The Milang Programming Language

Matthew Spellings

Contents

The Milang Programming Language

Milang is a minimalist functional programming language with:

- **Zero keywords** — everything is a function or operator, including `if`
- **Haskell-like syntax** — clean, whitespace-sensitive, expression-oriented
- **Partial evaluation** as the core compilation model — the compiler reduces your program as far as possible at compile time, then emits C code for what remains
- **Capability-based IO** — side effects flow through an explicit `world` value
- **Five annotation domains** — types (`::`) , traits (`:~`), docs (`:?`), parse declarations (`:!`), and values (`=`)

Note: the lazy binding operator `:=` is a variant of the value domain (it creates a cached thunk) and is not a separate annotation domain.

Milang is designed around three guiding principles: extreme simplicity, aggressive compile-time evaluation, and explicit capabilities for side effects.

There are no special syntactic forms in Milang — control flow, conditionals, and data construction are expressed with ordinary functions and operators. This uniform surface makes programs concise and composable, and helps both authors and tooling reason about code consistently.

Partial evaluation is the heart of the compiler: any expression that can be resolved at compile time is evaluated by the compiler itself. The result is that high-level abstractions often carry zero runtime cost — configuration, macro-like computation, and many optimizations happen automatically while compiling into straightforward C.

Milang uses an explicit capability-based IO model: the program entry point receives a `world` record that contains sub-records like `io` and `process`. By passing only the capabilities a function needs, you restrict what it can do. The compiler targets C and emits code suitable for `gcc` or `clang`, which makes Milang programs portable and fast.

The repository contains focused examples covering language features referenced throughout this guide.

Quick Example

```
Shape = {Circle radius; Rect width height}

area s = s ->
  Circle = 3.14 * s.radius * s.radius
  Rect = s.width * s.height

main world =
  world.io.println (area (Circle 5))
  world.io.println (area (Rect 3 4))
```

```
78.5
12
```

How It Compiles

```
milang source -> parse -> import resolution -> partial evaluation -> C codegen
 ↪  -> gcc
```

The partial evaluator is the heart of milang: it reduces all compile-time-known expressions to values, leaving only runtime-dependent code in the output. This means zero runtime overhead for abstractions that are fully known at compile time.

Part I

Getting Started

Chapter 1

Installation

Milang is built from source using the Haskell toolchain and compiles programs to C via `gcc`.

Pre-built Binaries

Automated builds for Windows, macOS, and Linux (static) are available at: **https://github.com/klarh/milang/actions/workflows/build.yml**

Download the artifact for your platform from any successful workflow run — no Haskell toolchain needed.

Prerequisites

You need three things installed:

Tool	Minimum version	Purpose
GHC	9.6+	Haskell compiler (builds the milang compiler itself)
cabal	3.10+	Haskell build tool
gcc	any recent	C compiler (milang emits C, then calls gcc to produce binaries)

Ubuntu / Debian

```
sudo apt install ghc cabal-install build-essential
```

Arch Linux

```
sudo pacman -S ghc cabal-install base-devel
```

macOS (Homebrew)

```
brew install ghc cabal-install gcc
```

Building from Source

Clone the repository and build:

```
git clone <repository>
cd milang
make
```

`make` runs `cabal build` inside the `core/` directory.

If you prefer to do it manually:

```
cd core
cabal update
cabal build
```

For a statically linked compiler build (Linux) using Podman, you can use the provided Makefile in the core/ directory:

```
cd core
make -f Makefile.static
# now ./milang is available
```

Verifying the Installation

Start the REPL to confirm everything works:

```
./milang repl
```

You should see a λ> prompt. Try evaluating an expression:

```
λ> 2 + 3
5
```

Press Ctrl-D to exit.

Run the test suite to make sure the compiler is healthy:

```
make test
```

This compiles and runs every `.mi` file in the repository's test suite. A successful run prints something like `Passed: 60, Failed: 0`.

Chapter 2

Hello World

This guide walks through creating, running, and compiling your first Milang program and explains common variants useful when learning the language.

Your First Program

Create a file called hello.mi with this content:

```
main world =
  world.io.println "Hello, Milang!"
```

```
Hello, Milang!
```

Run it with the bundled binary:

```
./milang run hello.mi
```

Expected output:

```
Hello, Milang!
```

What `main` and `world` mean

- `main` is the program entry point by convention (not a language keyword).
- `world` is an explicit record that carries runtime capabilities: `world.io` (console and file IO), `world.process` (exec/exit), `world.argv`, and helpers like `getEnv`.
- Only code that receives the appropriate part of `world` can perform the corresponding effects — pass only what you need to follow the principle of least privilege.

Printing and Helpers

`println` appends a newline; `print` does not. Prefer small helpers that accept only the sub-record they need:

```
greet io name = io.println ("Hello, " + name + "!")

main world =
  greet world.io "Alice"
```

```
Hello, Alice!
```

This makes `greet` unable to access process or filesystem capabilities.

