms: the directives that reopen each group, outermost first, then items, then one #endif per group.
Compiler.scan_conditionals	Records the open groups after each conditional directive of the tokenized unit, and marks layout attributes where written or where a macro expands to one.

Compiler

Compiler.leading_preproc

```
List Compiler.leading_preproc(Compiler c)
```

Returns source-ordered preprocessor nodes in the preceding trivia.

Spaces and comments remain trivia rather than becoming AST nodes.

Source: src/preprocess.x:289

Compiler.note_object_macro

```
void Compiler.note_object_macro(Compiler c, String content)
```

Records the name of the #define line content so a bare atom spelled the same way inside a literal can be flagged and a declaration prefix can be read. The directive after #define is scanned as x2c tokens. An #undef drops the name, so later source reads it as an ordinary identifier.

Source: src/preprocess.x:312

Compiler.note_preprocessor_macros

```
void Compiler.note_preprocessor_macros(Compiler c, List directives)
```

Records object-like #define names and removes names after #undef. Source storage determines visibility independently of directives.

Source: src/preprocess.x:302

Compiler.preproc_branch_state

```
int Compiler.preproc_branch_state(int state)
```

Returns a group's hidden-arm state after its #elif or #else: 2 when the group's state was 1, and 0 otherwise.

Source: src/preprocess.x:79

Compiler.preproc_conditional_kind

```
Symbol Compiler.preproc_conditional_kind(String text)
```

Classifies the preprocessor line text as a conditional directive: <open> for #if, #ifdef, and #ifndef, <branch> for #elif and #else forms, <close> for #endif, or 0 for any other line.

Source: src/preprocess.x:29

Compiler.preproc_directive

```
String Compiler.preproc_directive(String text)
```

Returns the preprocessor line text without its # and the blanks around the directive.

Source: src/preprocess.x:22

Compiler.preproc_include_target

```
String Compiler.preproc_include_target(String text, int &angle)
```

Returns the file named by the #include line text, or NULL for any other line. angle is 1 for a <...> name and 0 otherwise. Text after the name, such as a comment, is ignored.

Source: src/preprocess.x:85

Compiler.preproc_marker_file

```
String Compiler.preproc_marker_file(String text, int &line)
```

Returns the file that the host preprocessor's line marker text, # N "file" flags, names and sets line to N, or returns NULL for any other line.

Source: src/preprocess.x:99

Compiler.preproc_open_state

```
int Compiler.preproc_open_state(String text)
```

Returns the hidden-arm state of the conditional group that text opens: 2 when C never takes its first arm, because the condition requires a never-defined name or is 0; 1 when C never takes the arms after its first #else, because the condition is exactly !defined(NAME); and 0 otherwise.

Source: src/preprocess.x:42

Compiler.preproc_track_arms

```
List Compiler.preproc_track_arms(List arms, String text)
```

Follows the conditional groups open after the preprocessor line `text`. `arms` holds one entry per open group, innermost first, listing the `preproc` nodes that select that group's current arm.

Source: `src/preprocess.x:413`

Compiler.preproc_within_arms

```
List Compiler.preproc_within_arms(List arms, List items)
```

Returns `items` inside the conditional arms `arms` tracked by `Compiler.preproc_track_arms`: the directives that reopen each group, outermost first, then `items`, then one `#endif` per group.

Source: `src/preprocess.x:426`

Compiler.scan_conditionals

```
void Compiler.scan_conditionals(Compiler c)
```

Records the open groups after each conditional directive of the tokenized unit, and marks layout attributes where written or where a macro expands to one.

Source: `src/preprocess.x:146`

Design notes

Preprocessor lines stay in the token stream and become `preproc` nodes, so generated C keeps them where they were written. This module owns what x2c reads from those lines without running a preprocessor: which conditional arms C can reach, and what the directives before each form say about layout and the unit's `#define` names.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/report.x`

Command progress and completion receipts.

Functions

Function	Summary
`report_configure`	Resets process reporting for one command.
`report_duration`	Formats microseconds as integer `us`, rounded whole `ms`, or seconds with two decimal places.
`report_generated`	Writes the muted receipt for `n` generated C files and as many headers, `bytes` in all, when receipts are enabled.
`report_line`	Writes one newline-terminated receipt to stderr when receipts are enabled.
`report_now_us`	Returns monotonic time in microseconds, or zero when the clock read fails.
`report_phase`	Writes a muted phase receipt when receipts are enabled.
`report_progress`	Updates the terminal's transient progress line when transient mode is active.
`report_receipts`	Returns whether stable completion receipts are currently enabled.
`report_size`	Formats bytes as `B`, `KiB`, or `MiB` using binary unit boundaries.
`report_suspend`	Clears the active transient line from stderr, if this process drew one.
`Path.regular_bytes`	Returns the size of a regular file.

Functions

report_configure

```
void report_configure( int quiet, int plain, Symbol color_mode, int verbose, int dry_run, int inspecting)
```

Resets process reporting for one command. Quiet, verbose, dry-run, and inspection modes disable receipts. Transient progress additionally requires terminal stderr and non-plain output. Plain output disables color; automatic color respects terminal capability and `NO_COLOR`.

Source: `src/report.x:48`

report_duration

```
String report_duration(unsigned long microseconds)
```

Formats microseconds as integer `us`, rounded whole `ms`, or seconds with two decimal places.

Source: `src/report.x:258`

report_generated

```
void report_generated(int n, unsigned long long bytes)
```

Writes the muted receipt for `n` generated C files and as many headers, `bytes` in all, when receipts are enabled.

Source: `src/report.x:193`

report_line

```
void report_line(Symbol tone, String line)
```

Writes one newline-terminated receipt to stderr when receipts are enabled. In transient mode the receipt first clears the terminal line, which another process may be drawing, and `line` must be non-NULL.

Source: `src/report.x:165`

report_now_us

```
unsigned long report_now_us(void)
```

Returns monotonic time in microseconds, or zero when the clock read fails. The value measures elapsed time; it is not a wall-clock timestamp.

Source: `src/report.x:239`

report_phase

```
void report_phase( Symbol phase, int count, String noun, int cached, unsigned long microseconds)
```

Writes a muted phase receipt when receipts are enabled. A fully cached nonempty phase is marked up to date; a partial cache reports its cached count, and every receipt includes the elapsed time.

Source: `src/report.x:176`

report_progress

```
void report_progress(Symbol phase, int done, int total, String detail)
```

Updates the terminal's transient progress line when transient mode is active. Updates start after 125 ms and incomplete work is limited to one update per 50 ms. `detail` may be NULL; output is clipped to the configured terminal width and has no newline.

Source: `src/report.x:91`

report_receipts

```
int report_receipts(void)
```

Returns whether stable completion receipts are currently enabled.

Source: `src/report.x:80`

report_size

```
String report_size(unsigned long long bytes)
```

Formats bytes as `B`, `KiB`, or `MiB` using binary unit boundaries. Byte counts are exact; larger units use one decimal place.

Source: `src/report.x:267`

report_suspend

```
void report_suspend(void)
```

Clears the active transient line from stderr, if this process drew one. Forked workers inherit the state but leave the line to their parent.

Source: `src/report.x:153`

`Path`

Path.regular_bytes

```
unsigned long long Path.regular_bytes(String path)
```

Returns the size of a regular file. NULL, a failed `stat`, or a non-regular path returns zero.

Source: `src/report.x:249`

Design notes

The reporter writes only to stderr. Its transient mode uses one carriage- return line and never takes terminal input or changes terminal modes. Processes sharing a terminal, such as parallel Make recipes, take turns owning that line through a lock on the terminal device.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/sourceview.x`

Request-owned source overlays.

Functions

Function	Summary
`SourceView.exists`	Returns readable-file presence, including unsaved new files.
`SourceView.init`	Initializes empty overlays in the current request lifetime.
`SourceView.is_changed`	Returns whether this logical file has an unsaved overlay.
`SourceView.read`	Reads through the request overlay, falling back to a regular disk file.
`SourceView.set`	Stores an immutable snapshot under its logical source path.

`SourceView`

SourceView.exists

```
int SourceView.exists(SourceView sources, String path)
```

Returns readable-file presence, including unsaved new files.

Source: `src/sourceview.x:46`

SourceView.init

```
void SourceView.init(SourceView sources)
```

Initializes empty overlays in the current request lifetime.

Source: `src/sourceview.x:28`

SourceView.is_changed

```
int SourceView.is_changed(SourceView sources, String path)
```

Returns whether this logical file has an unsaved overlay.

Source: `src/sourceview.x:42`

SourceView.read

`int SourceView.read(SourceView sources, String path, String volatile &text)`

Reads through the request overlay, falling back to a regular disk file. The return value distinguishes an empty file from a failed read. Disk text belongs to the calling unit; configured snapshots remain borrowed.

Source: `src/sourceview.x:56`

SourceView.set

`void SourceView.set(SourceView s, String path, String text, int changed)`

Stores an immutable snapshot under its logical source path. Empty text is a present snapshot, not a request to fall back to the disk file.

Source: `src/sourceview.x:36`

Public types

Type	Kind	Summary
`SourceView`	class	Owns configured immutable overlays in the request's active Context.

SourceView

`class SourceView struct { Map overlays, dirty_paths; } *`

Owns configured immutable overlays in the request's active Context. A frontend and its units borrow this view. Text read from disk belongs to the reading unit, so the view never caches that shorter-lived text.

Source: `src/sourceview.x:17`

Design notes

Logical paths keep their identity when an editor supplies unsaved text. Configure this view before opening a frontend unit; its immutable values and Map must outlive every compiler borrowing it.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/stage.x`

The values crossing between meta code and the program.

Functions

Function	Summary
`Compiler.check_meta_call`	Refuses a run-time call to a `meta` function this compiler derived compile-time only.
`Compiler.folded_constant`	Returns the value of the folded constant `node`, or void when part of it is only known at run time.
`Compiler.literal_text_value`	Returns the value of a String or character literal from its source `spelling`, quotes included: adjacent pieces are unescaped on their own and joined, as C does, and a character is its code.
`Compiler.meta_argument`	Returns the value the argument expression `node` of a `$` call passes to a parameter of type `want`, or of no declared type when `want` is NULL: a constant, captured syntax, or the result of another `$` call, which `call` evaluates.
`Compiler.meta_is_comptime_only`	Returns whether `fn` is a `meta` function this compiler recorded as compile-time only, whose runtime form the unit does not emit.
`Compiler.meta_value_expression`	Returns literal code for a compile-time `value`, preserving `declared` when supplied.

`Compiler`

Compiler.check_meta_call

```
void Compiler.check_meta_call(Compiler c, List callee, Token origin)
```

Refuses a run-time call to a `meta` function this compiler derived compile-time only.

Such a function reaches a `Meta` operation, so it exists only inside a compiler and the unit emits no definition for it. Unchecked, the call reaches the linker as an undefined symbol, which names the C spelling and not the source. Another `meta` function may call it: calling one is what makes the caller compile-time only too, so a body being parsed under the marker is left alone.

Source: `src/stage.x:375`

Compiler.folded_constant

```
Var Compiler.folded_constant(Compiler c, Var node)
```

Returns the value of the folded constant `node`, or void when part of it is only known at run time. Literal folding hoists a constant into the compiler cache and leaves `(cache ID)`, a graph of ids over `cons`, `var` and `string` leaves.

Source: `src/stage.x:138`

Compiler.literal_text_value

```
Var Compiler.literal_text_value(String spelling)
```

Returns the value of a String or character literal from its source `spelling`, quotes included: adjacent pieces are unescaped on their own and joined, as C does, and a character is its code.

Source: `src/stage.x:212`

Compiler.meta_argument

```
Var Compiler.meta_argument( Compiler c, List node, Type want, Token site, MetaCall call)
```

Returns the value the argument expression `node` of a `$` call passes to a parameter of type `want`, or of no declared type when `want` is NULL: a constant, captured syntax, or the result of another `$` call, which `call` evaluates. Captured literal syntax reaches a parameter that is not syntax as the literal's value, a `TypeInfo` parameter as the description of its type, and a `Source` parameter with its source text. Anything else is reported at `site`.

Source: `src/stage.x:63`

Compiler.meta_is_comptime_only

```
int Compiler.meta_is_comptime_only(Compiler c, List fn)
```

Returns whether `fn` is a `meta` function this compiler recorded as compile-time only, whose runtime form the unit does not emit.

Source: `src/stage.x:386`

Compiler.meta_value_expression

```
List Compiler.meta_value_expression( Compiler c, Type declared, Var value, Token site)
```

Returns literal code for a compile-time `value`, preserving `declared` when supplied. An Array or Map, at any depth, becomes a literal that builds a fresh collection on every execution; other data comes from the literal cache. A cycle or a collection held twice is reported at `site`. Returns NULL for code Lists or values without a literal representation.

Source: `src/stage.x:240`

Public types

Type	Kind	Summary
`MetaCall`	callback	Evaluates a nested $ call among a call's arguments.

MetaCall

`static typedef Var (*MetaCall)(Compiler c, List expression, Token site)`

Evaluates a nested `$` call among a call's arguments.

Source: `src/stage.x:46`

Design notes

This file owns what crosses between meta code and the program: the arguments a `$` call passes, evaluated from constants, captured syntax, and other `$` calls; the literal code a result becomes; and the rule that a function reaching a compiler operation has no runtime form. A unit's `meta` group and its emission are in `meta-group.x`; the compiler's side of the project helper is in `meta-helper-client.x`.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/statements.x`

X2c statement parsing.

Functions

Function	Summary
`Compiler.finish_return_statement`	Builds a return node for an optional expression without consuming tokens.
`Compiler.parse_block_item`	Parses one block-position declaration, statement, or macro insertion.
`Compiler.parse_block_items`	Parses block items after an already-consumed opening brace through `}` in a new lexical scope.
`Compiler.parse_callable_body`	Parses a callable's outer block in its active parameter scope.
`Compiler.parse_case_body`	Parses a source match arm's guard and body, declaring its typed `types` captures, inside the arm scope that the binder opened.
`Compiler.parse_compound_statement`	Parses a compound body after its opening brace and consumes the closing `}`, returning an origin-anchored `(block ...)` node.
`Compiler.parse_governed`	Parses the statement a control keyword or statement macro governs, or a block item at `<block>`.
`Compiler.parse_match_row_argument`	Parses one MatchRow macro argument with the ordinary match-arm owner.
`Compiler.parse_source_clause`	Parses the clause `kind` of a source `if`, `while`, `do`, `for`, `match`, or `try`, with the punctuation after it, when the binder reaches that clause.
`Compiler.parse_statement`	Parses and binds one statement or statement-position macro at the current token.
`Compiler.with_binding`	Returns the binding of the current identifier when it names a live `with` expression, or NULL.

`Compiler`

Compiler.finish_return_statement

`List Compiler.finish_return_statement(Compiler c, List expr)`

Builds a return node for an optional expression without consuming tokens. A present expression is resolved in the current `Sym` scope and includes the current `return_type` for later conversion.

Source: `src/statements.x:303`

Compiler.parse_block_item

`List Compiler.parse_block_item(Compiler c)`

Parses one block-position declaration, statement, or macro insertion. The caller owns the surrounding scope; a macro insertion may return a `(seq ...)` node containing several block items.

Source: `src/statements.x:702`

Compiler.parse_block_items

```
List Compiler.parse_block_items(Compiler c, int anchor_items)
```

Parses block items after an already-consumed opening brace through `}` in a new lexical scope. `anchor_items` records statement origins.

Source: `src/statements.x:745`

Compiler.parse_callable_body

```
List Compiler.parse_callable_body(Compiler c)
```

Parses a callable's outer block in its active parameter scope.

Source: `src/statements.x:752`

Compiler.parse_case_body

```
List Compiler.parse_case_body(Compiler c, List types)
```

Parses a source match arm's guard and body, declaring its typed `types` captures, inside the arm scope that the binder opened.

Source: `src/statements.x:475`

Compiler.parse_compound_statement

```
List Compiler.parse_compound_statement(Compiler c)
```

Parses a compound body after its opening brace and consumes the closing `}`, returning an origin-anchored `(block ...)` node.

Source: `src/statements.x:740`

Compiler.parse_governed

```
List Compiler.parse_governed(Compiler c, Symbol position)
```

Parses the statement a control keyword or statement macro governs, or a block item at `<block>`. Directives written before it stay in front of it in a `(group DIRECTIVE... STATEMENT)`, which emits without braces, so each directive stays where C read it. A conditional group they open also takes the statement of each later arm and the closing directive, so a statement macro that wraps its body in braces keeps the whole group inside them. A later statement in the same arm follows the governed one, as in C.

Source: `src/statements.x:620`

Compiler.parse_match_row_argument

```
List Compiler.parse_match_row_argument(Compiler c)
```

Parses one MatchRow macro argument with the ordinary match-arm owner.

Source: `src/statements.x:427`

Compiler.parse_source_clause

```
List Compiler.parse_source_clause(Compiler c, Symbol kind)
```

Parses the clause `kind` of a source `if`, `while`, `do`, `for`, `match`, or `try`, with the punctuation after it, when the binder reaches that clause. An absent `else`, `for` header, `catch`, or `finally` clause is NULL.

Source: `src/statements.x:223`

Compiler.parse_statement

`List Compiler.parse_statement(Compiler c)`

Parses and binds one statement or statement-position macro at the current token. On return, the cursor follows the complete statement and any temporary `Sym` scopes opened by the statement have been closed.

Source: `src/statements.x:74`

Compiler.with_binding

`List Compiler.with_binding(Compiler c)`

Returns the binding of the current identifier when it names a live `with` expression, or NULL.

Source: `src/statements.x:206`

Design notes

`Compiler.parse_statement` dispatches each statement form to one production, which consumes its tokens in grammar order and returns its node. Preprocessor directives between statements stay in the tree, so each one emits where C read it. Return checking uses the return type that declaration parsing recorded.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/toolchain.x`

Host preprocessing, compilation, archive, and link actions.

Functions

Function	Summary
`tool_capture`	Runs the host tool `arguments` without a shell, captures both streams, and returns its shell-style status.
`ToolAction.as_program`	Inherits the standard streams and suppresses the failure summary.
`ToolAction.new`	Creates a `Scope`-owned action that reports nonzero status by default.
`ToolAction.run`	Starts and waits for the action, returning its final status.
`ToolAction.start`	Starts the action without a shell and returns a `Scope`-owned execution.
`ToolRun.ready`	Checks whether an execution can be waited without blocking.
`ToolRun.wait`	Waits for an execution, forwards its captured streams, and returns its shell-style status.
`Toolchain.archive_action`	Builds but does not start an `ar rcs` action in object-list order.
`Toolchain.compile_action`	Builds but does not start one C compilation action.
`Toolchain.for_meta`	Creates the toolchain that builds meta code with `cc`.
`Toolchain.link_action`	Builds but does not start a host-compiler link action.
`Toolchain.meta_cc`	Returns the host C compiler that builds meta code, which runs in this process's host: `explicit`, `X2C_META_CC`, `META_CC`, or `cc`, never the target compiler.
`Toolchain.module_action`	Builds but does not start the link of a native module.
`Toolchain.new`	Creates a `Scope`-owned host toolchain with the tools, native options, and verbose and dry-run modes of `request`, and resolves its native layout.
`Toolchain.preprocess`	Runs the configured C preprocessor without a shell.
`Toolchain.preprocess_action`	Builds the preprocessor action whose output identifies an object.
`Toolchain.search_directories`	Returns the directories the C compiler searches for headers and libraries without explicit options, as it reports them, plus the `lib` directory beside each reported `include` directory.

Functions

tool_capture

```
int tool_capture(List arguments, String &output, String &errors)
```

Runs the host tool `arguments` without a shell, captures both streams, and returns its shell-style status. A tool that cannot start returns 127 and leaves the reason in `errors`.

Source: `src/toolchain.x:298`

`ToolAction`

ToolAction.as_program

```
void ToolAction.as_program(ToolAction action)
```

Inherits the standard streams and suppresses the failure summary.

Source: `src/toolchain.x:151`

ToolAction.new

```
ToolAction ToolAction.new( Symbol phase, List arguments, int verbose, int dry_run)
```

Creates a `Scope`-owned action that reports nonzero status by default. The action retains `arguments` without copying them.

Raises: `<alloc-fail>` when the action cannot be allocated.

Source: `src/toolchain.x:141`

ToolAction.run

```
int ToolAction.run(ToolAction action)
```

Starts and waits for the action, returning its final status.

Raises: the same construction and capture-reading causes as `ToolAction.start` and `ToolRun.wait`.

Source: `src/toolchain.x:242`

ToolAction.start

```
ToolRun ToolAction.start(ToolAction action)
```

Starts the action without a shell and returns a `Scope`-owned execution. Verbose and dry-run actions print their quoted argv to stderr. A dry run starts no child.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the execution or argv.

Source: `src/toolchain.x:251`

`ToolRun`

ToolRun.ready

```
int ToolRun.ready(ToolRun t)
```

Checks whether an execution can be waited without blocking. A dry run and a tool that could not start are ready immediately.

Source: `src/toolchain.x:269`

ToolRun.wait

```
int ToolRun.wait(ToolRun execution)
```

Waits for an execution, forwards its captured streams, and returns its shell-style status. Signals return `128 + signal`, a tool that could not start returns 127, and a dry run returns 0. Captured output goes to stderr; program actions inherit standard streams.

Raises: `<io-fail>`, `<bad-arg>`, `<size-limit>`, or `<alloc-fail>` while reading either capture as a `String`.

Source: `src/toolchain.x:279`

`Toolchain`

Toolchain.archive_action

```
ToolAction Toolchain.archive_action( Toolchain t, String output, List objects)
```

Builds but does not start an `ar rcs` action in object-list order. The action does not remove an existing archive, so callers requiring exact membership must unlink `output` before it runs.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the action.

Source: `src/toolchain.x:197`

Toolchain.compile_action

```
ToolAction Toolchain.compile_action( Toolchain t, String source, String object, String depfile,
List gen_dirs)
```

Builds but does not start one C compilation action. Generated include directories precede the x2c include directory and configured compiler arguments. The action requests dependency output at `depfile` with `object` as its target.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the action.

Source: `src/toolchain.x:167`

Toolchain.for_meta

```
Toolchain Toolchain.for_meta(String cc)
```

Creates the toolchain that builds meta code with `cc`. It selects the archiver as `Toolchain.new` does and has no native options.

Source: `src/toolchain.x:80`

Toolchain.link_action

```
ToolAction Toolchain.link_action(Toolchain t, String output, List inputs)
```

Builds but does not start a host-compiler link action. Input order is preserved; configured linker arguments, the matching x2c runtime archive, and `-lm` follow the inputs.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the action.

Source: `src/toolchain.x:207`

Toolchain.meta_cc

```
String Toolchain.meta_cc(String explicit)
```

Returns the host C compiler that builds meta code, which runs in this process's host: `explicit`, `X2C_META_CC`, `META_CC`, or `cc`, never the target compiler.

Source: `src/toolchain.x:132`

Toolchain.module_action

```
ToolAction Toolchain.module_action( Toolchain t, String output, List inputs, String loader)
```

Builds but does not start the link of a native module. The module leaves the x2c runtime unresolved and uses the loading process's copy. macOS links a bundle against `loader`, so a runtime function that executable lacks fails this link. Other hosts link a shared object whose calls to its own functions never bind to a same-named compiler function; a missing runtime function fails when the compiler loads it.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the action.

Source: `src/toolchain.x:221`

Toolchain.new

```
Toolchain Toolchain.new(CliRequest request)
```

Creates a `Scope`-owned host toolchain with the tools, native options, and verbose and dry-run modes of `request`, and resolves its native layout. Tool selection is explicit value, `X2C_CC` or `X2C_AR`, `CC` or `AR`, the installed toolchain record, then `cc` or `ar`. Explicit tool `Strings` and option `Lists` are borrowed.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the toolchain or its canonical layout.

Source: `src/toolchain.x:69`

Toolchain.preprocess

```
int Toolchain.preprocess( Toolchain t, const char *fname, List include_dirs, const char *imacros,
String &output, String &errors, String &dependencies)
```

Runs the configured C preprocessor without a shell. `include_dirs` precede the system directories, and the runtime header directory follows them, so a generated `string.h` never replaces the C library's. System headers keep their include directives when the host supports that mode. `output`, `errors`, and `dependencies` are cleared before use. Source and include paths remain distinct argv elements, and stdout and stderr are captured separately. The temporary depfile is read into `dependencies` when possible and removed on returning paths, including a handled `<io-fail>` while reading it. A non-returning `<bad-arg>`, `<size-limit>`, or `<alloc-fail>` may transfer before removal. Returns the shell-style child status, 127 when the preprocessor cannot start, or -1 for invalid arguments or local setup failure. This operation does not consult `dry_run`.

Raises: `<io-fail>`, `<bad-arg>`, `<size-limit>`, or `<alloc-fail>` while constructing arguments or reading captured text.

Source: `src/toolchain.x:349`

Toolchain.preprocess_action

```
ToolAction Toolchain.preprocess_action( Toolchain t, String source, String output, List gen_dirs)
```

Builds the preprocessor action whose output identifies an object. It uses the compilation's native flags and include order and keeps line markers, so source locations belong to the identity.

Raises: `<alloc-fail>` or `<size-limit>` while constructing the action.

Source: `src/toolchain.x:180`

Toolchain.search_directories

```
List Toolchain.search_directories(Toolchain t)
```

Returns the directories the C compiler searches for headers and libraries without explicit options, as it reports them, plus the `lib` directory beside each reported `include` directory. A compiler that reports none contributes none.

Source: `src/toolchain.x:413`

Public types

Type	Kind	Summary
`ToolAction`	struct	Describes one `Scope`-owned host-tool argv action and its reporting policy.

Type	Kind	Summary
ToolRun	struct	Tracks one Scope-owned started action and its job.
Toolchain	struct	Holds resolved host tools, native layout, and borrowed option Lists.

ToolAction

```
typedef struct ToolAction { Symbol phase, List arguments, int verbose, dry_run, inherit_stdio, report; } *ToolAction
```

Describes one Scope-owned host-tool argv action and its reporting policy. The arguments List is retained without copying, follows its owning canonical pool, and must remain valid through the action's execution.

Source: src/toolchain.x:28

ToolRun

```
typedef struct ToolRun { ToolAction action, Job job, String start_error; } *ToolRun
```

Tracks one Scope-owned started action and its job. A tool that could not start has no job and keeps the message its ToolRun.wait reports. The borrowed action must remain valid through that wait.

Source: src/toolchain.x:37

Toolchain

```
typedef struct Toolchain { String cc, ar, include_dir, runtime_lib, List cpp_args, cc_args, ld_args; int verbose, dry_run, keep_system_includes, inline_macro_expansions; } *Toolchain
```

Holds resolved host tools, native layout, and borrowed option Lists. The record returned by Toolchain.new is Scope-owned. Its Strings follow their owning canonical pools, which may be ancestors; cpp_args, cc_args, and ld_args are retained without copying.

Source: src/toolchain.x:19

Design notes

Resolves host tools and builds typed argv. Every action runs as a lib/process.x command, without a shell.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/translation.x

Synchronous compiler translation requests.

Functions

Function	Summary
translate_request	Translates a request synchronously.

Functions

translate_request

```
int translate_request( CliRequest request, Map unit_dirs, void *context, void (*progress)(void *context, String input, int completed))
```

Translates a request synchronously. Optional per-input directories and a parent-process progress callback let a caller own build state. Without a callback, translation reports its own progress and receipt.

Source: src/translation.x:206

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/type-ledger.x

The compiler's Var tag tables, projected from the ledger.

Functions

Function	Summary
Type.builtin_var_tags	Returns the process-lifetime Var tag of each builtin type.
Type.var_tag_rows	Returns each constant-row tag's (top mask bottom) encoding row.

Type

Type.builtin_var_tags

`Map Type.builtin_var_tags(void)`

Returns the process-lifetime `Var` tag of each builtin type.

Source: `src/type-ledger.x:39`

Type.var_tag_rows

`Map Type.var_tag_rows(void)`

Returns each constant-row tag's `(top mask bottom)` encoding row.

Source: `src/type-ledger.x:42`

Design notes

`lib/var-tags.x` holds the ledger. This unit is the one compiler unit that imports it, so the units that include `type.x` do not load the ledger. Both tables have process lifetime.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

src/type.x

X2c semantic types.

Functions

Function	Summary
List.type_from_ast	Returns the semantic Type represented by a complete (declare ...) AST.
Type.begin_unit	Starts an empty set of source-declared Var rows for one translation unit.
Type.declaration_ast	Returns a complete (declare ...) AST for type and binding.
Type.declaration_base	Returns the declaration specifiers t without the pointer, array, or function modifiers an alias in them contributes, and sets modifiers to those modifiers, for Ast.append_modifiers to give each declarator.
Type.end_unit	Restores the enclosing unit's source-declared Var rows before the current unit's Scope is released.
Type.fixed_var_tag	Returns the process-lifetime Var tag fixed for type, or zero.
Type.parameter_ast	Returns a complete (param ...) AST for type and binding.
Type.parts	Answers Type.parts, declared in lib/meta.x, through type_declaration_parts, which a project's helper shares.
Type.register_var_adoption	Replaces a registered type's inferred Var tag with tag, or with the fixed tag of representation when tag is zero.
Type.register_var_tag	Registers one named type's unit-local Var tag and exact forward converter.
Type.var_converter	Returns the unit-local forward Var converter for the canonical form of type, or NULL.

Function	Summary
Type.var_tag	Returns the unit-local Var tag for type, falling back to its fixed tag.
Type.var_tag_row	Reads the encoding row of tag into top, mask, and bottom and reports whether one exists.

List

List.type_from_ast

```
Type List.type_from_ast(List ast)
```

Returns the semantic Type represented by a complete (declare ...) AST. A declaration with one binding is unwrapped to that binding's Type; multiple bindings return their Types in source order.

Source: src/type.x:27

Type

Type.begin_unit

```
Map Type.begin_unit(void)
```

Starts an empty set of source-declared Var rows for one translation unit. Returns the enclosing unit's rows for restoration at close.

Source: src/type.x:295

Type.declaration_ast

```
List Type.declaration_ast(Type type, List binding)
```

Returns a complete (declare ...) AST for type and binding. A NULL binding produces an abstract declaration.

Source: src/type.x:194

Type.declaration_base

```
List Type.declaration_base(Type t, List &modifiers)
```

Returns the declaration specifiers t without the pointer, array, or function modifiers an alias in them contributes, and sets modifiers to those modifiers, for Ast.append_modifiers to give each declarator. Storage and source specifier text, such as ("_Noreturn"), stay on the base.

Source: src/type.x:175

Type.end_unit

```
void Type.end_unit(Map enclosing)
```

Restores the enclosing unit's source-declared Var rows before the current unit's Scope is released.

Source: src/type.x:304

Type.fixed_var_tag

```
Symbol Type.fixed_var_tag(Type type)
```

Returns the process-lifetime Var tag fixed for type, or zero.

Source: src/type.x:234

Type.parameter_ast

```
List Type.parameter_ast(Type type, List binding)
```

Returns a complete (param ...) AST for type and binding. A NULL binding produces an unnamed parameter.

Source: `src/type.x:202`

Type.parts

`List Type.parts(Type type)`

Answers `Type.parts`, declared in `lib/meta.x`, through `type_declaration_parts`, which a project's helper shares.

Source: `src/type.x:168`

Type.register_var_adoption

`void Type.register_var_adoption(Type type, Type representation, Symbol tag)`

Replaces a registered type's inferred `Var` tag with `tag`, or with the fixed tag of `representation` when `tag` is zero. Missing rows and untagged representations leave the table unchanged.

Source: `src/type.x:283`

Type.register_var_tag

`void Type.register_var_tag(Type t, String name, String converter)`

Registers one named type's unit-local `Var` tag and exact forward converter. The first row for a canonical `Type` wins. A `NULL` type, name, or converter, or no active unit, leaves the table unchanged.

Source: `src/type.x:271`

Type.var_converter

`String Type.var_converter(Type type)`

Returns the unit-local forward `Var` converter for the canonical form of `type`, or `NULL`.

Source: `src/type.x:245`

Type.var_tag

`Symbol Type.var_tag(Type type)`

Returns the unit-local `Var` tag for `type`, falling back to its fixed tag.

Source: `src/type.x:224`

Type.var_tag_row

`int Type.var_tag_row( Symbol tag, unsigned long &top, unsigned long &mask, unsigned long &bottom)`

Reads the encoding row of `tag` into `top`, `mask`, and `bottom` and reports whether one exists. A tag whose decoded form carries a validity clause, an immediate width, or a user registration has no constant row.

Source: `src/type.x:256`

Design notes

Converts compiler declaration syntax to semantic types and owns the translation unit's Var registrations. Structural Type operations live in lib/type.x.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

src/utils.x

System utilities for environment discovery and workers.

Functions

Function	Summary
`fnv_bytes`	Returns `hash` extended with `length` `bytes` by 64-bit FNV-1a.
`fnv_file`	Returns `hash` extended with the contents of the file at `path`.
`x2c_get_executable`	Returns the borrowed resolved executable path, or NULL when unavailable.
`x2c_get_root`	Returns the borrowed repository root, or NULL before it is configured.
`x2c_initialize_command_environment`	Initializes an external command with the compiler identity embedded when it was built.
`x2c_initialize_environment`	Initializes compiler paths and default include `List`s once.
`x2c_set_root`	Overrides the repository root and rebuilds its default include `List`s.
`Compiler.cache_root`	Returns the per-user cache root: `X2C_CACHE_DIR`, `XDG_CACHE_HOME/x2c`, or `~/.cache/x2c`, whichever is set first, or NULL when none is.
`Compiler.command_environment`	Returns the home, driver, and identity supplied to external commands.
`Compiler.command_path`	Returns an executable command in this compiler's libexec, or NULL.
`Compiler.cpp_include_dirs`	Returns the borrowed preprocessor `List` `<root>/lib`, then `<root>/src`.
`Compiler.default_include_dirs`	Returns the borrowed default include `List`, `<root>/include/x2c`.
`Compiler.driver_executable`	Returns the matching compiler executable for a recursive translation.
`Compiler.home_dir`	Returns the discovered or configured home, or NULL when there is none.
`Compiler.home_libexec`	Returns the command directory for this checkout or installed home.
`Compiler.home_packages`	Returns `<home>/packages`, which may not exist, or NULL without a home.
`Compiler.identity`	Returns the active compiler's identity in 16 hexadecimal digits.
`Compiler.is_package_source`	Recognizes a package's `src/` files or its package-named legacy entry.
`Compiler.module_stamp`	Returns the stamp a native module records: `x2c-module-stamp:` and the running compiler's identity.
`Compiler.package_directory`	Returns the package directory that holds `path` below one of `roots`: the root's child on the way to `path`, compared by canonical path, when that child's name is an identifier.
`Compiler.run_command`	Runs an external command with inherited streams and returns its status.
`Compiler.stage_dir`	Returns the directory of a compiler staged at `<home>/builds/<stage>/`, or NULL for any other compiler.
`Diagnostics.driver_error`	Prints `x2c: error: <message>` to stderr and exits with status 2.
`Diagnostics.host_error`	Reports a caught `<not-found>` or `<io-fail>` through `Diagnostics.driver_error` as its operation, path or program, and system reason.
`Env.find_program`	Returns the spelling of the first `PATH` candidate for the program `name` that this process may execute, searched as `execvp` searches, or NULL.
`Job.decode_wait_status`	Decodes the wait status `raw` of a child that has ended.
`Path.content_identity`	Returns the identity a compiler executable at `path` has when it runs: the FNV-1a hash of its bytes, or NULL when `path` is NULL or unreadable.
`Path.identifier_hash`	Hashes unit filename spelling for stable generated C identifiers.
`Path.is_layout_source`	Reports whether `path` names a file in the indentation syntax.
`Path.is_x2c_source`	Reports whether `path` is x2c source: a `.x` or `.xp` file, or a file of any other name whose first line is a shebang, which is a script.
`Path.lock_file`	Locks the file `p`, creating it, and returns a descriptor that holds the lock until it is closed or the process exits.
`Path.publish_files`	Replaces each file named in `outputs`, a List of alternating paths and texts.
`SourceView.package_entry`	Resolves the first readable package entry under `roots`, preferring `.x` to `.xp`, using the importing compiler's source view.

Functions

fnv_bytes

```
uint64_t fnv_bytes(uint64_t hash, const void *bytes, size_t length)
```

Returns `hash` extended with `length` `bytes` by 64-bit FNV-1a.

Source: `src/utils.x:404`

fnv_file

```
uint64_t fnv_file(uint64_t hash, String path, int &ok)
```

Returns `hash` extended with the contents of the file at `path`. A missing or unreadable file clears `ok`.

Source: `src/utils.x:389`

x2c_get_executable

```
String x2c_get_executable(void)
```

Returns the borrowed resolved executable path, or NULL when unavailable.

Source: `src/utils.x:185`

x2c_get_root

```
String x2c_get_root(void)
```

Returns the borrowed repository root, or NULL before it is configured. The root is absolute with symbolic links resolved, the one spelling paths below the home are compared in.

Source: `src/utils.x:182`

x2c_initialize_command_environment

```
void x2c_initialize_command_environment( const char *argv0, String embedded_identity)
```

Initializes an external command with the compiler identity embedded when it was built. A driver-supplied identity must match that compiler.

Source: `src/utils.x:86`

x2c_initialize_environment

```
void x2c_initialize_environment(const char *argv0)
```

Initializes compiler paths and default include `Lists` once. The executable is resolved from the host, `argv0`, or `PATH`. The home is `X2C_HOME` when it is set and not empty; otherwise discovery walks from the executable and then from the current directory to a directory holding `include/` and `etc/compiler-sdk.xlisp`, a source checkout or an installed prefix alike. The root is kept absolute with symbolic links resolved, the one spelling every path below the home is compared in. Without a home the root is the current directory and `Compiler.home_dir` reports none. The compiler identity is taken here, before any later work could observe a replaced executable. An already configured root leaves all state unchanged.

Source: `src/utils.x:79`

x2c_set_root

```
void x2c_set_root(String root)
```

Overrides the repository root and rebuilds its default include `Lists`. The root is resolved as environment setup resolves it. The rebuilt values must remain valid until the next override or the process no longer uses them.

Source: `src/utils.x:168`

`Compiler`

Compiler.cache_root

```
String Compiler.cache_root(void)
```

Returns the per-user cache root: `X2C_CACHE_DIR`, `XDG_CACHE_HOME/x2c`, or `~/.cache/x2c`, whichever is set first, or NULL when none is.

Source: `src/utils.x:497`

Compiler.command_environment

`Map Compiler.command_environment(void)`

Returns the home, driver, and identity supplied to external commands.

Source: `src/utils.x:245`

Compiler.command_path

`String Compiler.command_path(String name)`

Returns an executable command in this compiler's libexec, or NULL.

Source: `src/utils.x:237`

Compiler.cpp_include_dirs

`List Compiler.cpp_include_dirs(void)`

Returns the borrowed preprocessor `List` `<root>/lib`, then `<root>/src`. `<root>/src` is present only when the home has that directory. Returns NULL before environment setup.

Source: `src/utils.x:278`

Compiler.default_include_dirs

`List Compiler.default_include_dirs(void)`

Returns the borrowed default include `List`, `<root>/include/x2c`. Returns NULL before environment setup.

Source: `src/utils.x:272`

Compiler.driver_executable

`String Compiler.driver_executable(void)`

Returns the matching compiler executable for a recursive translation. External commands use the driver's X2C path or their adjacent installed compiler. An unavailable or changed compiler returns NULL.

Source: `src/utils.x:190`

Compiler.home_dir

`String Compiler.home_dir(void)`

Returns the discovered or configured home, or NULL when there is none.

Source: `src/utils.x:206`

Compiler.home_libexec

`String Compiler.home_libexec(void)`

Returns the command directory for this checkout or installed home.

Source: `src/utils.x:225`

Compiler.home_packages

`String Compiler.home_packages(void)`

Returns `<home>/packages`, which may not exist, or NULL without a home.

Source: `src/utils.x:219`

Compiler.identity

```
String Compiler.identity(void)
```

Returns the active compiler's identity in 16 hexadecimal digits. Ordinary compiler startup hashes its executable with FNV-1a; an external command uses the identity embedded from the compiler that built it. Returns NULL if ordinary startup could not read its executable.

Source: `src/utils.x:365`

Compiler.is_package_source

```
int Compiler.is_package_source(String directory, String path)
```

Recognizes a package's `src/` files or its package-named legacy entry. Other files under the package directory are consumers.

Source: `src/utils.x:352`

Compiler.module_stamp

```
String Compiler.module_stamp(void)
```

Returns the stamp a native module records: `x2c-module-stamp:` and the running compiler's identity. Returns NULL when the executable cannot be read. Only the compiler that built a module loads it.

Source: `src/utils.x:371`

Compiler.package_directory

```
String Compiler.package_directory(List roots, String path)
```

Returns the package directory that holds `path` below one of `roots`: the root's child on the way to `path`, compared by canonical path, when that child's name is an identifier. Returns NULL for any other path.

Source: `src/utils.x:318`

Compiler.run_command

```
int Compiler.run_command(String name, List arguments)
```

Runs an external command with inherited streams and returns its status. Resolves only this compiler's libexec and passes literal arguments without a shell. An unavailable command reports an error and exits with status 2.

Source: `src/utils.x:259`

Compiler.stage_dir

```
String Compiler.stage_dir(void)
```

Returns the directory of a compiler staged at `<home>/builds/<stage>/`, or NULL for any other compiler.

Source: `src/utils.x:211`

Diagnostics

Diagnostics.driver_error

```
void Diagnostics.driver_error(const char *message)
```

Prints `x2c: error: <message>` to stderr and exits with status 2. The streams are flushed and `atexit` handlers do not run, so the call is safe inside a `try` body or a catch arm, whose records those handlers would otherwise find still live.

Source: `src/utils.x:476`

Diagnostics.host_error

`void Diagnostics.host_error(List detail)`

Reports a caught `<not-found>` or `<io-fail>` through `Diagnostics.driver_error` as its operation, path or program, and system reason.

Source: `src/utils.x:486`

`Env`

Env.find_program

`String Env.find_program(String name)`

Returns the spelling of the first `PATH` candidate for the program `name` that this process may execute, searched as `execvp` searches, or NULL. An empty entry names the current directory.

Source: `src/utils.x:284`

`Job`

Job.decode_wait_status

`int Job.decode_wait_status(int raw, int &signal)`

Decodes the wait status `raw` of a child that has ended. Returns its exit status, `128 + signal` when a signal ended it, or −1 otherwise, and stores that signal in `signal`, or 0 when no signal ended it.

Source: `src/utils.x:463`

`Path`

Path.content_identity

`String Path.content_identity(String path)`

Returns the identity a compiler executable at `path` has when it runs: the FNV-1a hash of its bytes, or NULL when `path` is NULL or unreadable.

Source: `src/utils.x:379`

Path.identifier_hash

`String Path.identifier_hash(String filename)`

Hashes unit filename spelling for stable generated C identifiers.

Source: `src/utils.x:414`

Path.is_layout_source

`int Path.is_layout_source(String path)`

Reports whether `path` names a file in the indentation syntax.

Source: `src/utils.x:311`

Path.is_x2c_source

`int Path.is_x2c_source(String path)`

Reports whether `path` is x2c source: a `.x` or `.xp` file, or a file of any other name whose first line is a shebang, which is a script.

Source: `src/utils.x:297`

Path.lock_file

```
int Path.lock_file(Path p, int wait)
```

Locks the file `p`, creating it, and returns a descriptor that holds the lock until it is closed or the process exits. Returns −1 when `wait` is zero and another process holds the lock.

Source: `src/utils.x:426`

Path.publish_files

```
void Path.publish_files(List outputs)
```

Replaces each file named in `outputs`, a List of alternating paths and texts. Every text is written and closed in a process-specific sibling of its path before the first rename, so a failed write replaces no destination. The renames then run in order: each destination holds its old contents or its new ones, and a failed rename leaves the earlier destinations replaced.

Raises: `<not-found>` or `<io-fail>`, after removing the siblings.

Source: `src/utils.x:446`

SourceView

SourceView.package_entry

```
String SourceView.package_entry( SourceView sources, List roots, String name, String &directory)
```

Resolves the first readable package entry under `roots`, preferring `.x` to `.xp`, using the importing compiler's source view. Returns its directory through `directory`, or NULL when the package does not exist.

Source: `src/utils.x:334`

Public types

Type	Kind	Summary
`Compiler`	struct	Names the translation state that `src/compiler/compiler.x` defines.
`Diagnostics`	struct	Names the positioned diagnostic store that `src/diagnostics.x` defines.

Compiler

```
typedef struct Compiler *Compiler
```

Names the translation state that `src/compiler/compiler.x` defines.

Source: `src/utils.x:31`

Diagnostics

```
typedef struct Diagnostics *Diagnostics
```

Names the positioned diagnostic store that `src/diagnostics.x` defines.

Source: `src/utils.x:34`

Design notes

Owns the environment the driver modules share: where the executable, home, stage, and packages are, which files are x2c source, the compiler's identity, how a file is locked or replaced, and the driver's fatal error line. Host tools run through `lib/process.x`.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

src/callables/callables.x

Callable values lowered to C helpers.

Functions

Function	Summary
Compiler.adapt_lambda_arg	Adapts a lowered noncapturing lambda helper to a typed callback.
Compiler.capture_environment	Binds the file-static context type `name` with the field rows `fields`, for callable defers and function-pointer `Func` values.
Compiler.context_func_adapter	Builds an adapter with `type`'s public signature that passes its own Func as the first argument of `target`, before the declared arguments.
Compiler.func_parameters	Returns `(SIGNATURE (STORAGE READ)...)` for an adapter of a `Var` function taking the lambda parameter rows `entries`: its Func signature as a cached `List`, and each argument as `_func_reads` reads it from `argv` of the Func `fn`.
Compiler.func_signature	Returns the canonical signature shared by native and meta Func adapters.
Compiler.lift_func_expression	Converts a resolved function-like expression to `Func` when supported.
Compiler.lower_typed_adapter_expr	Lowers a resolved `tadapt` expression to a typed callback helper.
Compiler.maybe_adapt_func_arg	Adapts a direct native function argument when `FuncAdapter` is expected.
Compiler.prepend_setup	Puts `setup` at the entry of a block or typed expression `body`.

`Compiler`

Compiler.adapt_lambda_arg

```
List Compiler.adapt_lambda_arg(Compiler c, List argument, List expected_type)
```

Adapts a lowered noncapturing lambda helper to a typed callback. `argument` must be a resolved helper reference produced by `Compiler.lower_lambda_expr`, optionally wrapped in parentheses. `expected_type` must describe a fixed, nonvariadic function. Unless its parameters and result are already `Var`, a queued static helper converts callback arguments to the lowered lambda's original parameter types before calling it, then converts its `Var` result to the expected return type. Already compatible or unsupported shapes pass through unchanged.

Source: `src/callables/callbacks.x:167`

Compiler.capture_environment

```
List Compiler.capture_environment(Compiler c, List name, List fields)
```

Binds the file-static context type `name` with the field rows `fields`, for callable defers and function-pointer `Func` values. The complete typedef is bound at once, so each field keeps its member type. It binds at file scope whatever local scope is open.

Source: `src/callables/adapters.x:84`

Compiler.context_func_adapter

```
List Compiler.context_func_adapter(Compiler c, Type type, List target)
```

Builds an adapter with `type`'s public signature that passes its own Func as the first argument of `target`, before the declared arguments. The target can read per-instance bindings with `Func.context`.

Source: `src/callables/adapters.x:215`

Compiler.func_parameters

```
List Compiler.func_parameters( Compiler c, List entries, List fn, List argv)
```

Returns `(SIGNATURE (STORAGE READ)...)` for an adapter of a `Var` function taking the lambda parameter rows `entries`: its Func signature as a cached `List`, and each argument as `_func_reads` reads it from `argv` of the Func `fn`.

Source: `src/callables/adapters.x:282`

Compiler.func_signature

`List Compiler.func_signature(Compiler c, Type type)`

Returns the canonical signature shared by native and meta Func adapters.

Source: `src/callables/adapters.x:386`

Compiler.lift_func_expression

`List Compiler.lift_func_expression(Compiler c, List expression)`

Converts a resolved function-like expression to `Func` when supported. Existing `Func` values pass through. Direct fixed functions reuse a file-static handle; other function-typed and function-pointer expressions are evaluated once and copied into a new context-bound `Func`, with null pointers producing null `Func`. Lambda expressions are lowered first, and unrelated expressions pass through unchanged. Public inline functions reach the queued helpers through generated bridge functions.

Source: `src/callables/values.x:13`

Compiler.lower_typed_adapter_expr

`List Compiler.lower_typed_adapter_expr(Compiler c, List expression)`

Lowers a resolved `tadapt` expression to a typed callback helper. `expression` must have the resolved shape `(expr TARGET (tadapt ORIGIN (expr SOURCE (ident BINDING))))`. Compatible helpers are cached by source binding and target type, queued with `Compiler.add_early`, and returned as typed identifiers; other expressions pass through unchanged.

Source: `src/callables/callbacks.x:55`

Compiler.maybe_adapt_func_arg

`List Compiler.maybe_adapt_func_arg( Compiler c, List argument, List expected_type)`

Adapts a direct native function argument when `FuncAdapter` is expected. A noncapturing lambda is lowered to its direct function designator. Other arguments must be resolved direct designators, possibly parenthesized, cast, or addressed; an indirect function-pointer value is rejected. A function already having the adapter's pointee type passes through, and new helpers are cached and queued with `Compiler.add_early`.

Source: `src/callables/adapters.x:126`

Compiler.prepend_setup

`List Compiler.prepend_setup(Compiler c, List body, List setup)`

Puts `setup` at the entry of a block or typed expression `body`.

Source: `src/callables/adapters.x:97`

Design notes

A callable value that C cannot spell directly becomes a static helper queued with `Compiler.add_early`. Every `Func` helper shares one `FuncAdapter` ABI: arguments are read in index order from the argument vector, and the result is boxed into a `Var`. `src/components/lambda-functions.x` lowers lambdas.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/cleanup/cleanup.x

Cleanup regions and the transfers that leave them.

Functions

Function	Summary
`Ast.defer_captures`	Writes captured addresses in the order capture selection established.
`Compiler.lower_cleanup`	Lowers the cleanup regions and transfers of the completed top-level `node`; other nodes pass through.
`Compiler.lower_defer_region`	Returns the region that runs `finalizer` when `body` leaves.
`Compiler.rewrite_defer_list`	Returns `stmts` with each `defer` statement and the statements after it replaced by one region; a list without `defer` returns unchanged.
`Compiler.static_value_is_runtime`	Reports whether the static initializer `value` has to run at runtime, because it calls, allocates, or reads an object other than a function name.

`Ast`

Ast.defer_captures

```
List Ast.defer_captures(List environment, List records)
```

Writes captured addresses in the order capture selection established.

Source: `src/cleanup/defer.x:17`

`Compiler`

Compiler.lower_cleanup

```
List Compiler.lower_cleanup(Compiler c, List node)
```

Lowers the cleanup regions and transfers of the completed top-level `node`; other nodes pass through. Unit normalization calls this at function completion. Expressions cannot contain an unlifted function, so no second unit-tree traversal is needed.

Source: `src/cleanup/transfers.x:64`

Compiler.lower_defer_region

```
List Compiler.lower_defer_region(Compiler c, List body, List finalizer)
```

Returns the region that runs `finalizer` when `body` leaves. Ordinary cleanup statements take the callable chain; lexical transfers and unsupported capture types keep the landing-frame path.

Source: `src/cleanup/defer.x:55`

Compiler.rewrite_defer_list

```
List Compiler.rewrite_defer_list(Compiler c, List stmts)
```

Returns `stmts` with each `defer` statement and the statements after it replaced by one region; a list without `defer` returns unchanged.

Source: `src/cleanup/defer.x:31`

Compiler.static_value_is_runtime

```
int Compiler.static_value_is_runtime(Compiler c, List value, Map runtime)
```

Reports whether the static initializer `value` has to run at runtime, because it calls, allocates, or reads an object other than a function name. `runtime` holds the function-local statics already known to run that way, or is `NULL` at file scope.

Source: `src/cleanup/statics.x:120`

Design notes

A cleanup region is a `defer` body, a region of a landing form such as a `try` body or catch arm, or the rest of a block after a static local whose initializer runs at runtime. Normalization turns each `defer` statement into a region. When a function is complete, each `try` takes the landing form its component answers, and one walk places each region's exits on every transfer that leaves it, rejects a jump into a region, and keeps the locals a `sigsetjmp` landing reads either `volatile` or escaped.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/cli/cli.x`

X2c command-line parsing and presentation.

Functions

Function	Summary
`CliRequest.command_arguments`	Copies the driver's already-read vector, or returns NULL for direct invocation.
`CliRequest.dependency_option`	Returns whether `s` contains a driver-owned dependency option.
`CliRequest.inspects`	Returns whether `request` selects a terminating inspection or dump mode.
`CliRequest.is_builtin`	Reports whether a raw command name belongs to the built-in parser.
`CliRequest.new`	Constructs a request with the command's ordinary CLI defaults.
`CliRequest.package_options`	Reads a package's native response options, expanding literal `{package}` after tokenization.
`CliRequest.package_roots`	Returns the package roots `request` searches: its explicit `--package-dir` and manifest directories in order, then the home's `packages/` directory when it exists.
`CliRequest.parse`	Expands response files and parses `argv[1..]` into one validated request.
`CliRequest.parse_arguments`	Parses already-read arguments, preserving deferred script expansion.
`CliRequest.read_arguments`	Reads initial arguments.
`CliRequest.require_input`	Exits with a driver error unless `input` names a regular file.
`Compiler.version`	Returns the version line `--version` prints, without a newline.
`TargetKind.named`	Returns the target kind spelled `spelling`, or 0 when no kind has that spelling.
`TargetKind.of`	Returns the row of target kind `kind`; any other kind reads as an executable.

`CliRequest`

CliRequest.command_arguments

```
Array CliRequest.command_arguments(int argc, char **argv)
```

Copies the driver's already-read vector, or returns NULL for direct invocation. The caller owns the Array; no word is expanded again.

Source: `src/cli/responses.x:21`

CliRequest.dependency_option

```
int CliRequest.dependency_option(String s)
```

Returns whether `s` contains a driver-owned dependency option. Recognizes `-MMD`, `-MP`, `-MF`, and `-MT` as leading spellings or in a comma-delimited pass-through argument; `NULL` returns zero.

Source: `src/cli/parsing.x:553`

CliRequest.inspects

```
int CliRequest.inspects(CliRequest request)
```

Returns whether `request` selects a terminating inspection or dump mode.

Source: `src/cli/requests.x:65`

CliRequest.is_builtin

```
int CliRequest.is_builtin(const char *word)
```

Reports whether a raw command name belongs to the built-in parser.

Source: `src/cli/parsing.x:655`

CliRequest.new

```
CliRequest CliRequest.new(Symbol command)
```

Constructs a request with the command's ordinary CLI defaults.

Source: `src/cli/requests.x:37`

CliRequest.package_options

```
CliRequest CliRequest.package_options(String path, String package)
```

Reads a package's native response options, expanding literal `{package}` after tokenization. Only native include/define/thread options, ordered archive/library/framework inputs, run-time library search directories, and the `-Wl,` and `-Xlinker` linker pass-throughs are admitted. `cc_args` and `ld_args` serve native actions; no source-preprocessing options are returned.

Source: `src/cli/parsing.x:581`

CliRequest.package_roots

```
List CliRequest.package_roots(CliRequest request)
```

Returns the package roots `request` searches: its explicit `--package-dir` and manifest directories in order, then the home's `packages/` directory when it exists. A root named twice is searched twice and resolves the same entries. Explicit directories are borrowed; the result is a fresh `List` only when the home directory is appended.

Source: `src/cli/requests.x:73`

CliRequest.parse

```
CliRequest CliRequest.parse(int argc, char **argv)
```

Expands response files and parses `argv[1..]` into one validated request. Help and version requests print and exit with status zero. An empty command line prints help and exits with status two; other invalid input reports an error and exits with status two. The result has the active `Scope` and canonical-pool lifetimes described by `CliRequest`.

Raises: `<alloc-fail>` or `<size-limit>` while expanding response files or constructing request values.

Source: `src/cli/parsing.x:255`

CliRequest.parse_arguments

```
CliRequest CliRequest.parse_arguments(Array args)
```

Parses already-read arguments, preserving deferred script expansion.

Source: `src/cli/parsing.x:261`

CliRequest.read_arguments

```
Array CliRequest.read_arguments(int argc, char **argv)
```

Reads initial arguments. Script options defer response-file expansion until parsing, so program arguments remain literal. The caller owns the returned Array.

Source: `src/cli/responses.x:9`

CliRequest.require_input

`void CliRequest.require_input(String input)`

Exits with a driver error unless `input` names a regular file. A wildcard or directory operand adds a note on what to pass instead.

Source: `src/cli/requests.x:95`

`Compiler`

Compiler.version

`String Compiler.version(void)`

Returns the version line `--version` prints, without a newline.

Source: `src/cli/requests.x:62`

`TargetKind`

TargetKind.named

`Symbol TargetKind.named(String spelling, String &refusal)`

Returns the target kind spelled `spelling`, or 0 when no kind has that spelling. `refusal` says why this compiler refuses a known spelling, and is NULL otherwise.

Source: `src/cli/requests.x:18`

TargetKind.of

`TargetKind *TargetKind.of(Symbol kind)`

Returns the row of target kind `kind`; any other kind reads as an executable.

Source: `src/cli/requests.x:28`

Public types

Type	Kind	Summary
`CliRequest`	struct	Holds one compiler command and its command-specific inputs and options.
`TargetKind`	struct	Describes one target kind: its spelling on the command line and in a manifest, the noun a report uses, and the default output file name around the target's name.

CliRequest

```
typedef struct CliRequest { Symbol command, List inputs, run_args, include_dirs, package_dirs, cpp_args; List cc_args, ld_args, native_modules, extensions; Map collection_packages; String out_dir, dep_file, dep_target, manifest; String target, profile, output, build_dir, temps_dir, label, state_seed; String cc, meta_cc, ar, compile_commands, sha256, index, Symbol kind; String diagnostics_file, Symbol color_mode; Symbol dump; int jobs, debugging, verbose, dry_run, quiet, plain, no_deps; int no_phony_deps, compile_only, kind_explicit, save_temps, no_cpp; int max_errors; int source_map, source_facts, live_symbols, cpp_symbols; int force, rebuild, clean; int fatal_warnings, no_interfaces; SourceView sources; } *CliRequest
```

Holds one compiler command and its command-specific inputs and options. `Lists` produced by `CliRequest.parse` preserve CLI order. Copies are shallow: request storage and referenced canonical values keep their producing `Scope` and pool lifetimes.

Source: `src/cli/cli.x:18`

TargetKind

`typedef struct TargetKind { Symbol kind, String spelling, noun, prefix, suffix; } TargetKind`

Describes one target kind: its spelling on the command line and in a manifest, the noun a report uses, and the default output file name around the target's name.

Source: `src/cli/cli.x:44`

Design notes

Each option is one table row: the commands that accept it, its spellings, its help text, and the request field it sets. Parsing and help read the same rows.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/collect/collect.x`

Source-ordered shallow symbol collection and replay.

Functions

Function	Summary
`Compiler.add_linked_meta_provider_hashes`	Retains source proofs and one definition hash per linked provider.
`Compiler.callable_symbols`	Returns the collected source's public type families and the declarations of `names`, including their receiver rows.
`Compiler.collect_package`	Collects a package once and installs its public surface in the current unit.
`Compiler.collect_symbols`	Collects the current translation unit's declarations into `globs`.
`Compiler.complete_meta_hashes`	Completes a linked function's own source hashes before a shallow call.
`Compiler.configure_interfaces`	Creates the process cache and names the directories searched for `.xi` interfaces.
`Compiler.configure_package`	Selects a source file's own package, as its standalone translation does.
`Compiler.declared_adoptions`	Returns the `(adopt ...)` rows that the current file's collected entry declares, in no particular order.
`Compiler.forget_provisional_entries`	Drops the entries collected without declaration defaults while the shared compile-time session was filled, or by a project meta build.
`Compiler.include_function_declarations`	Adds the public functions an included unit itself declares.
`Compiler.include_type_dependencies`	The semantic type rows reached by an ordinary source include.
`Compiler.include_typedef_names`	The typedef names published by the files that the current unit's include of `target` reaches, including its transitive includes.
`Compiler.included_compile_time_effects`	Prepares included compile-time effects after full parsing resets macros.
`Compiler.inline_type_dependencies`	Records the actual type dependencies of typed public inline bodies.
`Compiler.install_included_effects`	Installs the definitions of each include among the directives before the cursor.
`Compiler.interface_text`	Returns the compiler's own collected contribution as interface text, or NULL when the unit has not collected its symbols or the compiler's identity is unknown, since no compiler could replay that interface.
`Compiler.is_prelude_source`	Reports whether the canonical `path` is a compiler prelude source.
`Compiler.linked_meta_definitions_current`	Reports whether a provider's linked definition hashes still match.
`Compiler.linked_meta_provider_current`	Reports whether a linked provider and the files it read are unchanged.
`Compiler.linked_prelude_records`	Returns the interface record text of each prelude component the compiler links, by home-portable path: the datum its `.xi` holds after the compiler identity.

Function	Summary
Compiler.loaded_call_type	Returns the native calling type of a loaded function.
Compiler.loaded_provider	Returns this source's native definitions with their canonical declarations and the public type rows needed to check other units' declarations.
Compiler.meta_provider_hashes	Returns the shared function hashes of an advertised meta provider.
Compiler.module_source	Returns the x2c source at `path`, whose own text is `text`, with its file-scope `module` block replaced by the text of each member it names, in order, or NULL when the text has no block.
Compiler.name_loaded_provider	Names declared bindings through one loaded provider.
Compiler.name_meta_provider_bindings	Selects helper C spellings for public definitions owned by `path`.
Compiler.prelude_interface	Returns the path of the first prelude interface this compiler wrote, or NULL when there is none or the compiler's identity is unknown.
Compiler.prelude_sources	Returns the compiler-owned prelude sources relative to its home.
Compiler.publish_inline_types	Promotes the selected type families after the owning ordinary full parse.
Compiler.publishes_type_family	Reports whether this unit's interface publishes the type `family`.
Compiler.publishes_typedef	Reports whether this unit's selected interface publishes `name`.
Compiler.record_conformance_rows	Replaces the recorded conformance rows of the current file's collected entry with `rows`, which join the declaration map its interface publishes, as `record_generated_symbol` adds its rows.
Compiler.record_generated_symbol	Records one generated public callable in the declaration map that the current file's collected entry contributes, which is the map its interface publishes.
Compiler.refresh_source_collections	Makes the next collection recheck source files and interfaces after an interactive edit.
Compiler.replay_package_imports	Replays package imports and public compile-time definitions in source order.
Compiler.runtime_function_declarations	The runtime umbrella includes each listed module's generated header.
SourceView.resolve_include	The file the include of `target` from `includer_dir` names, searched as collection searches `dirs`, or NULL.

`Compiler`

Compiler.add_linked_meta_provider_hashes

```
void Compiler.add_linked_meta_provider_hashes(Compiler c, Map rows)
```

Retains source proofs and one definition hash per linked provider.

Source: `src/collect/providers.x:110`

Compiler.callable_symbols

```
Map Compiler.callable_symbols(Compiler c, List names)
```

Returns the collected source's public type families and the declarations of `names`, including their receiver rows. Includes contribute their selected declarations, without replaying compile-time effects. Unselected functions and variables do not enter the result.

Source: `src/collect/declarations.x:407`

Compiler.collect_package

```
void Compiler.collect_package(Compiler c, String name, Token token)
```

Collects a package once and installs its public surface in the current unit. `name` resolves through the compiler's registered package roots. A cold entry include closure is parsed in package mode; cached entries replay their

previously collected declarations. Public declarations and protocol rows enter the current symbol state, and dependencies enter the importing compiler. Replay also merges recorded function definitions. `token` locates lookup and public-surface errors.

Source: `src/collect/packages.x:25`

Compiler.collect_symbols

```
Map Compiler.collect_symbols(Compiler c, Map globs)
```

Collects the current translation unit's declarations into `globs`. The compiler must have its filename, text, and include paths prepared. `globs` is the base symbol map and is mutated; a null value starts from an empty map. A prelude unit first replays the runtime contribution, or walks `lib/x2c.x` cold when it must not skip runtime headers. The source and raw-include closure then merge in source order, and the returned map is `globs`. Collection also updates dependencies, function definitions, and macro state. Keyword alias maps and seen-name state are file-local and restored when each file walk ends.

Source: `src/collect/walk.x:18`

Compiler.complete_meta_hashes

```
void Compiler.complete_meta_hashes(Compiler c)
```

Completes a linked function's own source hashes before a shallow call. A lookahead entry never replaces the file's ordinary contribution.

Source: `src/collect/providers.x:93`

Compiler.configure_interfaces

```
void Compiler.configure_interfaces(String out_dir, int cold)
```

Creates the process cache and names the directories searched for `.xi` interfaces. `out_dir` is the current translation output directory, or NULL. Home files mirror their home-relative path under the compiler's stage directory when it runs from `<home>/builds/`, otherwise under the home. A `cold` process reads no interface and still writes its own. Call it before opening any translation unit's Context.

Source: `src/collect/interfaces.x:27`

Compiler.configure_package

```
void Compiler.configure_package(Compiler c, List roots, String filename)
```

Selects a source file's own package, as its standalone translation does. A cold provider does not register an import in its including unit.

Source: `src/collect/walk.x:361`

Compiler.declared_adoptions

```
List Compiler.declared_adoptions(Compiler c)
```

Returns the `(adopt ...)` rows that the current file's collected entry declares, in no particular order. A file that imports a package declares none here: the walk that records them for a cold entry does not install a package's surface.

Source: `src/collect/declarations.x:599`

Compiler.forget_provisional_entries

```
void Compiler.forget_provisional_entries(void)
```

Drops the entries collected without declaration defaults while the shared compile-time session was filled, or by a project meta build. Call once that session is published, and after the meta build's parses.

Source: `src/collect/cache.x:165`

Compiler.include_function_declarations

```
void Compiler.include_function_declarations( Compiler c, String target, int angle, Map available)
```

Adds the public functions an included unit itself declares. Its collected include closure also contains private source includes, which its generated header need not publish. Those cannot supply declarations here. Meta signatures alone do not establish a runtime declaration either. Emission reads the entries collection already installed without invalidating them.

Source: `src/collect/declarations.x:175`

Compiler.include_type_dependencies

```
List Compiler.include_type_dependencies( Compiler c, String target, int angle, Map seen)
```

The semantic type rows reached by an ordinary source include. Each file contributes once to `seen`; the current unit does not contribute through a cycle back to its own still-open header.

Source: `src/collect/declarations.x:135`

Compiler.include_typedef_names

```
List Compiler.include_typedef_names( Compiler c, String target, int angle, Map seen)
```

The typedef names published by the files that the current unit's include of `target` reaches, including its transitive includes. A file already in `seen` is skipped with the files it reaches, and each file reached is added to `seen`. NULL when the include does not resolve to x2c source; a runtime module adds nothing the prelude has not declared.

Source: `src/collect/declarations.x:122`

Compiler.included_compile_time_effects

```
Map Compiler.included_compile_time_effects(Compiler c, Map globs)
```

Prepares included compile-time effects after full parsing resets macros. The ordered cache walk counts each file once and groups effects by the canonical path of the unit's direct include. The parser installs those effects when it reaches that include.

Source: `src/collect/walk.x:597`

Compiler.inline_type_dependencies

```
Map Compiler.inline_type_dependencies(Compiler c, List ast)
```

Records the actual type dependencies of typed public inline bodies.

Source: `src/collect/declarations.x:438`

Compiler.install_included_effects

```
void Compiler.install_included_effects(Compiler c)
```

Installs the definitions of each include among the directives before the cursor.

Source: `src/collect/walk.x:642`

Compiler.interface_text

```
String Compiler.interface_text(Compiler c)
```

Returns the compiler's own collected contribution as interface text, or NULL when the unit has not collected its symbols or the compiler's identity is unknown, since no compiler could replay that interface. A contribution that the interface grammar cannot spell is reported as an `emit` diagnostic.

Source: `src/collect/interfaces.x:266`

Compiler.is_prelude_source

`int Compiler.is_prelude_source(String path)`

Reports whether the canonical `path` is a compiler prelude source.

Source: `src/collect/walk.x:68`

Compiler.linked_meta_definitions_current

`int Compiler.linked_meta_definitions_current(Compiler c, String path)`

Reports whether a provider's linked definition hashes still match. Changes outside definitions do not require a project helper.

Source: `src/collect/providers.x:54`

Compiler.linked_meta_provider_current

`int Compiler.linked_meta_provider_current(Compiler c, String path)`

Reports whether a linked provider and the files it read are unchanged. The linked inventory retains the ordinary collector's dependency proof.

Source: `src/collect/providers.x:8`

Compiler.linked_prelude_records

`Map Compiler.linked_prelude_records(Compiler c)`

Returns the interface record text of each prelude component the compiler links, by home-portable path: the datum its `.xi` holds after the compiler identity. Each is walked cold by a compiler of its own, so the records hold this compiler's collection rather than a table linked into it; the unit's own entries for them are restored afterwards. A record the interface grammar cannot spell is left out.

Source: `src/collect/interfaces.x:368`

Compiler.loaded_call_type

`Type Compiler.loaded_call_type(Compiler c, Type type)`

Returns the native calling type of a loaded function. Array parameters decay to pointers, and a parameter's outer qualifiers affect only its local copy. Keep the declared type for inspection and reload checks.

Source: `src/collect/providers.x:176`

Compiler.loaded_provider

`List Compiler.loaded_provider(Compiler c)`

Returns this source's native definitions with their canonical declarations and the public type rows needed to check other units' declarations.

Source: `src/collect/providers.x:156`

Compiler.meta_provider_hashes

`Map Compiler.meta_provider_hashes(Compiler c, String provider)`

Returns the shared function hashes of an advertised meta provider.

Source: `src/collect/providers.x:106`

Compiler.module_source

```
ModuleSource Compiler.module_source(Compiler c, String path, String text)
```

Returns the x2c source at `path`, whose own text is `text`, with its file-scope `module` block replaced by the text of each member it names, in order, or NULL when the text has no block. Member paths are relative to the file that names them, and their own blocks are spliced in turn. A member's quoted include of a file beside it is spelled from the entry's directory, so it names the same file there.

Source: `src/collect/splice.x:28`

Compiler.name_loaded_provider

```
void Compiler.name_loaded_provider( Compiler c, String path, int index, List provider)
```

Names declared bindings through one loaded provider. Its own source declarations remain authoritative; private bindings never link out.

Source: `src/collect/providers.x:208`

Compiler.name_meta_provider_bindings

```
void Compiler.name_meta_provider_bindings( Compiler c, String path, int index)
```

Selects helper C spellings for public definitions owned by `path`. Bindings, rather than source tokens, keep unrelated private names and fields unchanged. Native supplier declarations retain their C names.

Source: `src/collect/providers.x:141`

Compiler.prelude_interface

```
String Compiler.prelude_interface(void)
```

Returns the path of the first prelude interface this compiler wrote, or NULL when there is none or the compiler's identity is unknown. Its source hashes are not checked.

Source: `src/collect/interfaces.x:244`

Compiler.prelude_sources

```
List Compiler.prelude_sources(void)
```

Returns the compiler-owned prelude sources relative to its home. Their runtime implementations are already linked into the shipped binaries.

Source: `src/collect/walk.x:51`

Compiler.publish_inline_types

```
void Compiler.publish_inline_types(Compiler c, List ast)
```

Promotes the selected type families after the owning ordinary full parse. The header and Xi then use the same complete family selection.

Source: `src/collect/declarations.x:482`

Compiler.publishes_type_family

```
int Compiler.publishes_type_family(Compiler c, List family)
```

Reports whether this unit's interface publishes the type `family`.

Source: `src/collect/declarations.x:227`

Compiler.publishes_typedef

```
int Compiler.publishes_typedef(Compiler c, String name)
```

Reports whether this unit's selected interface publishes `name`.

Source: `src/collect/declarations.x:222`

Compiler.record_conformance_rows

```
void Compiler.record_conformance_rows(Compiler c, Map rows)
```

Replaces the recorded conformance rows of the current file's collected entry with `rows`, which join the declaration map its interface publishes, as `record_generated_symbol` adds its rows. A row an earlier cold walk recorded may sit in an earlier map. The cache retains each key and value.

Source: `src/collect/declarations.x:575`

Compiler.record_generated_symbol

```
void Compiler.record_generated_symbol( Compiler c, String name, Type signature)
```

Records one generated public callable in the declaration map that the current file's collected entry contributes, which is the map its interface publishes. A file without a collected declaration map records nothing. The cache retains `signature`.

Source: `src/collect/declarations.x:555`

Compiler.refresh_source_collections

```
void Compiler.refresh_source_collections(void)
```

Makes the next collection recheck source files and interfaces after an interactive edit. Drops cache indexes, retaining their backing rows: already-open units may still use those declarations.

Source: `src/collect/cache.x:147`

Compiler.replay_package_imports

```
void Compiler.replay_package_imports( Compiler c, Map globs, Map rows, Array exports)
```

Replays package imports and public compile-time definitions in source order. The shadow borrows the unit's macro state and package registries. The full parse collects effects for the include line and leaves the unit's own definitions at their source sites.

Source: `src/collect/packages.x:194`

Compiler.runtime_function_declarations

```
void Compiler.runtime_function_declarations(Compiler c, Map available)
```

The runtime umbrella includes each listed module's generated header. Compiler components in the semantic prelude supply no runtime header.

Source: `src/collect/declarations.x:184`

`SourceView`

SourceView.resolve_include

```
String SourceView.resolve_include( SourceView s, List dirs, String includer_dir, String target,
int angle)
```

The file the include of `target` from `includer_dir` names, searched as collection searches `dirs`, or NULL.

Source: `src/collect/walk.x:452`

Design notes

Raw collection scans includes without running cpp. Each cold walk records declaration maps and included paths at their source positions; cache and interface replay consume that same order so declaration precedence

does not depend on whether a file was already collected. A translated unit writes its own contribution beside its generated C as a `.xi` interface, and the runtime prelude is `lib/x2c.xi`. The compiler's own prelude components are linked into it as the records their interfaces would hold.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/compiler/compiler.x`

One x2c unit's translation state and its two parses.

Functions

Function	Summary
`Compiler.absolute_home_path`	Returns the absolute path that a `Compiler.portable_path` spelling names.
`Compiler.add_early`	Appends a declaration the kernel generated to the unit's pending support.
`Compiler.add_init`	Appends the lowered statement `stmt` to the file initialization area `phase`.
`Compiler.add_support`	Appends the declaration `decl` to the unit's pending support as it is: a definition offered to every function rule, or a written one.
`Compiler.add_translation_dependency`	Records a path dependency not embedded in generated C.
`Compiler.anchor_origin`	Wraps a parsed node in the source location of its opening token.
`Compiler.at_completion`	Reports whether the parser is at the private REPL completion marker.
`Compiler.at_word`	Reports whether the current token is the identifier `word`.
`Compiler.cache`	Interns a constant key and returns its stable `(cache id)` reference.
`Compiler.cache_cons_cell`	Caches a cons cell when both parts have immutable cache forms.
`Compiler.cache_literal_list`	Returns a runtime `List` expression for cached compiler-owned syntax.
`Compiler.cache_literal_var`	Interns an immutable value and returns its `(cache id)` reference.
`Compiler.canonical_path`	Returns the spelling that identifies the file at `path`.
`Compiler.close_child`	Retains a child's final diagnostics and closes its owned Lisp session.
`Compiler.collect_compile_time_definition`	Collects macro definitions or keyword aliases before the full source parse.
`Compiler.collect_unit_macro`	Expands the file-scope unit macro at the cursor for collection and reports whether it did.
`Compiler.copy_source_declaration`	Carries declaration metadata with one actual symbol contribution.
`Compiler.define_pattern_binders`	Declares pattern locals and returns their capture-token/binding pairs.
`Compiler.emitted_binding_name`	Returns a binding's selected emitted spelling.
`Compiler.expect`	Requires and consumes the current token type.
`Compiler.finish_collected_declaration`	Records declaration visibility and meta facts, then skips its body.
`Compiler.free_lisp`	Destroys a compiler's owned `Lisp` session, if any.
`Compiler.freeze_declaration_syntax`	Retains exported syntax across translation units and cached interfaces.
`Compiler.fresh_name`	Allocates the next compiler-private C spelling for `stem`.
`Compiler.full_parse`	Parses and types the positioned source against `globs`.
`Compiler.gensym`	Returns a fresh semantic identity for an anonymous aggregate.
`Compiler.mark_completion`	Retags the token beginning at `position` as a private completion marker.
`Compiler.mark_reference_present`	Records a nonnull optional parameter for the current lexical path.
`Compiler.match_capture_value`	Returns the typed expression reading one native match capture slot.
`Compiler.match_pattern_binders`	Returns definite binders from a typed `Match` pattern AST.
`Compiler.match_pattern_is_static`	Reports whether a typed `Match` pattern has a fully static value graph.
`Compiler.match_pattern_value`	Recovers a pattern value graph, using `x2c-dyn` for computed values.

Function	Summary
`Compiler.match_value_is_static`	Reports whether a recovered pattern value graph is fully static.
`Compiler.merge_source_declarations`	Carries declaration metadata beside a completed symbol-map merge.
`Compiler.merge_translation_dependencies`	Merges another translation's dependency rows into this compiler.
`Compiler.new`	Creates a compiler with independent package and generated-name state.
`Compiler.new_shared`	Creates a compiler sharing its owner's package and generated-name state.
`Compiler.next`	Consumes the current token and advances past following trivia.
`Compiler.optional_reference_test`	Returns the optional-reference parameter tested by `condition`, or NULL.
`Compiler.own_diagnostics`	Routes this compiler's diagnostic store through its own printer.
`Compiler.owns_package_source`	Reports whether an x2c source belongs to the current package root.
`Compiler.pattern_binder_declarations`	Builds ordinary declarations from a pattern scope's issued bindings.
`Compiler.peek`	Returns the non-trivia token type `steps` from parser position.
`Compiler.portable_path`	Returns `path` relative to the canonical x2c home when it lies below the home, otherwise `path`.
`Compiler.present_references`	Returns the optional references proven present in this lexical path.
`Compiler.queue_declaration_effect`	Queues a source Lisp form until declaration production needs its state.
`Compiler.read_source`	Reads a source through the request view and retains exact response bytes.
`Compiler.record_inline_function`	Records the header linkage of a public inline definition.
`Compiler.record_origin`	Records a token's location and returns its one-based occurrence.
`Compiler.record_source_declaration`	Records a physical declaration using the binding's actual scope and key.
`Compiler.record_source_reference`	Records a resolved reference without inventing spans for constructed ASTs.
`Compiler.replay_declaration_source`	Binds a retained declaration production at the cursor, then advances past its source.
`Compiler.require_input`	Raises `<incomplete>` when a required grammar item reaches the supplied input boundary.
`Compiler.restore_reference_presence`	Restores the optional-reference facts saved before a lexical path.
`Compiler.return_unit_state`	Returns that state to `owner`, so the next segment starts where this one finished and any Lisp environment this segment created stays alive after the shadow is released.
`Compiler.run_declaration_effects`	Runs pending effects for declaration production or CPP macro evaluation.
`Compiler.select_declaration_defaults`	Selects the owning file's declaration defaults after all its segments.
`Compiler.settle_reference`	Marks the optional reference `binding` present after an `if` when the arm that runs without it cannot fall through: `no` when the true arm proves presence, `yes` otherwise.
`Compiler.shallow_parse`	Collects file-scope declarations into `globals` without parsing bodies.
`Compiler.shallow_parse_overlay`	Collects declarations with reads over `base` then `overlay`.
`Compiler.share_meta_group`	Shares `owner`'s pending `meta` group and the definitions it reads, which belong with the Lisp session that holds the group's stubs.
`Compiler.skip_collected_script_statement`	Skips a collected script statement, or diagnoses one beside `main`.
`Compiler.skip_script_statement`	Moves past one run of a script unit's statement tokens, through a `;` or a closing `}` outside every bracket.
`Compiler.start_collection`	Starts a collection pass with the built-in macros.
`Compiler.take_unit_state`	Takes over `owner`'s macro, object-like `#define`, include, keyword, literal cache, and Lisp state for one segment of a collected file.
`Compiler.take_word`	Consumes the identifier `word` when current and reports whether it matched.
`Compiler.test`	Consumes `type` when current and reports whether it matched.

Function	Summary
`Compiler.thaw_declaration_syntax`	Restores exported syntax in the importing unit's parsing lifetime.
`Compiler.tokenize`	Scans source and positions the compiler at its first non-trivia token.
`Compiler.var`	Boxes the compiler's pointer; the caller keeps the compiler.
`Map.merge_translation_dependency`	Merges one translation dependency, preserving an existing content hash.
`Path.real`	Returns the real path of the existing file at `path`, or NULL when `path` is NULL or does not resolve.
`Path.resolved_absolute`	Returns `Path.absolute(path)`, resolving an existing file through `Path.real`.
`Pending.area`	Returns the storage of the destination `area`.
`Pending.checkpoint`	Returns the mark that `restore` returns every destination to.
`Pending.initializes`	Whether any file initialization area holds a statement.
`Pending.reset`	Gives every destination fresh, empty storage.
`Pending.restore`	Drops the code queued since `mark`.
`Symbol.group_step`	Returns 1 for a token type that opens a delimited group, −1 for one that closes a group, and 0 otherwise.
`Token.after_group`	Returns the first non-trivia token after the group `t` opens, or after `t` when it opens no group.
`Token.group_close`	Returns the token that closes the group `t` opens, `t` itself when it opens no group, or the `eof` token when the group never closes.
`Token.skip_trivia`	Returns the first token at or after `token` that is not a space, comment, or preprocessor line.
`Var.compiler`	Recovers the compiler pointer boxed by `Compiler.var`.

`Compiler`

Compiler.absolute_home_path

```
String Compiler.absolute_home_path(String spelling)
```

Returns the absolute path that a `Compiler.portable_path` spelling names.

Source: `src/compiler/source-files.x:87`

Compiler.add_early

```
void Compiler.add_early(Compiler c, List decl)
```

Appends a declaration the kernel generated to the unit's pending support. It is implementation, so its definitions are written.

Source: `src/compiler/pending.x:5`

Compiler.add_init

```
void Compiler.add_init(Compiler c, Symbol phase, List stmt)
```

Appends the lowered statement `stmt` to the file initialization area `phase`. Bound code that still needs lowering, such as a `String` literal, would reach C as is.

Source: `src/compiler/pending.x:18`

Compiler.add_support

```
void Compiler.add_support(Compiler c, List decl)
```

Appends the declaration `decl` to the unit's pending support as it is: a definition offered to every function rule, or a written one.

Source: `src/compiler/pending.x:11`

Compiler.add_translation_dependency

```
void Compiler.add_translation_dependency(Compiler c, String path)
```

Records a path dependency not embedded in generated C.

Source: `src/compiler/source-files.x:99`

Compiler.anchor_origin

```
List Compiler.anchor_origin(Compiler c, List node, Token token)
```

Wraps a parsed node in the source location of its opening token.

A null node remains null. Macro construction uses its active `m-origin` marker; otherwise a node without a token remains unwrapped.

Source: `src/compiler/source-locations.x:29`

Compiler.at_completion

```
int Compiler.at_completion(Compiler c)
```

Reports whether the parser is at the private REPL completion marker.

Source: `src/compiler/parsing/tokens.x:257`

Compiler.at_word

```
int Compiler.at_word(Compiler c, String word)
```

Reports whether the current token is the identifier `word`. A contextual keyword such as `with` or `as` is an ordinary identifier elsewhere. This query does not consume tokens.

Source: `src/compiler/parsing/tokens.x:194`

Compiler.cache

```
List Compiler.cache(Compiler c, List key)
```

Interns a constant key and returns its stable `(cache id)` reference.

Source: `src/compiler/literal-cache.x:4`

Compiler.cache_cons_cell

```
List Compiler.cache_cons_cell(Compiler c, List head, List tail)
```

Caches a cons cell when both parts have immutable cache forms.

Returns `NULL` when runtime literals are required or either part cannot be represented by the immutable cache graph.

Source: `src/compiler/literal-cache.x:28`

Compiler.cache_literal_list

```
List Compiler.cache_literal_list(Compiler c, List values)
```

Returns a runtime `List` expression for cached compiler-owned syntax.

`values` may contain nested `Lists`, `Strings`, integer `Vars`, and `Symbols`.

Source: `src/compiler/literal-cache.x:90`

Compiler.cache_literal_var

```
List Compiler.cache_literal_var(Compiler c, Var value)
```

Interns an immutable value and returns its `(cache id)` reference. `value` is a `List` of such values, a `String`, a number with its tag, or a `Symbol`; any other value returns NULL.

Source: `src/compiler/literal-cache.x:45`

Compiler.canonical_path

```
String Compiler.canonical_path(Compiler c, String path)
```

Returns the spelling that identifies the file at `path`. Through a source view it is the absolute path, because an unsaved file need not exist on disk; otherwise it is the real path, or `path` itself when that does not resolve.

Source: `src/compiler/source-files.x:17`

Compiler.close_child

```
void Compiler.close_child(Compiler c, Compiler child)
```

Retains a child's final diagnostics and closes its owned Lisp session.

Source: `src/compiler/lifecycle.x:25`

Compiler.collect_compile_time_definition

```
void Compiler.collect_compile_time_definition(Compiler c, int keyword)
```

Collects macro definitions or keyword aliases before the full source parse. The owning source reports malformed definitions during its full parse, which can recover at later forms. Included providers reject them here.

Source: `src/compiler/parsing/collection.x:77`

Compiler.collect_unit_macro

```
int Compiler.collect_unit_macro(Compiler c)
```

Expands the file-scope unit macro at the cursor for collection and reports whether it did. A required expansion reports its errors. A tried expansion that fails keeps nothing and leaves the cursor at the invocation, so collection skips it and the full parse reports it.

Source: `src/compiler/declarations/production.x:66`

Compiler.copy_source_declaration

```
void Compiler.copy_source_declaration( Compiler c, Map target, Map source, List key)
```

Carries declaration metadata with one actual symbol contribution.

Source: `src/compiler/source-locations.x:82`

Compiler.define_pattern_binders

```
List Compiler.define_pattern_binders(Compiler c, List pattern)
```

Declares pattern locals and returns their capture-token/binding pairs.

Source: `src/compiler/patterns.x:89`

Compiler.emitted_binding_name

```
String Compiler.emitted_binding_name(Compiler c, List binding)
```

Returns a binding's selected emitted spelling.

A binding without an explicit emission rename uses its identity spelling.

Source: `src/compiler/binding-facts.x:10`

Compiler.expect

```
Symbol Compiler.expect(Compiler c, Symbol type)
```

Requires and consumes the current token type.

A mismatch reports a parse diagnostic. An active recovery boundary raises `<malformed>`; without one, diagnostic reporting exits.

Source: `src/compiler/parsing/tokens.x:149`

Compiler.finish_collected_declaration

```
void Compiler.finish_collected_declaration( Compiler c, List declaration, Token meta, int native)
```

Records declaration visibility and meta facts, then skips its body. The declaration is already bound by the shared top-level parser.

Source: `src/compiler/parsing/collection.x:99`

Compiler.free_lisp

```
void Compiler.free_lisp(Compiler c)
```

Destroys a compiler's owned `Lisp` session, if any.

A borrowed session is left alive. Owned sessions not freed here are destroyed when the compiler's Scope ends. This is final compiler cleanup: it clears the diagnostic store and the compiler must not be reused. At process exit, root Scope cleanup follows shutdown hooks and canonical pool cleanup. Close explicitly while any native session dependencies live.

Source: `src/compiler/lifecycle.x:138`

Compiler.freeze_declaration_syntax

```
Var Compiler.freeze_declaration_syntax(Compiler c, Var syntax)
```

Retains exported syntax across translation units and cached interfaces. Tokens and origin indices become portable source data; marker-shaped user Lists are escaped so thawing preserves their values.

Source: `src/compiler/declarations/portable-syntax.x:11`

Compiler.fresh_name

```
String Compiler.fresh_name(Compiler c, String stem)
```

Allocates the next compiler-private C spelling for `stem`. Related compilers increment the same per-stem counter.

Source: `src/compiler/names.x:6`

Compiler.full_parse

```
List Compiler.full_parse(Compiler c, Map globs, int generated_symbols)
```

Parses and types the positioned source against `globs`.

The result is a source-ordered top-level AST. This resets macro state and protocol resolution; origins live for the whole unit. `generated_symbols` publishes external adapter signatures before parsing.

Source: `src/compiler/parsing/parse.x:12`

Compiler.gensym

```
List Compiler.gensym(Compiler c)
```

Returns a fresh semantic identity for an anonymous aggregate. Identities are scoped to the compiler's current file and numbered per file, so every process mints the same sequence for one file and two files never share an identity. No emission path prints one.

Source: `src/compiler/names.x:21`

Compiler.mark_completion

```
void Compiler.mark_completion(Compiler c, int position)
```

Retags the token beginning at `position` as a private completion marker.

Source: `src/compiler/parsing/tokens.x:246`

Compiler.mark_reference_present

```
void Compiler.mark_reference_present(Compiler c, List binding)
```

Records a nonnull optional parameter for the current lexical path.

Source: `src/compiler/binding-facts.x:22`

Compiler.match_capture_value

```
List Compiler.match_capture_value(Var binder, int index)
```

Returns the typed expression reading one native match capture slot.

Source: `src/compiler/patterns.x:108`

Compiler.match_pattern_binders

```
List Compiler.match_pattern_binders( Compiler c, List pattern, List &?possible)
```

Returns definite binders from a typed `Match` pattern AST.

When `possible` is non-null, stores every binder appearing on any path.