Handling Command-Line Arguments

A more advanced "Hello World" might greet someone by name, using command-line arguments. The `world.argv` list contains the arguments. The following example, which you can save as `hello_argv.mi`, demonstrates this. It uses a helper function to safely get an argument or fall back to a default value.

```
-- main entrypoint
main world =
  name = fromMaybe "World" (at' 1 world.argv)
  world.io.println ("Hello, " + name + "!")
```

```
Hello, World!
```

Run this from your terminal:

```
# With no arguments
./milang run hello_argv.mi
# Expected output: Hello, World!

# With an argument
./milang run hello_argv.mi "Universe"
# Expected output: Hello, Universe!
```

This example shows several concepts:

- `world.argv`: A list of strings from the command line.
- `at'`: A prelude function to safely get an element from a list by index. It returns a `Maybe` value. (`at'` takes index first; `at` takes list first for use as an operator: `xs `at` 1`).
- `fromMaybe`: A prelude function that unwraps a `Maybe`, returning a default value if `Nothing`.

This pattern of using helpers to safely extract information is common in Milang.

Script Mode (quick experiments)

When a file does not define `main` that takes a parameter, `milang run` executes in script mode: every top-level binding is evaluated and printed. This is ideal for short tests and REPL-style exploration.

```
x = 6 * 7
y = x + 1
```

```
x = 42
y = 43
```

Script-mode output prints name/value pairs for top-level bindings (prelude/internal bindings are hidden).

Printing non-strings and Maybe values

Use `toString` to render non-string values. Many standard library functions return `Maybe` to handle operations that might fail, like converting a string to a number. For example, `toInt` returns `Just(number)` on success and `Nothing` on failure.

Use `toString` to safely print these `Maybe` values.

```
main world =
  world.io.println (toString (toInt "42"))
  world.io.println (toString (toInt "abc"))
```

```
Just(42)
Nothing
```

This will print:

```
Just(42)
Nothing
```

The `Maybe` type is how Milang handles optional values, avoiding nulls and making error handling more explicit. You can use pattern matching to safely unwrap these values.

Compiling to C

Emit the generated C and compile it:

```
./milang compile hello.mi hello.c
gcc hello.c -o hello
./hello
```

The C file embeds the milang runtime; you only need a standard C toolchain.

Using the REPL

Start the REPL for interactive experimentation:

```
./milang repl
```

Example session:

```
> 2 + 3
5
> f x = x * x
> f 8
64
> map f [1, 2, 3, 4]
[1, 4, 9, 16]
```

Bindings persist across lines; you may rethink and refine definitions live. Many common functions like `map`, `filter`, and `fold` are available automatically because they are part of the prelude.

Next Steps

- Read the full syntax cheatsheet.
- Inspect reduction with `./milang reduce` (see Partial Evaluation).
- Try the larger examples in the repository root.

Chapter 3

How Milang Works

Milang's compilation pipeline has four stages:

```
source.mi -> Parser -> Import Resolution -> Partial Evaluator -> C Codegen ->
 ↪  gcc
```

Each stage is a pure transformation of the AST, except for import resolution (which reads files and URLs) and the final gcc invocation.

1. Parser

The parser is indentation-sensitive — nested blocks are determined by whitespace, similar to Haskell or Python.

There are **zero keywords** in milang. Everything that looks like a keyword — `if`, `import`, `true`, `false` — is actually a function or value defined in the prelude. The parser only needs to recognize:

- **Bindings** across five annotation domains: `=` (value), `:` (type), `:~` (traits), `:?` (docs), `:!` (parse)
- **Expressions**: literals, application, operators, lambdas, records, lists, pattern match (`->`)
- **Operators**: parsed with configurable precedence (`:!` declarations can define new ones)

The output is a single `Expr` AST type with variants like `IntLit`, `App`, `Lam`, `Namespace`, `Record`, `Match`, and so on.

2. Import Resolution

When the parser encounters `import "path.mi"`, the import resolver:

1. **Reads the file** (local path or URL)
2. **Parses it** into an AST
3. **Recursively resolves** its imports
4. **Returns a record** of the module's exported bindings

Import types:

Syntax	Source
`import "lib/utils.mi"`	Local file (relative to importing file)
`import "https://example.com/lib.mi"`	URL (downloaded and cached)
`import "/usr/include/math.h"`	C header (extracts function signatures for FFI)

URL security: URL imports must be pinned with a SHA-256 hash using `import'` and a hash record. The `milang pin` command fetches imports and writes the hashes back into your source file. The hash covers the content of the import and all of its transitive sub-imports (a Merkle hash), so any tampering is detected.

Circular imports are handled by returning only the non-import bindings from the cycle and marking the recursive reference as a lazy thunk.

3. Partial Evaluator

The partial evaluator is the heart of the compiler. It walks the AST and **reduces every expression it can** given the values it knows at compile time.