Source: `src/compiler/patterns.x:78`

Compiler.match_pattern_is_static

```
int Compiler.match_pattern_is_static(Compiler c, List pattern)
```

Reports whether a typed `Match` pattern has a fully static value graph.

Source: `src/compiler/patterns.x:66`

Compiler.match_pattern_value

```
Var Compiler.match_pattern_value(Compiler c, Var node)
```

Recovers a pattern value graph, using `x2c-dyn` for computed values.

Source: `src/compiler/patterns.x:8`

Compiler.match_value_is_static

```
int Compiler.match_value_is_static(Var value)
```

Reports whether a recovered pattern value graph is fully static.

Source: `src/compiler/patterns.x:57`

Compiler.merge_source_declarations

```
void Compiler.merge_source_declarations(Compiler c, Map target, Map source)
```

Carries declaration metadata beside a completed symbol-map merge.

Source: `src/compiler/source-locations.x:93`

Compiler.merge_translation_dependencies

```
void Compiler.merge_translation_dependencies(Compiler c, Map dependencies)
```

Merges another translation's dependency rows into this compiler.

Source: `src/compiler/source-files.x:104`

Compiler.new

```
Compiler Compiler.new(void)
```

Creates a compiler with independent package and generated-name state.

Source: `src/compiler/lifecycle.x:63`

Compiler.new_shared

```
Compiler Compiler.new_shared(Compiler owner)
```

Creates a compiler sharing its owner's package and generated-name state.

Source: `src/compiler/lifecycle.x:66`

Compiler.next

```
void Compiler.next(Compiler c)
```

Consumes the current token and advances past following trivia.

This updates the unmatched-brace stack. An unmatched `}` reports a parse diagnostic, which raises `<malformed>` under recovery and otherwise exits.

Source: `src/compiler/parsing/tokens.x:163`

Compiler.optional_reference_test

```
List Compiler.optional_reference_test( Compiler c, List condition, int &truth)
```

Returns the optional-reference parameter tested by `condition`, or NULL. `truth` is set to whether the condition's true arm proves that the caller supplied an object. Only a direct truth or null test proves presence.

Source: `src/compiler/binding-facts.x:48`

Compiler.own_diagnostics

```
void Compiler.own_diagnostics(Compiler c)
```

Routes this compiler's diagnostic store through its own printer.

Compilers that share a store call this when taking the diagnostic stream back from another compiler.

Source: `src/compiler/lifecycle.x:20`

Compiler.owns_package_source

```
int Compiler.owns_package_source(Compiler c, String path)
```

Reports whether an x2c source belongs to the current package root. Native headers and source outside that root keep their own names.

Source: `src/compiler/source-files.x:26`

Compiler.pattern_binder_declarations

```
List Compiler.pattern_binder_declarations( Compiler c, List bindings, List handle)
```

Builds ordinary declarations from a pattern scope's issued bindings. A catch reads through its handler; a match reads its capture slots.

Source: `src/compiler/patterns.x:115`

Compiler.peek

```
Symbol Compiler.peek(Compiler c, int steps)
```

Returns the non-trivia token type `steps` from parser position.

Zero reads the current token; positive and negative steps count forward and backward through non-trivia tokens. The parser cursor is unchanged.

Source: `src/compiler/parsing/tokens.x:92`

Compiler.portable_path

```
String Compiler.portable_path(String path)
```

Returns `path` relative to the canonical x2c home when it lies below the home, otherwise `path`. Interfaces, macro definitions, retained declarations, and generated identities spell paths this way, so they do not depend on where the home is installed.

Source: `src/compiler/source-files.x:81`

Compiler.present_references

```
List Compiler.present_references(Compiler c)
```

Returns the optional references proven present in this lexical path.

Source: `src/compiler/binding-facts.x:18`

Compiler.queue_declaration_effect

```
void Compiler.queue_declaration_effect( Compiler c, String form, Token first, Token after)
```

Queues a source Lisp form until declaration production needs its state. Files without declaration producers keep ordinary full-parse evaluation. A null `after` replays an included effect without an owning source span.

Source: `src/compiler/declarations/production.x:11`

Compiler.read_source

```
int Compiler.read_source(Compiler c, String path, String volatile &text)
```

Reads a source through the request view and retains exact response bytes.

Source: `src/compiler/source-files.x:6`

Compiler.record_inline_function

```
void Compiler.record_inline_function(Compiler c, List declaration)
```

Records the header linkage of a public inline definition. Semantic function types discard storage, so this declaration fact survives there.

Source: `src/compiler/parsing/collection.x:140`

Compiler.record_origin

```
int Compiler.record_origin(Compiler c, Token token)
```

Records a token's location and returns its one-based occurrence.

A null token returns zero. Callers pass the token that opened a construct, so an `(at N node)` wrapper retains its start for transform diagnostics after `c.token` has reached end of file.

Source: src/compiler/source-locations.x:13

Compiler.record_source_declaration

```
void Compiler.record_source_declaration( Compiler c, List binding, Token first, Token after)
```

Records a physical declaration using the binding's actual scope and key.

Source: src/compiler/source-locations.x:38

Compiler.record_source_reference

```
void Compiler.record_source_reference( Compiler c, List binding, Type type, Token first, Token after)
```

Records a resolved reference without inventing spans for constructed ASTs.

Source: src/compiler/source-locations.x:73

Compiler.replay_declaration_source

```
List Compiler.replay_declaration_source(Compiler c)
```

Binds a retained declaration production at the cursor, then advances past its source. Returns NULL when collection retained no production.

Source: src/compiler/declarations/production.x:127

Compiler.require_input

```
void Compiler.require_input(Compiler c)
```

Raises <incomplete> when a required grammar item reaches the supplied input boundary. The caller sets a token in the current token stream after tokenizing; tokenization clears it. Semantic failures do not call this.

Source: src/compiler/parsing/tokens.x:140

Compiler.restore_reference_presence

```
void Compiler.restore_reference_presence(Compiler c, List before)
```

Restores the optional-reference facts saved before a lexical path.

Source: src/compiler/binding-facts.x:38

Compiler.return_unit_state

```
void Compiler.return_unit_state(Compiler c, Compiler owner)
```

Returns that state to owner, so the next segment starts where this one finished and any Lisp environment this segment created stays alive after the shadow is released.

Source: src/compiler/lifecycle.x:54

Compiler.run_declaration_effects

```
void Compiler.run_declaration_effects(Compiler c)
```

Runs pending effects for declaration production or CPP macro evaluation.

Source: src/compiler/declarations/production.x:27

Compiler.select_declaration_defaults

```
Map Compiler.select_declaration_defaults( Compiler c, String path, Map symbols, Array parts, Map definitions, Map statics)
```

Selects the owning file's declaration defaults after all its segments. The selected signatures join ordinary declarations before protocol and body binding; discarded candidates never bind their bodies. Returns added signatures for the caller to retain in the header-cache lifetime.

Source: `src/compiler/declarations/defaults.x:23`

Compiler.settle_reference

```
void Compiler.settle_reference( Compiler c, List binding, int true_is_present, List yes, List no)
```

Marks the optional reference `binding` present after an `if` when the arm that runs without it cannot fall through: `no` when the true arm proves presence, `yes` otherwise. A missing `no` arm falls through.

Source: `src/compiler/binding-facts.x:31`

Compiler.shallow_parse

```
void Compiler.shallow_parse(Compiler c, Map globals)
```

Collects file-scope declarations into `globals` without parsing bodies.

Source: `src/compiler/parsing/collection.x:4`

Compiler.shallow_parse_overlay

```
void Compiler.shallow_parse_overlay(Compiler c, Map base, Map overlay)
```

Collects declarations with reads over `base` then `overlay`.

Writes go to `overlay`, which captures exactly what this translation contributes above `base`.

Source: `src/compiler/parsing/collection.x:18`

Compiler.share_meta_group

```
void Compiler.share_meta_group(Compiler c, Compiler owner)
```

Shares `owner`'s pending `meta` group and the definitions it reads, which belong with the Lisp session that holds the group's stubs.

Source: `src/compiler/lifecycle.x:34`

Compiler.skip_collected_script_statement

```
int Compiler.skip_collected_script_statement(Compiler c)
```

Skips a collected script statement, or diagnoses one beside `main`. Called after top-level directives establish source visibility.

Source: `src/compiler/parsing/scripts.x:83`

Compiler.skip_script_statement

```
void Compiler.skip_script_statement(Compiler c)
```

Moves past one run of a script unit's statement tokens, through a `;` or a closing `}` outside every bracket. A statement that ends early this way leaves its remainder as the next run, and runs are rejoined in order.

Source: `src/compiler/parsing/scripts.x:97`

Compiler.start_collection

```
void Compiler.start_collection(Compiler c)
```

Starts a collection pass with the built-in macros. It parses no bodies, and an import it reads installs the protocols visible to it when a template or `meta` body first asks for one.

Source: `src/compiler/parsing/collection.x:58`

Compiler.take_unit_state

```
void Compiler.take_unit_state(Compiler c, Compiler owner)
```

Takes over `owner`'s macro, object-like `#define`, include, keyword, literal cache, and Lisp state for one segment of a collected file. Segments are one translation unit, so retained macro bodies index the unit's literal cache and use its Lisp environment.

Source: `src/compiler/lifecycle.x:43`

Compiler.take_word

```
int Compiler.take_word(Compiler c, String word)
```

Consumes the identifier `word` when current and reports whether it matched.

Source: `src/compiler/parsing/tokens.x:199`

Compiler.test

```
inline int Compiler.test(Compiler c, Symbol type)
```

Consumes `type` when current and reports whether it matched.

Source: `src/compiler/parsing/tokens.x:185`

Compiler.thaw_declaration_syntax

```
Var Compiler.thaw_declaration_syntax(Compiler c, Var syntax)
```

Restores exported syntax in the importing unit's parsing lifetime.

Source: `src/compiler/declarations/portable-syntax.x:66`

Compiler.tokenize

```
void Compiler.tokenize(Compiler c, char *text)
```

Scans source and positions the compiler at its first non-trivia token.

The compiler borrows `text` for diagnostics and macro source capture until translation finishes.

Source: `src/compiler/parsing/tokens.x:8`

Compiler.var

```
Var Compiler.var(Compiler c)
```

Boxes the compiler's pointer; the caller keeps the compiler.

Source: `src/compiler/lifecycle.x:5`

Map

Map.merge_translation_dependency

```
void Map.merge_translation_dependency(Map m, String path, Var content_hash)
```

Merges one translation dependency, preserving an existing content hash.

Source: `src/compiler/source-files.x:91`

Path

Path.real

```
String Path.real(String path)
```

Returns the real path of the existing file at `path`, or NULL when `path` is NULL or does not resolve. Each spelling is resolved once per process.

Source: `src/compiler/source-files.x:43`

Path.resolved_absolute

```
String Path.resolved_absolute(String path)
```

Returns `Path.absolute(path)`, resolving an existing file through `Path.real`.

Source: `src/compiler/source-files.x:56`

Pending

Pending.area

```
Array Pending.area(Pending &p, Symbol area)
```

Returns the storage of the destination `area`.

Source: `src/compiler/pending.x:28`

Pending.checkpoint

```
PendingMark Pending.checkpoint(Pending &p)
```

Returns the mark that `restore` returns every destination to.

Source: `src/compiler/pending.x:38`

Pending.initializes

```
int Pending.initializes(Pending &p)
```

Whether any file initialization area holds a statement.

Source: `src/compiler/pending.x:32`

Pending.reset

```
void Pending.reset(Pending &p)
```

Gives every destination fresh, empty storage.

Source: `src/compiler/pending.x:23`

Pending.restore

```
void Pending.restore(Pending &p, PendingMark mark)
```

Drops the code queued since `mark`.

Source: `src/compiler/pending.x:46`

Symbol

Symbol.group_step

```
int Symbol.group_step(Symbol s)
```

Returns 1 for a token type that opens a delimited group, −1 for one that closes a group, and 0 otherwise. Every opener spelling ends in the `(`, `[`, or `{` that its closer matches.

Source: `src/compiler/parsing/tokens.x:214`

Token

Token.after_group

`Token Token.after_group(Token t)`

Returns the first non-trivia token after the group `t` opens, or after `t` when it opens no group. A group that never closes yields `eof`.

Source: `src/compiler/parsing/tokens.x:238`

Token.group_close

`Token Token.group_close(Token t)`

Returns the token that closes the group `t` opens, `t` itself when it opens no group, or the `eof` token when the group never closes.

Source: `src/compiler/parsing/tokens.x:229`

Token.skip_trivia

`Token Token.skip_trivia(Token token)`

Returns the first token at or after `token` that is not a space, comment, or preprocessor line. The stream must end in `eof`.

Source: `src/compiler/parsing/tokens.x:112`

`Var`

Var.compiler

`Compiler Var.compiler(Var value)`

Recovers the compiler pointer boxed by `Compiler.var`.

Source: `src/compiler/lifecycle.x:8`

Public types

Type	Kind	Summary
`Ancestor`	struct	One syntax the parser is inside, which `x2c_enclosing` answers: a function body with its declaration in `node`, and `lowering` set while the transform lowers it, a block item with the token or origin it starts at, or an initialized declarator with its declaration base in `base` and its `(bind ...)` node in `node`.
`Compiler`	struct	Holds mutable state for one source translation.
`Diagnostics`	struct	Names the positioned diagnostic store routed by a `Compiler`.
`GenNames`	struct	Holds scope-owned generated-name state shared by related compilers.
`ModuleSource`	struct	Describes an x2c source whose `module` block names member files: `text` holds the file's own text, `own`, with each member's text in place of the block.
`Pending`	struct	Holds generated code awaiting insertion, by `Pending.area` destination: unit support declarations, which `Compiler.transform` lowers after the unit, then the lowered statements of each file initialization area.
`PendingMark`	struct	Holds each destination's length for `Pending.restore`.
`ScriptUnit`	struct	Describes the `#!` script unit a translation unit was started from: its canonical path, its first line as written, and whether it defines `main` itself.
`SlotWatch`	struct	Names the slots a rewrite's first lowering watches; see "prepared replacements" in macros.x.
`Sym`	struct	Names the scope-owned semantic table belonging to one compiler.
`SymScope`	struct	Holds the three mutable maps that form one lexical semantic scope.
`SymTxn`	struct	Holds one reversible semantic transaction in caller storage.

Ancestor

```
typedef struct Ancestor { Symbol what; Token token; int origin, lowering, callback; List base, node; } Ancestor
```

One syntax the parser is inside, which `x2c_enclosing` answers: a function body with its declaration in `node`, and `lowering` set while the transform lowers it, a block item with the token or origin it starts at, or an initialized declarator with its declaration base in `base` and its `(bind ...)` node in `node`.

Source: `src/compiler/compiler.x:99`

Compiler

```
typedef struct Compiler { String filename, text, root_dir; String package, List package_dirs, package_source_dirs; Map package_roots, package_aliases, package_members, package_effects; Token token; Token input_boundary; Tokenizer tokenizer; Array line_markers; ModuleSource unit_source; List return_type, include_dirs; Map deps; List aggregate_type, macro_stack, declaration_effects, Sym sym; Map evaluated_effects; int expansion_floor; SymScope params; Map key_ids, macros, kw_aliases; List binding_rows, binding_sealed; Map object_macros; List arms; Map arm_stacks; Array layout_marks, packed_marks; Token directives_taken; Map protocols, conforms, protocol_helpers; Map proto_cache; Map adoptions; Map macro_holes; Map local_macro_captures; int macro_application; List lambda_scopes; Array match_types; Map imports; Map included_effects; Map init_tokens, static_init_deps, fn_defs; Array id_keys; String init_fn, fini_fn; Pending pending, int prelude; Map meta_comptime, meta_regions, meta_hashes, meta_calls; struct MetaHelper *meta_helper; Map native_meta; Map project_meta; Array unit_nodes, meta_group; Map meta_group_bound; int meta_build; int load_source; int runtime_inc, runtime_hdrs, collect_protocols, shallow, source_private; int module_declaration; int replaying_declaration; int public_bodies; int interface_provider, signature_only; int collection_native; Map collection_native_files; Array pending_inline_bodies; int layout; int meta_body; int capture_unit; List active_rewrites; List applying; Map rewrite_rules; Map prepared_rewrites; SlotWatch slot_watch; Token meta_statement; int import_protocols; int in_pattern, match_is, runtime_literals, inline_header; int permanent_literals; int builtin_defs, in_proto, macro_count, recovery_depth; int declaration_produced; int needs_exception; int local_macro_capture_scopes; int open_linkage; String fn_name, Diagnostics diagnostics, Array braces, import_stack; Block ancestors; Array placements; String lines_text, Array line_starts; ScriptUnit unit_script, script; Lisp macro_lisp, int borrowed_lisp; int inherited_lisp; GenNames names; Array origins, int origin, source_map; SourceView sources; int source_facts, source_primary; Array source_occurrences; Map source_definitions, source_declarations, source_texts; } *Compiler
```

Holds mutable state for one source translation.

The structure, its state, and its owned compile-time `Lisp` session belong to the current `Scope`. `Compiler.free_lisp` permits early session cleanup; otherwise Scope teardown releases it. All Lisp calls must return first.

Source: `src/compiler/compiler.x:113`

Diagnostics

```
typedef struct Diagnostics *Diagnostics
```

Names the positioned diagnostic store routed by a `Compiler`.

Source: `src/compiler/compiler.x:26`

GenNames

```
typedef struct GenNames { Map counters, adapters, file_scope_owners; int next_binding, issued; } *GenNames
```

Holds scope-owned generated-name state shared by related compilers.

`Compiler.new` initializes every valid instance; callers borrow it from the compiler rather than constructing or freeing it. `next_binding` is the last binding identity issued in the translation unit: collection, shadow, package, and full-parse compilers all draw from it, so a binding number names one live declaration across every symbol table in the unit. Exported rows use their interface's own numbering, which starts at 1; imported templates rebind those references in the consuming unit.

Source: src/compiler/compiler.x:62

ModuleSource

```
typedef struct ModuleSource { String text, own; Array markers; Map members; } *ModuleSource
```

Describes an x2c source whose `module` block names member files: `text` holds the file's own text, `own`, with each member's text in place of the block. `markers` are the line-marker rows that give each region of `text` its file, line, and byte offset there, with the text that precedes that offset on its line and the prefix that file's statics take. The first row marks the start of `text`. `members` maps each member's canonical path to its content hash.

Source: src/compiler/compiler.x:46

Pending

```
typedef struct Pending { Array areas[PENDING_AREAS]; } Pending
```

Holds generated code awaiting insertion, by `Pending.area` destination: unit support declarations, which `Compiler.transform` lowers after the unit, then the lowered statements of each file initialization area.

Source: src/compiler/compiler.x:82

PendingMark

```
typedef struct PendingMark { int lengths[PENDING_AREAS]; } PendingMark
```

Holds each destination's length for `Pending.restore`.

Source: src/compiler/compiler.x:85

ScriptUnit

```
typedef struct ScriptUnit { String path, shebang; int defines_main; } *ScriptUnit
```

Describes the `#!` script unit a translation unit was started from: its canonical path, its first line as written, and whether it defines `main` itself. Every compiler of that unit shares one record, so collection sees the same script settings the full parse does.

Source: src/compiler/compiler.x:33

SlotWatch

```
typedef struct SlotWatch *SlotWatch
```

Names the slots a rewrite's first lowering watches; see "prepared replacements" in macros.x.

Source: src/compiler/compiler.x:92

Sym

```
typedef struct Sym *Sym
```

Names the scope-owned semantic table belonging to one compiler.

Source: src/compiler/compiler.x:88

SymScope

```
typedef struct SymScope { Map symbols, bindings, enumerators, macros; } SymScope
```

Holds the three mutable maps that form one lexical semantic scope.

Active scopes have initialized maps. The zero value is only the sentinel returned when an empty scope stack is popped.

Source: src/compiler/compiler.x:72

SymTxn

```
typedef struct SymTxn { Compiler c; int scope_index, next_binding, active, String initializer_name; String shutdown_name, Map counters; int local_macro_names, mark, meta_group_size, ancestors, placements; SymScope scope; Map statics, binding_facts; Map source_definitions; int source_occurrences; int extended, Map adapters; PendingMark pending, int origin_count, origin, needs_exception; } SymTxn
```

Holds one reversible semantic transaction in caller storage. The zero value is inactive. Keep an active transaction in one object; copying it does not coordinate completion. The symbol table's undo log holds the rows it may restore from `mark`; the staged generated-name counters and other snapshots live in its initial Scope.

Source: `src/compiler/compiler.x:292`

Design notes

A `Compiler` owns the translation state of one unit: its token cursor, symbol table, and declaration state. Collection parses the unit shallowly, skipping function bodies; the full parse produces its AST. The related compilers of one unit share its package registries, generated names, and binding numbers. The symbol table is `Sym`, in `symtab/symtab.x`.

Diagnostics may exit immediately or raise `<malformed>` while a recovery boundary is active.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/components/bracket-access.x`

Bracket access policy: Array and Map mutation through their typed helpers, and any other participant's read, store, and update through its indexing members, with each operand evaluated once, in order. Getter resolution remains with admission.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/components/collection-literals.x`

Array and Map literals.

Functions

Function	Summary
`array_literal`	Builds an Array literal from its elements.
`map_literal`	Builds a Map literal from its keys and values, alternating.

Functions

array_literal

```
meta Code array_literal(Code code)
```

Builds an Array literal from its elements.

Source: `src/components/collection-literals.x:14`

map_literal

```
meta Code map_literal(Code code)
```

Builds a Map literal from its keys and values, alternating.

Source: `src/components/collection-literals.x:32`

Design notes

A collection literal builds its container with the counted constructor, which receives each element converted to `Var`, in source order; an empty brace among the elements is an empty Map. The compiler keeps the elements in source order; see `src/transform/literals.x`.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/components/delegate-members.x

Delegate fields.

Design notes

A field declared `delegate T name;` answers the method calls its aggregate lacks. The parser marks each such field, and the rule below is registered for the mark, so only a call that finds no member on a receiver whose aggregate declares one tests it. The rule searches the receiver's delegate fields depth first in field order and rebuilds the call through the one path that finds the method. More than one path, or a cycle with no path, is an error.

Each definition precedes the definitions that call it: the compiler settles whether a linked copy reaches a compile-time operation when the copy binds.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/components/destructuring.x

Flat List destructuring.

Design notes

A destructuring reads its source once as a `List` and gives each target the element at its position, left to right; the compiler converts each element as an ordinary assignment or initialization does. The parser has declared the targets, so each rule answers bound code that keeps their bindings: `T (a, b) = source;` declares `a` and `b` before the read, `(T1 a, T2 b) = source;` declares each initialized from its element, an assignment statement reads in its own block, and an assignment expression keeps its source in a result local as its value. The List local and the result local are fresh names the carrier asks for.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/components/dynamic-operators.x

Dynamic operators.

Design notes

An operation with a `Var` operand that no protocol member resolved lowers to the dynamic runtime. A binary operator boxes both operands for `Var.binary`; a compound assignment, increment, or decrement calls the update helper of its target's storage, which takes the target's address once and stores once.

Each definition precedes the definitions that call it: the compiler settles whether a linked copy reaches a compile-time operation when the copy binds.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/components/interpolation.x

Interpolated Strings.

Functions

Function	Summary
`interpolation_lowering`	Joins an interpolated String's parts.

Functions

interpolation_lowering

`meta Code interpolation_lowering(Code code)`

Joins an interpolated String's parts.

Source: `src/components/interpolation.x:14`

Design notes

A `%"..."` literal reaches its rule as the `segments` form, each part already a `String`: cached text, a converted `$` insertion, or raw text that constructed syntax spells. The rule joins the parts with `String.join` over a `List` chain of them, which the compiler boxes and evaluates once each, left to right.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/components/lambda-functions.x`

Lambda lowering.

Design notes

A lambda becomes a static function among the unit's support declarations. Its parameters keep their declarators, a bare one is a `Var`, and it returns a `Var`: a bare return and the end of a block body return none, and an expression body returns its value. A noncapturing lambda's value is that function.

A capturing lambda's function also takes each binding it captures, a value as its own type and a reference as a reference. Its value is a `Func` that copies each captured value, or the address a reference captures, into a context record when the lambda is evaluated, left to right. The `Func`'s adapter reads the arguments and calls the function with them and the record's fields. A public inline function reaches the unit's private record through a bridge function.

A local or parameter that a lambda shares by reference moves into a Scope cell, so it outlives the function that declares it; a function rule rewrites each function definition whose lambdas share one.

Each definition precedes the definitions that call it: the compiler settles whether a linked copy reaches a compile-time operation when the copy binds.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/components/pattern-match.x`

Match statements.

Design notes

A match lowers to C text around its subject, patterns, and bodies: a block that reads the subject once as a List and switches on the bits of its head. Lowering steps into the nodes the text holds, and the `switch` is the barrier the cleanup walk knows, so an arm's `break` leaves the match and `continue` reaches the enclosing loop. Each arm tests its pattern, declares its binders, runs its body, and breaks; an arm that fails falls into the next. A static pattern of literals, unique binders, typed captures, nested Lists, and a final `*` is tested in place by nested `if` tests on cursors over the subject. Any other pattern captures into a buffer: a macro-valued case through its macro, and the rest through the Match runtime, at a static site when the pattern's value is the same each time it runs.

Each definition precedes the definitions that call it: the compiler settles whether a linked copy reaches a compile-time operation when the copy binds.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/components/printf-format.x

Var values in printf-family formats.

Design notes

A printf-family call whose format is a static literal passes each Var value the format consumes as the C type its conversion reads: a numeric conversion converts the value through `Var.convert`, `%c` and a `*` width or precision read an int, and `%s` displays it through `Var.str`. The format is read once, left to right. A Var value that no conversion consumes, or one at a conversion with no native reading, is an error. Each family member is one call pattern, keyed by its callee.

Each definition precedes the definitions that call it: the compiler settles whether a linked copy reaches a compile-time operation when the copy binds.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/components/protocol-adapters.x

Protocol adapters and Var descriptors.

Design notes

The compiler presents each resolved conformance whose forward converter the unit defines, and whose participant is not numeric, as a `conform` form. It has settled each member's owner and storage, reported their collisions and missing conversions, and introduced the generated names:

(conform BASE PARTICIPANT FORWARD REVERSE BINDER DESCRIPTOR (ROW...))

FORWARD and REVERSE spell the participant's converters to and from BASE, and BINDER is the protocol's type variable. A ROW is `(MEMBER STATUS SOURCE EXPECTED TEMPLATE PART...)`: STATUS is `base-dflt`, `implmntd`, or `no-member`, SOURCE the function that implements the member, EXPECTED its signature for the participant, and TEMPLATE its signature in the protocol. Each PART asks for one function NAME, a binding. `(default NAME STORAGE)` is the forwarding member `PARTICIPANT_MEMBER`, `static` when STORAGE is `static`, which calls SOURCE. `(inherited NAME EXPECTED)` is the prototype of that member, which another conformance defines with the signature EXPECTED. `(thunk NAME ERASED)` is the descriptor slot MEMBER's function, whose signature ERASED types each protocol variable `Var`. DESCRIPTOR is `(METHODS NAME VAR-TAG AREA [TAG])` for a `Var` participant with its own descriptor, and empty otherwise: the method table's name, the registered type name and its Var tag, the `unit-init` area, and an explicit tag. A rule answers file-scope code for the unit's support, or `(seq)` when it placed everything.

This rule places each function and the static method table among the unit's support declarations, and the table's assignment and the descriptor's registration in the unit's initialization. A forwarding member or thunk passes each parameter the protocol types BINDER through a converter: FORWARD for a forwarding member, REVERSE for a thunk. A thunk that prints a boxed struct or union, a copy that has an identity only before unboxing, guards its rendering path.

Each definition precedes the definitions that call it: the compiler settles whether a linked copy reaches a compile-time operation when the copy binds.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/components/raise.x

Raise.

Design notes

A raise records one structured Error. Its rule converts the code to a Symbol and each detail key and value to a Var, promoting a C string literal value to a String first. A value whose type is not immutable is reported and replaced by null, so the translation goes on and fails when it finishes; the runtime rejects what a List or Var

detail holds. The compiler lowers the converted raise's parts in source order and builds their literals when the raise runs.
Each definition precedes the definitions that call it: the compiler settles whether a linked copy reaches a compile-time operation when the copy binds.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/components/string-switch.x

Switches on a String.

Design notes

A switch on a String selects its case by content. Each label the switch owns becomes its index, so the body stays an ordinary integer switch on a helper's result. The helper returns the index of the first label equal to the selector under ordinary ==, or 0. Switches on the same type with the same labels share one helper.
Each definition precedes the definitions that call it: the compiler settles whether a linked copy reaches a compile-time operation when the copy binds.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/components/try-catch.x

Try, catch, and finally.

Design notes

A try lowers to the landing form `(landing CODE ROWS)` that the cleanup walk finishes: its frame, its finalizer as a region that leaves the frame on a transfer out of it, the exits that close its catch site, claim the finalizer, and leave the frame, then its body and each catch arm as regions that run those exits on every transfer out of them. The frame lands when something raises, and the catch site selects the arm. A clause's facts are `(HANDLE STATE (ARM...) PATTERN...)`: the handler the parser introduced, the catch site's initial state, the arms' tokens, and the patterns of the filtered arms, which precede the default arm.
Each definition precedes the definitions that call it: the compiler settles whether a linked copy reaches a compile-time operation when the copy binds.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/emit/emit.x

C tokens from normalized x2c ASTs.

Functions

Function	Summary
`Compiler.emit`	Emits a bound, typed, transform-normalized AST sequence as flat C tokens.

Compiler

Compiler.emit

```
List Compiler.emit(Compiler c, List ast, Map cache_bindings)
```

Emits a bound, typed, transform-normalized AST sequence as flat C tokens. Source mapping adds `src-at`/ID pairs consumed by the formatter. The compiler must own the AST's binding facts and origins, and continue the translation session's shared generated-name state. This operation does not bind, transform, or choose header and source placement; generation supplies any added scaffolding in the same normalized grammar.

`cache_bindings` maps semantic cache ids to generated slots; NULL keeps raw ids for inspection. It preserves the top-level AST sequence and advances generated-name counters as it allocates temporaries. Returned canonical `Lists` and `Strings` are owned by pools active during emission; promote them before releasing those pools if the tokens must survive.

Source: `src/emit/dispatch.x:92`

Design notes

This module owns emission: it turns a bound, typed, transform-normalized AST into the token `Lists` that formatting prints as C. One stack-local Emitter holds the current function's name, static objects, and native aliases, so emission is reentrant and a failed translation cannot contaminate later units. `transform.x` has already placed each cleanup region's statements on the exits that leave it.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/expressions/expressions.x`

Expression syntax and its resolution.

Functions

Function	Summary
`Compiler.apply_macro_value`	Applies the Macro value `expr` to `supplied` argument expressions.
`Compiler.check_explicit_converter`	Reports `parsed` when it is the explicit converter call resolved last and `target` converts its receiver on its own: either side is Var, the types share one C type, or the receiver declares a converter to the target.
`Compiler.complete_iter_chain`	Completes an eligible resolved `Iter` call chain for immediate consumption.
`Compiler.convert_expression`	Adds operations to convert a resolved expression AST to `target`.
`Compiler.convert_segment_to_string`	Converts a resolved interpolation segment to `String` when available.
`Compiler.converter_call`	The call to the converter that `type`, or the first of its typedef names that declares one, provides for `target`, applied to `expr`, or NULL when none declares one.
`Compiler.func_call_parts`	Returns the callee and arguments of a typed `Func` call, or NULL for any other expression.
`Compiler.is_builtin_converter_call`	The global builtin boxers and scalar formatters only observe their arguments.
`Compiler.needs_resolution`	True when syntax still needs ordinary binding or macro substitution.
`Compiler.parse_assignment`	Parses one right-associative assignment expression.
`Compiler.parse_conditional`	Parses a binary expression and its optional conditional tail.
`Compiler.parse_expression`	Parses an assignment expression and any following comma expressions.
`Compiler.parse_macro_expression_target`	Parses one macro target through the cast-expression grammar.
`Compiler.parse_parenthesized_statement`	Parses a statement beginning with `(`.
`Compiler.parse_primary`	Parses one primary expression or expression-valued macro slot.
`Compiler.parse_variable`	Parses and resolves one complex identifier expression.
`Compiler.postfix_completions`	Returns sorted visible field and method names that resolve on `receiver` through `access`.
`Compiler.printf_static_format`	Returns the C spelling of the format a printf-family call consumes when it is known at translation time, or `NULL`.
`Compiler.promote_string_literal`	Converts a C string literal to `String` where no C meaning applies: as a method receiver, a `foreach` collection, or a raise detail.
`Compiler.require_var_tag`	Returns the exact Var tag for a type test, rejecting types without one.
`Compiler.resolve_expression`	Resolves and type-annotates one expression AST in current compiler state.

Function	Summary
Compiler.resolve_map_entry	Resolves the key and value of one (map-entry key value) AST row.
Compiler.resolve_postfix_member	Resolves one field or method selection without consuming parser tokens.
Compiler.var_tag_expression	Builds an exact tag expression, deferring macro type slots until binding.
List.printf_family	Returns the printf-family entry a callee names, or NULL.

Compiler

Compiler.apply_macro_value

```
List Compiler.apply_macro_value( Compiler c, List expr, List supplied, Token origin)
```

Applies the Macro value expr to supplied argument expressions. Inside a template body the call is retained so the template can capture the value it applies; elsewhere it is an ordinary Macro_apply over the argument values.

Source: src/expressions/syntax.x:533

Compiler.check_explicit_converter

```
void Compiler.check_explicit_converter( Compiler c, List parsed, Type target, int context)
```

Reports parsed when it is the explicit converter call resolved last and target converts its receiver on its own: either side is Var, the types share one C type, or the receiver declares a converter to the target. The call then changes nothing but the spelling. context is 0 for a typed destination, 1 for an interpolation hole, which displays every value through Var.str, and 2 for a printf-family value, which the format converts when it is a Var.

Source: src/expressions/conversions.x:48

Compiler.complete_iter_chain

```
List Compiler.complete_iter_chain(Compiler c, List expression)
```

Completes an eligible resolved Iter call chain for immediate consumption. It accepts only a typed identifier call of the form (expr R (call (expr ((func P) T) (ident B)) (args A))), where R and the last formal in P canonicalize to Iter. Iter arguments are completed recursively; a call missing only that last formal receives the hidden destination. Variadic calls and Iter_unzip are returned unchanged.

Source: src/expressions/calls.x:416

Compiler.convert_expression

```
List Compiler.convert_expression(Compiler c, List expr, Type target)
```

Adds operations to convert a resolved expression AST to target. The result may contain converter, boxing, unboxing, Func, reference, or composite-literal operations. Returns the original expression when C performs the conversion implicitly; an unsupported x2c conversion reports a type error through c. Synthesized operations may add generated bindings or immutable literal entries to compiler state.

Source: src/expressions/conversions.x:224

Compiler.convert_segment_to_string

```
List Compiler.convert_segment_to_string(Compiler c, List expr)
```

Converts a resolved interpolation segment to String when available. A missing String conversion is expected: the transform boxes that segment to Var and renders it at runtime.

A declared numeric converter keeps its formatting; other numeric segments use Var.str. A segment statically spelled Var also uses Var.str for every runtime tag. The ordinary Var-to-String conversion is not equivalent: it extracts only a String payload and yields empty String for every other tag.

Source: src/expressions/conversions.x:417

Compiler.converter_call

`List Compiler.converter_call(Compiler c, List expr, Type type, Type target)`

The call to the converter that `type`, or the first of its typedef names that declares one, provides for `target`, applied to `expr`, or NULL when none declares one.

Source: `src/expressions/conversions.x:575`

Compiler.func_call_parts

`List Compiler.func_call_parts(Compiler c, Var content)`

Returns the callee and arguments of a typed `Func` call, or NULL for any other expression. `content` is the body of the call's `expr` node. Each argument is `(func-arg value address source)`: the argument boxed as a Var, or `(no-value)` when it has no Var form; its address, or 0; and its type, which is `(expr ("List") (ident reference))` for a null argument that takes the callee's own type.

Source: `src/expressions/calls.x:355`

Compiler.is_builtin_converter_call

`int Compiler.is_builtin_converter_call(Compiler c, List expr)`

The global builtin boxers and scalar formatters only observe their arguments. A custom converter or a shadowed callee may change state.

Source: `src/expressions/conversions.x:628`

Compiler.needs_resolution

`int Compiler.needs_resolution(Compiler c, Var value)`

True when syntax still needs ordinary binding or macro substitution.

Source: `src/expressions/resolution.x:80`

Compiler.parse_assignment

`List Compiler.parse_assignment(Compiler c)`

Parses one right-associative assignment expression. A parenthesized identifier list on the left becomes a destructuring assignment only for `=`. `c.token` stops after the expression.

Source: `src/expressions/syntax.x:38`

Compiler.parse_conditional

`List Compiler.parse_conditional(Compiler c)`

Parses a binary expression and its optional conditional tail. The false arm recurses at conditional precedence, making `?:` right-associative, and `c.token` stops after the expression.

Source: `src/expressions/syntax.x:87`

Compiler.parse_expression

`List Compiler.parse_expression(Compiler c)`

Parses an assignment expression and any following comma expressions. A comma expression retains source order and takes the type of its final value. `c.token` stops at the first token outside the expression.

Source: `src/expressions/syntax.x:13`

Compiler.parse_macro_expression_target

```
List Compiler.parse_macro_expression_target(Compiler c)
```

Parses one macro target through the cast-expression grammar. Parsing starts at `c.token` and leaves it at the first token after the target.

Source: `src/expressions/syntax.x:279`

Compiler.parse_parenthesized_statement

```
List Compiler.parse_parenthesized_statement(Compiler c)
```

Parses a statement beginning with `(`.

The ordinary parameter parser consumes the contents once: one anonymous parameter is a cast type, while named parameters are destructuring declarations. Everything following an ordinary parenthesized expression resumes at the postfix tail it had already reached. This entry consumes the terminating `;` and returns `(stmnt expression)` or an origin-anchored `(dstrdecl ...)`.

Source: `src/expressions/syntax.x:209`

Compiler.parse_primary

```
List Compiler.parse_primary(Compiler c)
```

Parses one primary expression or expression-valued macro slot. Dispatch starts at `c.token` to the selected literal, identifier, grouping, or macro parser and leaves the token after that primary form.

Source: `src/expressions/syntax.x:607`

Compiler.parse_variable

```
List Compiler.parse_variable(Compiler c)
```

Parses and resolves one complex identifier expression. Parsing starts at `c.token` and leaves it after the identifier.

Source: `src/expressions/syntax.x:665`

Compiler.postfix_completions

```
List Compiler.postfix_completions( Compiler c, Type receiver, Symbol access)
```

Returns sorted visible field and method names that resolve on `receiver` through `access`.

Source: `src/expressions/access.x:327`

Compiler.printf_static_format

```
String Compiler.printf_static_format(Compiler c, Var format)
```

Returns the C spelling of the format a printf-family call consumes when it is known at translation time, or `NULL`. That is a quoted C string literal, the canonical `String` one becomes, or the `String_new` of one; any other format, such as a variable or an object macro, is not readable here. A `String` literal's text is spelled as a C literal.

Source: `src/expressions/calls.x:504`

Compiler.promote_string_literal

```
List Compiler.promote_string_literal(Compiler c, List expr)
```

Converts a C string literal to `String` where no C meaning applies: as a method receiver, a `foreach` collection, or a raise detail. Parentheses and a conditional whose arms are both literals count as the literal; any other expression is returned unchanged.

Source: `src/expressions/conversions.x:176`

Compiler.require_var_tag

```
Symbol Compiler.require_var_tag( Compiler c, Type target, Token origin)
```

Returns the exact Var tag for a type test, rejecting types without one. Enums retain no identity after boxing and cannot be tested this way.

Source: `src/expressions/resolution.x:324`

Compiler.resolve_expression

```
List Compiler.resolve_expression(Compiler c, List input, Token origin)
```

Resolves and type-annotates one expression AST in current compiler state. Existing `expr` type annotations are resolved semantic types. Already-resolved trees without unresolved descendants and non-expression inputs are returned unchanged; abstract declarations use the declaration binder. `origin` anchors diagnostics and generated operations that must retain source position.

Source: `src/expressions/resolution.x:55`

Compiler.resolve_map_entry

```
List Compiler.resolve_map_entry(Compiler c, List input, Token origin)
```

Resolves the key and value of one `(map-entry key value)` AST row. Any other shape is reported at `origin` as a parse error.

Source: `src/expressions/resolution.x:450`

Compiler.resolve_postfix_member

```
List Compiler.resolve_postfix_member( Compiler c, Type receiver_type, List field, Symbol access, int call_context)
```

Resolves one field or method selection without consuming parser tokens. `field` is a single-name `List` and `access` is `.` or `->`. The result is a `(field access type)`, `(method binding signature)`, or `(ambiguous ...)` row, or NULL when no member is visible. Method lookup is enabled only by `call_context` and records the selected binding in `c.sym`.

Source: `src/expressions/access.x:177`

Compiler.var_tag_expression

```
List Compiler.var_tag_expression(Compiler c, Type target, Token origin)
```

Builds an exact tag expression, deferring macro type slots until binding.

Source: `src/expressions/resolution.x:337`

List

List.printf_family

```
const PrintfFn *List.printf_family(List l)
```

Returns the printf-family entry a callee names, or `NULL`. A resolved user function that happens to use a libc spelling is not one.

Source: `src/expressions/calls.x:483`

Public types

Type	Kind	Summary
PrintfFn	struct	A printf-family function: its name, the indexes of its format and first value arguments, and whether only a spelling no declaration resolves names it.

PrintfFn

`typedef struct PrintfFn { const char *name, int fmt_arg, first_arg, unresolved; } PrintfFn`

A printf-family function: its name, the indexes of its format and first value arguments, and whether only a spelling no declaration resolves names it.

Source: `src/expressions/expressions.x:19`

Design notes

The parser reads expressions with C precedence and resolves each node as it builds it; constructed syntax enters the same resolver. Resolution gives a node its type and binding, and its last step converts the value to the type its destination declares. That conversion lives here; the walk of a brace initializer over its subobjects lives in `initializers.x`.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/generate/generate.x`

The generated header and source of one unit.

Functions

Function	Summary
`Compiler.definition_rows`	Returns one row for each function, foreign alias, and typedef that the lowered unit `ast` defines, in source order: `(function NAME DISPLAY TYPE PARAMETERS LINE DOC STATIC ORIGIN DECLARATOR SPAN)` or `(typedef NAME BASE MODIFIERS SPAN)`.
`Compiler.dump_definitions`	Prints the `--dump-definitions` projection of the lowered unit `ast`: `(unit PATH)`, the module comment as `(module TEXT)` when the file opens with one, then one row per `Compiler.definition_rows` entry.
`Compiler.generate_code`	Writes the generated C header and source for one lowered translation unit.
`Compiler.generate_code_text`	Returns the generated header and source of the lowered `ast` as `(hfile htext cfile ctext)`, named from `basename`, without writing them.
`Compiler.init_statements`	Returns the statements queued for `phase`, in the order they were added.
`Compiler.initializer_function`	Returns the function `type name(void)` running `body`.
`Compiler.place_source_prelude`	Places generated `declarations` after source types and includes, before the first function or captured initializer that can use them.
`Compiler.run_once_body`	Returns a run-once initializer body: `before`, then a return when `guard` is set that sets it otherwise, then `after`.
`Init.enter`	Returns `function` with its entry setup: the protocol initializer and the type initializer run once, and a `guarded` entry calls the region's initializer unless its guard is set.

`Compiler`

Compiler.definition_rows

`List Compiler.definition_rows(Compiler c, List ast)`

Returns one row for each function, foreign alias, and typedef that the lowered unit `ast` defines, in source order: `(function NAME DISPLAY TYPE PARAMETERS LINE DOC STATIC ORIGIN DECLARATOR SPAN)` or `(typedef NAME BASE MODIFIERS SPAN)`. `ORIGIN` is `<source>`, `<macro>` for a Unit macro or generated body, or `<alias>`. `DECLARATOR` is the token range of an authored function's declarator, and `SPAN` the token range and privacy of the top-level form that produced the definition; either is empty when the compiler made the definition. `LINE` is 1 and `DOC` empty for a definition without authored source.

Source: `src/generate/definitions.x:15`

Compiler.dump_definitions

`void Compiler.dump_definitions(Compiler c, List ast)`

Prints the `--dump-definitions` projection of the lowered unit `ast`: `(unit PATH)`, the module comment as `(module TEXT)` when the file opens with one, then one row per `Compiler.definition_rows` entry. The command-line reference in the book describes the fields.

Source: `src/generate/definitions.x:108`

Compiler.generate_code

`void Compiler.generate_code(Compiler c, List ast, String dir)`

Writes the generated C header and source for one lowered translation unit. `ast` must be the normalized result of `transform_ast` for this compiler; its filename, symbols, binding facts, cache keys, and initialization state must still describe that same unit. `dir` must already exist. Generation partitions the AST, materializes caches and once-only initialization, and writes or replaces `<dir>/<source-stem>.h`, `.c`, and the `.xi` interface of a collected unit through one `Path.publish_files`, so a failed write replaces none of them. It appends generated bindings and initialization work to the compiler and is not idempotent. Failures are reported as `emit` diagnostics.

Source: `src/generate/output.x:35`

Compiler.generate_code_text

`List Compiler.generate_code_text(Compiler c, List ast, String basename)`

Returns the generated header and source of the lowered `ast` as `(hfile htext cfile ctext)`, named from `basename`, without writing them. It affects the compiler as `Compiler.generate_code` does.

Source: `src/generate/output.x:47`

Compiler.init_statements

`List Compiler.init_statements(Compiler c, Symbol phase)`

Returns the statements queued for `phase`, in the order they were added.

Source: `src/generate/initialization.x:199`

Compiler.initializer_function

`List Compiler.initializer_function( Compiler c, Type type, List name, List body)`

Returns the function `type name(void)` running `body`.

Source: `src/generate/initialization.x:6`

Compiler.place_source_prelude

`List Compiler.place_source_prelude( Compiler c, List source, List declarations)`

Places generated `declarations` after source types and includes, before the first function or captured initializer that can use them. An outer conditional containing that first use follows the declarations. A unit without such a use gets them at its end, after the declarations they may assign.

Source: `src/generate/output.x:102`

Compiler.run_once_body

`List Compiler.run_once_body(Compiler c, List before, List guard, List after)`

Returns a run-once initializer body: `before`, then a return when `guard` is set that sets it otherwise, then `after`.

Source: `src/generate/initialization.x:12`

`Init`

Init.enter

`List Init.enter(Init &i, List function, int guarded)`

Returns `function` with its entry setup: the protocol initializer and the type initializer run once, and a `guarded` entry calls the region's initializer unless its guard is set.

Source: `src/generate/initialization.x:49`

Public types

Type	Kind	Summary
`Init`	struct	Holds one region's initialization.

Init

`typedef struct Init { Compiler c, String initializer, List guard, entry, shutdown, synthetic; Map reachable; } Init`

Holds one region's initialization. `guard` is set once the region has initialized, and `entry` is the function a guarded entry calls. In the source, `initializer` names the type initializer, `shutdown` registers the unit's shutdown function, `synthetic` is the synthetic initializer or NULL, and `reachable` holds the entries a cache-only file guards, or is NULL when every public entry is guarded.

Source: `src/generate/generate.x:37`

Design notes

This module owns a unit's generated files: it splits the lowered AST into header and source without changing source order, gives the source its once-only initialization and static prototypes, wraps the header in its guard, and publishes both with the unit's interface. The definition rows the interface lists, and `--dump-definitions` prints, come from the same lowered AST.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/initializers/initializers.x`

Brace initializer conversion.

Functions

Function	Summary
`Compiler.convert_compound_literal`	Keeps a compound literal's native type definition at its original scope.
`Compiler.convert_initializer`	Converts an initializer using its declared native object for array bounds.
`Compiler.initializer_field_path`	Returns the initializer path selecting one visible aggregate field.
`Compiler.initializer_native_types`	Returns native definition/reference types for a compound literal.
`Compiler.initializer_rows`	Returns (original cases) rows; each case is (native-condition path destination value).
`Compiler.initializer_slot`	Selects a native subobject without evaluating it when used by sizeof.

`Compiler`

Compiler.convert_compound_literal

`List Compiler.convert_compound_literal( Compiler c, List value, Type type, Type native_type)`

Keeps a compound literal's native type definition at its original scope.

Source: `src/initializers/conversion.x:456`

Compiler.convert_initializer

`List Compiler.convert_initializer( Compiler c, List value, Type type, List target)`

Converts an initializer using its declared native object for array bounds.

Source: `src/initializers/conversion.x:7`

Compiler.initializer_field_path

`List Compiler.initializer_field_path( Compiler c, Type type, List field)`

Returns the initializer path selecting one visible aggregate field. Anonymous aggregate members remain explicit path frames, so consumers observe the same member promotion as native initializer conversion.

Source: `src/initializers/subobjects.x:145`

Compiler.initializer_native_types

`List Compiler.initializer_native_types(Compiler c, Type type)`

Returns native definition/reference types for a compound literal. Macro expansion stays in the original cast; named tags let later sizeof expressions reuse that exact layout without a new scope.

Source: `src/initializers/conversion.x:469`

Compiler.initializer_rows

`List Compiler.initializer_rows( Compiler c, Type root, List items, List target)`

Returns (original cases) rows; each case is (native-condition path destination value). Explicit braces start a nested walk. Scalar runs map their ordinal through the native dimensions; other inputs retain possible cursor continuations. A NULL condition is unconditional, and a NULL destination is excess.

Source: `src/initializers/subobjects.x:26`

Compiler.initializer_slot

`List Compiler.initializer_slot(Compiler c, List target, List path)`

Selects a native subobject without evaluating it when used by sizeof.

Source: `src/initializers/subobjects.x:321`

Design notes

A brace initializer converts each value for the subobject that C's initialization order gives it. This unit owns that walk over field order and array bounds. A bound only the native compiler knows becomes a condition on an unevaluated `*(T *) 0` target, and a value converts under each condition that can select it. `expressions.x` converts the single values.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/literals/literals.x

X2c literal parsing.

Functions

Function	Summary
Compiler.literal_cell	Conses the element expression `head` onto the cells `tail`, or appends the List a `(splice EXPR)` head holds, and folds the cell into the literal cache when both parts are constant.

Function	Summary
Compiler.parse_array_literal	Parses a quoted Array literal into a typed, source-ordered (array ...) node and consumes its closing].
Compiler.parse_atomic_literal	Parses the current atomic token into a typed expression and advances once.
Compiler.parse_catch_pattern_literal	Parses a filtered-catch %() payload into a typed List pattern.
Compiler.parse_list_literal	Parses a List literal beginning at (or %(and returns its typed expression after consuming).
Compiler.parse_macro_pattern_arguments	Parses a macro pattern's parenthesized arguments and returns the static pattern of each: a binder or wildcard, a nested macro pattern, or a %(...) List pattern.
Compiler.parse_map_entries	Parses comma-separated Map entries up to but not including }.
Compiler.parse_map_entry	Parses one Map entry without consuming its following comma or }.
Compiler.parse_map_literal	Parses a quoted Map literal into a typed, source-ordered (map ...) node and consumes its closing }.
Compiler.parse_raise_literal	Parses the %() payload following raise into a (raise CODE (args ...)) node and consumes its closing).
Compiler.parse_string_literal	Parses a percent String literal and returns its typed expression after the closing quote.
Compiler.parse_symbol_set_literal	Parses a %<<...>> literal into an immutable ordered SymbolSet.
Compiler.symbol_set_expression	Builds a typed SymbolSet expression from source-ordered Symbol values.
Compiler.try_parse_macro_pattern_insertion	Parses a pattern's ${$NAME(...)}, where NAME is a macro, into its derived pattern, or returns NULL without consuming tokens.
Compiler.typed_match_pattern	Applies each typed capture's predicate to every unquoted occurrence.

Compiler

Compiler.literal_cell

```
List Compiler.literal_cell(Compiler c, List head, List tail)
```

Conses the element expression head onto the cells tail, or appends the List a (splice EXPR) head holds, and folds the cell into the literal cache when both parts are constant.

Source: src/literals/quotation.x:256

Compiler.parse_array_literal

```
List Compiler.parse_array_literal(Compiler c)
```

Parses a quoted Array literal into a typed, source-ordered (array ...) node and consumes its closing].

Source: src/literals/collections.x:121

Compiler.parse_atomic_literal

```
List Compiler.parse_atomic_literal(Compiler c)
```

Parses the current atomic token into a typed expression and advances once. Pattern and macro-hole state control binder validation and quoting, while shallow parsing permits provisional numeric types.

Source: src/literals/atoms.x:34

Compiler.parse_catch_pattern_literal

```
List Compiler.parse_catch_pattern_literal(Compiler c)
```

Parses a filtered-catch `%()` payload into a typed `List` pattern. The call consumes the closing `)`. A code `Symbol`, binder, or pattern may be followed by `*` patterns or `(key pattern)` pairs; pattern and runtime-literal state is restored on every exit.

Source: `src/literals/payloads.x:98`

Compiler.parse_list_literal

```
List Compiler.parse_list_literal(Compiler c)
```

Parses a `List` literal beginning at `(` or `%(` and returns its typed expression after consuming `)`. Pattern parsing sets and restores `match_is`; `runtime_literals` disables stable-cell caching.

Source: `src/literals/quotation.x:35`

Compiler.parse_macro_pattern_arguments

```
List Compiler.parse_macro_pattern_arguments(Compiler c)
```

Parses a macro pattern's parenthesized arguments and returns the static pattern of each: a binder or wildcard, a nested macro pattern, or a `%(...)` List pattern.

Source: `src/literals/quotation.x:146`

Compiler.parse_map_entries

```
List Compiler.parse_map_entries(Compiler c)
```

Parses comma-separated `Map` entries up to but not including `}`. `Entry`-position macro sequences are flattened in source order.

Source: `src/literals/collections.x:206`

Compiler.parse_map_entry

```
List Compiler.parse_map_entry(Compiler c)
```

Parses one `Map` entry without consuming its following comma or `}`. A bare identifier key is an Atom; any other key is an expression. A direct row returns a resolved `(map-entry KEY VALUE)` node; an entry-position macro may return `(seq ...)` for its caller to splice.

Source: `src/literals/collections.x:221`

Compiler.parse_map_literal

```
List Compiler.parse_map_literal(Compiler c)
```

Parses a quoted Map literal into a typed, source-ordered `(map ...)` node and consumes its closing `}`.

Source: `src/literals/collections.x:155`

Compiler.parse_raise_literal

```
List Compiler.parse_raise_literal(Compiler c)
```

Parses the `%()` payload following `raise` into a `(raise CODE (args ...))` node and consumes its closing `)`. The code is a bare `Symbol` or a `$name` reference, detail keys are bare `Symbols`, each keyed detail has one value, and payload literals bypass the compiler cache.

Source: `src/literals/payloads.x:32`

Compiler.parse_string_literal

```
List Compiler.parse_string_literal(Compiler c)
```

Parses a percent `String` literal and returns its typed expression after the closing quote. Static segments enter the compiler cache unless `runtime_literals` is set; interpolated segments remain source ordered.

Source: `src/literals/strings.x:18`

Compiler.parse_symbol_set_literal

```
List Compiler.parse_symbol_set_literal(Compiler c)
```

Parses a `%<<...>>` literal into an immutable ordered `SymbolSet`. Entries must be literal compact `Symbols`; source order defines dense indexes and an equal encoded `Symbol` reports a duplicate diagnostic.

Source: `src/literals/collections.x:30`

Compiler.symbol_set_expression

```
List Compiler.symbol_set_expression(Compiler c, List values, int &duplicate)
```

Builds a typed `SymbolSet` expression from source-ordered `Symbol` values. Stores the first duplicate index, or −1, through `duplicate`; a duplicate returns NULL.

Source: `src/literals/collections.x:72`

Compiler.try_parse_macro_pattern_insertion

```
List Compiler.try_parse_macro_pattern_insertion(Compiler c, int content)
```

Parses a pattern's `${$NAME(...)}`, where NAME is a macro, into its derived pattern, or returns NULL without consuming tokens. `content` selects the bare content, for the content position of an `(expr TYPE ...)` shell.

Source: `src/literals/quotation.x:184`

Compiler.typed_match_pattern

```
List Compiler.typed_match_pattern(Compiler c, List pattern, List types)
```

Applies each typed capture's predicate to every unquoted occurrence.

Source: `src/literals/quotation.x:336`

Design notes

Each literal parses to a typed expression. A `List` cell or `String` segment with no dynamic reference enters the compiler cache and is built once; `runtime_literals` builds every one at run time instead. Lambda literals parse in `lambdas.x`.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/macros/macros.x`

Source macros and the compile-time Lisp they run.

Functions

Function	Summary
x2c_template_call	Returns the syntax that invokes the template `stored`, a macro name spelled as a String or a local definition, with `values`: what a template call in a staged `meta` body evaluates to.
Compiler.add_library_dependencies	Records the compile-time Lisp libraries as dependencies of `c`, as each use of its session does.
Compiler.after_hole	Returns the token after the named or expression hole at the cursor.
Compiler.applying_shipped	Whether the rule whose meta function now runs is shipped, so that the names it calls resolve to the copies the compiler links.

Function	Summary
`Compiler.bind_code_value`	Binds a code-value carrier that a meta call returned outside a macro value application as an application binds it, under a transaction that also covers its effects.
`Compiler.bind_macro_lisp_statement`	Binds a compile-time Lisp code `value` returned to a statement at `context`.
`Compiler.bind_meta_operation`	Binds `function`, the operation `lib/meta.x` declares as `name`, under its Lisp name in `lisp`.
`Compiler.capture_macro_value`	Returns an anonymous macro definition as a `Macro` value.
`Compiler.ensure_macro_lisp`	Opens the unit's compile-time Lisp session when it has none.
`Compiler.evaluate_declaration_effect`	Evaluates a queued source Lisp form with its original diagnostic site.
`Compiler.evaluate_declaration_recipe`	Evaluates a declaration recipe after its owning source is collected.
`Compiler.evaluate_macro_rows`	Evaluates a macro slot and returns its syntax as a row sequence.
`Compiler.evaluate_macro_slot`	Evaluates an active template's `(macro-slot ...)` value, or a pending Macro value application into the invocation that expands it.
`Compiler.evaluate_meta_value`	Evaluates the explicit meta call `expression` at `site` with the active expansion's captures visible to the SDK.
`Compiler.expand_macro_invocation_node`	Expands canonical macro capture rows and binds the result at `position`.
`Compiler.has_rewrites`	Tests one operation's registry without invoking a translator.
`Compiler.inherit_library_comptime`	Marks the shared session's compile-time-only definitions in a fresh compiler pass, which reads them without defining them.
`Compiler.inherits_import`	Whether the shared session evaluated the file at `path`.
`Compiler.install_builtin_macros`	Installs the compiler-shipped source macros into `c` once.
`Compiler.install_compile_time_effects`	Installs included compile-time definitions in their source order.
`Compiler.keyword_form_is_definition`	Returns whether the current tokens begin a `keyword NAME $macro` alias.
`Compiler.land_quotation`	Binds quoted `syntax` with its `fresh` rows at `position`, as the expansion of its quotation would.
`Compiler.library_filling`	Answers whether the shared compile-time session is still being filled, before `lib/meta.x` has defined the syntax builders.
`Compiler.lift_macro_lisp_expression`	Converts a compile-time Lisp value into a bound expression AST.
`Compiler.local_macro_form_is_definition`	Returns whether the current tokens begin a local macro definition.
`Compiler.lower_replacement`	Lowers `replacement`, which `Compiler.rewrite_lowered` bound for the lowered statement `source`, and keeps each part of `source` it holds as it is, so no statement rule is offered that part again.
`Compiler.lower_rewrite`	Binds and lowers a replacement while its rule remains active.
`Compiler.macro_form_is_definition`	Returns whether the current tokens have macro-definition introducer form.
`Compiler.macro_introduced_name`	Returns the current declaration's definition-local identity.
`Compiler.macro_invocation_collection`	Returns how collection treats the macro invocation at the cursor.
`Compiler.macro_invocation_site`	Resolves a stored macro invocation marker to its source token.
`Compiler.macro_lisp_starts_declaration`	Returns whether tokens after a Lisp form or explicit meta call continue a declaration.
`Compiler.macro_pattern_at`	Returns the `(` after a visible macro's `$NAME` at `at`, storing its spelling and definition, or NULL.
`Compiler.macro_starts_target_at`	Returns whether the parser claims the macro invocation at the cursor for `position`.
`Compiler.macro_tag_name`	Returns a template's local binding for tag `name` of `kind`, or NULL when it names a visible public tag.
`Compiler.macro_targets_unit`	Reports whether the macro invocation at the cursor produces file-scope syntax, directly or through the target it decorates.

Function	Summary
`Compiler.macro_value_literal`	Returns the literal expression that carries `value`, a macro definition or one of its references, as run-time data.
`Compiler.marked_rewrite`	Matches the member rules registered for each field keyword that `aggregate` declares a field with, as `Compiler.rewrite` matches those keyed by a receiver type.
`Compiler.open_macro_library`	Builds the shared compile-time session, leaving it open for the caller to fill and then publish.
`Compiler.parse_keyword_definition`	Parses and installs one source-local `keyword` alias.
`Compiler.parse_macro_definition`	Parses the macro definition at the current token into a `macrodef List`.
`Compiler.parse_macro_lisp_expression`	Parses a compile-time Lisp form in an expression position.
`Compiler.parse_macro_lisp_shallow`	Loads import dependencies and queues source Lisp effects.
`Compiler.parse_macro_lisp_top_level`	Evaluates one source Lisp form or imports an .xlisp dependency.
`Compiler.parse_macro_quotation`	Parses a quotation at `$!` into the code it builds from the locals its body names.
`Compiler.peek_macro_hole`	Returns the registered hole descriptor for a named or expression hole.
`Compiler.place`	Places `code` where `where` says, under the active expansion's transaction: `(after-statement)` after the enclosing block item or initialized declarator; `(block-exit)` on every exit from the enclosing block from that point on, as a `defer` placed after it; `(function-entry)` at the entry of the enclosing function, at parse or transform time, in placement order before its first statement; `(unit-support)` among the unit's support declarations, once per KEY with `(unit-support KEY)`; or `(unit-init)` in the unit's file initialization, in AREA with `(unit-init AREA)`.
`Compiler.publish_macro_definition_node`	Publishes a canonical `macrodef` in source order and returns `node`.
`Compiler.publish_macro_library`	Prepares and freezes the shared session and makes it every unit's parent.
`Compiler.rebound`	Records that binding built the statement `bound` again from `source`, a value the active lowering watches, so it watches `bound` as well.
`Compiler.rebuild_expression`	Rebuilds an expression from a pending Macro value application, preserving its established root `type`, child stage, and source wrappers.
`Compiler.rebuild_function`	Rebuilds a bound function through a Function decorator without binding it again.
`Compiler.rebuild_statement`	Rebuilds a statement template as a canonical unbraced `seq`.
`Compiler.rebuild_unit_function`	Constructs a fresh function from a Unit template after lowering.
`Compiler.record_compile_time_effect`	Records a public source effect where its completed definition stands.
`Compiler.record_comptime`	Records that `name` is compile-time only in `c`, and in the shared session when it is being filled.
`Compiler.report_lisp_failure`	Reports that the compile-time Lisp `source` failed with `error` at `invocation`.
`Compiler.reset_macro_library`	Ends the shared compile-time library before its build-target Context is reclaimed.
`Compiler.rewrite`	Matches registered patterns for one operation, then binds the first replacement.
`Compiler.rewrite_lowered`	Matches registered patterns for the statement `source`, which the transform has lowered, then binds the first replacement as `Compiler.rewrite` does.
`Compiler.rewrite_written`	Binds the replacement a shipped rule gives the lowered statement `source` of `kind`, which the kernel wrote while lowering, as `Compiler.rewrite_lowered` does.
`Compiler.shared_definitions`	Returns the published definition keys, or null before publication.
`Compiler.shares_meta_definition`	Answers whether the published shared session already holds the definition of `name` from this file, which a unit reading the file again leaves alone.
`Compiler.skip_macro_invocation`	Consumes a macro invocation name and its balanced argument list.

Function	Summary
Compiler.skip_named_type_declaration	Consumes a NamedType target already projected by owning-source collection.
Compiler.source_path	Returns the canonical path of the source file.
Compiler.take_code_value	Consumes a (code-value STAGE CODE EFFECTS) carrier a producer returned into a macro value application, or that Compiler.bind_code_value binds as one.
Compiler.try_parse_macro_expression	Parses and resolves a direct or keyword-alias expression macro.
Compiler.try_parse_macro_member	Parses a member name in a template.
Compiler.try_parse_macro_pattern	Parses case NAME(?a, *b) where NAME selects a macro, as $name or as a Macro variable, into the pattern expression that recognizes code the macro builds.
Compiler.try_parse_macro_slot	Parses a macro hole or Lisp slot for role while reading a template.
Compiler.try_parse_macro_subpattern	Parses $NAME(P, ...) in a pattern, where each argument is a pattern, into the pattern for code NAME builds with each argument pattern in its hole.
Compiler.try_parse_macro_target_at	Parses a direct or keyword-alias macro at the requested syntax position.
Compiler.watched_step	Lowers ast when it is a value the active lowering watches; NULL otherwise.
Compiler.written_support	The bound file-scope code bound that the rule now writing gives, written as the kernel's when that rule is shipped.

Functions

x2c_template_call

```
List x2c_template_call(Var stored, List values)
```

Returns the syntax that invokes the template stored, a macro name spelled as a String or a local definition, with values: what a template call in a staged meta body evaluates to.

Source: src/macros/templates.x:261

Compiler

Compiler.add_library_dependencies

```
void Compiler.add_library_dependencies(Compiler c)
```

Records the compile-time Lisp libraries as dependencies of c, as each use of its session does.

Source: src/macros/lisp.x:410

Compiler.after_hole

```
Token Compiler.after_hole(Compiler c)
```

Returns the token after the named or expression hole at the cursor. Consumes neither its $ or @ prefix nor its optional braces.

Source: src/macros/definitions/slots.x:109

Compiler.applying_shipped

```
int Compiler.applying_shipped(Compiler c)
```

Whether the rule whose meta function now runs is shipped, so that the names it calls resolve to the copies the compiler links.

Source: src/macros/rewrites.x:443

Compiler.bind_code_value

```
List Compiler.bind_code_value( Compiler c, List carrier, Symbol position, Type return_type)
```

Binds a code-value carrier that a meta call returned outside a macro value application as an application binds it, under a transaction that also covers its effects.

Source: `src/macros/values.x:340`

Compiler.bind_macro_lisp_statement

```
List Compiler.bind_macro_lisp_statement( Compiler c, List value, Symbol context)
```

Binds a compile-time Lisp code `value` returned to a statement at `context`.

Source: `src/macros/lisp.x:153`

Compiler.bind_meta_operation

```
void Compiler.bind_meta_operation(Lisp lisp, String name, Var function)
```

Binds `function`, the operation `lib/meta.x` declares as `name`, under its Lisp name in `lisp`. A definition the compile-time libraries already give that name wins, because it adapts the arguments.

Source: `src/macros/lisp.x:602`

Compiler.capture_macro_value

```
List Compiler.capture_macro_value(Compiler c, List definition)
```

Returns an anonymous macro definition as a `Macro` value. Macro values the body applies are captured where the literal is written, so a later application applies the same children.

Source: `src/macros/values.x:217`

Compiler.ensure_macro_lisp

```
void Compiler.ensure_macro_lisp(Compiler c)
```

Opens the unit's compile-time Lisp session when it has none. An Each use records the library files as dependencies, and a session loads them once.

Source: `src/macros/lisp.x:13`

Compiler.evaluate_declaration_effect

```
void Compiler.evaluate_declaration_effect( Compiler c, String form, Token invocation)
```

Evaluates a queued source Lisp form with its original diagnostic site.

Source: `src/macros/lisp.x:103`

Compiler.evaluate_declaration_recipe

```
Var Compiler.evaluate_declaration_recipe( Compiler c, Atom callback, List arguments)
```

Evaluates a declaration recipe after its owning source is collected. The callback is a Lisp name and arguments are retained canonical values.

Source: `src/macros/templates.x:243`

Compiler.evaluate_macro_rows

```
List Compiler.evaluate_macro_rows(Compiler c, Var value)
```

Evaluates a macro slot and returns its syntax as a row sequence. `(seq ...)` contributes its children; every other result contributes one row.

Source: `src/macros/templates.x:91`

Compiler.evaluate_macro_slot

```
Var Compiler.evaluate_macro_slot(Compiler c, Var value)
```

Evaluates an active template's `(macro-slot ...)` value, or a pending Macro value application into the invocation that expands it. Non-slots and slots outside an expansion are returned unchanged. A splice slot's `List` result is wrapped as `(seq ...)` for its syntax position.

Source: `src/macros/templates.x:32`

Compiler.evaluate_meta_value

```
Var Compiler.evaluate_meta_value( Compiler c, List expression, Token site, int slot)
```

Evaluates the explicit meta call `expression` at `site` with the active expansion's captures visible to the SDK. A nonzero `slot` marks a template slot, which reports a call the project meta build leaves.

Source: `src/macros/templates.x:130`

Compiler.expand_macro_invocation_node

```
List Compiler.expand_macro_invocation_node( Compiler c, Var stored, List arguments, Token invocation, Symbol position)
```

Expands canonical macro capture rows and binds the result at `position`. `stored` is a `macrodef` or visible macro name. The active expansion stack supplies Lisp bindings, source location, and recursion checks; the definition's fresh rows allocate invocation-local names. This method does not begin a semantic transaction.

Source: `src/macros/expansion.x:22`

Compiler.has_rewrites

```
int Compiler.has_rewrites(Compiler c, Symbol point, Var kind)
```

Tests one operation's registry without invoking a translator.

Source: `src/macros/rewrites.x:230`

Compiler.inherit_library_comptime

```
void Compiler.inherit_library_comptime(Compiler c)
```

Marks the shared session's compile-time-only definitions in a fresh compiler pass, which reads them without defining them.

Source: `src/macros/lisp.x:465`

Compiler.inherits_import

```
int Compiler.inherits_import(String path)
```

Whether the shared session evaluated the file at `path`. A unit that is that file, as when a library is linted, inherits its Lisp forms.

Source: `src/macros/lisp.x:448`

Compiler.install_builtin_macros

```
void Compiler.install_builtin_macros(Compiler c)
```

Installs the compiler-shipped source macros into `c` once.

Source: `src/macros/lisp.x:513`

Compiler.install_compile_time_effects

```
void Compiler.install_compile_time_effects(Compiler c, List rows)
```

Installs included compile-time definitions in their source order. Keyword aliases retain the definition visible when their source declared them, even if a later macro definition uses the same name.

Source: `src/macros/invocations/recognition.x:41`

Compiler.keyword_form_is_definition

```
int Compiler.keyword_form_is_definition(Compiler c)
```

Returns whether the current tokens begin a `keyword NAME $macro` alias. This query does not consume tokens.

Source: `src/macros/invocations/recognition.x:105`

Compiler.land_quotation

```
List Compiler.land_quotation( Compiler c, List fresh, Var syntax, Symbol position, Type return_type)
```

Binds quoted `syntax` with its `fresh` rows at `position`, as the expansion of its quotation would.

Source: `src/macros/templates.x:433`

Compiler.library_filling

```
int Compiler.library_filling(void)
```

Answers whether the shared compile-time session is still being filled, before `lib/meta.x` has defined the syntax builders.

Source: `src/macros/lisp.x:472`

Compiler.lift_macro_lisp_expression

```
List Compiler.lift_macro_lisp_expression( Compiler c, Var value, Token invocation)
```

Converts a compile-time Lisp value into a bound expression AST. Scalars, immutable values, representable Array/Map roots, compiler-issued identifiers, and nonempty code `Lists` are accepted; `invocation` locates an unsupported result.

Source: `src/macros/lisp.x:139`

Compiler.local_macro_form_is_definition

```
int Compiler.local_macro_form_is_definition(Compiler c)
```

Returns whether the current tokens begin a local macro definition. This query does not consume tokens.

Source: `src/macros/invocations/recognition.x:95`

Compiler.lower_replacement

```
List Compiler.lower_replacement(Compiler c, List replacement, List source)
```

Lowers `replacement`, which `Compiler.rewrite_lowered` bound for the lowered statement `source`, and keeps each part of `source` it holds as it is, so no statement rule is offered that part again.

Source: `src/macros/rewrites.x:255`

Compiler.lower_rewrite

```
List Compiler.lower_rewrite( Compiler c, Symbol point, Var kind, List source, Symbol position, Type expected, Token site)
```

Binds and lowers a replacement while its rule remains active. The source expressions the replacement holds lower outside the rule, so a nested use of the rule applies again. A statement replacement, or a lowered one,

also lowers the statements its source held outside the rule, and lowers once, so a new statement it writes stays outside the rule. Code a shipped rule's replacement writes, such as the support call an interpolated String or a collection literal lowers to, is not source, so only shipped rules see it. An expression retains the result type already established for its parent. A replacement whose template and slot signatures an earlier use in the unit prepared lowers only its captured values.

Source: `src/macros/rewrites.x:286`

Compiler.macro_form_is_definition

```
int Compiler.macro_form_is_definition(Compiler c)
```

Returns whether the current tokens have macro-definition introducer form. This query does not consume tokens.

Source: `src/macros/invocations/recognition.x:81`

Compiler.macro_introduced_name

```
List Compiler.macro_introduced_name(Compiler c, String spelling)
```

Returns the current declaration's definition-local identity. An active macro-definition locals map is required.

Source: `src/macros/definitions/templates.x:264`

Compiler.macro_invocation_collection

```
Symbol Compiler.macro_invocation_collection(Compiler c)
```

Returns how collection treats the macro invocation at the cursor. `<required>` covers every file-scope `Unit` macro, declaration-unit macro, named-type decorator, and local `Unit` macro whose template contains protocol or adoption rows. `<tried>` covers Unit decorators and other local `Unit` macros. Other invocations return zero.

Source: `src/macros/invocations/recognition.x:358`

Compiler.macro_invocation_site

```
Token Compiler.macro_invocation_site(Compiler c, Var site)
```

Resolves a stored macro invocation marker to its source token. Nested template markers use the active expansion's invocation; unresolved markers return NULL.

Source: `src/macros/expansion.x:235`

Compiler.macro_lisp_starts_declaration

```
int Compiler.macro_lisp_starts_declaration(Compiler c)
```

Returns whether tokens after a Lisp form or explicit meta call continue a declaration. The balanced argument group is inspected without moving the compiler cursor; a visible source macro retains its own grammar.

Source: `src/macros/definitions/slots.x:181`

Compiler.macro_pattern_at

```
Token Compiler.macro_pattern_at( Compiler c, Token at, String &spelling, Var &stored)
```

Returns the `(` after a visible macro's `$NAME` at `at`, storing its spelling and definition, or NULL. The cursor does not move.

Source: `src/macros/values.x:106`

Compiler.macro_starts_target_at

```
int Compiler.macro_starts_target_at(Compiler c, Symbol position)
```

Returns whether the parser claims the macro invocation at the cursor for `position`. A bare keyword alias is claimed only where its result fits. This query does not consume tokens.

Source: `src/macros/invocations/recognition.x:172`

Compiler.macro_tag_name

```
List Compiler.macro_tag_name( Compiler c, Symbol kind, String name, int definition)
```

Returns a template's local binding for tag `name` of `kind`, or NULL when it names a visible public tag. A tag the template defines or declares is a template local, apart from ordinary names of the same spelling. A tag it only references keeps its public spelling unless the template later defines or declares it.

Source: `src/macros/definitions/templates.x:277`

Compiler.macro_targets_unit

```
int Compiler.macro_targets_unit(Compiler c)
```

Reports whether the macro invocation at the cursor produces file-scope syntax, directly or through the target it decorates. A script unit keeps such an invocation at file scope. This query does not consume tokens.

Source: `src/macros/invocations/recognition.x:402`

Compiler.macro_value_literal

```
List Compiler.macro_value_literal(Compiler c, List value)
```

Returns the literal expression that carries `value`, a macro definition or one of its references, as run-time data.

Source: `src/macros/values.x:12`

Compiler.marked_rewrite

```
List Compiler.marked_rewrite( Compiler c, Type aggregate, List source, Token site)
```

Matches the member rules registered for each field keyword that `aggregate` declares a field with, as `Compiler.rewrite` matches those keyed by a receiver type.

Source: `src/macros/rewrites.x:216`

Compiler.open_macro_library

```
Lisp Compiler.open_macro_library(Compiler c)
```

Builds the shared compile-time session, leaving it open for the caller to fill and then publish. Returns the session, or null when this home cannot preload, in which case every unit falls back to its own load, which reports against the unit that needed it.

The session is not published until `Compiler.publish_macro_library`, so a unit opened in between still builds its own.

Source: `src/macros/lisp.x:358`

Compiler.parse_keyword_definition

```
Atom Compiler.parse_keyword_definition(Compiler c)
```

Parses and installs one source-local `keyword` alias. The named macro must already be visible; the alias captures that definition and consumes its terminating semicolon. Returns the alias.

Source: `src/macros/invocations/recognition.x:117`

Compiler.parse_macro_definition

```
List Compiler.parse_macro_definition(Compiler c)
```

Parses the macro definition at the current token into a `macrodef List`. A source-level definition is published immediately; a definition inside a template remains syntax for later binding at its expansion site.

Source: `src/macros/definitions/reading.x:20`

Compiler.parse_macro_lisp_expression

```
List Compiler.parse_macro_lisp_expression(Compiler c)
```

Parses a compile-time Lisp form in an expression position. Macro-definition parsing records a deferred slot; ordinary parsing evaluates the form in the translation unit's Lisp session and lifts it.

Source: `src/macros/lisp.x:124`

Compiler.parse_macro_lisp_shallow

```
void Compiler.parse_macro_lisp_shallow(Compiler c)
```

Loads import dependencies and queues source Lisp effects. Declaration projection forces preceding effects exactly once; otherwise full parsing keeps the ordinary source-order evaluation.

Source: `src/macros/lisp.x:69`

Compiler.parse_macro_lisp_top_level

```
List Compiler.parse_macro_lisp_top_level(Compiler c)
```

Evaluates one source Lisp form or imports an .xlisp dependency.

Source: `src/macros/lisp.x:52`

Compiler.parse_macro_quotation

```
List Compiler.parse_macro_quotation(Compiler c)
```

Parses a quotation at `$!` into the code it builds from the locals its body names.

Source: `src/macros/definitions/quotations.x:19`

Compiler.peek_macro_hole

```
List Compiler.peek_macro_hole(Compiler c)
```

Returns the registered hole descriptor for a named or expression hole. `$NAME` and `${expression}` insert one value; `@NAME` and `@{expression}` splice a sequence. Returns NULL without consuming a non-hole spelling.

Source: `src/macros/definitions/slots.x:95`

Compiler.place

```
void Compiler.place(Compiler c, List where, Var code)
```

Places `code` where `where` says, under the active expansion's transaction: `(after-statement)` after the enclosing block item or initialized declarator; `(block-exit)` on every exit from the enclosing block from that point on, as a `defer` placed after it; `(function-entry)` at the entry of the enclosing function, at parse or transform time, in placement order before its first statement; `(unit-support)` among the unit's support declarations, once per KEY with `(unit-support KEY)`; or `(unit-init)` in the unit's file initialization, in AREA with `(unit-init AREA)`.

Source: `src/macros/values.x:305`

Compiler.publish_macro_definition_node

```
List Compiler.publish_macro_definition_node(Compiler c, List node)
```

Publishes a canonical `macrodef` in source order and returns `node`. A later definition with the same name affects only later invocations.

Source: src/macros/invocations/recognition.x:14

Compiler.publish_macro_library

```
void Compiler.publish_macro_library(Compiler c, Lisp shared)
```

Prepares and freezes the shared session and makes it every unit's parent. shared must be the session Compiler.open_macro_library returned.

Source: src/macros/lisp.x:430

Compiler.rebound

```
void Compiler.rebound(Compiler c, Var source, List bound)
```

Records that binding built the statement bound again from source, a value the active lowering watches, so it watches bound as well. A statement without an origin, such as an unbraced body, keeps nothing else that identifies it when a binder it uses binds again.

Source: src/macros/rewrites.x:694

Compiler.rebuild_expression

```
List Compiler.rebuild_expression(Compiler c, Type type, List application)
```

Rebuilds an expression from a pending Macro value application, preserving its established root type, child stage, and source wrappers. Binding, capture collection, hygiene, and effects do not run, and a statement expression's template-origin wrappers are omitted.

Source: src/macros/templates.x:347

Compiler.rebuild_function

```
List Compiler.rebuild_function(Compiler c, List target, List application)
```

Rebuilds a bound function through a Function decorator without binding it again. The template keeps the target's return type and declarator.

Source: src/macros/templates.x:369

Compiler.rebuild_statement

```
List Compiler.rebuild_statement(Compiler c, List application)
```

Rebuilds a statement template as a canonical unbraced seq. Bound children keep their identities and origins; template-origin wrappers are omitted because this path does not open an invocation.

Source: src/macros/templates.x:355

Compiler.rebuild_unit_function

```
List Compiler.rebuild_unit_function(Compiler c, List application)
```

Constructs a fresh function from a Unit template after lowering. The caller supplies its bound name and lowered children; binding does not run.

Source: src/macros/templates.x:361

Compiler.record_compile_time_effect

```
void Compiler.record_compile_time_effect( Compiler c, List row, Token start)
```

Records a public source effect where its completed definition stands.

Source: src/macros/invocations/recognition.x:24

Compiler.record_comptime

```
void Compiler.record_comptime(Compiler c, String name)
```

Records that `name` is compile-time only in `c`, and in the shared session when it is being filled.

Source: `src/macros/lisp.x:456`

Compiler.report_lisp_failure

```
void Compiler.report_lisp_failure( Compiler c, Token invocation, List error, String source)
```

Reports that the compile-time Lisp `source` failed with `error` at `invocation`.

Source: `src/macros/diagnostics.x:357`

Compiler.reset_macro_library

```
void Compiler.reset_macro_library(void)
```

Ends the shared compile-time library before its build-target Context is reclaimed. A later target creates a fresh session in its own Context.

Source: `src/macros/lisp.x:629`

Compiler.rewrite

```
List Compiler.rewrite( Compiler c, Symbol point, Var kind, List source, Symbol position, Type expected, Token site)
```

Matches registered patterns for one operation, then binds the first replacement. A translator declines with void, null, or its input itself. Active rules cannot re-enter. A statement of the same kind that a statement replacement writes or copies, such as its source, takes the rules after the one that replaced it.

Source: `src/macros/rewrites.x:238`

Compiler.rewrite_lowered

```
List Compiler.rewrite_lowered( Compiler c, Symbol point, Var kind, List source, Symbol position, Type expected, Token site)
```

Matches registered patterns for the statement `source`, which the transform has lowered, then binds the first replacement as `Compiler.rewrite` does. Binding keeps the source, and each part of it the replacement holds, as lowered.

Source: `src/macros/rewrites.x:247`

Compiler.rewrite_written

```
List Compiler.rewrite_written( Compiler c, Symbol kind, List source, Type expected)
```

Binds the replacement a shipped rule gives the lowered statement `source` of `kind`, which the kernel wrote while lowering, as `Compiler.rewrite_lowered` does. A statement the kernel writes is not source, so no program rule sees it.

Source: `src/macros/rewrites.x:267`

Compiler.shared_definitions

```
Map Compiler.shared_definitions(Compiler c)
```

Returns the published definition keys, or null before publication. The borrowed map is the shared session's source identity table.

Source: `src/macros/lisp.x:477`

Compiler.shares_meta_definition

```
int Compiler.shares_meta_definition(Compiler c, String name)
```

Answers whether the published shared session already holds the definition of `name` from this file, which a unit reading the file again leaves alone.

Source: `src/macros/lisp.x:485`

Compiler.skip_macro_invocation

```
void Compiler.skip_macro_invocation(Compiler c)
```

Consumes a macro invocation name and its balanced argument list. The invocation terminator or following decorator target remains current.

Source: `src/macros/invocations/recognition.x:378`

Compiler.skip_named_type_declaration

```
int Compiler.skip_named_type_declaration(Compiler c)
```

Consumes a NamedType target already projected by owning-source collection. CPP scanning does not produce the declaration or parse its fields again.

Source: `src/macros/invocations/recognition.x:389`

Compiler.source_path

```
String Compiler.source_path(Compiler c, String file)
```

Returns the canonical path of the source `file`. A relative name that is not a file resolves against the home.

Source: `src/macros/lisp.x:292`

Compiler.take_code_value

```
int Compiler.take_code_value( Compiler c, Var input, Var &value, int &retained)
```

Consumes a `(code-value STAGE CODE EFFECTS)` carrier a producer returned into a macro value application, or that `Compiler.bind_code_value` binds as one. Effects are applied in order under the application's transaction, and their tokens are replaced in the code. `retained` reports a bound or lowered stage, which ordinary binding leaves untouched. Lowered code is not searched for a leftover binder. Returns 0 for any other value.

Source: `src/macros/values.x:255`

Compiler.try_parse_macro_expression

```
List Compiler.try_parse_macro_expression(Compiler c)
```

Parses and resolves a direct or keyword-alias expression macro. Returns NULL without consuming an identifier that is not an applicable alias; a direct `$` invocation must resolve to a visible expression form.

Source: `src/macros/invocations/expressions.x:15`

Compiler.try_parse_macro_member

```
List Compiler.try_parse_macro_member(Compiler c)
```

Parses a member name in a template. A singular `Name` hole there supplies the spelling it captured, without hygienic renaming.

Source: `src/macros/definitions/slots.x:197`

Compiler.try_parse_macro_pattern

```
List Compiler.try_parse_macro_pattern(Compiler c)
```

Parses `case NAME(?a, *b)` where NAME selects a macro, as `$name` or as a `Macro` variable, into the pattern expression that recognizes code the macro builds. Returns NULL without consuming tokens for any other case.

Source: `src/macros/values.x:40`

Compiler.try_parse_macro_slot

```
List Compiler.try_parse_macro_slot(Compiler c, Symbol role)
```

Parses a macro hole or Lisp slot for `role` while reading a template. Returns role-shaped syntax containing `(macro-bind ...)` or `(macro-slot ...)`, or NULL when ordinary grammar owns the current tokens; successful parsing advances the cursor.

Source: `src/macros/definitions/slots.x:14`

Compiler.try_parse_macro_subpattern

```
List Compiler.try_parse_macro_subpattern(Compiler c, int content)
```

Parses `$NAME(P, ...)` in a pattern, where each argument is a pattern, into the pattern for code NAME builds with each argument pattern in its hole. It matches the shelled expression or its bare content; `content` selects the content alone, for a position whose pattern already writes the `(expr TYPE ...)` shell. Returns NULL without consuming tokens when no macro NAME is visible.

Source: `src/macros/values.x:142`

Compiler.try_parse_macro_target_at

```
List Compiler.try_parse_macro_target_at(Compiler c, Symbol position)
```

Parses a direct or keyword-alias macro at the requested syntax position. Returns NULL without consuming a macro hole or an invocation that `macro_starts_target_at` does not claim; template parsing returns a deferred `(seq (macro-invoke ...))`, and ordinary parsing returns the bound expansion.

Source: `src/macros/invocations/targets.x:54`

Compiler.watched_step

```
Ast Compiler.watched_step(Compiler c, Ast ast)
```

Lowers `ast` when it is a value the active lowering watches; NULL otherwise. A preparing watch also records what lowering produced. A value that an enclosing replacement's source held as well lowers outside that replacement's rule too, and that replacement still watches its own values within it.

Source: `src/macros/rewrites.x:707`

Compiler.written_support

```
List Compiler.written_support(Compiler c, List bound)
```

The bound file-scope code `bound` that the rule now writing gives, written as the kernel's when that rule is shipped. A rule writes while its meta function runs, until a rule it reaches binds, and while its replacement binds and lowers. What a program writes, or a placement outside any rule, is source.

Source: `src/macros/rewrites.x:434`

Design notes

This module owns macro expansion. Expansion fills a definition's template from an invocation's captures and binds the result through the operations that bind parsed source. A definition is a compiler-only `macrodef` List: its holes, a Match pattern over an invocation's capture rows, and a template of the binders that pattern captures. Template slots and `$(...)` forms run in the unit's compile-time Lisp session, a child of the shared library session filled once per target.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/meta-group/meta-group.x`

A unit's meta group, emitted as C.

Functions

Function	Summary
`Compiler.bind_meta_group`	Binds the session's group function `name` when it is not bound yet, by staging the group, and reports at `site` a function that cannot run.
`Compiler.finish_meta_functions`	Closes compile-time-only calls across later meta definitions.
`Compiler.group_meta_function`	Records a native callable body `fn` in the unit's helper group.
`Compiler.groups_meta`	Answers whether a `meta` function or value belongs to the unit's group: a parse meets it outside a macro definition while the project meta build parses the unit or a session stages it.
`Compiler.meta_call_missing`	Returns why the group function `name` has no compile-time entry, from its type.
`Compiler.meta_calls_wait`	Answers whether a call of a project `meta` function waits for the translation: the project meta build's own parse has no helper yet.
`Compiler.meta_cc`	Returns the C compiler and the runtime header directory that build `meta` code, or NULL before `Compiler.use_meta_toolchain`.
`Compiler.meta_cc_identity`	Returns the identity of the C compiler at `cc`: its path and content hash.
`Compiler.meta_cc_run`	Runs `arguments`, a C compiler command building the group C in `directory`, and returns NULL, or else its first error, which names `directory` when a group's C is the cause.
`Compiler.meta_reaches_compile_time`	Answers whether a `meta` body reaches the compiler itself: it names a compile-time-only function or a compiler operation, or constructs a template.
`Compiler.prepare_loaded_functions`	Adds a loaded unit's public functions and object accessors to its targets.
`Compiler.prepare_loaded_main`	Exposes a loaded unit's main through the ordinary helper target table.
`Compiler.public_native_function`	Answers whether a definition has public native linkage.
`Compiler.refuse_meta_call`	Reports at `site` that the `meta` function `name` cannot run at compile time, and `why`.
`Compiler.refuse_record_meta_call`	Reports at `site` that the group function `name` cannot run at compile time when its result is a struct or union, which its type alone decides.
`Compiler.stage_meta_in_process`	Enables in-process staging of `meta` groups for compiler sessions.
`Compiler.stage_session_function`	Stages one session factory in an isolated native emission.
`Compiler.use_meta_build_directory`	Directs each provider group into `directory`, using `owners` to name their headers by table, or stops that when `directory` is NULL.
`Compiler.use_meta_toolchain`	Selects the C compiler `cc` that builds the `meta` code of the units this process translates, with the runtime headers this compiler was built with, found from its installed headers in `include_dir`.
`Compiler.write_meta_build`	Writes the group of a unit the project meta build parsed into the build directory as `group-K.c` and `group-K.h`, K being its table, with the x2c sources it read in `group-K.deps`, or its failure in `group-K.failure`.

`Compiler`

Compiler.bind_meta_group

```
void Compiler.bind_meta_group(Compiler c, String name, Token site)
```

Binds the session's group function `name` when it is not bound yet, by staging the group, and reports at `site` a function that cannot run.

Source: `src/meta-group/staging.x:54`

Compiler.finish_meta_functions

```
void Compiler.finish_meta_functions(Compiler c, Array nodes)
```

Closes compile-time-only calls across later meta definitions. Retained group bodies hold the bound calls, including literal insertions. Select runtime definitions only after that relation is complete.

Source: `src/meta-group/selection.x:62`

Compiler.group_meta_function

```
void Compiler.group_meta_function(Compiler c, List fn)
```

Records a native callable body `fn` in the unit's helper group.

Source: `src/meta-group/selection.x:28`

Compiler.groups_meta

```
int Compiler.groups_meta(Compiler c)
```

Answers whether a `meta` function or value belongs to the unit's group: a parse meets it outside a macro definition while the project meta build parses the unit or a session stages it.

Source: `src/meta-group/selection.x:18`

Compiler.meta_call_missing

```
String Compiler.meta_call_missing(Compiler c, String name)
```

Returns why the group function `name` has no compile-time entry, from its type.

Source: `src/meta-group/selection.x:283`

Compiler.meta_calls_wait

```
int Compiler.meta_calls_wait(Compiler c)
```

Answers whether a call of a project `meta` function waits for the translation: the project meta build's own parse has no helper yet. A loaded helper's parse calls the ordinary helper of its sources.

Source: `src/meta-group/selection.x:25`

Compiler.meta_cc

```
String Compiler.meta_cc(String &include_dir)
```

Returns the C compiler and the runtime header directory that build `meta` code, or NULL before `Compiler.use_meta_toolchain`.

Source: `src/meta-group/toolchain.x:31`

Compiler.meta_cc_identity

```
String Compiler.meta_cc_identity(String cc)
```

Returns the identity of the C compiler at `cc`: its path and content hash.

Source: `src/meta-group/toolchain.x:38`

Compiler.meta_cc_run

```
String Compiler.meta_cc_run(List arguments, String directory)
```

Runs `arguments`, a C compiler command building the group C in `directory`, and returns NULL, or else its first error, which names `directory` when a group's C is the cause.

Source: `src/meta-group/toolchain.x:47`

Compiler.meta_reaches_compile_time

```
int Compiler.meta_reaches_compile_time(Compiler c, Var node)
```

Answers whether a `meta` body reaches the compiler itself: it names a compile-time-only function or a compiler operation, or constructs a template. Such a function has no runtime form.

Source: `src/meta-group/selection.x:41`

Compiler.prepare_loaded_functions

```
void Compiler.prepare_loaded_functions(Compiler c, Array nodes)
```

Adds a loaded unit's public functions and object accessors to its targets. Meta closure must finish first so these entries retain runtime semantics. Existing meta entries and the adapted main keep their original targets.

Source: `src/meta-group/loaded.x:79`

Compiler.prepare_loaded_main

```
void Compiler.prepare_loaded_main(Compiler c, Array nodes)
```

Exposes a loaded unit's main through the ordinary helper target table. Native calls retain its declared signature; the public argc/argv entry takes an Array of Strings. A renamed main still returns zero on falling through its closing brace. Source text and tokens are unchanged.

Source: `src/meta-group/loaded.x:56`

Compiler.public_native_function

```
int Compiler.public_native_function( Compiler c, List function, String name)
```

Answers whether a definition has public native linkage. Main is excluded because its process entry needs a separate adapter.

Source: `src/meta-group/selection.x:273`

Compiler.refuse_meta_call

```
void Compiler.refuse_meta_call( Compiler c, String name, Token site, String why)
```

Reports at `site` that the `meta` function `name` cannot run at compile time, and `why`.

Source: `src/meta-group/selection.x:302`

Compiler.refuse_record_meta_call

```
void Compiler.refuse_record_meta_call(Compiler c, String name, Token site)
```

Reports at `site` that the group function `name` cannot run at compile time when its result is a struct or union, which its type alone decides.

Source: `src/meta-group/selection.x:310`

Compiler.stage_meta_in_process

```
void Compiler.stage_meta_in_process(void)
```

Enables in-process staging of `meta` groups for compiler sessions.

Source: `src/meta-group/selection.x:13`

Compiler.stage_session_function

```
Map Compiler.stage_session_function( Compiler c, List declarations, Func prepare, String &failure)
```

Stages one session factory in an isolated native emission. `prepare` receives this compiler and returns the factory's typed definition; it can place supporting functions through the ordinary compiler APIs. `declarations` supplies session types. Returns the loaded module's targets, or NULL with `failure` set. Publishes no evaluator bindings.

Source: `src/meta-group/staging.x:92`

Compiler.use_meta_build_directory

```
void Compiler.use_meta_build_directory( String directory, Array owners, Map providers)
```

Directs each provider group into `directory`, using `owners` to name their headers by table, or stops that when `directory` is NULL.

Source: `src/meta-group/staging.x:13`

Compiler.use_meta_toolchain

```
void Compiler.use_meta_toolchain(String cc, String include_dir)
```

Selects the C compiler `cc` that builds the `meta` code of the units this process translates, with the runtime headers this compiler was built with, found from its installed headers in `include_dir`.

Source: `src/meta-group/toolchain.x:9`

Compiler.write_meta_build

```
void Compiler.write_meta_build(Compiler c)
```

Writes the group of a unit the project meta build parsed into the build directory as `group-K.c` and `group-K.h`, K being its table, with the x2c sources it read in `group-K.deps`, or its failure in `group-K.failure`. Included native and type providers use the same group header and object even when they have no `meta` functions.

Source: `src/meta-group/staging.x:27`

Design notes

A bodied `meta` function a project defines runs as native code: in the project's helper program, which the project meta build compiles from each unit's `meta` group before translation (`meta-project.x`), or in a native module that a session stages in process. This module owns the group, from the functions it holds to the C it emits.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/meta-native/meta-native.x`

Meta functions and the native code they call.

Functions

Function	Summary
`x2c_register_extension`	Registers package `name`'s compile-time part, linked into the compiler, whose `targets` returns its name-to-`Func` Map.
`Compiler.add_native_module`	Records the name-to-`Func` Map that the entry of the native module loaded from absolute `path` returns.
`Compiler.apply_meta_function`	Applies the meta function spelled `name` to the evaluated `arguments` at `site`, outside any expansion.
`Compiler.bind_linked_meta`	Binds the bodied `meta` definition `fn`, of function type `type`, to the compiler's linked copy of it when the two definition texts hash the same, or always for a prelude component's, instead of staging it.
`Compiler.bind_native_meta`	Binds an included native `meta` function the first time compile-time code calls `name`.
`Compiler.bind_project_meta`	Binds an advertised meta function and reports whether it was found.

Function	Summary
`Compiler.compiler_targets`	Returns the compiler's own targets, the operations `lib/meta.x` declares with a bodyless `meta` prototype, loaded as the native module every request selects first.
`Compiler.evaluate_meta_expression`	Executes an explicit meta call and inserts its result at a code boundary.
`Compiler.evaluate_meta_statement`	Executes an explicit meta call written as a whole statement at `context`.
`Compiler.extension_archive`	Returns the archive of the linked packages' objects that the build of this compiler kept beside it, named by the compiler's identity, or NULL when the compiler links none.
`Compiler.install_collected_meta_function`	Installs the stub of the bodied `meta` definition `declaration` whose body collection skips, so a file-scope constant after it can call it in the project's helper.
`Compiler.install_meta_declaration`	Applies a contextual `meta` marker to one initialized file-static value.
`Compiler.install_meta_function`	Installs a bodied `meta` function.
`Compiler.install_native_meta_effects`	Records the native advertisements retained by included interfaces.
`Compiler.install_native_meta_function`	Installs a prototype-only `meta` function from the compiler's trusted native target registry.
`Compiler.install_project_meta_effect`	Installs an included meta function's evaluator advertisement.
`Compiler.linked_meta_provider_source`	Returns a linked provider's source and dependency hashes.
`Compiler.links_extension`	Reports whether package `name`'s compile-time part is linked into the compiler, so its import loads no module.
`Compiler.links_meta_function`	Reports whether the compiler links the `meta` function `name`.
`Compiler.load_native_module`	Loads the native module at `path` once per process and returns its absolute path.
`Compiler.matches_macro`	Tests a registered macro with the candidate's global and source bindings.
`Compiler.native_meta_accepts`	Answers whether the native target `function` matches the declared `signature`.
`Compiler.native_meta_module`	Returns the selected file-backed native module that supplies `name`, with its native `type`, or NULL for a linked or absent target.
`Compiler.native_meta_targets`	Returns the declared native targets advertised by `meta` interface rows, in the row form `lib/lisp-targets.x` generates its target inventory from, or only those declared in the files `paths` names when it is not empty.
`Compiler.native_module_loaded`	Reports whether the native module at absolute `path` is loaded.
`Compiler.native_module_targets`	Returns the name-to-`Func` Map of the loaded native module at absolute `path`.
`Compiler.preload_native_module`	Loads the native module at absolute `path` when this compiler built it and the platform loads modules.
`Compiler.project_meta_uses_linked`	Answers whether a provider's function can use its linked body.
`Compiler.record_native_meta_effect`	The shallow interface retains the advertisement separately from the C declaration.
`Compiler.record_project_meta_effect`	Records the provider of a public bodied meta function in its interface.
`Compiler.run_meta_call`	Runs the explicit meta call `expression` at `site`.
`Compiler.select_native_modules`	Selects the loaded native modules, by absolute path, that bodyless `meta` prototypes bind in the current request.
`Compiler.select_package_module`	Selects package `name`'s native module, when it has one, after the modules already selected, and records it as a prerequisite of the unit.
`Compiler.supplies_native_meta`	Reports whether the compiler itself supplies the native function `name`.

Functions

x2c_register_extension

```
void x2c_register_extension(const char *name, Map (*targets)(void))
```

Registers package `name`'s compile-time part, linked into the compiler, whose `targets` returns its name-to-`Func` Map. The registration unit `x2c build --extension` generates calls it from a constructor.

Source: `src/meta-native/modules.x:237`

`Compiler`

Compiler.add_native_module

```
void Compiler.add_native_module(String path, Map (*entry)(void))
```

Records the name-to-`Func` Map that the entry of the native module loaded from absolute `path` returns. The Funcs, names, signatures, and path last for the process.

Source: `src/meta-native/modules.x:178`

Compiler.apply_meta_function

```
Var Compiler.apply_meta_function( Compiler c, String name, List arguments, Token site)
```

Applies the meta function spelled `name` to the evaluated `arguments` at `site`, outside any expansion. A call the project meta build leaves for the translation returns void.

Source: `src/meta-native/calls.x:78`

Compiler.bind_linked_meta

```
int Compiler.bind_linked_meta(Compiler c, List fn, Type type)
```

Binds the bodied `meta` definition `fn`, of function type `type`, to the compiler's linked copy of it when the two definition texts hash the same, or always for a prelude component's, instead of staging it. Returns whether it did; another edited definition is staged as user code.

Source: `src/meta-native/linked.x:60`

Compiler.bind_native_meta

```
int Compiler.bind_native_meta(Compiler c, String name)
```

Binds an included native `meta` function the first time compile-time code calls `name`. Returns whether the macro session now binds it. A declaration-producing macro can call one during collection, before the parse installs the advertisements, so the first lookup there installs the ones visible so far.

Source: `src/meta-native/bindings.x:64`

Compiler.bind_project_meta

```
int Compiler.bind_project_meta(Compiler c, String name)
```

Binds an advertised meta function and reports whether it was found. Its provider supplies the helper table and private helpers.

Source: `src/meta-native/functions.x:229`

Compiler.compiler_targets

```
Map Compiler.compiler_targets(void)
```

Returns the compiler's own targets, the operations `lib/meta.x` declares with a bodyless `meta` prototype, loaded as the native module every request selects first.

Source: `src/meta-native/modules.x:39`

Compiler.evaluate_meta_expression

```
List Compiler.evaluate_meta_expression( Compiler c, List expression, Token site)
```

Executes an explicit meta call and inserts its result at a code boundary.

Source: `src/meta-native/calls.x:13`

Compiler.evaluate_meta_statement

```
List Compiler.evaluate_meta_statement( Compiler c, List expression, Symbol context, Token site)
```

Executes an explicit meta call written as a whole statement at `context`. A code List result binds there, so the call may return a statement as well as an expression.

Source: `src/meta-native/calls.x:21`

Compiler.extension_archive

```
String Compiler.extension_archive(void)
```

Returns the archive of the linked packages' objects that the build of this compiler kept beside it, named by the compiler's identity, or NULL when the compiler links none. Project meta code links it.

Source: `src/meta-native/modules.x:255`

Compiler.install_collected_meta_function

```
void Compiler.install_collected_meta_function( Compiler c, List declaration, Token marker)
```

Installs the stub of the bodied `meta` definition `declaration` whose body collection skips, so a file-scope constant after it can call it in the project's helper. The full parse installs it again. The compiler's linked copy answers while its source hashes agree; otherwise the definition replaces an earlier binding with a helper stub. A group that stages in process waits for the full parse.

Source: `src/meta-native/functions.x:43`

Compiler.install_meta_declaration

```
void Compiler.install_meta_declaration( Compiler c, List declaration, Token marker)
```

Applies a contextual `meta` marker to one initialized file-static value. The unit's staged `meta` group holds the compile-time instance, which its module initializes for each unit.

Source: `src/meta-native/functions.x:184`

Compiler.install_meta_function

```
void Compiler.install_meta_function(Compiler c, List fn, Token marker)
```

Installs a bodied `meta` function. The project meta build compiled it into the project's helper, so the session holds a stub that calls it there; the REPL's stub stages the session's group at the first call instead, and the project meta build's own parse only groups it. A function whose body reaches a compiler operation, a template, or a compile-time-only function has no runtime form. A body that lets its own storage outlive a call is rejected where the storage leaves.

Source: `src/meta-native/functions.x:20`

Compiler.install_native_meta_effects

```
void Compiler.install_native_meta_effects(Compiler c, Map globs)
```

Records the native advertisements retained by included interfaces. Each binds on first use, so a unit with no compile-time code pays nothing.

Source: `src/meta-native/bindings.x:42`

Compiler.install_native_meta_function

```
void Compiler.install_native_meta_function( Compiler c, List declaration, Token marker)
```

Installs a prototype-only `meta` function from the compiler's trusted native target registry. The declaration keeps its ordinary runtime form.

Source: `src/meta-native/bindings.x:77`

Compiler.install_project_meta_effect

```
void Compiler.install_project_meta_effect(Compiler c, List row)
```

Installs an included meta function's evaluator advertisement.

Source: `src/meta-native/functions.x:215`

Compiler.linked_meta_provider_source

```
List Compiler.linked_meta_provider_source(String provider)
```

Returns a linked provider's source and dependency hashes.

Source: `src/meta-native/linked.x:99`

Compiler.links_extension

```
int Compiler.links_extension(String name)
```

Reports whether package `name`'s compile-time part is linked into the compiler, so its import loads no module.

Source: `src/meta-native/modules.x:246`

Compiler.links_meta_function

```
int Compiler.links_meta_function(String name)
```

Reports whether the compiler links the `meta` function `name`.

Source: `src/meta-native/linked.x:96`

Compiler.load_native_module

```
String Compiler.load_native_module(String path)
```

Loads the native module at `path` once per process and returns its absolute path. Loading runs the module's code inside the compiler, so it happens only on request. A module from another compiler, a file that is not a module, or an unsupported platform prints a diagnostic and exits.

Source: `src/meta-native/modules.x:101`

Compiler.matches_macro

```
int Compiler.matches_macro(Compiler c, MacroMatcher &matcher, List code)
```

Tests a registered macro with the candidate's global and source bindings. A pattern derived without them reads none.

Source: `src/meta-native/functions.x:142`

Compiler.native_meta_accepts

```
int Compiler.native_meta_accepts(Compiler c, Var function, List signature)
```

Answers whether the native target `function` matches the declared `signature`. A declared `Func` parameter matches a target's `Var` parameter, which takes the compile-time callable and adapts it. Aliases of one native type match each other.

Source: `src/meta-native/bindings.x:178`

Compiler.native_meta_module

```
String Compiler.native_meta_module(Compiler c, String name, Type &type)
```

Returns the selected file-backed native module that supplies `name`, with its native `type`, or NULL for a linked or absent target.

Source: `src/meta-native/bindings.x:214`

Compiler.native_meta_targets

```
List Compiler.native_meta_targets(List paths)
```

Returns the declared native targets advertised by `meta` interface rows, in the row form `lib/lisp-targets.x` generates its target inventory from, or only those declared in the files `paths` names when it is not empty. Sorting makes that inventory independent of Map order.

Source: `src/meta-native/bindings.x:305`

Compiler.native_module_loaded

```
int Compiler.native_module_loaded(String path)
```

Reports whether the native module at absolute `path` is loaded.

Source: `src/meta-native/modules.x:203`

Compiler.native_module_targets

```
Map Compiler.native_module_targets(String path)
```

Returns the name-to-`Func` Map of the loaded native module at absolute `path`.

Source: `src/meta-native/modules.x:208`

Compiler.preload_native_module

```
void Compiler.preload_native_module(String path)
```

Loads the native module at absolute `path` when this compiler built it and the platform loads modules. A process that loads a module before it forks translation workers lets them inherit it; anything else is left for the import to report.

Source: `src/meta-native/modules.x:119`

Compiler.project_meta_uses_linked

```
int Compiler.project_meta_uses_linked( Compiler c, String name, String provider)
```

Answers whether a provider's function can use its linked body. Copied definitions retain precise callees; runtime bodies retain their provider's source dependencies, including private helpers.

Source: `src/meta-native/linked.x:34`

Compiler.record_native_meta_effect

```
void Compiler.record_native_meta_effect( Compiler c, List declaration, Token marker)
```

The shallow interface retains the advertisement separately from the C declaration. That lets a client install the trusted evaluator binding without repeating the marker in every translation unit.

Source: `src/meta-native/bindings.x:15`

Compiler.record_project_meta_effect

`void Compiler.record_project_meta_effect( Compiler c, List declaration, Token marker)`

Records the provider of a public bodied meta function in its interface. The provider owns both its native body and its compile-time state.

Source: `src/meta-native/functions.x:201`

Compiler.run_meta_call

`Var Compiler.run_meta_call( Compiler c, List expression, Token site, int slot)`

Runs the explicit meta call `expression` at `site`. The project meta build's own parse leaves a project function's call for the translation: an expression takes a placeholder, and a template `slot`, which has none, is reported.

Source: `src/meta-native/calls.x:71`

Compiler.select_native_modules

`void Compiler.select_native_modules(List paths)`

Selects the loaded native modules, by absolute path, that bodyless `meta` prototypes bind in the current request. The first module in `paths` that defines a name supplies it.

Source: `src/meta-native/modules.x:53`

Compiler.select_package_module

`void Compiler.select_package_module( Compiler c, String name, String root, Token token)`

Selects package `name`'s native module, when it has one, after the modules already selected, and records it as a prerequisite of the unit. The module is `<root>/builds/<name>.module`; a worker loads it itself when the process has not. A module from another compiler, or one on a platform that loads none, is reported at the import `token`.

Source: `src/meta-native/modules.x:73`

Compiler.supplies_native_meta

`int Compiler.supplies_native_meta(String name)`

Reports whether the compiler itself supplies the native function `name`. Such a function exists only inside a compiler, so a `meta` function that reaches it has no runtime form.

Source: `src/meta-native/modules.x:214`

Design notes

A bodied `meta` function runs at compile time through a stub that calls its compiled copy, and a bodyless `meta` prototype binds a native function that the compiler links, a selected native module defines, or a linked package supplies. This unit owns those bindings, the region summaries a native signature implies, and the native modules, which a request loads once per process and never unloads.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/meta-sdk/meta-sdk.x`

The compiler's answers to `lib/meta.x` operations.

Functions

Function	Summary
x2c_binding_for	Answers x2c.binding.for, declared in lib/meta.x.
x2c_binding_fresh	Answers x2c.binding.fresh, declared in lib/meta.x.
x2c_diagnostic_error_at	Answers x2c.diagnostic.error.at, declared in lib/meta.x.
x2c_diagnostic_fail	Answers x2c.diagnostic.fail, declared in lib/meta.x.
x2c_diagnostic_fail_at	Answers x2c.diagnostic.fail.at, declared in lib/meta.x.
x2c_diagnostic_warn	A warning reports where it is raised and returns, so a macro can keep expanding.
x2c_embed_text	Answers x2c.embed.text, declared in lib/meta.x.
x2c_enclosing	Answers x2c.enclosing, declared in lib/meta.x.
x2c_ident	Answers x2c.ident, declared in lib/meta.x.
x2c_invocation_column	Answers x2c.invocation.column, declared in lib/meta.x.
x2c_invocation_file	Answers x2c.invocation.file, declared in lib/meta.x.
x2c_invocation_line	Answers x2c.invocation.line, declared in lib/meta.x.
x2c_meta_definition_hashes	Answers x2c.meta.definition.hashes, declared in lib/meta.x.
x2c_place	Answers x2c.place, declared in lib/meta.x.
Code.arguments	Answers Code.arguments, declared in lib/meta.x.
Code.binding_spelling	Answers Code.binding_spelling, declared in lib/meta.x.
Code.body	Answers Code.body, declared in lib/meta.x.
Code.bound	Answers Code.bound, declared in lib/meta.x, with the carrier Compiler.take_code_value consumes.
Code.call_in_order	Answers Code.call_in_order, declared in lib/meta.x.
Code.convert	Answers Code.convert, declared in lib/meta.x.
Code.declared_bindings	Returns an expression reading each binding declaration declares.
Code.exits	Answers Code.exits, declared in lib/meta.x.
Code.format	Answers Code.format, declared in lib/meta.x.
Code.func_parameters	Answers Code.func_parameters, declared in lib/meta.x.
Code.is_static_pattern	Answers Code.is_static_pattern, declared in lib/meta.x.
Code.lowered	Answers Code.lowered, declared in lib/meta.x, with the carrier Compiler.take_code_value consumes.
Code.name	Answers Code.name, declared in lib/meta.x.
Code.native_signature	Returns the Func signature of the function syntax syntax.
Code.parameter	Answers Code.parameter, declared in lib/meta.x.
Code.pattern_value	Answers Code.pattern_value, declared in lib/meta.x.
Code.promoted	Answers Code.promoted, declared in lib/meta.x.
Code.register_after_initialization	Adds post-initialization block items while the declaration owner retains the original declarators, their type identity, and initialization order.
Code.register_rewrite	Answers Code.register_rewrite, declared in lib/meta.x: the pattern's form selects the family and its keys.
Code.source_text	Answers Code.source_text, declared in lib/meta.x.
Code.type	Answers the compiler-backed captured-code type query.
Code.value	Reads a constant through the compiler's existing constant decoder.
Compiler.bind_sdk_primitives	Binds the internal primitives the compile-time SDK library wraps into lisp, under their _x2c. names.
Compiler.expanding	Returns the compiler running the current compile-time call.

Function	Summary
MetaContext.current	Returns the context of the running compile-time call, which an evaluation changes with $let for its own extent.
MetaContext.fresh_binding	Returns a fresh binding whose spelling starts with stem.
MetaContext.function_reference	Returns the typed expression naming the function name, or an empty List when no function of that name is visible.
MetaContext.invocation_location	Returns the location of the active macro invocation.
MetaContext.literal_list	Returns values, such as a Lisp-built signature, as a cached literal of the expanding unit.
MetaContext.reject	Reports message and notes at the active invocation and never returns, so the rejected operation's caller cannot continue with a missing answer.
MetaContext.source_description	Returns what a meta parameter declared Source receives for the captured syntax value: ((text T) (file F) (syntax value)), where T is the text the developer wrote and F the file it is in.
MetaContext.type_description	Returns what a meta parameter declared TypeInfo receives for the captured syntax value: ((name N) (kind K) (type T) (fields F) (methods M)).
Type.aggregate	Answers Type.aggregate, declared in lib/meta.x.
Type.element	Answers Type.element, declared in lib/meta.x.
Type.fields	Answers Type.fields, declared in lib/meta.x.
Type.getter	Answers Type.getter, declared in lib/meta.x, with the getter that bracket admission selected.
Type.is_named	Tests type ancestry through the active symbol table.
Type.is_text	Tests the text forms accepted by string arithmetic.
Type.is_value	Answers Type.is_value, declared in lib/meta.x.
Type.layout	Answers Type.layout, declared in lib/meta.x.
Type.marked_fields	Answers Type.marked_fields, declared in lib/meta.x.
Type.members	Answers Type.members, declared in lib/meta.x.
Type.numeric	Resolves numeric typedefs through the active symbol table.
Type.parameters	Answers Type.parameters, declared in lib/meta.x.
Type.protocol_member	Selects a callable protocol member in the current compilation context.
Type.resolve	Answers Type.resolve, declared in lib/meta.x.
Type.resolve_member	Answers Type.resolve_member, declared in lib/meta.x.
Type.return_type	Answers Type.return_type, declared in lib/meta.x.
Type.reverse_name	Answers Type.reverse_name, declared in lib/meta.x.
Type.tag_name	Answers Type.tag_name, declared in lib/meta.x.
Type.update_helper	Answers Type.update_helper, declared in lib/meta.x.

Functions

x2c_binding_for

```
List x2c_binding_for(Var key, String role)
```

Answers x2c.binding.for, declared in lib/meta.x.

Source: src/meta-sdk/syntax.x:245

x2c_binding_fresh

```
List x2c_binding_fresh(String role)
```

Answers x2c.binding.fresh, declared in lib/meta.x.

Source: src/meta-sdk/syntax.x:237

x2c_diagnostic_error_at

```
void x2c_diagnostic_error_at( Var node, Symbol category, String message, List notes)
```

Answers `x2c.diagnostic.error.at`, declared in `lib/meta.x`.

Source: `src/meta-sdk/context.x:68`

x2c_diagnostic_fail

```
void x2c_diagnostic_fail(String message, List notes)
```

Answers `x2c.diagnostic.fail`, declared in `lib/meta.x`.

Source: `src/meta-sdk/context.x:54`

x2c_diagnostic_fail_at

```
void x2c_diagnostic_fail_at( Var node, Symbol category, String message, List notes)
```

Answers `x2c.diagnostic.fail.at`, declared in `lib/meta.x`.

Source: `src/meta-sdk/context.x:60`

x2c_diagnostic_warn

```
void x2c_diagnostic_warn(String message, List notes)
```

A warning reports where it is raised and returns, so a macro can keep expanding. Failure stays separate because it never returns.

Source: `src/meta-sdk/context.x:77`

x2c_embed_text

```
String x2c_embed_text(Var requested)
```

Answers `x2c.embed.text`, declared in `lib/meta.x`.

Source: `src/meta-sdk/sources.x:53`

x2c_enclosing

```
Code x2c_enclosing(Symbol what)
```

Answers `x2c.enclosing`, declared in `lib/meta.x`.

Source: `src/meta-sdk/syntax.x:318`

x2c_ident

```
List x2c_ident(String spelling)
```

Answers `x2c.ident`, declared in `lib/meta.x`.

Source: `src/meta-sdk/syntax.x:229`

x2c_invocation_column

```
int x2c_invocation_column(void)
```

Answers `x2c.invocation.column`, declared in `lib/meta.x`.

Source: `src/meta-sdk/context.x:50`

x2c_invocation_file

```
String x2c_invocation_file(void)
```

Answers `x2c.invocation.file`, declared in `lib/meta.x`.

Source: `src/meta-sdk/context.x:42`

x2c_invocation_line

```
int x2c_invocation_line(void)
```

Answers `x2c.invocation.line`, declared in `lib/meta.x`.

Source: `src/meta-sdk/context.x:46`

x2c_meta_definition_hashes

```
Map x2c_meta_definition_hashes(void)
```

Answers `x2c.meta.definition.hashes`, declared in `lib/meta.x`.

Source: `src/meta-sdk/primitives.x:22`

x2c_place

```
void x2c_place(List where, Code code)
```

Answers `x2c.place`, declared in `lib/meta.x`.

Source: `src/meta-sdk/syntax.x:326`

`Code`

Code.arguments

```
List Code.arguments(Code parameters)
```

Answers `Code.arguments`, declared in `lib/meta.x`.

Source: `src/meta-sdk/syntax.x:278`

Code.binding_spelling

```
String Code.binding_spelling(Var syntax)
```

Answers `Code.binding_spelling`, declared in `lib/meta.x`.

Source: `src/meta-sdk/syntax.x:191`

Code.body

```
List Code.body(Code function)
```

Answers `Code.body`, declared in `lib/meta.x`.

Source: `src/meta-sdk/syntax.x:272`

Code.bound

```
Code Code.bound(Code c, List effects)
```

Answers `Code.bound`, declared in `lib/meta.x`, with the carrier `Compiler.take_code_value` consumes.

Source: `src/meta-sdk/syntax.x:103`

Code.call_in_order

```
Code Code.call_in_order(Code member, List arguments, Type type)
```

Answers `Code.call_in_order`, declared in `lib/meta.x`. The call binds its temporaries here, so the result is a carrier that binds no further.

Source: `src/meta-sdk/syntax.x:91`

Code.convert

```
Code Code.convert(Code value, Type target)
```

Answers `Code.convert`, declared in `lib/meta.x`.

Source: `src/meta-sdk/syntax.x:118`

Code.declared_bindings

```
List Code.declared_bindings(List declaration)
```

Returns an expression reading each binding `declaration` declares.

Source: `src/meta-sdk/syntax.x:151`

Code.exits

```
int Code.exits(Code statement)
```

Answers `Code.exits`, declared in `lib/meta.x`.

Source: `src/meta-sdk/syntax.x:112`

Code.format

```
String Code.format(Code value)
```

Answers `Code.format`, declared in `lib/meta.x`.

Source: `src/meta-sdk/syntax.x:124`

Code.func_parameters

```
List Code.func_parameters(Code parameters, Code fn, Code argv)
```

Answers `Code.func_parameters`, declared in `lib/meta.x`.

Source: `src/meta-sdk/syntax.x:106`

Code.is_static_pattern

```
int Code.is_static_pattern(Code pattern)
```

Answers `Code.is_static_pattern`, declared in `lib/meta.x`.

Source: `src/meta-sdk/syntax.x:130`

Code.lowered

```
Code Code.lowered(Code c, List effects)
```

Answers `Code.lowered`, declared in `lib/meta.x`, with the carrier `Compiler.take_code_value` consumes.

Source: `src/meta-sdk/syntax.x:99`

Code.name

```
String Code.name(Code function)
```

Answers `Code.name`, declared in `lib/meta.x`.

Source: `src/meta-sdk/syntax.x:265`

Code.native_signature

```
List Code.native_signature(List syntax)
```

Returns the `Func` signature of the function syntax `syntax`. A native binding stores the same signature as an ordinary Func adapter.

Source: `src/meta-sdk/syntax.x:307`

Code.parameter

```
Code Code.parameter(Code function, String wanted)
```

Answers `Code.parameter`, declared in `lib/meta.x`.

Source: `src/meta-sdk/syntax.x:290`

Code.pattern_value

```
Var Code.pattern_value(Code pattern)
```

Answers `Code.pattern_value`, declared in `lib/meta.x`.

Source: src/meta-sdk/syntax.x:136

Code.promoted

```
Code Code.promoted(Code value)
```

Answers Code.promoted, declared in lib/meta.x.

Source: src/meta-sdk/syntax.x:142

Code.register_after_initialization

```
Code Code.register_after_initialization(Code c, Macro shape, List holes)
```

Adds post-initialization block items while the declaration owner retains the original declarators, their type identity, and initialization order.

Source: src/meta-sdk/rewrites.x:170

Code.register_rewrite

```
Code Code.register_rewrite(Code function, List pattern, List holes)
```

Answers Code.register_rewrite, declared in lib/meta.x: the pattern's form selects the family and its keys. A function rule is keyed by return type, which Compiler.normalize dispatches once per definition before its body lowers.

Source: src/meta-sdk/rewrites.x:106

Code.source_text

```
String Code.source_text(Var value)
```

Answers Code.source_text, declared in lib/meta.x.

Source: src/meta-sdk/sources.x:40

Code.type

```
Type Code.type(Code value)
```

Answers the compiler-backed captured-code type query.

Source: src/meta-sdk/syntax.x:49

Code.value

```
Var Code.value(Code code)
```

Reads a constant through the compiler's existing constant decoder.

Source: src/meta-sdk/syntax.x:75

Compiler

Compiler.bind_sdk_primitives

```
void Compiler.bind_sdk_primitives(Lisp lisp)
```

Binds the internal primitives the compile-time SDK library wraps into lisp, under their _x2c. names.

Source: src/meta-sdk/primitives.x:87

Compiler.expanding

```
Compiler Compiler.expanding(void)
```

Returns the compiler running the current compile-time call. A slot function compiled into the compiler reads its facts through it.

Source: `src/meta-sdk/context.x:29`

`MetaContext`

MetaContext.current

```
MetaContext *MetaContext.current(void)
```

Returns the context of the running compile-time call, which an evaluation changes with `$let` for its own extent.

Source: `src/meta-sdk/context.x:25`

MetaContext.fresh_binding

```
Var MetaContext.fresh_binding(String stem)
```

Returns a fresh binding whose spelling starts with `stem`.

Source: `src/meta-sdk/syntax.x:256`

MetaContext.function_reference

```
List MetaContext.function_reference(String name)
```

Returns the typed expression naming the function `name`, or an empty List when no function of that name is visible. This reads the symbol table only, so any compile-time Lisp evaluation may call it, including one outside a macro expansion.

Source: `src/meta-sdk/syntax.x:179`

MetaContext.invocation_location

```
List MetaContext.invocation_location(void)
```

Returns the location of the active macro invocation.

Source: `src/meta-sdk/context.x:35`

MetaContext.literal_list

```
List MetaContext.literal_list(List values)
```

Returns `values`, such as a Lisp-built signature, as a cached literal of the expanding unit.

Source: `src/meta-sdk/primitives.x:57`

MetaContext.reject

```
void MetaContext.reject(String message, List notes)
```

Reports `message` and `notes` at the active invocation and never returns, so the rejected operation's caller cannot continue with a missing answer. With no active invocation it is a bad state.

Source: `src/meta-sdk/context.x:95`

MetaContext.source_description

```
List MetaContext.source_description(Var value)
```

Returns what a `meta` parameter declared `Source` receives for the captured syntax `value`: `((text T) (file F) (syntax value))`, where `T` is the text the developer wrote and `F` the file it is in.

Source: `src/meta-sdk/sources.x:153`

MetaContext.type_description

`List MetaContext.type_description(Var value)`

Returns what a `meta` parameter declared `TypeInfo` receives for the captured syntax `value`: `((name N) (kind K) (type T) (fields F) (methods M))`. `T` is the canonical type of `value` and `N` its name, or "" when it has none. `K` is `struct`, `union`, `enum`, `pointer`, `scalar`, or `other`, `F` lists the `(name type)` rows of a struct or union's named fields, and `M` the names of its direct dotted methods.

Source: `src/meta-sdk/types.x:192`

`Type`

Type.aggregate

`Type Type.aggregate(Type type)`

Answers `Type.aggregate`, declared in `lib/meta.x`.

Source: `src/meta-sdk/types.x:40`

Type.element

`Type Type.element(Type t)`

Answers `Type.element`, declared in `lib/meta.x`.

Source: `src/meta-sdk/types.x:89`

Type.fields

`List Type.fields(Type t)`

Answers `Type.fields`, declared in `lib/meta.x`.

Source: `src/meta-sdk/types.x:170`

Type.getter

`Code Type.getter(Type type)`

Answers `Type.getter`, declared in `lib/meta.x`, with the getter that bracket admission selected.

Source: `src/meta-sdk/types.x:69`

Type.is_named

`int Type.is_named(Type type, String name)`

Tests type ancestry through the active symbol table.

Source: `src/meta-sdk/types.x:21`

Type.is_text

`int Type.is_text(Type type)`

Tests the text forms accepted by string arithmetic.

Source: `src/meta-sdk/types.x:33`

Type.is_value

`int Type.is_value(Type type)`

Answers `Type.is_value`, declared in `lib/meta.x`.

Source: `src/meta-sdk/types.x:121`

Type.layout

`List Type.layout(Type type)`

Answers `Type.layout`, declared in `lib/meta.x`.

Source: `src/meta-sdk/types.x:115`

Type.marked_fields

`List Type.marked_fields(Type aggregate, Symbol mark)`

Answers `Type.marked_fields`, declared in `lib/meta.x`.

Source: `src/meta-sdk/types.x:46`

Type.members

`List Type.members(Type type)`

Answers `Type.members`, declared in `lib/meta.x`.

Source: `src/meta-sdk/types.x:147`

Type.numeric

`Type Type.numeric(Type type)`

Resolves numeric typedefs through the active symbol table.

Source: `src/meta-sdk/types.x:27`

Type.parameters

`List Type.parameters(Type type)`

Answers `Type.parameters`, declared in `lib/meta.x`.

Source: `src/meta-sdk/types.x:92`

Type.protocol_member

`Code Type.protocol_member(Type type, String name)`

Selects a callable protocol member in the current compilation context. The member's own implementation receives no recursive selection.

Source: `src/meta-sdk/types.x:59`

Type.resolve

`Type Type.resolve(Type type)`

Answers `Type.resolve`, declared in `lib/meta.x`.

Source: `src/meta-sdk/types.x:109`

Type.resolve_member

`List Type.resolve_member(Type type, String name, int call)`

Answers `Type.resolve_member`, declared in `lib/meta.x`.

Source: `src/meta-sdk/types.x:52`

Type.return_type

`Type Type.return_type(Type type)`

Answers `Type.return_type`, declared in `lib/meta.x`.

Source: src/meta-sdk/types.x:99

Type.reverse_name

```
String Type.reverse_name(String base, String participant)
```

Answers Type.reverse_name, declared in lib/meta.x.

Source: src/meta-sdk/types.x:103

Type.tag_name

```
Symbol Type.tag_name(String name)
```

Answers Type.tag_name, declared in lib/meta.x.

Source: src/meta-sdk/types.x:131

Type.update_helper

```
String Type.update_helper(Type type)
```

Answers Type.update_helper, declared in lib/meta.x.

Source: src/meta-sdk/types.x:80

Public types

Type	Kind	Summary
MetaContext	struct	What the running compile-time call answers from.

MetaContext

```
typedef struct MetaContext { Compiler expander, evaluator; String definition_file; Map captures; int has_bindings; Token site; } MetaContext
```

What the running compile-time call answers from. expander is the compiler of the active macro expansion, definition_file the source of its definition, captures its complete captured syntax by identity, and has_bindings whether it bound captured values. evaluator is the compiler evaluating Lisp and site where that evaluation stands, which a nested import changes without an expansion. Each evaluation installs its part with $let, so a captured entry is valid only while its expansion is active.

Source: src/meta-sdk/meta-sdk.x:25

Design notes

Compile-time code asks the compiler about syntax, types, names, and source through the operations lib/meta.x declares, and the built-in macros ask through their own entries here. A native operation has no Compiler parameter, so each answer reads the running call's MetaContext, which the evaluations in macros.x install for their extent. An operation that cannot answer reports at the active invocation and never returns.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

src/parse/parse.x

X2c declarations, parsed from source or constructed.

Functions

Function	Summary
Ast.declared_return	The declared result type of the function declaration, which its return statements convert to, or NULL.

Function	Summary
`Compiler.bind_callable_body`	Binds a callable's outer block in its active parameter scope.
`Compiler.bind_catch_arm`	Binds one catch arm's `body` in a scope of its own, with declarations of the definite binders of `pattern`, read through `handle`, before it.
`Compiler.bind_match_arm`	Binds one match arm's `body` in a scope of its own, which defines the definite binders of `pattern` when the arm `binds`.
`Compiler.bind_parameter`	Binds one constructed parameter in the current scope, including nested function modifiers and reference parameter facts.
`Compiler.bind_source_statement`	Binds the source `if`, `while`, `do`, `for`, `match`, or `try` at the cursor through the same binders as constructed syntax.
`Compiler.bind_syntax`	Binds parser-shaped `syntax` at `context` into current compiler state.
`Compiler.bind_template_local`	Installs a definition-local template binding or typedef provisionally.
`Compiler.check_reference_placement`	Rejects transparent references in object and function-result types.
`Compiler.close_placements`	Gives the placements made since `first` the `result` of the expansion that made them.
`Compiler.defines_main`	Reports whether the unit's tokens define a function named `main` at file scope.
`Compiler.definition_doc`	Returns the body of the doc comment that documents the definition starting at `start`, or NULL.
`Compiler.enclosing`	Returns the innermost `what` around the parser, or around the function body the transform lowers, as captured code, or NULL when there is none: the declarator whose initializer is being parsed directly in the current block item, as its one-declarator declaration; the block item, or the one the transform lowers, as its origin anchor; the function, as a definition whose body is still the empty `(seq)`; or the unit, as a `String` expression of its path.
`Compiler.enter`	Enters `row` as the innermost syntax `Compiler.enclosing` answers.
`Compiler.enter_function`	Enters the function definition `declaration`, whose body the transform lowers next, so a rule lowering that body places code at its entry.
`Compiler.exit_function`	Leaves the function row at `depth` that `Compiler.enter_function` began.
`Compiler.finish_foreign_alias`	Constructs a foreign alias from one direct function declaration and target.
`Compiler.finish_initializers`	Lowers block declarations whose declarators placed code after themselves, or that an after-initialization rewrite follows, to declarations followed by that code in source order, preserving their installed bindings and the enclosing lifetime.
`Compiler.function_entry`	Returns the code placed at the entry of the function row at `depth` since it was entered or last asked, in placement order, or NULL.
`Compiler.leave_item`	Leaves the block item `enter` began and returns the code placed after it, bound in its block, or NULL.
`Compiler.meta_form_is_declaration`	Reports whether the cursor begins a contextual top-level `meta` declaration: a function or an initialized file-static value.
`Compiler.package_alias_spelling`	Returns the folded package-member spelling at the current token, or NULL.
`Compiler.parse_basic_identifier`	Consumes one `ident` token and returns its spelling as a one-item `List`.
`Compiler.parse_complex_identifier`	Parses the current identifier or dotted owner/member as a one-item name.
`Compiler.parse_declaration_argument`	Parses one non-function, non-typedef `Decl` macro argument.
`Compiler.parse_declaration_row`	Parses one x2c declaration row and leaves its terminator current.
`Compiler.parse_declarator_argument`	Parses one declarator row for a macro argument without installing it.
`Compiler.parse_enumerator`	Parses one enumerator for enum type `context` and returns its AST.
`Compiler.parse_enumerators`	Parses comma-separated enumerators up to the current closing brace.
`Compiler.parse_field`	Parses one field for aggregate type `context` and returns its AST.
`Compiler.parse_fields`	Parses aggregate fields in source order up to the current closing brace.
`Compiler.parse_function_definition`	Parses one function declaration and its required compound body.

Function	Summary
Compiler.parse_function_target	Parses one function decorator target and returns its resulting AST.
Compiler.parse_import_declaration	Parses and registers one `import` declaration, including its semicolon.
Compiler.parse_named_type	Captures a name followed by an ordinary complete type and semicolon.
Compiler.parse_optional_identifier	Consumes and returns the current identifier, or NULL without consuming.
Compiler.parse_parameter	Parses one parameter and returns `(param TYPE DECLARATOR)` or `(...)`.
Compiler.parse_parameter_list	Parses a nonempty comma-separated parameter `List` in source order.
Compiler.parse_simple_declaration	Parses one declaration group and leaves its terminating token current.
Compiler.parse_static_assert	Parses a C assertion declaration; native C owns constant-expression checks.
Compiler.parse_submission	Parses one submission from the current token stream.
Compiler.parse_top_level	Parses one full top-level form through the shared classifier.
Compiler.parse_top_level_mode	Parses one top-level form and applies its source-ordered compiler effects.
Compiler.parse_type_name	Parses a type specifier with qualifiers and pointer/reference modifiers.
Compiler.parse_type_operand	Parses a declaration that sits inside an expression or a `for` header, such as a cast's type name, and returns it as one `(decl ...)` node.
Compiler.place_after	Records `code` to follow the innermost block item, or the initialized declarator whose initializer is being parsed directly in it.
Compiler.place_entry	Records `code` for the entry of the innermost function definition, after its parameters bind and before its first statement.
Compiler.protocol_form_starts	Reports whether the cursor begins a protocol declaration or adoption, including its `meta` and `static` markers.
Compiler.script_statement_executes	Reports whether the top-level item at the cursor is a script statement that runs, rather than a declaration: an expression, control flow, a `with` block, or a statement macro.
Compiler.script_statement_starts	Reports whether the top-level item at the cursor is one of a script unit's statements, which become `main`'s body.
Compiler.skip_linkage_brace	Consumes one brace of a C linkage specification at file scope and reports whether it did.
Compiler.source_clause	The source clause `kind`, which the binder parses when it reaches it.
Compiler.take_meta_marker	Consumes the `meta` marker at the cursor and the contextual `native` marker that may follow it, and returns the `meta` token.
Compiler.test_declaration	Tests whether the current token can begin a declaration without consuming.
Compiler.test_static_assert	Reports whether the current identifier starts a C static assertion.

Ast

Ast.declared_return

```
Type Ast.declared_return(List declaration)
```

The declared result type of the function `declaration`, which its `return` statements convert to, or NULL.

Source: `src/parse/functions.x:102`

Compiler

Compiler.bind_callable_body

```
List Compiler.bind_callable_body(Compiler c, List syntax, Type return_type)
```

Binds a callable's outer block in its active parameter scope. Ordinary constructed blocks open their own scope before using this operation.

Source: `src/parse/binding.x:591`

Compiler.bind_catch_arm

`List Compiler.bind_catch_arm( Compiler c, List pattern, Var body, Token start, List handle)`

Binds one catch arm's `body` in a scope of its own, with declarations of the definite binders of `pattern`, read through `handle`, before it. Binder diagnostics use `start`. A source arm's body parses here, and a template's declares its binders where its expansion binds it. A parsed arm's body already begins with them, so it binds unchanged.

Source: `src/parse/binding.x:446`

Compiler.bind_match_arm

`List Compiler.bind_match_arm( Compiler c, List pattern, Var body, Token start, int binds, List types)`

Binds one match arm's `body` in a scope of its own, which defines the definite binders of `pattern` when the arm `binds`. Binder diagnostics use `start`. A source arm's body parses here with the typed captures `types` that its pattern recorded.

Source: `src/parse/binding.x:547`

Compiler.bind_parameter

`List Compiler.bind_parameter(Compiler c, List base, List declarator)`

Binds one constructed parameter in the current scope, including nested function modifiers and reference parameter facts.

Source: `src/parse/declarators.x:505`

Compiler.bind_source_statement

`List Compiler.bind_source_statement(Compiler c)`

Binds the source `if`, `while`, `do`, `for`, `match`, or `try` at the cursor through the same binders as constructed syntax.

Source: `src/parse/binding.x:325`

Compiler.bind_syntax

`List Compiler.bind_syntax( Compiler c, Var syntax, Symbol context, Type return_type)`

Binds parser-shaped `syntax` at `context` into current compiler state. `context` is `<unit>`, `<block>`, `<field>`, `<enumerator>`, `<map-entry>`, `<statement>`, or `<expression>`. The input must evaluate to a nonempty AST `List` valid for the requested position. Bindings, types, scopes, and expressions are resolved in source order; `return_type` applies only while descendants are bound. This method mutates `Sym` and does not open a semantic transaction.

Source: `src/parse/binding.x:23`

Compiler.bind_template_local

`void Compiler.bind_template_local( Compiler c, List key, List type, List context)`

Installs a definition-local template binding or typedef provisionally. Later template types name a local typedef by its identity, so each expansion refers to that expansion's private typedef.

Source: `src/parse/declarators.x:292`

Compiler.check_reference_placement

`void Compiler.check_reference_placement(Compiler c, Type type)`

Rejects transparent references in object and function-result types. References inside a function's parameter list remain supported.

Source: src/parse/declarations.x:291

Compiler.close_placements

```
void Compiler.close_placements(Compiler c, int first, List result)
```

Gives the placements made since `first` the `result` of the expansion that made them.

Source: src/parse/placement.x:121

Compiler.defines_main

```
int Compiler.defines_main(Compiler c)
```

Reports whether the unit's tokens define a function named `main` at file scope. A script unit that does is an ordinary program: its declarations stay at file scope and it may not have top-level statements. Both parse passes read the same tokens, so they agree before either parses.

Source: src/parse/units.x:233

Compiler.definition_doc

```
String Compiler.definition_doc(Compiler c, Token start)
```

Returns the body of the doc comment that documents the definition starting at `start`, or NULL. A doc comment opens with two asterisks and ends on the line before `start` or on its line. Macro templates carry the text until their selected body binds at the invocation.

Source: src/parse/units.x:591

Compiler.enclosing

```
List Compiler.enclosing(Compiler c, Symbol what)
```

Returns the innermost `what` around the parser, or around the function body the transform lowers, as captured code, or NULL when there is none: the declarator whose initializer is being parsed directly in the current block item, as its one-declarator declaration; the block item, or the one the transform lowers, as its origin anchor; the function, as a definition whose body is still the empty `(seq)`; or the unit, as a `String` expression of its path.

Source: src/parse/placement.x:31

Compiler.enter

```
void Compiler.enter(Compiler c, Ancestor row)
```

Enters `row` as the innermost syntax `Compiler.enclosing` answers. Its owner leaves it with `Block.pop`, or with `Compiler.leave_item` for a block item.

Source: src/parse/placement.x:20

Compiler.enter_function

```
int Compiler.enter_function(Compiler c, List declaration, int callback)
```

Enters the function definition `declaration`, whose body the transform lowers next, so a rule lowering that body places code at its entry. Returns the row's depth for `Compiler.leave_function`.

Source: src/parse/placement.x:60

Compiler.exit_function

```
void Compiler.exit_function(Compiler c, int depth)
```

Leaves the function row at `depth` that `Compiler.enter_function` began.

Source: src/parse/placement.x:87

Compiler.finish_foreign_alias

`List Compiler.finish_foreign_alias( Compiler c, List declaration, List native_syntax)`

Constructs a foreign alias from one direct function declaration and target. `native_syntax` must resolve to a different direct identifier and, when typed, a function. Storage is limited to `static` or `inline`, when present, and variadic parameters are rejected.

Source: `src/parse/units.x:762`

Compiler.finish_initializers

`List Compiler.finish_initializers( Compiler c, List declaration, Token origin)`

Lowers block declarations whose declarators placed code after themselves, or that an after-initialization rewrite follows, to declarations followed by that code in source order, preserving their installed bindings and the enclosing lifetime.

Source: `src/parse/placement.x:169`

Compiler.function_entry

`List Compiler.function_entry(Compiler c, int depth)`

Returns the code placed at the entry of the function row at `depth` since it was entered or last asked, in placement order, or NULL. The code binds in a scope of its own with the function's result type; its caller lowers it while the row is still entered, so a placement that lowering makes lands at the entry too.

Source: `src/parse/placement.x:74`

Compiler.leave_item

`List Compiler.leave_item(Compiler c)`

Leaves the block item `enter` began and returns the code placed after it, bound in its block, or NULL. A declarator in the item that placed code is incomplete unless its block declaration took that code.

Source: `src/parse/placement.x:131`

Compiler.meta_form_is_declaration

`int Compiler.meta_form_is_declaration(Compiler c)`

Reports whether the cursor begins a contextual top-level `meta` declaration: a function or an initialized file-static value.

Source: `src/parse/units.x:308`

Compiler.package_alias_spelling

`String Compiler.package_alias_spelling(Compiler c)`

Returns the folded package-member spelling at the current token, or NULL. The token must be an unshadowed imported alias followed by `.` and an identifier. This lookahead does not consume tokens.

Source: `src/parse/declarators.x:606`

Compiler.parse_basic_identifier

`List Compiler.parse_basic_identifier(Compiler c)`

Consumes one `ident` token and returns its spelling as a one-item `List`.

Source: `src/parse/declarators.x:619`

Compiler.parse_complex_identifier

```
List Compiler.parse_complex_identifier(Compiler c)
```

Parses the current identifier or dotted owner/member as a one-item name. Package aliases and imported method spellings are folded through the current `Sym`, and all accepted tokens are consumed.

Source: `src/parse/declarators.x:544`

Compiler.parse_declaration_argument

```
List Compiler.parse_declaration_argument(Compiler c)
```

Parses one non-function, non-typedef `Decl` macro argument. Returns a single declaration without consuming the invocation delimiter; flat destructuring may omit an initializer in this position.

Source: `src/parse/declarations.x:110`

Compiler.parse_declaration_row

```
List Compiler.parse_declaration_row(Compiler c)
```

Parses one x2c declaration row and leaves its terminator current. A mixed-type comma row returns a `seq` of declarations in source order; a single declaration returns directly.

Source: `src/parse/declarations.x:15`

Compiler.parse_declarator_argument

```
List Compiler.parse_declarator_argument(Compiler c)
```

Parses one declarator row for a macro argument without installing it. Its enclosing declaration supplies the base type when expanded.

Source: `src/parse/declarators.x:210`

Compiler.parse_enumerator

```
List Compiler.parse_enumerator(Compiler c, Type context)
```

Parses one enumerator for enum type `context` and returns its AST. Outside macro-template parsing, its binding is published immediately so later initializers and enumerators can resolve it.

Source: `src/parse/types.x:489`

Compiler.parse_enumerators

```
List Compiler.parse_enumerators(Compiler c, List context)
```

Parses comma-separated enumerators up to the current closing brace. Returns their flat source-order `List` and leaves the brace unconsumed.

Source: `src/parse/types.x:473`

Compiler.parse_field

```
List Compiler.parse_field(Compiler c, List context)
```

Parses one field for aggregate type `context` and returns its AST. Ordinary fields publish their binding in the aggregate's field scope and consume their terminating semicolon; macro forms follow their own syntax.

Source: `src/parse/types.x:344`

Compiler.parse_fields

```
List Compiler.parse_fields(Compiler c, List context)
```

Parses aggregate fields in source order up to the current closing brace. Returns a flat field `List`, publishes delegate-field metadata, and leaves the closing brace unconsumed.

Source: `src/parse/types.x:329`

Compiler.parse_function_definition

```
List Compiler.parse_function_definition(Compiler c)
```

Parses one function declaration and its required compound body. The parameter bindings are active while the body is parsed, and the first token after the closing brace remains current.

Source: `src/parse/functions.x:12`

Compiler.parse_function_target

```
List Compiler.parse_function_target(Compiler c)
```

Parses one function decorator target and returns its resulting AST. A compatible unit macro at the current token is expanded first; otherwise an ordinary function definition is required.

Source: `src/parse/functions.x:28`

Compiler.parse_import_declaration

```
List Compiler.parse_import_declaration(Compiler c)
```

Parses and registers one `import` declaration, including its semicolon. The alias defaults to the package name; `with` members add source-ordered local spellings. These spellings affect source resolution only; the package name in the returned AST drives the generated header include.

Source: `src/parse/units.x:342`

Compiler.parse_named_type

```
List Compiler.parse_named_type(Compiler c)
```

Captures a name followed by an ordinary complete type and semicolon. The reserved typedef identity permits fields to refer to their enclosing named type before its pointer or value representation is complete.

Source: `src/parse/types.x:412`

Compiler.parse_optional_identifier

```
List Compiler.parse_optional_identifier(Compiler c)
```

Consumes and returns the current identifier, or NULL without consuming.

Source: `src/parse/declarators.x:626`

Compiler.parse_parameter

```
List Compiler.parse_parameter(Compiler c)
```

Parses one parameter and returns `(param TYPE DECLARATOR)` or `(...)`. Outside macro-template parsing, a named parameter is installed in the current `Sym` scope. The following delimiter remains current.

Source: `src/parse/declarators.x:493`

Compiler.parse_parameter_list

```
List Compiler.parse_parameter_list(Compiler c)
```

Parses a nonempty comma-separated parameter `List` in source order. The caller owns the parameter scope; the first non-comma delimiter remains current.

Source: `src/parse/declarators.x:476`

Compiler.parse_simple_declaration

`List Compiler.parse_simple_declaration(Compiler c)`

Parses one declaration group and leaves its terminating token current. Declared names are installed in `Sym` as their declarators are completed; the result is one `declare`, `typedef`, or initialized `dstrdecl` AST.

Source: `src/parse/declarations.x:35`

Compiler.parse_static_assert

`List Compiler.parse_static_assert(Compiler c)`

Parses a C assertion declaration; native C owns constant-expression checks.

Source: `src/parse/declarations.x:130`

Compiler.parse_submission

`List Compiler.parse_submission(Compiler c, int end_position)`

Parses one submission from the current token stream. `end_position` is the byte offset after supplied input, before any synthetic closing text. Missing required syntax raises `<incomplete>`; trailing items are rejected. Temporary parser scopes and captured parameters are restored on every exit. The caller owns the semantic transaction and commits after execution.

Source: `src/parse/units.x:208`

Compiler.parse_top_level

`List Compiler.parse_top_level(Compiler c)`

Parses one full top-level form through the shared classifier.

Source: `src/parse/units.x:200`

Compiler.parse_top_level_mode

`List Compiler.parse_top_level_mode(Compiler c, int skip_body)`

Parses one top-level form and applies its source-ordered compiler effects. Returns its AST, or NULL when a keyword definition, top-level Lisp form, linkage brace, or compile-time-only `meta` function only updates compiler state, with the first following token current. With `skip_body`, collection uses the same classifier and bound declarations but skips runtime bodies. This continuation is independent of the compiler's shallow-parse state.

Source: `src/parse/units.x:16`

Compiler.parse_type_name

`Type Compiler.parse_type_name(Compiler c)`

Parses a type specifier with qualifiers and pointer/reference modifiers. Returns its flat `Type` AST and leaves the first following token current.

Source: `src/parse/types.x:126`

Compiler.parse_type_operand

`List Compiler.parse_type_operand(Compiler c, Type *type)`

Parses a declaration that sits inside an expression or a `for` header, such as a cast's type name, and returns it as one `(decl ...)` node. When `type` is not null, it receives the declared `Type`.

Source: `src/parse/declarations.x:41`

Compiler.place_after

```
void Compiler.place_after(Compiler c, Var code)
```

Records `code` to follow the innermost block item, or the initialized declarator whose initializer is being parsed directly in it. The expansion that placed it gives it its result when it finishes.

Source: `src/parse/placement.x:108`

Compiler.place_entry

```
void Compiler.place_entry(Compiler c, Var code)
```

Records `code` for the entry of the innermost function definition, after its parameters bind and before its first statement. While parsing, a lambda body is inside the definition that holds it; while lowering, a lambda body is the body of its own lifted function, and a finalizer the kernel wrote as a callback has no entry a program can place at.

Source: `src/parse/placement.x:96`

Compiler.protocol_form_starts

```
int Compiler.protocol_form_starts(Compiler c)
```

Reports whether the cursor begins a protocol declaration or adoption, including its `meta` and `static` markers. This query does not consume tokens.

Source: `src/parse/units.x:300`

Compiler.script_statement_executes

```
int Compiler.script_statement_executes(Compiler c)
```

Reports whether the top-level item at the cursor is a script statement that runs, rather than a declaration: an expression, control flow, a `with` block, or a statement macro. This query does not consume tokens.

Source: `src/parse/units.x:291`

Compiler.script_statement_starts

```
int Compiler.script_statement_starts(Compiler c)
```

Reports whether the top-level item at the cursor is one of a script unit's statements, which become `main`'s body. Preprocessor lines, imports, protocols, compile-time definitions and Lisp, file-scope macro invocations, `typedef`, `static`, and `extern` declarations, linkage braces, type definitions, and function prototypes and definitions stay at file scope. This query does not consume tokens.

Source: `src/parse/units.x:254`

Compiler.skip_linkage_brace

```
int Compiler.skip_linkage_brace(Compiler c)
```

Consumes one brace of a C linkage specification at file scope and reports whether it did. `extern "C" {` opens a group; a `}` at file scope closes the innermost open group, because every other file-scope form consumes its own braces. The declarations between them stay at file scope. The `#ifdef __cplusplus` arm of the usual header guard is skipped, so only an unguarded group reaches this operation.

Source: `src/parse/units.x:109`

Compiler.source_clause

```
Var Compiler.source_clause(Symbol kind)
```

The source clause `kind`, which the binder parses when it reaches it.

Source: `src/parse/binding.x:307`

Compiler.take_meta_marker

`Token Compiler.take_meta_marker(Compiler c, int &?native)`

Consumes the `meta` marker at the cursor and the contextual `native` marker that may follow it, and returns the `meta` token. `native` binds the definition after it the way a bodyless prototype would. `*native`, when requested, reports whether that marker was present.

Source: `src/parse/units.x:321`

Compiler.test_declaration

`int Compiler.test_declaration(Compiler c)`

Tests whether the current token can begin a declaration without consuming. Typedefs, package aliases, and macro-hole kinds are resolved through the current compiler state.

Source: `src/parse/declarations.x:157`

Compiler.test_static_assert

`int Compiler.test_static_assert(Compiler c)`

Reports whether the current identifier starts a C static assertion.

Source: `src/parse/declarations.x:125`

Design notes

This module owns declarations and the top-level forms that hold them. One classifier reads each top-level form in collection and in the full parse. Each declarator installs its name in the current Sym before its initializer resolves, and syntax that macros and compile-time Lisp construct binds through the same declaration operations. Expressions, statements, and literals parse in their own modules.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/project/project.x`

X2c project manifests.

Functions

Function	Summary
`CliRequest.project_manifest`	Returns the explicit or nearest readable project manifest, or NULL.
`ProjectBuild.plan`	Parses a project manifest and returns its selected target's build plan.
`ProjectBuild.valid_name`	Returns whether a target, profile, or manifest field name contains only letters, digits, underscores, and hyphens, and is not empty.

`CliRequest`

CliRequest.project_manifest

`String CliRequest.project_manifest(CliRequest c)`

Returns the explicit or nearest readable project manifest, or NULL. Discovery uses the same request view as project parsing.

Source: `src/project/planning.x:41`

`ProjectBuild`

ProjectBuild.plan

`ProjectBuild ProjectBuild.plan(CliRequest request)`

Parses a project manifest and returns its selected target's build plan. `request` must be a build or run request with no explicit operands. The result contains each dependency once before its consumer and lowers manifest fields and command-line overrides to ordinary `CliRequest` values without executing build actions. Manifest discovery, parsing, validation, or target-selection failures print a diagnostic and exit with status 2.

Source: `src/project/planning.x:12`

ProjectBuild.valid_name

`int ProjectBuild.valid_name(String name)`

Returns whether a target, profile, or manifest field name contains only letters, digits, underscores, and hyphens, and is not empty.

Source: `src/project/manifest.x:322`

Public types

Type	Kind	Summary
`ProjectBuild`	struct	Links native build requests in dependency-first order.

ProjectBuild

`typedef struct ProjectBuild { CliRequest request; struct ProjectBuild *next; } *ProjectBuild`

Links native build requests in dependency-first order. `ProjectBuild.plan` returns the head. Each node and copied `CliRequest` struct is owned by the current `Scope` and needs no individual cleanup; its `String` and `List` fields retain their canonical pool lifetimes and may share values with the command request.

Source: `src/project/project.x:19`

Design notes

A project manifest defines source membership, deterministic glob expansion, target relationships, and named profiles. Resolved targets lower to the same CliRequest a direct build uses.

Provisional API: This reference documents current compiler behavior. Compatibility is not yet promised.

`src/protocol/protocol.x`

Protocols from declaration to conform forms.

Functions

Function	Summary
`Compiler.conformance_rows`	Returns the interface rows of the resolved conformances `adoptions` declare.
`Compiler.discard_helper`	Returns `(binding signature)` for a generated helper that calls the function `binding` of type `signature` and then discards the unnamed argument temporaries `which` selects (bit `n` for argument `n`).
`Compiler.dump_conformance`	Prints stable conformance rows for typedefs in `globs`.
`Compiler.forward_parameters`	Returns `(declarations arguments)` for a helper that forwards its parameters of `types`: each declaration names a fresh parameter `a0`, `a1`, ..., and each argument reads it.
`Compiler.generate_protocol_adapters`	Inserts native aliases and presents other resolved conformances as `conform` forms.
`Compiler.install_generated_protocol_symbols`	Publishes external native alias and ordinary adapter signatures.

Function	Summary
Compiler.native_protocol_alias	Reports whether a generated member names a native protocol host function.
Compiler.parse_protocol_declaration	Parses a protocol body or concrete adoption at the current token.
Compiler.protocol_discard_helper	The discard_helper for participant's protocol member.
Compiler.protocol_member_names	Returns unique member spellings from the participant's visible adopted conformances.
Compiler.protocol_members_for	Returns the resolved conformance for participant and base, if any.
Compiler.protocol_rejects_direct_member	Reports whether conformance supersedes an ambient direct member: 0 when it does not, 1 when the participant's own adoption does, and 2 when an ancestor's adoption does.
Compiler.protocol_update_helper	Returns a generated helper for a direct protocol-backed update.
Compiler.publish_protocol_node	Validates and installs one normalized protocol or adoption node.
Compiler.rebuild_protocols	Rebuilds the per-unit protocol and adoption registries from symbols.
Compiler.record_source_typedef	Remembers a source typedef's declaration and visibility, which native alias insertion reads for its participant.
Compiler.resolve_as_declared	Resolves the adoptions visible in symbols as a declaring unit's full parse resolves its own: before the parse, and again, after the generated members are published, where the parse publishes each one.
Compiler.resolve_protocol_member	Resolves a protocol member for participant.
Compiler.resolve_protocols	Resolves every visible adoption into the current conformance registry.
Compiler.reverse_converter_spelling	Returns the conventional reverse converter spelling.
Compiler.wrapper_function	Returns the function result name(params) { body } bound in this unit.

Compiler

Compiler.conformance_rows

```
Map Compiler.conformance_rows(Compiler c, List adoptions)
```

Returns the interface rows of the resolved conformances adoptions declare.

Source: src/protocol/resolution.x:211

Compiler.discard_helper

```
List Compiler.discard_helper( Compiler c, List binding, Type signature, String stem, int which)
```

Returns (binding signature) for a generated helper that calls the function binding of type signature and then discards the unnamed argument temporaries which selects (bit n for argument n). A discarded argument is one the compiler produced for this call alone, so its discard member may release what it owns before the enclosing scope ends. Returns null when no selected argument type has a discard member, or an ordinary pointer or aggregate result may borrow an argument.

Source: src/protocol/call-helpers.x:131

Compiler.dump_conformance

```
void Compiler.dump_conformance(Compiler c, Map globs)
```

Prints stable conformance rows for typedefs in globs. Rows are ordered by participant and protocol and identify whether each adoption is owned by this unit, so prelude and live symbol modes can be compared.

Source: src/protocol/lookup.x:163

Compiler.forward_parameters

`List Compiler.forward_parameters(Compiler c, List types)`

Returns `(declarations arguments)` for a helper that forwards its parameters of `types`: each declaration names a fresh parameter `a0`, `a1`, ..., and each argument reads it.

Source: `src/protocol/call-helpers.x:29`

Compiler.generate_protocol_adapters

`List Compiler.generate_protocol_adapters(Compiler c, List ast)`

Inserts native aliases and presents other resolved conformances as `conform` forms. A native alias is inserted at the participant's inferred public or private boundary. Each other conformance whose forward converter the unit defines goes to the rule on the `conform` head, which `src/components/protocol-adapters.x` registers to write its adapters into the unit's pending support. The external conformances the unit declares are first recorded in its interface. Returns `ast` with native insertions applied.

Source: `src/protocol/adapters.x:111`

Compiler.install_generated_protocol_symbols

`void Compiler.install_generated_protocol_symbols(Compiler c)`

Publishes external native alias and ordinary adapter signatures. Protocols must already be resolved in the active symbol table.

Source: `src/protocol/adapters.x:150`

Compiler.native_protocol_alias

`int Compiler.native_protocol_alias(Compiler c, String name)`

Reports whether a generated member names a native protocol host function. Its defining header supplies the alias instead of a C prototype.

Source: `src/protocol/adapters.x:191`

Compiler.parse_protocol_declaration

`List Compiler.parse_protocol_declaration(Compiler c)`

Parses a protocol body or concrete adoption at the current token. The method consumes through the closing brace or semicolon. Full parsing publishes the normalized row immediately. Macro-hole parsing returns syntax for later binding; shallow parsing publishes only when protocol collection is enabled and otherwise returns the uninstalled node. A leading `meta` makes an adoption's witnesses available to compile-time code.

Source: `src/protocol/declarations.x:122`

Compiler.protocol_discard_helper

`List Compiler.protocol_discard_helper( Compiler c, Type participant, String member, int which)`

The `discard_helper` for `participant`'s protocol `member`.

Source: `src/protocol/call-helpers.x:157`

Compiler.protocol_member_names

`List Compiler.protocol_member_names(Compiler c, Type participant)`

Returns unique member spellings from the participant's visible adopted conformances. Resolution remains responsible for selecting a binding.

Source: `src/protocol/lookup.x:105`

Compiler.protocol_members_for

```
List Compiler.protocol_members_for(Compiler c, Type participant, Type base)
```

Returns the resolved conformance for `participant` and `base`, if any. Lookup canonicalizes the participant and may use the nearest adopted typedef ancestor. Native conformances install their generated bindings before the cached conformance row is returned.

Source: `src/protocol/lookup.x:21`

Compiler.protocol_rejects_direct_member

```
int Compiler.protocol_rejects_direct_member( Compiler c, Type participant, String member)
```

Reports whether conformance supersedes an ambient direct member: 0 when it does not, 1 when the participant's own adoption does, and 2 when an ancestor's adoption does. The answer is cached for the canonical participant and uses the first visible adopted ancestor that declares the member.

Source: `src/protocol/lookup.x:130`

Compiler.protocol_update_helper

```
String Compiler.protocol_update_helper( Compiler c, Type participant, String member, int postfix)
```

Returns a generated helper for a direct protocol-backed update. The resolved member must have exactly `(Participant, RHS) -> Participant`. A matching helper is emitted once into the unit's pending support; `postfix` selects whether it returns the old or stored value. Returns null when the member cannot implement this update shape.

Source: `src/protocol/call-helpers.x:62`

Compiler.publish_protocol_node

```
List Compiler.publish_protocol_node( Compiler c, List node, Token participant_token, Token representation_token)
```

Validates and installs one normalized protocol or adoption node. The node must carry a protocol record or supported adoption shape with its storage and source location. Installation invalidates cached protocol decisions and returns the canonical published node. Generated contexts may also retain that node in `Sym` for replay. A `(meta-protocol ADOPTION)` node publishes its adoption and marks it for compile-time code.

Source: `src/protocol/registry.x:249`

Compiler.rebuild_protocols

```
void Compiler.rebuild_protocols(Compiler c, Map symbols)
```

Rebuilds the per-unit protocol and adoption registries from `symbols`. Existing rows, helper decisions, and lookup caches are discarded; a null map leaves those registries empty. The conformance rows included interfaces recorded are kept for reuse. Conformance reset and resolution belong to `resolve_protocols`.

Source: `src/protocol/registry.x:505`

Compiler.record_source_typedef

```
void Compiler.record_source_typedef( Compiler c, String name, List declaration, int private)
```

Remembers a source typedef's declaration and visibility, which native alias insertion reads for its participant.

Source: `src/protocol/adapters.x:371`

Compiler.resolve_as_declared

```
void Compiler.resolve_as_declared(Compiler c, Map symbols)
```

Resolves the adoptions visible in `symbols` as a declaring unit's full parse resolves its own: before the parse, and again, after the generated members are published, where the parse publishes each one. Their declaring units report their failures, so nothing is reported.

Source: `src/protocol/resolution.x:227`

Compiler.resolve_protocol_member

```
List Compiler.resolve_protocol_member( Compiler c, Type participant, String member_name)
```

Resolves a protocol member for `participant`. Returns a `(binding signature)` pair for the selected implementation or null when no eligible resolved member exists; positive and negative results are cached. Inside the selected implementation itself the result is null, so the member's own body keeps the native operation.

Source: `src/protocol/lookup.x:216`

Compiler.resolve_protocols

```
void Compiler.resolve_protocols(Compiler c)
```

Resolves every visible adoption into the current conformance registry. Resolution starts from an empty registry; diagnostics are located only for adoptions owned by the current translation unit. An adoption another unit declared and recorded keeps that conformance, which is installed when it is first used.

Source: `src/protocol/resolution.x:145`

Compiler.reverse_converter_spelling

```
String Compiler.reverse_converter_spelling( Compiler c, String base_name, String infix, String participant)
```

Returns the conventional reverse converter spelling. A `Base.participant` reverse converter is declared under the participant's source spelling, and package mode rewrites that joined name as a whole, so the package prefix sits at the front of the derived binding instead of inside it. The split is keyed on a known package because a foreign header may spell __ in a type name.

Source: `src/protocol/registry.x:225`

Compiler.wrapper_function

```
List Compiler.wrapper_function( Compiler c, Type result, List binding, List params, List body)
```

Returns the function `result name(params) { body }` bound in this unit. `result` is the storage class and result type, so static, inline and external helpers share it; `params` are the parameter declarations, and `body` its lowered statements.

Source: `src/protocol/call-helpers.x:19`

Design notes

This module owns the protocol feature: it parses protocol and adoption declarations, keeps their rows in per-unit registries, resolves each adoption into a conformance, answers member and operator lookups, and presents each conformance as native aliases or a `conform` form, from which `src/components/protocol-adapters.x` writes the adapters. Registries are rebuilt per translation unit; a static row is visible only in the unit of its canonical source path. The unit that declares an external adoption resolves it and records the conformance in its interface, which an including unit reuses.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/regions/regions.x`

Values that can outlive the region that allocated them.

Functions

Function	Summary
Compiler.audit_regions	Computes one unit's region summaries and findings for an optional project audit.
Compiler.check_meta_regions	Rejects a meta function whose body breaks the rule Compiler.check_regions warns about.
Compiler.check_regions	Warns about values that can outlive the region that allocated them.
Compiler.has_region_row	Reports whether the runtime table proves the lifetime effects of the native function name.
Compiler.region_effect	The runtime table's effect row for the operation name, such as (open scope) or (exit), or void when the table has no row.
Compiler.region_result	The owner of the storage the runtime operation name returns: <scope> for the active Scope, <slot> for the Scope its first argument names, <pool> for the canonical-value pool, or 0 when nothing is known.
Compiler.region_wrapper	Reports whether the runtime operation name returns its argument's storage unchanged, as a Var box or its unboxing does.

Compiler

Compiler.audit_regions

```
Map Compiler.audit_regions( Compiler c, List ast, Map seed, Map effects, Array findings)
```

Computes one unit's region summaries and findings for an optional project audit. seed contains prior project-round summaries indexed by the emitted names visible in this unit. effects adds audit-only native contracts; neither input changes ordinary translation. Findings carry their function name and are returned without compiler diagnostics.

Source: src/regions/analysis.x:166

Compiler.check_meta_regions

```
void Compiler.check_meta_regions(Compiler c, List fn)
```

Rejects a meta function whose body breaks the rule Compiler.check_regions warns about. A compile-time call frees its locals when it returns, so an address of one that is returned or stored into a static, a parameter's object, or an unknown pointer would read freed memory. The first finding is reported as an error. fn must be the bound and typed definition. The walk reads the summaries of the meta functions installed before fn and records the summary of fn in meta_regions.

Source: src/regions/analysis.x:147

Compiler.check_regions

```
void Compiler.check_regions(Compiler c, List ast)
```

Warns about values that can outlive the region that allocated them. ast must be the bound and typed top-level unit, before transform lowering rewrites its defer and region forms. The call adds warnings to c and does not change ast.

Source: src/regions/analysis.x:126

Compiler.has_region_row

```
int Compiler.has_region_row(String name)
```

Reports whether the runtime table proves the lifetime effects of the native function name.

Source: src/regions/runtime.x:188

Compiler.region_effect

`Var Compiler.region_effect(String name)`

The runtime table's effect row for the operation `name`, such as `(open scope)` or `(exit)`, or void when the table has no row.

Source: `src/regions/runtime.x:192`

Compiler.region_result

`Symbol Compiler.region_result(String name)`

The owner of the storage the runtime operation `name` returns: `<scope>` for the active Scope, `<slot>` for the Scope its first argument names, `<pool>` for the canonical-value pool, or `0` when nothing is known. What the operation does to its arguments is a separate fact.

Source: `src/regions/runtime.x:172`

Compiler.region_wrapper

`int Compiler.region_wrapper(String name)`

Reports whether the runtime operation `name` returns its argument's storage unchanged, as a `Var` box or its unboxing does.

Source: `src/regions/runtime.x:184`

Design notes

A region is a `$scope()` block, a `Scope.retain` and `Scope.release` pair, a `$scope(&slot)` push, a `Pool.open` bracket, an `$auto` local, a Scope local that `Scope.destroy` ends, or the function's own storage, which an address taken with `&` borrows. The pass reads the typed forms the parser produced, before the transform driver rewrites them, so a region is still the call that opens it and the `defer` beside it that closes it. It warns when a value born in a region reaches storage that outlives the region and when a local is read after it was freed.

A function's summary is two facts: which owner supplies fresh returned storage, and where each parameter is sunk. The unit's functions reach a fixpoint over their summaries. A call into another unit has a summary only through the runtime table, so warnings do not depend on which units were translated before it. A `meta` function is walked when it is defined, against the summaries of the `meta` functions before it, and a finding there is an error, because a compile-time call frees its locals when it returns.

The warnings name departures from the lexical pattern. Raw C stores, pointer arithmetic, callbacks, and storage the runtime did not allocate stay outside them.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/symtab/symtab.x`

The compiler's semantic symbol table.

Functions

Function	Summary
`Compiler.aggregate_name`	Binds a local aggregate tag before its fields, preserving native spelling.
`Compiler.begin_semantic_transaction`	Begins a reversible transaction over the current semantic scope.
`Compiler.declared_spelling`	Returns `name` as `scoped_spelling` does for a declaration, recording a spelling that took `prefix` so the names the compiler derives from it are known to hold their prefix already.
`Compiler.drop_fact`	Deletes one semantic binding fact.
`Compiler.fact`	Returns the List one semantic binding fact holds, or NULL when it is absent.
`Compiler.file_prefixes`	Returns the static prefix of each file of the module unit.

Function	Summary
`Compiler.imported_providers`	Returns sorted imported packages that publish `name`.
`Compiler.imported_spelling`	Returns an unambiguous imported spelling for `name`, or `NULL`.
`Compiler.macro_definition_locals`	Returns the active macro definition's borrowed local map, or `NULL`.
`Compiler.macro_template_local`	Reports whether `binding` is a local of the active macro definition or of a template that encloses it; a nested definition's literal names share the enclosing template's names.
`Compiler.package_member_spelling`	Returns a visible `with` name's package-prefixed spelling, or `NULL`.
`Compiler.package_spelling`	Returns a name in the current package namespace, preserving prefixes.
`Compiler.private_prefix`	Returns the prefix that a file-scope declaration of `type` takes in the current module file, or NULL.
`Compiler.record_declaration_visibility`	Records the visibility of one parsed top-level declaration.
`Compiler.register_package_alias`	Registers one local package alias.
`Compiler.register_package_member`	Registers one package member under a bare local spelling.
`Compiler.scoped_spelling`	Returns `name` spelled in a declaration's scope: with `prefix`, a file prefix, or else in the current package.
`Compiler.semantic_binding_facts`	Returns the borrowed semantic-facts map indexed by binding.
`Compiler.set_fact`	Sets one semantic binding fact.
`Compiler.unprefixed`	Returns the source name of a spelling this unit gave a file prefix, or `name`.
`Sym.aggregate_of`	Resolves typedefs or one pointer layer to an aggregate tag, or `NULL`.
`Sym.at_file_scope`	Returns whether declarations currently bind at file scope.
`Sym.base_symbols`	Copies all base-scope symbols into a fresh map in source order.
`Sym.bind_identity`	Installs an existing binding with `ast`'s qualifier-preserving type.
`Sym.binding_is_local`	Reports whether `binding` belongs to a scope inside the base scopes.
`Sym.binding_is_local_before`	Reports whether `binding` belongs to a local scope below `scope_count`.
`Sym.close_local_scopes`	Sets aside the local scopes now open, so declarations bind at file scope until `Sym.reopen_scopes` restores the returned scopes.
`Sym.current_binding`	Returns `key`'s binding in the current scope, or `NULL`.
`Sym.current_symbols`	Returns the current scope's mutable symbol map, or `NULL`.
`Sym.declare`	Analyzes a complete declaration type and installs its stored declared type.
`Sym.declare_enumerator`	Associates an enumerator key with its owner in the active scope.
`Sym.declare_field_order`	Records declaration AST fields in source order after binding finishes.
`Sym.declare_resolved`	Installs an aggregate or named type whose parser resolved its spelling.
`Sym.declared_key`	Returns the key a declaration of `key` with type `ast` takes in this unit: its file-private or package spelling at file scope, else `key`.
`Sym.define`	Defines `key` and returns its stable binding in the active scope.
`Sym.define_global`	Defines `key` in the unit's writable base scope.
`Sym.define_macro`	Defines or replaces a macro in the active lexical scope.
`Sym.drop`	Deletes `key` from `map` so that an active transaction can restore it.
`Sym.end_log`	Stops one transaction's semantic write log.
`Sym.enumerator_owner`	Returns the current scope's enum owner for `key`, or zero.
`Sym.field_order`	Returns recorded fields in source order, or `NULL`.
`Sym.file_statics`	Returns the borrowed set of file-static declaration keys.
`Sym.get`	Returns `key`'s type, or `NULL`.
`Sym.get_exact`	Returns `key`'s type without package fallback, or `NULL`.
`Sym.global_symbols`	Returns the mutable global symbol map supplied to the latest reset.

Function	Summary
`Sym.has_local`	Reports whether a scope inside the base scopes declares `key`.
`Sym.has_local_macros`	Returns whether any lexical scope contains a local macro definition.
`Sym.introduce`	Allocates a fresh binding identity for a compiler-introduced spelling.
`Sym.introduced`	Returns how many introduced bindings no scope holds: those `Sym.introduce` made, as opposed to the ones each scope makes once for a name it resolves.
`Sym.is_array_type`	Reports whether `type` reaches the named `Array` value type.
`Sym.is_file_static`	Reports whether `binding` names a file-scope static of this unit.
`Sym.is_map_type`	Reports whether `type` reaches the named `Map` value type.
`Sym.is_named_value_type`	Reports whether `type` reaches a named value type before its definition.
`Sym.is_string_type`	Reports whether `type` reaches the named `String` value type.
`Sym.is_var_type`	Reports whether `type` reaches the named `Var` value type.
`Sym.local_type`	Resolves a block-local typedef to the type saved at its declaration.
`Sym.lookup`	Resolves an existing key and optionally stores its semantic type.
`Sym.lookup_field`	Returns an aggregate field's declared type, or `NULL`.
`Sym.lookup_macro`	Returns the innermost visible local macro named `name`, or `NULL`.
`Sym.mark_field`	Records that the field `name` of `aggregate` was declared with the field keyword `mark`, such as `delegate`, and that `aggregate` declares one.
`Sym.mark_static`	Marks a declaration key as file-static.
`Sym.marked_fields`	Returns the `(NAME TYPE)` rows of the fields of `aggregate` declared with `mark`, in declaration order.
`Sym.new`	Creates the empty symbol table that compiler `c` owns.
`Sym.next_typedef`	Resolves one typedef hop and counts against the shared cycle budget.
`Sym.normalize_declared_type`	Resolves typedef bases while retaining every declarator qualifier.
`Sym.pop_scope`	Pops the innermost scope, or returns an empty scope when none exists.
`Sym.push_new_scope`	Pushes a new empty lexical scope.
`Sym.push_scope`	Pushes a caller-supplied lexical scope while retaining its map objects.
`Sym.put`	Sets `map[key]` so that an active semantic transaction can restore it.
`Sym.reference`	Resolves `key` or creates a forward binding in the current scope.
`Sym.reference_global`	Resolves a global name or creates its forward binding in the base scope.
`Sym.reopen_scopes`	Reopens the local scopes `Sym.close_local_scopes` set aside.
`Sym.reset`	Resets symbol state to one base scope backed by `globals`.
`Sym.reset_overlay`	Resets symbol state to read `base` below the writable scope `overlay`.
`Sym.resolve_global`	Resolves a binding through base scopes and optionally stores its type.
`Sym.resolve_key`	Resolves a canonical typedef key to the end of its declared chain.
`Sym.resolve_numeric_type`	Resolves a numeric typedef without reducing semantic object types.
`Sym.scope_count`	Returns the number of semantic scopes, including base scopes.
`Sym.seed_var_tags`	Registers declared `T_var` converters in the active `Type` unit.
`Sym.set`	Sets a semantic type for `key` in the required active scope.
`Sym.sole_typedef`	Returns the name of the one file-scope typedef declared directly as `type`, or NULL when no typedef or several typedefs are.
`Sym.transacting`	Returns whether a semantic transaction is open.
`Sym.unit_symbols`	Returns a fresh map of the base symbols with the current scope's rows merged in.

Function	Summary
Sym.unshadow_visible	Emits every local object spelled `spelling` that the current scope sees, the enclosing ones too, under one fresh spelling, so a file-scope name a template keeps reaches the global there, and returns them.
Sym.var_tag_for_type	Returns a type's `Var` tag and optionally stores its resolved type.
Sym.visible_symbols	Returns the visible one-part source names and their semantic types.
SymTxn.commit	Publishes an active semantic transaction and makes rollback a no-op.
SymTxn.commit_transient	Commits an active transaction, retaining the original counters map.
SymTxn.local_macros_changed	Returns whether the transaction's active scope changed its macro map.
SymTxn.rollback	Restores every semantic value captured by an active transaction.

`Compiler`

Compiler.aggregate_name

```
Var Compiler.aggregate_name( Compiler c, Symbol kind, Var name, int definition)
```

Binds a local aggregate tag before its fields, preserving native spelling. A reference reuses the nearest visible tag; a definition or standalone forward declaration introduces the tag in the current lexical scope. A macro template names its tags as template locals.

Source: `src/symtab/declarations.x:276`

Compiler.begin_semantic_transaction

```
SymTxn Compiler.begin_semantic_transaction(Compiler c)
```

Begins a reversible transaction over the current semantic scope.

The transaction restores the current scope maps, file-static and binding facts, binding and generated-name counters, initializer names, and the parser's ancestors and placements. It does not snapshot parser position or other compiler state.

Source: `src/symtab/transactions.x:39`

Compiler.declared_spelling

```
String Compiler.declared_spelling(Compiler c, String prefix, String name)
```

Returns `name` as `scoped_spelling` does for a declaration, recording a spelling that took `prefix` so the names the compiler derives from it are known to hold their prefix already. A replayed collected declaration was spelled when it was collected, so its names are final and are recorded as they are.

Source: `src/symtab/namespaces.x:112`

Compiler.drop_fact

```
void Compiler.drop_fact(Compiler c, Var key)
```

Deletes one semantic binding fact.

Source: `src/symtab/bindings.x:189`

Compiler.fact

```
List Compiler.fact(Compiler c, Var key)
```

Returns the List one semantic binding fact holds, or NULL when it is absent.

Source: `src/symtab/bindings.x:179`

Compiler.file_prefixes

```
List Compiler.file_prefixes(Compiler c)
```

Returns the static prefix of each file of the module unit.

Source: `src/symtab/namespaces.x:135`

Compiler.imported_providers

```
List Compiler.imported_providers(Compiler c, String name)
```

Returns sorted imported packages that publish `name`.

Package-owned methods use `<package>__<Type>_<member>` internally while consumers retain the spelling from the package header. Sorting keeps ambiguity diagnostics deterministic.

Source: `src/symtab/namespaces.x:233`

Compiler.imported_spelling

```
String Compiler.imported_spelling(Compiler c, String name)
```

Returns an unambiguous imported spelling for `name`, or `NULL`.

Source: `src/symtab/namespaces.x:245`

Compiler.macro_definition_locals

```
Map Compiler.macro_definition_locals(Compiler c)
```

Returns the active macro definition's borrowed local map, or `NULL`.

Source: `src/symtab/scopes.x:189`

Compiler.macro_template_local

```
int Compiler.macro_template_local(Compiler c, List binding)
```

Reports whether `binding` is a local of the active macro definition or of a template that encloses it; a nested definition's literal names share the enclosing template's names.

Source: `src/symtab/scopes.x:197`

Compiler.package_member_spelling

```
String Compiler.package_member_spelling(Compiler c, String name)
```

Returns a visible `with` name's package-prefixed spelling, or `NULL`.

An ordinary declaration of the local spelling shadows the `with` name.

Source: `src/symtab/namespaces.x:218`

Compiler.package_spelling

```
String Compiler.package_spelling(Compiler c, String name)
```

Returns a name in the current package namespace, preserving prefixes.

Idempotence lets parsing and declaration rewrites share this operation.

Source: `src/symtab/namespaces.x:155`

Compiler.private_prefix

```
String Compiler.private_prefix(Compiler c, Type type)
```

Returns the prefix that a file-scope declaration of `type` takes in the current module file, or NULL. A static takes it, and so does each declaration of a `static typedef` or `static class`; a `module` declaration and every declaration outside a module unit do not.

Source: `src/symtab/namespaces.x:86`

Compiler.record_declaration_visibility

```
void Compiler.record_declaration_visibility(Compiler c, List declaration)
```

Records the visibility of one parsed top-level declaration. Lexical privacy and static storage mark bindings in `Sym`; typedef rows are retained for placing generated protocol declarations at the same boundary.

Source: `src/symtab/namespaces.x:9`

Compiler.register_package_alias

```
void Compiler.register_package_alias( Compiler c, String name, String alias, Token token)
```

Registers one local package alias.

Shallow collection and full parsing both see an import, so repeating the same package and alias is a no-op. Another binding of the local spelling is a parse error.

Source: `src/symtab/namespaces.x:167`

Compiler.register_package_member

```
void Compiler.register_package_member( Compiler c, String name, String member, String local, Token
member_token, Token local_token)
```

Registers one package member under a bare local spelling.

The package member must already exist in the symbol table. Repeating the same binding is a no-op; any conflicting local binding is an error.

Source: `src/symtab/namespaces.x:180`

Compiler.scoped_spelling

```
String Compiler.scoped_spelling(Compiler c, String prefix, String name)
```

Returns `name` spelled in a declaration's scope: with `prefix`, a file prefix, or else in the current package. A name the compiler derived from a spelling this unit already prefixed, such as `second__Node_new` or `Var_second__node` for a private class `second__Node`, stays unchanged; it is unique in the unit, so code generated outside the file still reaches it. Ordinary source spellings always take their file's prefix, even when they resemble another generated spelling.

Source: `src/symtab/namespaces.x:100`

Compiler.semantic_binding_facts

```
Map Compiler.semantic_binding_facts(Compiler c)
```

Returns the borrowed semantic-facts map indexed by binding.

Write it through `set_fact` and `drop_fact`, which a semantic transaction can restore.

Source: `src/symtab/bindings.x:175`

Compiler.set_fact

```
void Compiler.set_fact(Compiler c, Var key, Var value)
```

Sets one semantic binding fact.

Source: `src/symtab/bindings.x:185`

Compiler.unprefixed

```
String Compiler.unprefixed(Compiler c, String name)
```

Returns the source name of a spelling this unit gave a file prefix, or `name`. Var tags and descriptor names use that short name.

Source: `src/symtab/namespaces.x:146`

`Sym`

Sym.aggregate_of

```
Type Sym.aggregate_of(Sym s, Type type)
```

Resolves typedefs or one pointer layer to an aggregate tag, or `NULL`.

Source: `src/symtab/types.x:320`

Sym.at_file_scope

```
int Sym.at_file_scope(Sym s)
```

Returns whether declarations currently bind at file scope.

Source: `src/symtab/scopes.x:76`

Sym.base_symbols

```
Map Sym.base_symbols(Sym s)
```

Copies all base-scope symbols into a fresh map in source order.

Package collection uses this so it resolves against the prelude and the importing unit's already-visible includes without mutating either.

Source: `src/symtab/scopes.x:104`

Sym.bind_identity

```
List Sym.bind_identity(Sym s, List context, List binding, List ast)
```

Installs an existing binding with `ast`'s qualifier-preserving type.

Source: `src/symtab/declarations.x:255`

Sym.binding_is_local

```
int Sym.binding_is_local(Sym s, List binding)
```

Reports whether `binding` belongs to a scope inside the base scopes.

Source: `src/symtab/bindings.x:155`

Sym.binding_is_local_before

```
int Sym.binding_is_local_before(Sym s, List binding, int scope_count)
```

Reports whether `binding` belongs to a local scope below `scope_count`.

Counts beyond the current scope depth are clamped to that depth.

Source: `src/symtab/bindings.x:162`

Sym.close_local_scopes

```
Block Sym.close_local_scopes(Sym s)
```

Sets aside the local scopes now open, so declarations bind at file scope until `Sym.reopen_scopes` restores the returned scopes.

Source: `src/symtab/scopes.x:80`

Sym.current_binding

```
List Sym.current_binding(Sym s, List key)
```

Returns `key`'s binding in the current scope, or `NULL`.

Source: `src/symtab/bindings.x:141`

Sym.current_symbols

```
Map Sym.current_symbols(Sym s)
```

Returns the current scope's mutable symbol map, or `NULL`.

Source: `src/symtab/scopes.x:111`

Sym.declare

```
List Sym.declare(Sym s, List context, List key, List ast)
```

Analyzes a complete declaration type and installs its stored declared type.

Returns the declared binding. At file scope this also records static visibility; local declarations record automatic-storage and declared-type facts. Stored types discard storage and `inline` while retaining `const`, `restrict`, and `volatile`.

Source: `src/symtab/declarations.x:12`

Sym.declare_enumerator

```
void Sym.declare_enumerator(Sym s, List key, Symbol owner)
```

Associates an enumerator key with its owner in the active scope.

Source: `src/symtab/bindings.x:77`

Sym.declare_field_order

```
void Sym.declare_field_order(Sym s, Type type, List fields)
```

Records declaration AST fields in source order after binding finishes.

`Field` types already use member keys. Unnamed rows retain their type and an empty name so initializer traversal preserves anonymous subobjects.

Source: `src/symtab/types.x:276`

Sym.declare_resolved

```
List Sym.declare_resolved(Sym s, List context, List key, List ast)
```

Installs an aggregate or named type whose parser resolved its spelling.

Source: `src/symtab/declarations.x:16`

Sym.declared_key

```
List Sym.declared_key(Sym s, List context, List key, List ast)
```

Returns the key a declaration of `key` with type `ast` takes in this unit: its file-private or package spelling at file scope, else `key`. Retained declarations keep their already resolved spelling.

Source: `src/symtab/declarations.x:45`

Sym.define

```
List Sym.define(Sym s, List key, List type)
```

Defines `key` and returns its stable binding in the active scope.

Source: `src/symtab/bindings.x:18`

Sym.define_global

```
void Sym.define_global(Sym s, List key, List type)
```

Defines `key` in the unit's writable base scope.

The definition survives the expression scope that first resolved it. Its binding is issued at the first reference, as for a row an included file contributes, so the unit numbers its bindings the same whether it defined the row here or replayed it from an interface.

Source: `src/symtab/bindings.x:70`

Sym.define_macro

```
void Sym.define_macro(Sym s, Atom name, List definition)
```

Defines or replaces a macro in the active lexical scope.

Replacing a name already defined in this scope does not increase the local macro count.

Source: `src/symtab/scopes.x:167`

Sym.drop

```
void Sym.drop(Sym s, Map map, Var key)
```

Deletes `key` from `map` so that an active transaction can restore it.

Source: `src/symtab/transactions.x:12`

Sym.end_log

```
void Sym.end_log(Sym s)
```

Stops one transaction's semantic write log.

Source: `src/symtab/transactions.x:124`

Sym.enumerator_owner

```
Symbol Sym.enumerator_owner(Sym s, List key)
```

Returns the current scope's enum owner for `key`, or zero.

Source: `src/symtab/bindings.x:148`

Sym.field_order

```
List Sym.field_order(Sym s, Type type)
```

Returns recorded fields in source order, or `NULL`.

Source: `src/symtab/types.x:298`

Sym.file_statics

```
Map Sym.file_statics(Sym s)
```

Returns the borrowed set of file-static declaration keys.

Source: `src/symtab/scopes.x:125`

Sym.get

```
List Sym.get(Sym s, List key)
```

Returns `key`'s type, or `NULL`. A plain key tries the current module file's static spelling first and retries a miss in package space.

Source: src/symtab/bindings.x:196

Sym.get_exact

```
List Sym.get_exact(Sym s, List key)
```

Returns key's type without package fallback, or NULL.

Source: src/symtab/bindings.x:204

Sym.global_symbols

```
Map Sym.global_symbols(Sym s)
```

Returns the mutable global symbol map supplied to the latest reset.

Source: src/symtab/scopes.x:97

Sym.has_local

```
int Sym.has_local(Sym s, List key)
```

Reports whether a scope inside the base scopes declares key.

Source: src/symtab/bindings.x:207

Sym.has_local_macros

```
int Sym.has_local_macros(Sym s)
```

Returns whether any lexical scope contains a local macro definition.

Source: src/symtab/scopes.x:175

Sym.introduce

```
List Sym.introduce(Sym s, String spelling)
```

Allocates a fresh binding identity for a compiler-introduced spelling.

Source: src/symtab/bindings.x:138

Sym.introduced

```
int Sym.introduced(Sym s)
```

Returns how many introduced bindings no scope holds: those Sym.introduce made, as opposed to the ones each scope makes once for a name it resolves.

Source: src/symtab/bindings.x:130

Sym.is_array_type

```
int Sym.is_array_type(Sym s, Type type)
```

Reports whether type reaches the named Array value type.

Source: src/symtab/types.x:234

Sym.is_file_static

```
int Sym.is_file_static(Sym s, Var binding)
```

Reports whether binding names a file-scope static of this unit.

Source: src/symtab/declarations.x:101

Sym.is_map_type

```
int Sym.is_map_type(Sym s, Type type)
```

Reports whether `type` reaches the named `Map` value type.

Source: `src/symtab/types.x:238`

Sym.is_named_value_type

```
int Sym.is_named_value_type(Sym s, Type type, String name)
```

Reports whether `type` reaches a named value type before its definition.

Source: `src/symtab/types.x:241`

Sym.is_string_type

```
int Sym.is_string_type(Sym s, Type type)
```

Reports whether `type` reaches the named `String` value type.

Source: `src/symtab/types.x:230`

Sym.is_var_type

```
int Sym.is_var_type(Sym s, Type type)
```

Reports whether `type` reaches the named `Var` value type.

Source: `src/symtab/types.x:227`

Sym.local_type

```
Type Sym.local_type(Sym s, Type type)
```

Resolves a block-local typedef to the type saved at its declaration. File-scope names retain their semantic identity. Local alias definitions are resolved before they are installed, so one lookup crosses the whole local chain without consulting names shadowed since its declaration.

Source: `src/symtab/types.x:119`

Sym.lookup

```
List Sym.lookup(Sym s, List key, Type &?type)
```

Resolves an existing key and optionally stores its semantic type.

A symbol row without a binding receives a stable binding identity. A total miss returns `NULL` and stores `NULL` through `type` when provided.

Source: `src/symtab/bindings.x:254`

Sym.lookup_field

```
Type Sym.lookup_field(Sym s, Type type, List field)
```

Returns an aggregate field's declared type, or `NULL`. A member of an anonymous struct or union belongs to its enclosing aggregate in C, so unnamed rows are searched the way a designated initializer already reaches them.

Source: `src/symtab/types.x:256`

Sym.lookup_macro

```
List Sym.lookup_macro(Sym s, Atom name)
```

Returns the innermost visible local macro named `name`, or `NULL`.

Source: `src/symtab/scopes.x:178`

Sym.mark_field

```
void Sym.mark_field(Sym s, Type aggregate, Symbol mark, String name)
```

Records that the field `name` of `aggregate` was declared with the field keyword `mark`, such as `delegate`, and that `aggregate` declares one.

Source: `src/symtab/types.x:302`

Sym.mark_static

```
void Sym.mark_static(Sym s, List key)
```

Marks a declaration key as file-static.

Source: `src/symtab/scopes.x:128`

Sym.marked_fields

```
List Sym.marked_fields(Sym s, Type aggregate, Symbol mark)
```

Returns the `(NAME TYPE)` rows of the fields of `aggregate` declared with `mark`, in declaration order.

Source: `src/symtab/types.x:309`

Sym.new

```
Sym Sym.new(Compiler c)
```

Creates the empty symbol table that compiler `c` owns.

Source: `src/symtab/scopes.x:4`

Sym.next_typedef

```
Type Sym.next_typedef(Sym s, Type type, int &hops)
```

Resolves one typedef hop and counts against the shared cycle budget.

`hops` is an in-out counter initialized by the caller for the whole walk. Returns `NULL` for an unresolved link. The shared budget turns a cycle into the same diagnostic as full-chain resolution.

Source: `src/symtab/types.x:67`

Sym.normalize_declared_type

```
Type Sym.normalize_declared_type(Sym s, Type type)
```

Resolves typedef bases while retaining every declarator qualifier.

Source: `src/symtab/types.x:88`

Sym.pop_scope

```
SymScope Sym.pop_scope(Sym s)
```

Pops the innermost scope, or returns an empty scope when none exists.

Source: `src/symtab/scopes.x:65`

Sym.push_new_scope

```
void Sym.push_new_scope(Sym s)
```

Pushes a new empty lexical scope.

Source: `src/symtab/scopes.x:59`

Sym.push_scope

```
void Sym.push_scope(Sym s, SymScope scope)
```

Pushes a caller-supplied lexical scope while retaining its map objects.

Source: `src/symtab/scopes.x:62`

Sym.put

```
void Sym.put(Sym s, Map map, Var key, Var value)
```

Sets `map[key]` so that an active semantic transaction can restore it. Semantic writes to scope, file-static, and binding-fact maps use this.

Source: `src/symtab/transactions.x:6`

Sym.reference

```
List Sym.reference(Sym s, List key, Type &?type)
```

Resolves `key` or creates a forward binding in the current scope.

Stores `NULL` through `type` when no declaration supplies a type.

Source: `src/symtab/bindings.x:261`

Sym.reference_global

```
List Sym.reference_global(Sym s, List key)
```

Resolves a global name or creates its forward binding in the base scope. Local declarations cannot capture a retained macro's global reference.

Source: `src/symtab/bindings.x:274`

Sym.reopen_scopes

```
void Sym.reopen_scopes(Sym s, Block local)
```

Reopens the local scopes `Sym.close_local_scopes` set aside.

Source: `src/symtab/scopes.x:89`

Sym.reset

```
void Sym.reset(Sym s, Map globals)
```

Resets symbol state to one base scope backed by `globals`.

Later definitions mutate that caller-supplied map.

Source: `src/symtab/scopes.x:27`

Sym.reset_overlay

```
void Sym.reset_overlay(Sym s, Map base, Map overlay)
```

Resets symbol state to read `base` below the writable scope `overlay`.

Later definitions mutate `overlay`; `base` is only read.

Source: `src/symtab/scopes.x:36`

Sym.resolve_global

```
List Sym.resolve_global(Sym s, List key, Type &?type)
```

Resolves a binding through base scopes and optionally stores its type.

Returns `NULL` when no base scope contains the key.

Source: `src/symtab/bindings.x:268`

Sym.resolve_key

```
Type Sym.resolve_key(Sym s, Type key)
```

Resolves a canonical typedef key to the end of its declared chain.

Source: `src/symtab/types.x:20`

Sym.resolve_numeric_type

```
Type Sym.resolve_numeric_type(Sym s, Type type)
```

Resolves a numeric typedef without reducing semantic object types.

Returns `NULL` when resolution yields neither a numeric type nor a recognized builtin numeric typedef.

Source: `src/symtab/types.x:158`

Sym.scope_count

```
int Sym.scope_count(Sym s)
```

Returns the number of semantic scopes, including base scopes.

Source: `src/symtab/scopes.x:73`

Sym.seed_var_tags

```
void Sym.seed_var_tags(Sym s, Map symbols)
```

Registers declared `T_var` converters in the active `Type` unit.

Source: `src/symtab/bindings.x:57`

Sym.set

```
void Sym.set(Sym s, List key, List type)
```

Sets a semantic type for `key` in the required active scope.

Source: `src/symtab/bindings.x:6`

Sym.sole_typedef

```
Type Sym.sole_typedef(Sym s, Type type)
```

Returns the name of the one file-scope typedef declared directly as `type`, or NULL when no typedef or several typedefs are.

Source: `src/symtab/types.x:75`

Sym.transacting

```
int Sym.transacting(Sym s)
```

Returns whether a semantic transaction is open.

Source: `src/symtab/transactions.x:121`

Sym.unit_symbols

```
Map Sym.unit_symbols(Sym s)
```

Returns a fresh map of the base symbols with the current scope's rows merged in.

Source: `src/symtab/scopes.x:118`

Sym.unshadow_visible

```
List Sym.unshadow_visible(Sym s, String spelling)
```

Emits every local object spelled `spelling` that the current scope sees, the enclosing ones too, under one fresh spelling, so a file-scope name a template keeps reaches the global there, and returns them. Locals of other scopes keep their spellings for the native macros that name them.

Source: `src/symtab/declarations.x:234`

Sym.var_tag_for_type

```
Symbol Sym.var_tag_for_type(Sym s, Type type, Type &?resolved)
```

Returns a type's `Var` tag and optionally stores its resolved type.

`resolved` receives the final type even when the result is zero because no `Var` tag is registered. A null input stores `NULL` and returns zero.

Source: `src/symtab/types.x:206`

Sym.visible_symbols

```
List Sym.visible_symbols(Sym s)
```

Returns the visible one-part source names and their semantic types. Inner scopes win. The result is a fresh List; types remain borrowed.

Source: `src/symtab/scopes.x:132`

SymTxn

SymTxn.commit

```
void SymTxn.commit(SymTxn &?s)
```

Publishes an active semantic transaction and makes rollback a no-op. Its writes are already in the scope maps, so code holding a borrowed map observes a committed expansion. An absent or inactive transaction has no effect.

Source: `src/symtab/transactions.x:135`

SymTxn.commit_transient

```
void SymTxn.commit_transient(SymTxn &s)
```

Commits an active transaction, retaining the original counters map. The caller may then release the transaction's construction scope. Parsing and evaluation must allocate outside that temporary scope.

Source: `src/symtab/transactions.x:157`

SymTxn.local_macros_changed

```
int SymTxn.local_macros_changed(SymTxn &s)
```

Returns whether the transaction's active scope changed its macro map.

Source: `src/symtab/transactions.x:104`

SymTxn.rollback

```
void SymTxn.rollback(SymTxn &?s)
```

Restores every semantic value captured by an active transaction. An absent or inactive transaction has no effect.

Source: `src/symtab/transactions.x:177`

Public types

Type	Kind	Summary
Sym	alias	Holds a compiler's declarations in nested scopes.

Sym

```
module typedef struct Sym { Block scopes, Map globals, statics, binding_facts, prefixed; int base_scopes, local_macro_names; Block undo, int transactions; Compiler c; } *Sym
```

Holds a compiler's declarations in nested scopes. The lowest `base_scopes` contain file-scope declarations. Active transactions retain the rows each write replaces in `undo`.

Source: `src/symtab/symtab.x:122`

Design notes

`Sym` holds one compiler's declarations as a stack of scopes searched from inner to outer; its lowest scopes hold file-scope declarations. It issues the binding identity of each declared name, resolves typedef chains and aggregate fields, and records which file-scope names are private. A `SymTxn` stages the rows a macro binds, so a failed expansion leaves the table as it was.

> **Provisional API:** This reference documents current compiler behavior. Compatibility is not yet promised.

`src/transform/transform.x`

X2c AST transformation pipeline.

Functions

Function	Summary
Compiler.call_in_order	Calls the typed callable `callee` with `arguments`, each converted to its parameter and evaluated once, in source order, as a statement expression.
Compiler.lower_lambda_expr	Lowers a resolved lambda expression, under any position wrappers and parentheses, through the lambda rule of `src/components/lambda-functions.x`; other expressions pass through.
Compiler.normalize	Normalizes one bound and typed node.
Compiler.shipped_support	Adds to the unit's pending support the file-scope code a rule gives the form `source` of `kind`, bound and lowered with the support.
Compiler.transform	Lowers a bound and typed top-level AST to the normalized form consumed by emission.

`Compiler`

Compiler.call_in_order

```
List Compiler.call_in_order(Compiler c, List callee, List arguments)
```

Calls the typed callable `callee` with `arguments`, each converted to its parameter and evaluated once, in source order, as a statement expression.

Source: `src/transform/literals.x:225`

Compiler.lower_lambda_expr

```
List Compiler.lower_lambda_expr(Compiler c, List expression)
```

Lowers a resolved lambda expression, under any position wrappers and parentheses, through the lambda rule of `src/components/lambda-functions.x`; other expressions pass through. A noncapturing lambda typed `Func`, which only a meta body leaves unlifted, lifts to a `Func` of its native signature.

Source: `src/transform/expressions.x:9`

Compiler.normalize

```
Ast Compiler.normalize(Compiler c, Ast ast)
```

Normalizes one bound and typed node. Newly constructed syntax is normalized where it is produced; children enter the same operation, so completed units do not require another unit walk.

Source: `src/transform/traversal.x:33`

Compiler.shipped_support

```
void Compiler.shipped_support( Compiler c, Symbol kind, List source, String form)
```

Adds to the unit's pending support the file-scope code a rule gives the form `source` of `kind`, bound and lowered with the support. The kernel has no lowering of its own for `source`, so no rule ends the translation as `_shipped_lowering` does.

Source: `src/transform/traversal.x:121`

Compiler.transform

```
List Compiler.transform(Compiler c, List ast)
```

Lowers a bound and typed top-level AST to the normalized form consumed by emission. `c` must own the AST's bindings, origins, and conversion state. Current-node rewrites finish before child traversal; containing blocks absorb cleanup markers produced by declaration rewrites. Pending support declarations are lowered and appended after the input units. Their storage determines their interface visibility. The call may add generated origins or diagnostics to `c`.

Source: `src/transform/traversal.x:13`

Design notes

Lowers typed expressions, literals, and statements into the AST forms consumed by C emission. One recursive normalizer rewrites each node where it is produced, and children enter the same operation, so a unit is walked once. `callables.x` lowers the `Func` conversions it meets, shipped components lower the forms the kernel no longer lowers, and `cleanup.x` lowers each completed function's cleanup regions. The active Compiler reports errors.

Compiler and contributor internals

Architecture

x2c translates one `.x` file at a time into a `.c` file, a `.h` file, a `.xi` unit interface, and a `.d` dependency file. A C compiler then builds the C. Everything the language adds on top of C is decided during that translation: `Var` values, immutable `Lists` and `Strings`, interpolation, pattern matching, `defer`, and lambdas. What comes out is plain C calling into the runtime archive. Each compiler phase transforms the previous phase's output.

The compiler and the runtime are themselves x2c programs. A compiler written in the language it compiles cannot be built from source in one pass, so the repository carries pre-generated C to start from, then builds the compiler several times over and compares the results. Most of the repository layout supports that self-host build.

For the modules that implement a particular language feature, see the implementation map. For the language being implemented, see the language reference. For the runtime these phases target, see the standard library overview. The compiler API reference lists the functions and types exposed by each compiler module.

Repository topology

```
bin/        the bootstrap compiler binary
bootstrap/  checked-in portable C for the compiler and the runtime
builds/     self-host stages 0 through 3
commands/   external commands built against the current compiler
include/    links to runtime sources and stage-0 generated headers
lib/        x2c runtime sources
src/        x2c compiler sources
unittest/   runtime suites, compiler fixtures, probes, and benchmarks
examples/   curated executable and build-checked programs
docs/       this book: guide, reference, library, and internals
agents/     repository-facing documentation, contracts, and project skills
plans/      active plans-as-logs and archived execution records
etc/        shared Makefile rules, build configuration, built-in macros,
            and the compile-time Lisp environment and SDK
tools/      documentation and stage-comparison checkers
```

Three of those directories hold generated files. `builds/` holds whole staged builds and is not tracked. `include/x2c/` holds only symlinks: the runtime `.x` sources from `lib/`, and the `.h` files stage 0 generated from them. That is where `cc -iquote include/x2c` finds runtime headers. The C under `bootstrap/` is tracked, and it is re-emitted from a known-good stage 0 instead of edited by hand. The compiler under development is `builds/0/x2c`; see building the compiler for how it is built.

The compiler pipeline

The compiler lives under `src/`. `src/main.x` dispatches commands and runs the translation loop; `src/cli/cli.x` parses the command line. `src/frontend.x` owns configured source stages and isolated unit lifetimes shared with internal tools. Nested interning pools keep batch translation close to single-file peak memory. For each input file the translator runs one pipeline:

```
source
  -> tokenizing
  -> shallow parse and global environment discovery
  -> full parse and type annotation
  -> protocol conformance forms
  -> lowering
  -> generation and cache staging
  -> C token emission
  -> formatting and output (.c, .h, .xi, then .d)
```

Most phases have a flag that prints their output and stops, so the same file can be printed after each phase; the flags are in compiler options. The example below follows one file through the phases.

A worked example

```
String greet(String name) {
  return %"hello, $name";
}
```

Saved as `greet.x` in the repository root and translated into a scratch directory,

```
mkdir -p /tmp/x2c-arch
./builds/0/x2c translate --out-dir /tmp/x2c-arch greet.x
```

it produces `greet.h`:

```
/* auto-generated by x2c.  Do not edit! */

#pragma once

#ifndef __GUARD_0x10E936D3__
#define __GUARD_0x10E936D3__

#include "x2c.h"
String greet(String name);


#endif /* __GUARD_0x10E936D3__ */
```

and `greet.c`:

```
/* auto-generated by x2c.  Do not edit! */

#define __GUARD_0x10E936D3__
#include "x2c.h"

static String _0;

static int _init_guard_ = 0;

__attribute__((constructor, noinline, cold)) static void _file_init_(void);

__attribute__((constructor, noinline, cold)) static void _file_init_(void){
  x2c_initialize_protocols();
  if(_init_guard_) return;
  _init_guard_ = 1;
  _0 = String_new("hello, ");
}

String greet(String name){
  if(! _init_guard_) _file_init_();
  return String_join(NULL, cons(String_var(_0), cons(String_var(name), NULL)));
}
```

The interpolation became a `String_join` over a cons list. The constant `"hello, "` became a file-static `String` built once by a generated initializer. The sections below show where each change happens.

Tokenizing

The tokenizer reads source text and produces a positioned token array. `Compiler.tokenize` in `src/compiler/parsing/tokens.x` reads the file, hands the text to `Tokenizer.new` from `lib/tokenizer.x`, and scans it with the allocation-free character recognizers in `lib/scan.x`. Whitespace, comments, and preprocessor lines all become tokens. Later phases can therefore report positions in the original file, and directives can be re-emitted where they were written. `src/preprocess.x` classifies those directive tokens and turns the tokens of a conditional arm that C never takes into comments.

```
./builds/0/x2c translate --dump-tokens greet.x

(   1, 1  ) ident              String
(   1, 7  ) space              □
(   1, 8  ) ident              greet
(   1, 13 ) (                  (
```

```
(   1, 14 ) ident               String
(   1, 20 ) space               □
(   1, 21 ) ident               name
(   1, 25 ) )                   )
(   1, 26 ) space               □
(   1, 27 ) {                   {
(   1, 28 ) space               □□□
(   2, 3  ) return              return
(   2, 9  ) space               □
(   2, 10 ) %"                  %"
(   2, 12 ) segment             hello,□
(   2, 19 ) $                   $
(   2, 20 ) ident               name
(   2, 24 ) "                   "
(   2, 25 ) ;                   ;
(   2, 26 ) space               □
(   3, 1  ) }                   }
(   3, 2  ) space               □
```

□ and □ stand for space and newline. `%"` and `segment` are already token classes. The scanner recognizes interpolation; no later pass rescans string contents.

Shallow parse and global environment discovery

This phase reads the token stream and the declarations the file can see, and produces a `Map` from global name to type. This phase exists because x2c's typed lowering has to know the type of every name in scope, including names that come from C headers the compiler never parses in full.

The default path does not run the C preprocessor. Every unit starts from the implicit runtime prelude, the declarations of `lib/x2c.x` and everything it includes. `src/collect/collect.x` shallow-parses source text directly, splicing quote-includes inline at their include points in the same order the preprocessor would have produced them, resolving each through the same search order (the including file's directory, `.`, `src/`, `lib/`, then the `-I` chain), and overlays what it finds on the prelude. Source includes install their public declarations and compile-time definitions. Native runtime headers already covered by the prelude add no declarations. Unresolved angle includes are system headers the generated C re-includes; unresolved quote includes are driver errors. Each file is spliced once by real path, so include cycles terminate.

Collection also hashes this unit's function bodies and initialized file-static values while it skips them. Full parsing records each definition's referenced names in the same definition metadata. Linked-meta generation pairs those names with the hashes. An unchanged forward definition can therefore use its shipped reference list to check later dependencies before their bodies are parsed. Included source bodies belong to their own units and do not enter this unit's hash map.

Linked providers retain one source hash and their direct dependency hashes. Imported functions refer to that provider record. The collector checks the dependency graph once per translation process, using its existing source hashes. A changed provider or dependency requires rebuilding its meta code.

The public `x2c_meta_definition_hashes()` query still returns String hashes for definitions parsed so far. The private Lisp query `_x2c.meta.definition.calls` reads their existing reference lists for linked generation; it adds no public library callable or separate stored inventory.

Unit interfaces

A file's contribution is collected once per process. Translating a unit also writes it beside the generated C as a unit interface, `<stem>.xi`: the identity of the compiler that wrote it, the ordered declaration maps, include placeholders, the source content hash, function definitions, and the macro, Lisp, and embedded-text files the walk read with their hashes. Each declaration map holds its public rows: nonstatic declarations and complete type families needed by public declarations. Source-position rows also retain public macro and keyword definitions, Lisp effects, and provider meta-function identities. A translated unit's map also records each ordinary external conformance whose adoption it declares, as `(conformance BASE PARTICIPANT FORWARD REVERSE (bindings (NAME TYPE)...) (MEMBER STATUS SOURCE)...)` keyed by base and participant. It lists each type variable's binding and each member that did not resolve as `no-member`; the expected and template types follow from the protocol record. A unit that replays the map installs that conformance when it first uses it, instead of resolving the

adoption. When a cold walk collects a file that declares adoptions, the unit resolves them in a compiler that gathers only the prelude's and that file's declarations and records the same rows in the file's entry. A file that imports a package records none. Cold collection and interface replay use the same ordered parts, while the full parser installs included compile-time definitions at their include positions. Macro names become visible immediately; their templates are decoded into the current unit only on first use. The existing semantic write log restores that decode if its expansion rolls back. Local declaration productions stay in their owning unit's Context until full parsing consumes them. Their syntax, macro frames, and binding facts are not exported. Static helpers remain in their provider. Each provider has one function-hash table. Its meta advertisements identify the provider; collection and replay install the table before advertisements. The file holds one `(interface 6 "compiler" "path" "hash" (PARTS...) (DEFINITIONS...) (DEPENDENCIES...) (INCLUDE-DIRS...))` form in `%()` List syntax, with bare Atoms for its structural words and Strings for identifiers; the reader reads that one form without evaluating it. The API reference reads `--dump-definitions` output and joins authored prose to those definitions. Before walking a file's source, collection looks for its interface in the output directory, then in the directory that mirrors the file's home-relative path under the compiler's stage directory (or under an installed home), then in a package's `builds/`. An interface is used only when its recorded compiler, path, include search roots, and every hash still match; otherwise collection walks the source. Include search roots retain their order and distinguish home-relative, absolute, and cwd-relative spellings. Search candidates with cwd-relative spellings are checked in the current working directory. The compiler identity is a digest of the executable's contents, so a changed compiler never replays rows an earlier build collected. The prelude is the runtime `x2c.xi` interface from the library batch, so a stage build produces the prelude the next batch and the next stage replay when the stage compilers are byte-identical, as they are once the bootstrap matches the source. No tracked file is both an input and an output of a build. An installation writes its runtime interfaces with the compiler it installs. `x2c env prelude` prints the prelude interface this compiler wrote; an empty value means the compiler walks the source of `lib/x2c.x`, which costs about a quarter of a second per process. Replay must match the walk it replaces: `make proof-cold-collection` retranslates `lib/` and `src/` with the hidden `--no-interfaces` option, which reads no interface, and requires the stage 1 C, headers, and interfaces byte for byte.

`--cpp-symbols` and `--live-symbols` discover symbols through the host C preprocessor: the toolchain force-loads `lib/x2c.x` and runs `cc -E -P` as a child process. Paths stay separate argv elements, and stdout, stderr, and the real child status come back independently. A second `Compiler` shallow-parses that output instead.

Either way this is a discovery pass. Shallow parsing (`Compiler.shallow_parse`) skips function bodies, and full parsing, source diagnostics, and emitted C all come from the original positioned token stream. `--no-cpp` skips symbol discovery and is a diagnostic aid for C-only source. `--dump-symbols` prints the symbol table the compiler ends up with; the implicit runtime contributes a few thousand rows, so grep the output.

Full parse and type annotation

The full parse reads tokens and the global environment and produces the parsed AST: immutable `Lists`, tagged by node kind, annotated with resolved types. `Compiler.full_parse` in `src/compiler/parsing/parse.x` drives a recursive-descent grammar split across four modules -- `src/parse/parse.x` for declarations and top-level forms, including initializer recognition and the function-body boundary, `src/expressions/expressions.x` for the C precedence ladder plus dotted method sugar, `src/statements.x` for control flow, `match`, `try`/`catch`, and `defer`, and `src/literals/literals.x` for collection, interpolated `String`, and lambda literals. `src/meta-group/meta-group.x` stages a unit's bodied `meta` functions as native code, and `src/stage.x` carries values across that boundary. The built-in source macro in `src/macros/macros.x`, `etc/builtin-macros.x`, and the algorithms in `src/builtins.x` expand `foreach` during this pass. `src/type.x` owns the `List`-backed `Type` representation those modules consult; `src/ast.x` owns sequence placement and binding helpers; `src/symtab/symtab.x` owns lexical scopes and symbol lookup from inner to outer; `src/compiler/names.x` owns generated names; `src/compiler/parsing/tokens.x` raises `<malformed>` for a token mismatch, and `src/compiler/parsing/parse.x` holds the filtered recovery boundary that synchronizes after a parse diagnostic; `src/diagnostics.x` collects positioned diagnostics in order.

Macro definitions use that same recursive-descent parser in macro mode. Each semantic entry point accepts a typed parameter or compile-time Lisp slot and otherwise follows the ordinary grammar, so the stored template is an ordinary AST `List`. An invocation parses its arguments through those entry points, matches their

canonical capture rows once, replaces the template once, and hands the result to the ordinary recursive syntax binder. Nested invocations and Lisp slots are expanded when that binder reaches them; there is no second macro grammar or post-expansion validation pass.

```
./builds/0/x2c translate --dump-ast greet.x

(function ("String")
 (bind (binding 1 "greet")
  ((fnmod (params (param ("String") (bind (binding 2 "name") ()))))))
 (block
  (at 1 (return ("String")
          (expr ("String")
           (segments (cache 0)
            (segvar (expr ("String") (ident (binding 2 "name")))))))))
```

Types are attached as the tree is built. The function, its parameter, the `return`, and the interpolated expression all carry `("String")` already. Identifiers have become numbered bindings, and each statement is wrapped in an `(at N ...)` node. That number indexes the compiler's table of recorded source positions, so a later phase can still report the line the statement came from. Source preprocessor lines survive as `preproc` nodes, kept in source order at the top level and inside compound statements. The implicit runtime prelude is not a source token or AST node.

The parser has already cached the literal. The literal segment `"hello, "` holds no dynamic references, so `src/literals/literals.x` recorded it as a cacheable constant and the AST refers to it by slot:

```
./builds/0/x2c translate --dump-cache greet.x

0 ==> (string (expr ("String") (literal ("String") "hello, ")))
```

Lowering

Lowering turns the parsed AST into a lowered AST that the emitter can walk without knowing about x2c. `src/transform/transform.x` normalizes each node until its identity stops changing, because lowering one construct can expose another. `src/cleanup/cleanup.x` places function cleanup within the shared driver. `src/components/lambda-functions.x` lowers each lambda the driver meets; see builtin components. `src/callables/callables.x` converts callable values. It adds an adapter when a receiving callback type differs from a function's signature. A direct function gets one reusable file-static `Func` handle, and a function pointer is copied into a new `Func` context. A public inline function reaches these helpers through a generated bridge that also runs the owning unit's initializer. The helpers are queued as early declarations, normalized by the same driver, and appended to the unit.

Cleanup lowering completes each function after its body is normalized. A `try` first takes the landing form that `src/components/try-catch.x` returns; see builtin components. A program's rule on `try` applies here as well. Binding and lowering its replacement keep the already lowered parts of the `try` as they are. The pass then assigns a name to the runtime record of every `defer` region and landing frame, builds the statements that leave each region, and runs them wherever control leaves it: the region's own end, a `return` -- after saving the value, since cleanup may change what the expression read -- a `break` or `continue` that leaves the construct that bounds it, and an outward `goto`. The pass rejects a jump into a region, including the region a static local's runtime initializer opens over the rest of its block. The pass also keeps the locals and parameters a landing's parts write valid across `sigsetjmp`, as C requires: it qualifies them `volatile` unless their address escapes. Transfer rewriting follows expression and lambda normalization within the function.

```
./builds/0/x2c translate --dump-transforms greet.x

(function ("String")
 (bind (binding 1 "greet")
  ((fnmod (params (param ("String") (bind (binding 2 "name") ()))))))
 (block
  (at 2 (return
          (expr ("String")
           ("String_join(NULL, "
            (expr ("List")
```

```
(cons (expr ("Var")
       (call (expr ((func (("String"))) "Var")
              (ident (binding 3 "String_var")))
        (args (expr ("String") (cache 0)))))
 (cons (expr ("Var")
        (call (expr ((func (("String"))) "Var")
               (ident (binding 3 "String_var")))
         (args (expr ("String") (ident (binding 2 "name"))))))
  (nil)))) ")"))))))
```

The interpolation is gone, replaced by a `String_join` call over a cons list of boxed `Var` values, each `String_var` call naming its callee as a resolved binding carrying that function's type. `src/components/interpolation.x` wrote the join; see builtin components. The `return` no longer carries a type annotation; nothing downstream needs it. Some children are now raw C text. Lowering may produce emitter-ready fragments, so `"String_join(NULL, "` sits in the tree as a `String` instead of a call node.

Generation and cache staging

Generation splits the lowered AST for one unit into two ASTs, one per output file, and adds the generated initialization that makes cached constants work. `src/generate/generate.x` owns the sequence: drop whitespace and comment nodes, partition declarations and functions into header and source halves, ask `src/cache.x` to materialize the cache slots as file statics with an initializer, fold every file-level initialization block into one guarded `_file_init_` function, synthesize static prototypes, restore vertical spacing, add the unit's own header as its primary include, and patch `main` when the unit has one. Generation synthesizes already-lowered nodes, so it runs no further transform pass.

`src/cache.x` creates `_0` in the worked example. Parsing and lowering intern each constant through `Compiler.cache` in `src/compiler/literal-cache.x`, which assigns its slot; `src/cache.x` materializes those slots between lowering and emission: immutable `String`, boxed `Var`, and canonical `List`/cons graphs share generated storage, while mutable collections are copied at their use sites, so caching never changes identity semantics. An ordinary C literal promoted to `String` enters this same cache without changing its C escape spelling. Parenthesized and conditional raw-string expressions distribute promotion to their literal leaves; dynamic leaves retain their per-evaluation conversion and conditional evaluation still selects only one arm.

Generation computes the transitive cache graph used by each output region. Source-resident values keep the compact `_N` statics in the `.c` file. A public inline body instead gets deterministic, filename-hashed static slots in the generated header, so each consuming C translation unit remains self-contained. If the same key occurs in both regions, each region has a slot, but `String` and `List` canonicalization makes the resulting value identity the same.

The generated C provides two ways to run the guarded initializer. `__attribute__((constructor))` runs the source or TU-local header initializer at load time where the host supports it. Every nonstatic source function except the protocol bootstrap functions checks the file's guard. In a file whose only initialization is cache setup, only those that reach a cache, directly or through calls, check it. Every header function that reads a header cache checks the header's guard. Therefore initialization also runs on hosts without constructor support. These immutable values are allocated eagerly and retained for the process lifetime, so an allocation failure happens during load-time initialization instead of at the literal's source expression.

C token emission

Emission turns the lowered AST into a flat `List` of C tokens. `src/emit/emit.x` does this with one stack-local `Emitter` per translation unit, which holds the current function's name and static objects. Emission is therefore reentrant, and state from a failed unit does not reach the next unit. Lowering has already placed every cleanup statement, so emission prints frames, records, and statements where the AST puts them. Preprocessor nodes are re-emitted here too, with `.x` include targets rewritten to the generated `.h` they correspond to.

```
./builds/0/x2c translate --dump-code greet.x
```

```
String greet(String name){
  return String_join(NULL, cons(String_var(_0), cons(String_var(name), NULL)));
}
```

That is the emitted unit on its own. The include lines, the `_0` static, and `_file_init_` are added afterwards by generation. The final `greet.c` defines the header's guard macro and includes `x2c.h` itself, and `greet.h` holds the declaration.

Formatting and output

Formatting turns the C token `List` into text. `src/format.x` walks the tokens without reordering them: parenthesis depth suppresses statement breaks inside expressions, preprocessor tokens occupy their own lines, and a `Buffer` builds the final `String`. The result is valid C, with spacing determined by the emitter. `src/generate/generate.x` then writes the `.h`, `.c`, and `.xi` files together, named from the input basename inside the `--out-dir` directory. A write failure replaces none of them and becomes a diagnostic carrying the target path and the host error. `src/main.x` then writes the `.d` file through `src/deps.x`.

`unittest/compiler-fixtures/` pins the output of each phase. It stores expected tokens, AST, transformed AST, symbol tables, generated C, and exit status per fixture, so a change in any phase shows up as a diff.

Module ownership

Each module under `src/` and `commands/` has these responsibilities. `src/components/` groups the builtin components. It is a directory, not a module block, so its files share no private scope.

- the builtin components, described in builtin components: `src/components/bracket-access.x` (bracket access), `src/components/try-catch.x` (try, catch, and finally), `src/components/delegate-members.x` (delegate fields), `src/components/collection-literals.x` (Array and Map literals), `src/components/printf-format.x` (printf formats), `src/components/dynamic-operators.x` (dynamic operators), `src/components/raise.x` (raise), `src/components/pattern-match.x` (match), `src/components/interpolation.x` (interpolation), `src/components/destructuring.x` (destructuring), `src/components/lambda-functions.x` (lambdas), `src/components/protocol-adapters.x` (protocol members and descriptors), and `src/components/string-switch.x` (String switches);
- `src/operator-ledger.x` -- operator precedence, compound assignment, and protocol-member mappings from one compile-time ledger;
- `src/cli/cli.x`, `src/main.x` -- option metadata, parsing, and dispatch;
- `src/translation.x` -- synchronous translation, inspections, worker scheduling, ordered diagnostics, and signal forwarding;
- `src/frontend.x` -- configured source stages, process translation support, and sequential unit Context/Type lifetimes;
- `src/meta-project.x` -- the project meta build: the helper program that runs a project's meta functions, built and cached before translation;
- `src/meta-helper-client.x` -- the compiler's side of that helper: one request and one reply per call, with its timeout and failures;
- `commands/repl/` -- terminal interaction, inline editing and history, persistent submissions, and named inspection over the compiler session API;
- `src/project/project.x`, `src/toolchain.x` -- shared manifest membership, target relationships, and host compile/archive/link primitives;
- `commands/build/` -- application build and run orchestration, incremental state, script executable caching, locking, and execution;
- `src/report.x` -- dependency-free terminal progress and stable completion receipts on standard error;
- `commands/package/` -- package installation, removal, and listing under the x2c home, project lockfile preparation, and source-package builds through the matching compiler; fetch, digest, and extraction use host tools;
- `src/deps.x` -- x2c dependency parsing and atomic depfile publication;

- `src/compiler/compiler.x` -- shared compiler state, with members under `src/compiler/` for parsing (tokens, collection, the full parse, top-level definitions, and scripts), declaration production, generated names, lifecycle, binding facts, the literal cache, match patterns, pending support declarations, and source files and locations;
- `src/fields.x` -- macros that copy or set named fields and transfer a segment's compile-time state;
- `src/symtab/symtab.x` -- the symbol table: scopes, bindings, lookup, typedef resolution, and semantic transactions;
- `src/preprocess.x` -- preprocessor directive lines, conditional arms, and the directives before each form;
- `src/parse/parse.x`, `src/expressions/expressions.x`, `src/statements.x`, `src/literals/literals.x` -- grammar and AST construction;
- `src/parse-reports.x`, `src/expressions-reports.x` -- package and alias diagnostics, and expression typing diagnostics;
- `src/initializers/initializers.x` -- brace initializer conversion by C's initialization order;
- `src/lambdas.x` -- lambda parsing and the capture resolution shared by source and constructed lambdas;
- `src/macros/macros.x` -- compile-time macro definitions, imports, Lisp lifting, hygiene, and expansion;
- `src/meta-sdk/meta-sdk.x` -- the compiler's answers to `lib/meta.x` operations and the context of the running compile-time call;
- `src/meta-native/meta-native.x` -- meta functions, explicit meta calls, native meta bindings and their lifetimes, and native modules;
- `src/stage.x` -- the arguments and results that cross between meta code and program code;
- `src/meta-group/meta-group.x` -- a unit's meta group, its emission as C, and in-process staging;
- `src/builtins.x`, `src/linked-meta.x` -- the built-in macro algorithms and the shipped compile-time code compiled into the compiler;
- `src/ast.x` -- AST sequence placement and binding helpers;
- `src/ast-rewrite.x` -- AST child rewriting and iterative tree traversal;
- `src/grammar.x` -- source-form patterns and parsed-node accessors used by lowering;
- `src/type.x`, `src/protocol/protocol.x` -- type representation, conversions, protocol declarations, conformance, and native aliases, with the conformance form that presents each conformance's adapters to `src/components/protocol-adapters.x`;
- `src/type-ledger.x` -- the `Var` tag lookups projected from the tag ledger, kept in their own unit so the units that include `type.x` do not load it;
- `src/collect/collect.x` -- global environment discovery, unit interfaces, and the splicing of module blocks;
- `src/utils.x` -- repository discovery, the driver's fatal error line, and forked translation workers;
- `src/transform/transform.x` -- lowering to emitter-ready AST;
- `src/callables/callables.x` -- typed callback adapters and `Func` conversions lowered to C helpers;
- `src/adapter-memo.x` -- macros that cache compiler adapters and reuse each result by key;
- `src/cleanup/cleanup.x` -- which exits leave a cleanup region and which locals an error transfer preserves;
- `src/regions/regions.x` -- per-function region summaries and the warnings for a value that can outlive the region that allocated it;
- `src/cache.x` -- cached constants and their generated initialization;
- `src/generate/generate.x` -- header/source partitioning, unit initialization, include guards, and output writes;
- `src/emit/emit.x` -- AST to C tokens;
- `src/format.x` -- C tokens to text;
- `src/diagnostics.x` -- recorded diagnostics;
- `src/sourceview.x` -- request-owned source overlays and logical file paths;
- `commands/editor/editor.x` -- one-request diagnostics, definition, and hover transport over the configured frontend and source overlays.

The internal frontend is shared by the CLI, editor, and source graph tool. A `Frontend` borrows a configured `CliRequest`; its `ParsedUnit` retains the compiler, AST, diagnostics, and preprocessor output until explicit close. The stages are start/tokenize, collect, and parse. Failed stages return to the caller with readable diagnostics. Adapters choose printing and process exit; the default frontend collects without printing. Closing a unit releases its owned Lisp session, Type unit, and isolated Context before the next unit opens.

Process type/header caches and generated-name state still require sequential units. The frontend is not a concurrent or stable public compiler-library API. A user macro that aborts or exits still ends the process.

The optional editor worker in `tools/x2c-editor/` uses the same frontend and project configuration. `SourceView` supplies immutable unsaved text under each file's canonical path; an empty overlay remains a present file. The request owns overlays, while a parsed unit owns disk text and semantic results.

Declaration metadata follows the actual symbol-map contribution and key through collection and imports. The primary parser associates resolved bindings with physical token spans for definition and hover queries. It does not infer a source location for constructed syntax that has no physical token. Each editor request runs in a fresh process and collects source declarations without reading `.xi` interfaces, which do not carry their physical spans. This isolates process caches and macro failures from the editor service.

The VS Code adapter under `etc/vsc-extension/` converts compiler UTF-8 byte offsets to editor positions and discards responses after document changes or cancellation. It executes semantic requests only in trusted local workspaces. Syntax highlighting remains available independently of the native worker.

The runtime boundary

The compiler is an x2c program and uses the same runtime that generated programs use. There is no compiler-private collection library, so every runtime weakness the compiler hits is one a user program can hit. The main divisions:

- `lib/scope/scope.x` owns allocation lifetime, and `lib/pool/pool.x` the nested interning pools layered over it;
- `lib/var/var.x`, `lib/varconvert.x`, `lib/varops.x`, and `lib/dispatch.x` own tagged values, cause-raising conversion, operators, and dynamic dispatch;
- `lib/string/string.x`, `lib/symbol.x`, `lib/atom.x`, and `lib/list/list.x` own canonical immutable values;
- `lib/block.x`, `lib/buffer.x`, `lib/array.x`, and `lib/map.x` own mutable storage;
- `lib/iter.x` owns status-bearing traversal;
- `lib/match/match.x` owns list-pattern matching; `lib/match-plan.x` compiles patterns to plans, `lib/match-cache.x` retains them, and `lib/match-machine.x` executes them over the wordcode and state definitions in `lib/machine.x`;
- `lib/lisp/lisp.x` owns the embedded Lisp reader, session, and evaluator; the compiler uses it to read `.xi` interfaces and to run compile-time macros;
- `lib/func.x` owns generic native calls through generated adapters;
- `lib/tokenizer.x` and `lib/scan.x` own tokenization, so the compiler's first phase is library code;
- `lib/error/error.x` owns failure records, handlers, and policy;
- `lib/exception.x` owns `Error` transfer and cleanup frames, while `lib/file.x` and `lib/logger.x` provide file I/O and logging;
- `lib/common.x` supplies shared representation and initialization support;
- `lib/lib.x` contains the standalone `DisjointSet` utility.

`lib/lisp/lisp.x` evaluates recursively with tail calls; see production execution. Pattern matching executes compiled Match plans on MatchMachine.

`lib/x2c.x` is the generated source definition of the implicit runtime prelude. The generator leaves out the optional x2c system modules; they are built with the runtime and need an explicit source include. Runtime component headers keep selective includes while they build the aggregate. Every other generated header includes `x2c.h`.

The self-host build

Because `src/*.x` and `lib/*.x` are x2c, building them requires an x2c compiler, and the only way to get the first one is to start from C that an earlier x2c compiler emitted. That C is tracked under `bootstrap/`. It has to stay portable, since a C toolchain is all you have at that point.

```
bootstrap/  -- cc -->  bootstrap compiler
                              |  translates lib/ and src/
                              v
```

```
                        builds/0  ->  builds/1  ->  builds/2  ->  builds/3

generated C compared byte for byte:
  stage-diff-0   bootstrap/  against  builds/0
  stage-diff-1   builds/0    against  builds/1
  stage-diff-2   builds/1    against  builds/2
  stage-diff-3   builds/2    against  builds/3
```

Each arrow is one full translation of `lib/` and `src/` followed by a C build, and each stage directory holds both halves: the C that the *previous* compiler in the chain emitted, and the binary built from it. So `builds/0` is the bootstrap compiler's output, `builds/1` is stage 0's output, and so on. Stage 0 is the compiler everything else uses. Stages 1 through 3 show that it reproduces itself, and `make stage-3` builds them.

Staging matters because a change to `src/` changes two things at once: the compiler's behavior, and the program that compiler is asked to compile. Stage 0 shows only that the old compiler could translate the new source. Stage 1 is the first build where the new compiler compiles the new source, and stage 2 is the first build where a compiler produced *by* the new compiler does. Some bugs need both halves of the change to appear, such as a lowering that emits code its own new parser mishandles. Those surface at stage 1 or 2 and nowhere earlier.

The comparison is byte-exact. `tools/check-generated-stages.sh`, behind the `make stage-diff-0` through `make stage-diff-3` targets, compares the generated `.c`/`.h` file sets of two neighbouring stages and then compares every file byte for byte. `stage-diff-0` asks whether the bootstrap compiler re-emits the C it was built from; the rest ask whether each stage emits what the stage before it emitted. Once two adjacent stages agree, the chain has reached a fixed point.

From stage 1 on, a stage build also fails on a diagnostic. The stage Makefile passes the hidden `--fatal-warnings` option, which fails a unit that reports any warning or region finding, and compiles the generated C with `-Werror`. Stage 0 stays lenient, because the checked-in bootstrap compiler translates it and a source install compiles it with the host C compiler.

A byte comparison can catch problems that behavioral tests miss. An iteration order that depends on addresses, or a generated-name counter that carries across translation units, shows up here while every suite still passes. The same phase run twice has to produce the same bytes.

Builtin components

`src/components/bracket-access.x` registers bracket reads, stores, and updates, the `$indexed` and `$x2c.collection.store` forms, through ordinary macro patterns and quotations. `lib/rewrite.x` supplies the decorator. Array and Map mutations call their typed helpers; any other protocol participant calls its getter, `setindex`, `updateindex`, or `postfixindex` member through `Code.call_in_order`, which evaluates each operand once, in order. Binding, index admission, evaluation order, and cleanup remain compiler responsibilities.

`src/components/try-catch.x` registers the `$caught` and `$tried` source forms from `src/grammar.x` through the same decorator, both on the `try` head. Its translator returns a landing form, `(landing CODE ROWS)`: the frame and catch-site template, the finalizer, the exits that close the catch site, claim the finalizer, and leave the frame, and the body and each catch arm as regions. The finalizer is also a region. The cleanup walk described under lowering lowers the rows in order.

`src/components/delegate-members.x` registers a member call pattern whose receiver hole is `%(expr <delegate> *)`, a field keyword where a type would go. The parser marks each `delegate` field of an aggregate; a method call that ordinary lookup misses tests the rules registered for the marks its receiver's aggregate declares, after those keyed by the receiver's own type. The translator searches the delegate fields with `Type.marked_fields` and `Type.resolve_member`, reports an ambiguous path or a cycle, and returns the call through the field path as constructed syntax, which ordinary call resolution binds.

`src/components/collection-literals.x` registers the Array and Map literal forms, `$array_value` and `$map_value` from `src/grammar.x`, on the literal family keyed by the `array` and `map` heads. Each translator returns the counted constructor, `Array.update_n` or `Map.update_n` on a new container, with each element, or each key and value, converted to `Var` in source order; an empty brace among them is an empty Map, and an empty literal is the bare `Array.new()` or `Map.new()`. Each part runs once, left to right: once two or more parts read state, the

transform moves each part that reads state, through the last one that may change state, into a temporary ahead of the call, as "literal order" in `src/transform/literals.x` describes.

`src/components/printf-format.x` registers one call pattern for each printf-family member, on the call family keyed by the callee's spelling, so a call to any other function only looks its spelling up. The translator reads the static format with `Code.format`, walks it once, and has each `Var` value a conversion consumes converted with `Code.convert` to the C type that conversion reads, or displayed through `Var.str` for `%s`. The redundant `.str()` warning for those values stays with the conversion checks in `src/expressions/expressions.x`.

`src/components/dynamic-operators.x` registers its translators with Match patterns typed `Var` whose operator positions name a set, `(!or OP...)`, on the binary family for binary operators and compound assignments and the unary family for increments, decrements, and unary operators, keyed by each operator and `Var`. The transform dispatches an operation to those rules only when an operand has `Var` identity, after protocol operators resolved, presents it typed `Var`, and normalizes the replacement as it would its own lowering. A binary operator returns a `Var.binary` call through `Code.lowered`, so an operator chain does not rebind its remaining terms at each level; an update returns the lowered call of its storage's update helper with the target's address, the form the cleanup walk recognizes as a write. Truthiness stays a conversion the transform applies at condition positions, and slices, comparisons, and a participant's direct compound update remain in the transform. When a rewrite lowers its replacement, the source expressions the replacement holds lower outside the active rule, so a store nested in another store's value applies the same rule again. A rule whose pattern was derived without the subject's bindings matches without collecting them.

`src/components/raise.x` registers `raise` on the statement family keyed by the `raise` head. Its translator converts the code to a `Symbol` and each detail key and value to a `Var` with `Code.convert`, promotes a C string literal value with `Code.promoted`, and reports a value whose type is not immutable with `x2c_diagnostic_error_at`, which returns so the translation goes on. It returns the converted raise as a lowered carrier, which binding leaves as it is. The compiler lowers that raise's parts in source order, moving them into temporaries as literal parts move, and builds its literals at run time.

`src/components/pattern-match.x` registers `match` on the statement family keyed by the `match` head. Its translator returns a lowered carrier of verbatim C text around the subject, the patterns, and the arm bodies: a block that converts the subject to a `List` once with `Code.convert` and switches on the bits of the subject's head, with a label on the first arm of each head. `Code.pattern_value` gives each arm's head and binders. A cached pattern of literals, unique binders, typed captures, nested Lists, and a final `*` is tested in place by nested `if` tests on cursors over the subject, each binder a local read from its cell. A case on a visible macro whose derived pattern has that form is tested the same way after `Macro_case_root` starts it on the subject's view; another macro-valued case calls `Macro_case_capture_at`, and any other pattern calls the Match runtime through the capture buffer, at a static site when `Code.is_static_pattern` holds. The transform and the cleanup walk step into the nodes the text holds, and the parts of the match lower outside its rule, so a nested match applies it again. The generated `switch` is what an arm's `break` leaves.

`src/components/string-switch.x` registers `switch` on the statement family keyed by the `switch` head and declines a selector whose type is not `String` or an alias of it. Its translator replaces each label the switch owns by the label's index, leaving nested switches and expressions alone, and reports a label that is not a string literal with `x2c_diagnostic_error_at`. It returns a lowered carrier of the integer switch on a call of a static helper that returns the selector's index among the labels, or 0. The helper is placed once per selector type and label list under `(unit-support KEY)`, named by `x2c_binding_for` with the same key, so switches with the same labels share it. A program's `switch` rule applies to source switches only, not to the integer switch the component writes.

`src/components/interpolation.x` registers a `%"..."` literal on the literal family keyed by the `segments` head. The transform first converts each `$` insertion to a `String` as the parser converts its own, and a constructed String of one constant text reuses the parsed literal's cache slot. The translator joins the parts with `String.join` over a `List` chain of them: cached text, a converted insertion, or `String.new` of the raw text that constructed syntax spells. The compiler boxes the chain's parts and evaluates them once each, left to right, moving them ahead of the join when they move.

`src/components/destructuring.x` registers destructuring declarations on the statement family keyed by the `dstrdecl` head, an assignment statement keyed by `stmnt`, and an assignment expression on the literal family

keyed by `dstrasgn`. Its translators answer bound carriers: a `List` local read from the source with `Code.convert`, and each target given its element through the conversions lowering applies to an assignment or initialization. The expression form keeps the source in a result local as its value. The parser has already declared the targets, so binding leaves their declarations alone and they keep their scope and identity; the locals are `new-name` effects.

`src/components/lambda-functions.x` registers both lambda forms on the literal family keyed by the `lambda` head, which the transform offers each lambda it meets. A noncapturing lambda returns a bound carrier: the name of a static function placed among the unit's support declarations, whose body is the lambda's bound body as a lowered carrier, so it is not bound again. The function lowers with the unit's other support declarations, so it prepares its own cells and takes its own entry placements. An expression body carries the origin of the item that held the lambda, which `x2c_enclosing(<statement>)` returns while lowering, so a diagnostic in it points there. A capturing lambda's function also takes each captured binding as a parameter, a reference as a reference, so its body is not rewritten. The rule places a context record and an adapter, which reads the arguments through `Code.func_parameters` and calls the function with them and the record's fields, and returns the statement expression that evaluates the captures left to right and makes the `Func`. The component also registers a function rule for every definition: a local or parameter that a lambda in the body shares by reference moves into a Scope cell, so the binding outlives the call. A parameter is copied into its cell at entry, a declarator becomes the cell's declaration, a destructuring declaration copies its target after it, and every read goes through the cell; the replacement is bound again. Capture bindings have separate identities from their source bindings, so moving a source into a shared cell cannot retarget a sibling snapshot.

`src/components/protocol-adapters.x` registers on the `conform` head. Before the transform, the compiler presents each conformance whose forward converter the unit defines, and whose participant is not numeric, as a form `(conform BASE PARTICIPANT FORWARD REVERSE BINDER DESCRIPTOR (ROW...))`. The compiler has already chosen each member's owner and storage, reported collisions and missing conversions, published the external forwarding members' signatures, and bound the name of every function it asks for and of the method table. Each row is `(MEMBER STATUS SOURCE EXPECTED TEMPLATE PART...)`, whose parts are `(default NAME STORAGE)` for a forwarding member, `(inherited NAME EXPECTED)` for the prototype of a member another conformance defines, and `(thunk NAME ERASED)` for a descriptor slot. A function that completes a documented prototype keeps its prose. DESCRIPTOR is `(METHODS NAME VAR-TAG AREA [TAG])` for a `Var` participant with its own descriptor, and empty otherwise. The translator places each function and the table with `%(unit-support)`, and the table's assignment and the registration with `%(unit-init AREA)`, and returns `(seq)`. A rule on the head returns file-scope code, which joins the unit's support. When no rule returns, the translation ends with `no rule lowers this protocol conformance`. Native aliases stay in the compiler, because they enter at the participant typedef's header boundary.

The components and `src/grammar.x`, whose source forms they recognize, are compiler prelude sources.

A compiler runs its own linked copies of the components' translators, as it runs its own kernel. An edited component therefore changes the compiler built from it, not the compiler that translates the edit.

Implementation Map

x2c translates its extensions into C expressions, statements, and runtime calls. This chapter identifies the parser, transforms, runtime modules, and tests for each feature.

To follow a feature from syntax to generated C, start with its entry below and read the files in the listed order.

What the features *mean* is documented elsewhere:

- language reference: syntax and behavior
- idioms: how to use each feature well
- standard library overview: runtime contracts
- architecture: the pipeline as a whole

The compiler API reference and runtime module reference describe individual functions and types.

Inspecting a feature's lowering

To see how a compiler stage transforms a feature, dump the AST before and after that stage. Take a `foreach` loop:

```
int sum(List numbers) {
  int total = 0;
  foreach(Var n, numbers) total += n;
  return total;
}
```

Print the AST after lowering:

```
./builds/0/x2c translate --dump-transforms sum.x

(block (declare ("Var") (bindings (bind (binding 4 "n") ())))
 (declare ("List")
  (bindings
   (op = (bind (binding 5 "_x2c_gensym_macro_object_0") ())
    (expr ("List") (ident (binding 2 "numbers"))))))
 (declare ("List")
  (bindings
   (op = (bind (binding 6 "_x2c_gensym_macro_cursor_0") ())
    (expr ("List") (ident (binding 5 "_x2c_gensym_macro_object_0"))))))
 (declare ("Var")
  (bindings (bind (binding 7 "_x2c_gensym_macro_cursor_output_0") ())))
 (at 20
  (while
   (expr (int)
    (call ... (ident (binding 8 "List_try_next")))
    (args (expr ("List") (ident (binding 5 "_x2c_gensym_macro_object_0")))
     (expr (opt-ref "List") (op & ...))
     (expr (opt-ref "Var") (op & ...)))))
   (block ...))))
```

The built-in source macro in `src/macros/macros.x`, `etc/builtin-macros.x`, and `src/builtins.x` has replaced `foreach` with declarations and a `while` loop that calls `List_try_next`. The loop keeps the List, a cursor, and the current element in generated locals; `List_try_next` advances the cursor and stores the element through its reference arguments. The later transform pass replaces `total += n` with the run-time compound assignment helper for that scalar tag, and it passes the cursor and element by address. Compare `--dump-ast` with `--dump-transforms` to see those changes. The full set of dump flags is listed under compiler options.

Feature ownership

Collection and String literals

- Parse: `src/literals/literals.x`
- Lower/generate: `src/components/collection-literals.x`, `src/transform/transform.x`, `src/cache.x`
- Runtime: `lib/list/list.x`, `lib/array.x`, `lib/map.x`, `lib/string/string.x`
- Tests: literal-cache and mutable-empty fixtures, collection suites
- See also: collections

String interpolation

- Parse: `src/literals/literals.x` for `$name` and `${expr}`, which are separate sites
- Type/convert: `Compiler.convert_segment_to_string` in `src/expressions/conversions.x` selects the conversion for each segment. A segment that is already a `Var`, canonical or aliased, renders through `Var_str` whatever its run-time tag; a numeric segment takes its nearest declared `T_str` converter when one exists and is boxed and rendered through `Var_str` otherwise
- Lower/generate: `src/transform/transform.x` converts each insertion, then `src/components/interpolation.x` joins them; see builtin components
- Runtime: `lib/string/string.x`, plus `Var.str` for the boxed segments
- Tests: interpolation suite
- See also: values

Symbol and Atom literals

- Tokenize: `lib/tokenizer.x`, using the scanners in `lib/scan.x`
- Parse: `src/literals/literals.x`
- Lower/generate: `src/emit/emit.x`
- Runtime: `lib/symbol.x` for immediate `Symbols`, `lib/atom.x` for spellings that do not fit a `Symbol`
- Tests: symbol suite and checked symbol example
- See also: symbols and atoms

Indexing and slicing

- Parse/type: `src/expressions/expressions.x`, `src/type.x`
- Lower: `src/components/bracket-access.x` with ordinary ordered-call and conversion operations from `src/transform/transform.x`
- Runtime: collection helpers plus the shared bounds normalizer in `lib/common.x`
- Tests: index/slice suite and checked indexing example
- See also: collections

Method-style calls

- Parse/type: `src/expressions/expressions.x`, `src/type.x`
- Lower: `src/transform/transform.x`; a call that finds no member on a receiver with `delegate` fields goes through the one field path that provides it, which `src/components/delegate-members.x` finds
- Runtime: statically selected module function
- Tests: unit suites across runtime types and the delegate compiler fixtures

Protocol adoptions

- Parse, publish, and resolve declarations and adoptions, and choose each member's owner: `src/protocol/protocol.x`; operator mappings: `src/operator-ledger.x`
- Write each conformance's adapters: `src/components/protocol-adapters.x`; see builtin components
- Runtime descriptors and their registration: `lib/dispatch.x`; built-in adoptions: `lib/protocols.x`
- Tests: protocol, adoption, and conformance compiler fixtures and the protocol-boundaries probe
- See also: protocols

Compile-time macros and decorators

- Parse, import, hygiene, and expand: `src/macros/macros.x`
- Compiler operations for `meta` code, the `Code` and `Type` methods and the `x2c_` helpers: `lib/meta.x`
- Macro values, their application, and macro-valued `case` recognition: `lib/macrovalue/macrovalue.x`
- Compile-time Lisp boundary adapters: `etc/compiler-sdk.xlisp`
- `meta` arguments and results: `src/stage.x`; a unit's `meta` group and in-process staging: `src/meta-group/meta-group.x`; the project helper: `src/meta-project.x` builds it, `src/meta-helper-client.x` calls it, and `etc/meta-helper.x` answers, with the compiler queries that `tools/gen-meta-queries.sh` generates from `lib/meta.x` into `etc/meta-queries.x`

- Embedded native bindings: `etc/lisp-bindings.x`, `etc/lisp-bindings.xlisp`, `lib/lisp-targets.x`, `lib/lisp/lisp.x`, and `lib/func.x`
- Tests: macro import, template, decorator, inline-Lisp, and inferred-binding compiler fixtures; Lisp and `Func` suites; compile-time-macros, decorators, and inline-lisp examples
- See also: compile-time macros

Package imports and `with` names

- Parse: `src/parse/parse.x`
- Package name collection and resolution: `src/symtab/symtab.x`, `src/collect/collect.x`
- Manifest and native build ownership: `src/project/project.x`, `commands/build/build.x`, and `src/toolchain.x`
- Tests: import/package compiler fixtures and the import-greet example
- See also: packages and imports

Module blocks

- Splice the member files into the entry's text: `Compiler.module_source` in `src/collect/collect.x`, called for a unit's text in `src/frontend.x`, for every cold walk of an included file, and for the project `meta` scan in `src/meta-project.x`
- Member positions: the unit's `line_markers` rows, read by `Compiler.token_source`, `Compiler.token_column`, and `Compiler.text_source` in `src/diagnostics.x`; a replayed compile-time effect records its site in the member's own text through `Compiler.source_token`
- Member dependencies and the entry's content hash: `FileWalk` in `src/collect/collect.x`
- The `module` storage class: `_module_storage_form` in `src/parse/units.x` retypes the word as `static` under `Compiler.module_declaration`
- File-private statics: each marker row carries its file's prefix from `_file_prefix` in `src/collect/collect.x`; `_declared_key` in `src/symtab/symtab.x` respells a static's key with `Compiler.private_prefix` and `Compiler.declared_spelling`, which records prefixed spellings so derived names keep theirs; `Sym.get` and `_lookup_from` try the current file's key first, and `Compiler.unprefixed` restores the short name for Var tags and descriptors
- Tests: the `module-block` and `module-storage-class` compiler fixtures
- See also: module blocks

Exact Var-tag tests with `is` and `is not`

- Parse, type, and lower: `src/expressions/expressions.x` types each test as a `Var.is`, `Var.is_void`, or constant-row `Var.is_row` call; `src/type.x` supplies the tag facts
- Runtime tag representation: `lib/var/var.x`, `lib/common.x`
- Tests: is-type and is-not compiler fixtures plus `Var` suites
- See also: exact Var-tag tests

List destructuring

- Parse declarations and binders: `src/parse/parse.x`, `src/statements.x`
- Type access: `src/type.x`; lower: `src/components/destructuring.x`, dispatched from `src/transform/transform.x`
- Runtime source values: `lib/list/list.x`
- Tests: destructuring suite and list-destructuring compiler fixtures

Inline Lisp bindings

- Parse and expand `$lisp.bind`, `$lisp.binding`, and `$lisp.install`: `src/macros/macros.x`, `etc/lisp-bindings.x`, and `etc/lisp-bindings.xlisp`
- Runtime call boundary: `lib/lisp/lisp.x`, `lib/func.x`
- Tests: Lisp binding compiler fixtures, Lisp/`Func` suites, and the inline-lisp example
- See also: compile-time macros

Var boxing, conversion, operations, and dispatch

- Type selection: `src/type.x`
- Typedef identity and scopes: `src/symtab/symtab.x`, `src/statements.x`
- Conversion lowering: `src/expressions/expressions.x`
- Protocol operator selection: `src/expressions/expressions.x`
- Dynamic operators: `src/components/dynamic-operators.x`
- Truthiness and native emission: `src/transform/transform.x`, `src/emit/emit.x`
- Sentinel literal parsing and C spelling: `src/literals/literals.x`, `src/emit/emit.x`; expression context distinguishes the literal from C `void` type positions
- Runtime representation: `lib/common.x`, `lib/var/var.x`, `lib/scope/scope.x`
- Numeric metadata, conversion, and failure causes: `lib/varconvert.x`; shared representation declarations: `lib/common.x`
- Arithmetic, truthiness, and compound assignment: `lib/varops.x`
- Comparison policy and nonnumeric dispatch: `lib/dispatch.x`; exact numeric comparison mechanics delegate to `Var.integer_compare` and `Var.integer_floating_compare`
- Typedefs: declarations are file-scope; direct and chained aliases of `Var` use the same conversions and operators as `Var`
- Numeric conversion: all 15 numeric tags use one implementation that raises the appropriate `Error` cause on failure; `long`/`ulong`, `llong`/`ullong`, and `ldouble` name native `long`, `long long`, and `long double` families rather than fixed widths
- Limits: numeric-to-`String` conversion remains partial; other nonnumeric values use their declared conversions; helper-backed collection, enum, and bit-field compound assignment are unsupported; the C compiler checks ordinary C lvalue legality
- Sentinel contract: equality, identity, exact `is void`, and rendering inspect `void`; hashing, ordering, truthiness, iteration, conversion, arithmetic, and updates retain `<void-op>` behavior
- Tests: `Var`/`File`/VarOps suites and wide/custom, var-alias-crossings, var-numeric-lowering, var-numeric-conversions, var-native-lvalue-boundaries, var-truthy-`void`, var-nonnumeric-operator, var-unary-operator, var-helper-compound, var-enum-compound, var-bitfield-compound, and local-typedef compiler fixtures
- See also: values and Var

Scalar declarations, literals, and arithmetic

- Parse: `src/parse/parse.x`, `src/literals/literals.x`, `src/expressions/expressions.x`
- Canonical type owner: `src/type.x`
- Conversion: `src/transform/transform.x`, `src/expressions/expressions.x`
- Runtime representation: `lib/var/var.x`, `lib/scope/scope.x`
- Tests: canonical scalar fixture, invalid-scalar and overflow diagnostics, `Var` suite
- See also: values and Var

Lambdas

- Parse and resolve captures: `src/literals/literals.x`, `src/lambdas.x`
- Lower: `src/components/lambda-functions.x`, dispatched from `src/transform/transform.x`
- Direct `Func` calls and conversions: `src/callables/callables.x`, `src/expressions/expressions.x`
- Runtime: `lib/func.x` and receiving callback APIs
- Tests: captured-lambda and lambda-lowering fixtures, lambda and `Func` suites

Foreach

- Parse and expand: built-in source macro support in `src/macros/macros.x` and `etc/builtin-macros.x`; the expansion algorithm in `src/builtins.x`
- Lower/generate the expanded loop: `src/transform/transform.x`, `src/emit/emit.x`
- Runtime: `lib/iter.x` and collection adapters; `Iter.try_next` returns the traversal status
- Tests: `foreach` fixture, `Iter` suite, and checked example
- See also: iteration

Status-bearing Map operations

- Runtime owner: `lib/map.x`
- Literal boundary: `src/components/collection-literals.x`; see builtin components
- Generic routing: `lib/dispatch.x`
- Value-returning wrappers: `Map.get` and `Map.del`
- Cursor contract: structural mutation invalidates outstanding traversal state
- Tests: `Map` and `Iter` suites plus the `void`-sentinel and raw `Null` coverage
- See also: collections

Counted Array construction

- Runtime owner: `lib/array.x` through `Array.update_n` and `Array.push`
- Literal boundary: `src/components/collection-literals.x`; see builtin components
- Value contract: raw `Null` is data; `void` is rejected
- Tests: `Array` suite and the counted-literal compiler fixture
- See also: collections

Var values in printf-family formats

- Parse and type the call: `src/expressions/expressions.x`
- Lower: `src/components/printf-format.x`; see builtin components
- Runtime: `Var.convert` in `lib/varconvert.x` and `Var.str` in `lib/dispatch.x`
- Tests: the `printf-var` compiler fixtures
- See also: values and Var

Match statement

- Parse: `src/statements.x`
- Lower: `src/components/pattern-match.x`; see builtin components
- Native emission: `src/emit/emit.x`
- Runtime: `lib/match/match.x` and the plan modules listed in the runtime boundary; patterns and binding sets are ordinary `Lists` from `lib/list/list.x`
- Tests: match suites and checked nested example
- See also: pattern matching

String switch

- Parse: `src/statements.x`
- Lower: `src/components/string-switch.x`; see builtin components
- Tests: the `string-switch` and `rewrite-string-switch` compiler fixtures
- See also: C control flow

Raise, filtered catch, finally, and defer

- Parse: `src/statements.x`
- Lower/generate: `src/transform/transform.x`, `src/components/raise.x`, `src/components/try-catch.x` (see builtin components), `src/cleanup/cleanup.x`, `src/emit/emit.x`
- Runtime: `lib/exception.x` implements the frames and jumps for transfer and cleanup; `lib/error/error.x` owns handlers, policy, watermarks, accumulated records, matching, and transferring registration lifetime
- Tests: optimized `defer`/try and filtered-catch compiler fixtures, exception/`defer`/`Error` suites, and fatal/floor subprocess probes
- See also: errors and cleanup, and scopes and lifetime for what cleanup releases

Type-owned initialization

- Parse: `src/parse/parse.x`
- Lower/generate: `src/generate/generate.x`, `src/cache.x`
- Runtime: generated guard plus `TYPE.initialize()`

- Boundary: generation and cache staging lists the functions that check each guard
- Tests: type-initializer fixtures

Structured diagnostics

- Produce: all parser phases
- Collect/render: `src/diagnostics.x`
- Destination: standard error, written directly; `src/report.x` suspends the progress line first. `--diagnostics-file` also writes JSON Lines diagnostics
- Tests: parse-error, diagnostic-width, and invalid-list-splice fixtures plus the diagnostics suite

The exact location fields, one-based coordinates, token-width rendering, CLI default of 20 errors, low-level one-error default, and reusable store limit are documented in `agents/logger-and-diagnostics-guide.md`.

Compiler phase boundaries

The compiler pipeline gives the phase order and the modules in each phase, and module ownership lists each module's responsibility.

Shared Lisp initialization

The compiler and `Lisp.new()` load the same `etc/init.xlisp`: the evaluator primitives, macros, native bindings, and the standard algorithms (list traversal, numeric wrappers, selectors, predicates, and pattern helpers). The algorithms without callbacks are x2c functions in `lib/lisp-init.x`, which that file binds by name; `map`, `filter`, and `foldl` stay in Lisp. Those algorithms use fixed primitive operations internally; their public Lisp names remain replaceable. In particular, rebinding `car`, `cdr`, or `cons` does not change `filter`.

The compiler's built-in macro algorithms are compiler functions in `src/builtins.x`, bound into the shared compile-time session under the names the built-in macros call. They are bound at the compiler-only boundary; they are not part of `Lisp.new()`. Scope expansion, foreach lowering, and deferred class defaults reuse the ordinary compiler queries and syntax builders.

Native Lisp signature and binding algorithms are compiler functions in `src/builtins.x` under `_x2c.binding_*`, and `etc/lisp-bindings.xlisp` defines the Lisp entry points over them. Importing that file loads only functions. The compiler initializes binding rows and sealed groups when it installs the binding macros for each translation unit, so signature preloading cannot put mutable group state in the shared parent.

Building the Compiler

Language users can skip this chapter. It describes how the self-hosting repository builds itself and what a contributor is expected to rebuild.

A source checkout needs a GCC- or Clang-compatible C compiler, `ar`, GNU Make, Python 3, and `/bin/bash`. Python runs repository build tooling.

Check build prerequisites

From the repository root:

```
./configure
```

This reports missing core build tools together, with installation guidance. It checks the selected `CC` and `AR`, and `X2C_CC` when set. It does not compile anything or write a configuration file. `make configure` runs the same report. `make build` and `make build-safe` run it automatically before bootstrap work or cleaning.

For optional packages, use `./configure --packages`, or name only the ones you intend to build, such as `./configure --packages termbox2`. See the package instructions for native dependencies. These reports check command availability and required OpenSSL Perl modules; upstream configure scripts and compilers still check platform support.

Why a bootstrap exists

The compiler and runtime are written in x2c. A fresh machine cannot translate those `.x` sources until an x2c compiler already exists. The repository therefore carries portable C emitted by an earlier verified compiler.

The chain is:

```
tracked portable C
  -> bootstrap compiler
  -> stage 0 compiler and runtime
  -> self-host stages 1, 2, and 3
```

`make build-safe` builds through stage 0 and links `bin/x2c` to `bin/x2c-bootstrap`. `builds/0/x2c` is the development compiler. `make build-install` copies it to `bin/x2c-<branch>` and links `bin/x2c` there; see developer workflow. Language users stop at stage 0. `make stage-3` builds the self-host stages, which the self-host build explains.

Generated and tracked boundaries

- Runtime and compiler behavior is written in `.x` sources.
- Numbered build directories are generated and untracked.
- The tracked generated outputs are the portable bootstrap C under `bootstrap/`, the runtime aggregator `lib/x2c.x`, and what `make doc-generate` writes: `agents/x2c-module-catalog.md`, `docs/src/internals/compiler-api/`, `docs/src/library/modules/`, and `site/public/llms.txt` with `llms-full.txt`.

Do not hand-edit any of them; regenerate them with their targets.

Unit interfaces and native bindings

Each translated unit writes a unit interface, `<stem>.xi`, beside its `.c` and `.h`. A stage build leaves the runtime's interfaces under `builds/N/lib`. A later translation replays `lib/x2c.xi` from there as its prelude only when the same compiler wrote it; otherwise it reads `lib/x2c.x` from source once per process. These interfaces are untracked build output; Compiler Architecture describes their contents and validation.

Native `Func` bindings use canonical function `Type` `Lists`, for example `((func (("String"))) "String")`. Structural words are lowercase `Symbols`, identifier names are exact `Strings`, and a nullary function uses the single parameter `Type (void)`. Ordinary adapters and Lisp bindings build the same signature: a typedef parameter, reference target, or result keeps its declared name, such as `("Byte")` or `(& "Byte")`.

Contributor checks

`make doc-examples` compiles the book and website samples, including the PCRE2 import example. Prepare that optional package first with `make -C packages/pcre2 build`; the checker uses its generated headers and prepared dependency includes. It does not fetch dependencies itself.

For testing, style, debugging, and bootstrap-refresh instructions, start with `agents/README.md`.

See Compiler Architecture for the translation phases and the Implementation Map for feature ownership.

Literate Lisp in x2c

The reference interpreter implements x2c Lisp in one source file. It implements its reader, environments, closures, special forms, native registry, and standard vocabulary. Evaluation recurses over ordinary values; it never constructs a Lisp session, calls the production evaluator, or uses its call-frame machinery.

Build it from the repository root:

```
./builds/0/x2c build --output /tmp/reference-lisp \
  examples/programs/literate-lisp.x
/tmp/reference-lisp --selftest
/tmp/reference-lisp -e '(map (lambda (x) (* x x)) (list 1 2 3 4))'
/tmp/reference-lisp examples/data/reference-lisp/showcase.xlisp
```

With no arguments it reads forms from standard input, prompting when attached to a terminal. It accepts multiline input and continues after a failed form. The executable needs no initialization file or repository-relative lookup. The source also stands alone: its standard vocabulary is an ordinary quoted List literal, evaluated by this interpreter at startup. An explicit `import` reads the file requested by the Lisp program.

Reading the source

View this example at 120 columns. Top-level block comments explain Lisp; these comments stay within 75 columns and code within 79 columns. Terse line comments begin at column 81 and explain x2c syntax, conversions, and lifetime rules beside their uses. Notes beside the quoted standard library describe its runtime Lisp forms. Block comments begin at the left edge.

Start with `Interp.eval`, then `Interp.special`. The first reads like the evaluation rule: resolve an atom, return a literal, or evaluate a call. The second matches the grammar directly: `(quote form)`, `(def name form)`, `(lambda parameters body)`, and the other special forms. A matched shape supplies its parts; separate error helpers describe rejected forms. The small `$fail` statement macro constructs the ordinary `raise` syntax from a cause, an operation, and the remaining field/value pairs. Each call keeps its diagnostic details visible and preserves their value types and order.

`Interp.apply` receives values. Only `eval` decides which arguments to evaluate: ordinary calls evaluate left to right, while a macro receives its raw argument forms. Native argument marshalling lives with the native operations, so the evaluator contains only evaluation rules.

`Fn` is a closure: parameters, a body, captured values, and a macro flag. It uses the ordinary `Var` value tag `lambda`, preserving sorting, type queries, and native diagnostics. The record is defined entirely here; no production `Lambda` object or protocol is used.

`Env` is a stack-local link and a Map of bindings. Call bindings are freed when the call returns. Closure records and their capture Maps remain in the session Scope. Capturing copies a Var, preserving the identity and ownership of its referent. Ordinary String and List pools own canonical data.

These private records use plain structs. Class defaults would require extra initialization and comparison methods for their Map fields; the evaluator needs neither generic record boxing nor field comparison. `foreach` handles element traversal. The remaining `for` loops follow environment links or keep the parameter tail needed to bind a dotted rest argument.

Quasiquotation has two operations with distinct results: `quasiquote` returns one value; `quoted_elements` builds a containing List from the values contributed by its actual elements. An active `,@` can contribute several elements. The nesting depth determines when an unquote becomes active. `quoted_elements` walks siblings once, recursing only into nested forms. A bare `unquote`, `unquote-splicing`, or `quasiquote` atom inside an element sequence remains data; only a List headed by that name has quotation syntax. For example, `` `(a unquote (+ 1 2)) `` returns `(a unquote (+ 1 2))`, and `` `(a quasiquote ,(+ 1 2)) `` returns `(a quasiquote 3)`.

`Reader.scan` tokenizes one source batch. `Reader.read` returns successive forms, using recursive descent for Lists and reader prefixes. End of input is `void`; an unfinished or malformed form raises an error at its source position. Token storage has its own temporary Scope, while the forms outlive it. The REPL retains unfinished input and resumes from that form's start. `Repl.read_line` handles prompts, line input, and EOF. `Repl.read_unit` calls it when needed, then handles tokenization and parsing. It returns one form or `void` when no form is ready, setting

`done` at EOF. The `storage` scope owns the tokens between forms and is released before the source buffer changes. The single loop in `_repl` calls `read_unit`, passes the form to `Interp.eval`, and prints the result. Evaluation errors are caught per form.

`_install_natives` is one table of Lisp names, `bind` spellings, and ordinary x2c functions. A native whose Lisp name is `()` is available through `bind` without being installed globally. Function conversion supplies fixed native signatures; the small `$rest` macro states the shared convention for native functions that take every argument as a List. These are compile-time x2c facilities, not another runtime evaluator. The `lisp_*` strings are compatibility names; their implementations belong to this file or ordinary runtime modules.

The static `_stdlib` List supplies the remaining macros and higher-order functions in Lisp itself. x2c hoists its initialization; it needs no accessor function. Native definitions appear only in the table, rather than being declared again in this List. The few text adapters that return `Var` preserve signatures printed in Lisp error messages.

The source uses ordinary runtime collection and pattern operations, including `List.match` and replacement. Those operations retain their normal runtime implementation; this example reimplements only Lisp evaluation and uses x2c's collection and pattern operations unchanged.

Compatibility

The reference follows x2c Lisp, including behavior that differs from Scheme:

- Only the empty List is false; zero and the empty String are true.
- Ordinary arguments run left to right. Macros receive unevaluated forms.
- Special forms are callable identities, so aliases and shadowing work.
- Closures capture local values where they are made. Calls resolve parameters, captures, globals, and reserved names, without searching caller locals. Globals remain visible after rebinding.
- `def` writes globals. `eval` evaluates its supplied expression globally.
- `apply` consumes evaluated values and rejects macros and special forms, except for the built-in `apply` itself.
- `bind` looks up a fixed native registry; repeated lookup preserves identity.
- Completed effects remain when a later form fails to read or evaluate.

Native calls use x2c's existing checked `Func` boundary. Deliberate Lisp signature and form errors remain in the evaluator. No additional syntax validation or compiler-specific protection of `x2c.*` names is introduced.

Production closures also retain local names that mutable global or unknown callables may later read from current data. Captured local special identities keep quote opacity, quasiquote depth and nested parameter binding. Captures belong to the session; borrowed values keep their original owners and must remain live. The reference uses the same conservative capture rule: stable local special identities control quotation and binding, while global, parameter and computed callables retain possible local reads from their data. This can retain local values and their referents longer than their immediate use would require; it does not copy or promote borrowed objects.

The reference implements language behavior only. It omits the embedding APIs and the production stack limits, so sufficiently deep recursion can exhaust its native stack. Callable addresses are process-specific, as they are in the production implementation. Tests compare the two implementations; they do not prove identical behavior for every program.

Production execution

The production evaluator recurses on the C stack for every call except a call in tail position, which reuses its caller's frame. A self-recursive loop written in tail position therefore runs in constant native stack. The outermost call of an evaluation measures the stack. Nested calls may then use the smaller of 6 MB and four fifths of the thread's remaining stack, and a call beyond that raises `<call-stack>`. Each public entry, such as `Lisp.eval`, also opens a budget of 40,000,000 calls, tail calls included; `Lisp.call_budget` changes it. Exhausting it raises `<call-stack>` with reason `"steps"`. Free names resolve through the Lambda's captures and then the session globals and reserved names, never through caller locals. The production step budget and interruption flag stop evaluation through `<call-stack>` and `<interrupt>`. These shared Error causes allow catches, but reject

returning `ignore`, `log`, or `collect` policies. A later public entry opens a fresh budget; the session owner clears its interruption flag.

The showcase and comparison

The Lisp demonstration differentiates `x*x + 3*x` using recursive pattern matching, simplifies the result, evaluates both expressions at `x = 5`, and maps a captured offset over a List. Its result is:

```
((+ (* x x) (* 3 x)) (+ (+ x x) 3) 40 13 (11 12 13))
```

The example is listed in `examples/manifest.txt`, so `make examples` checks its output alongside the existing examples. The optional differential checker builds both implementations and compares exit status, stdout, and stderr:

```
x2c script examples/programs/check-reference-lisp --build
```

The corpus covers ordinary values, arithmetic, macros, capture, shadowing, reader failures, native errors, file imports, optional Lisp libraries, and REPL recovery. It avoids printing opaque addresses and performs no output normalization. Run it when changing this interpreter or the production Lisp semantics; it is not an additional publication gate.

Basic benchmarks

The optional runner compares the recursive interpreter with the production Lisp shell in `examples/programs/lisp.x`. Run it from the repository root:

```
python3 examples/programs/benchmark-reference-lisp.py --build
python3 examples/programs/benchmark-reference-lisp.py --repeat 10
python3 examples/programs/benchmark-reference-lisp.py --case fibonacci --show-source
```

The five workloads cover startup, recursive Fibonacci, map/fold operations, captured closures, and runtime Lisp macros. Each sample verifies an expected result. The runner alternates execution order and reports median milliseconds from fresh processes, including startup, library initialization, reading, evaluation, and output. These measure complete CLI runs; they do not isolate evaluator speed. The startup baseline is reported separately, without subtracting it from the other measurements.