Consider:

```
double x = x * 2
y = double 21
```

```
double = <closure>
y = 42
```

The partial evaluator sees that `double` is fully known and `21` is a literal, so it evaluates `double 21` at compile time. The result in the reduced AST is simply `y = 42` — the function call has been eliminated entirely.

Key techniques:

- **Strongly Connected Component (SCC) analysis** — bindings are sorted by dependency so each group can be reduced in order.
- **Depth-limited recursion** — recursive functions are unrolled a fixed number of times. If the result converges (reaches a base case), it becomes a compile-time constant. Otherwise, the function is left as runtime code.
- **Environment threading** — the evaluator carries a map of known bindings. When a binding's value is fully determined, it's substituted into all uses.

The partial evaluator **is** the optimizer. There is no separate optimization pass. Any abstraction that is fully known at compile time — constants, configuration, helper functions applied to literals, record construction — is resolved to a value before code generation.

4. C Code Generation

The code generator takes the reduced AST and emits a single, self-contained C file. This file includes:

- **An embedded runtime** — an arena allocator, a tagged union value type (`MiVal`), environment chains, and built-in functions.
- **Arena allocation** — all milang values are allocated from 1 MB arena blocks with 8-byte alignment. There is no garbage collector; arenas are freed in bulk.
- **Tagged unions** — every runtime value is a `MiVal` with a `tag` (`MI_INT`, `MI_FLOAT`, `MI_STRING`, `MI_CLOSURE`, `MI_RECORD`, etc.) and a payload.
- **Tail-call optimization** — `tail` calls are compiled to `goto` jumps, so recursive functions run in constant stack space.
- **Closures** — functions that capture variables are represented as a code pointer plus an environment chain of bindings.

The generated C compiles with `gcc` (or `clang`) and links against the standard C library:

```
gcc output.c -o program
```

The Key Insight

Because the partial evaluator runs at compile time, **high-level abstractions often have zero runtime cost**. A chain of helper functions, a configuration record, a computed lookup table — if the inputs are known at compile time, none of that code exists in the generated binary. Only expressions that depend on runtime values (IO, user input, command-line arguments) survive into the emitted C.

Debugging the Pipeline

Two compiler commands let you inspect intermediate stages:

- `milang dump file.mi` — shows the parsed AST before import resolution. Useful for checking how the parser interpreted your syntax.
- `milang reduce file.mi` — shows the AST after partial evaluation. This is what the code generator sees. Use it to verify that compile-time computation happened as expected.

```
./milang dump myfile.mi    # parsed AST
./milang reduce myfile.mi  # after partial evaluation
```

Part II

Language Reference

Chapter 4

Milang Syntax Cheatsheet

Milang is a functional language with **zero keywords**, Haskell-like syntax, and partial evaluation as the core compilation model. Everything is an expression.

Literals

```
42              -- integer
3.14            -- float
"hello"         -- string (supports \n \t \\ \")
"""
  multi-line    -- triple-quoted string (Swift-style margin stripping)
  string        -- closing """ indentation defines the margin
  """
[]              -- empty list (Nil record)
[1, 2, 3]      -- list literal (desugars to Cons/Nil chain)
```

Bindings

```
x = 42                    -- value binding
f x y = x + y             -- function binding (params before =)
lazy := expensive_calc    -- lazy binding (thunk, evaluated on first use)
```

Functions & Application

```
f x y = x + y             -- define: name params = body
f 3 4                     -- apply: juxtaposition (left-associative)
(\x -> x + 1)             -- lambda
(\x y -> x + y)           -- multi-param lambda
f 3 |> g                  -- pipe: g (f 3)
f >> g                    -- compose left-to-right: \x -> g (f x)
f << g                    -- compose right-to-left: \x -> f (g x)
```

Operators

All operators are just functions. Standard arithmetic, comparison, logical:

```
+ - * / % **              -- arithmetic (** is power)
== /= < > <= >=          -- comparison
&& ||                    -- logical (short-circuit)
not x                    -- logical negation (function, not operator)
+ `+`                   -- string concat (use `+` for both numeric and string)
:                        -- cons (right-assoc): 1 : 2 : [] = [1, 2]
```

Operators as functions and functions as operators:

```
(+) 3 4                  -- operator in prefix: 7
3 `add` 4                -- function in infix (backtick syntax)
```

Records

```
-- Anonymous record
point = {x = 3; y = 4}

-- Field access
point.x                       -- 3

-- Positional access (by declaration order)
point._0                      -- 3 (first field)

-- Record update
point2 = point <- {x = 10}      -- {x = 10, y = 4}

-- Nested access
world.io.println              -- chained field access

-- Destructuring
{x; y} = point               -- binds x=3, y=4
{myX = x; myY = y} = point   -- binds myX=3, myY=4

-- Parsing gotcha
-- When passing a record literal directly as an argument you may need to
 ↪  parenthesize
-- the literal or bind it to a name to avoid parse ambiguity. For example:
--   -- may need parentheses
--   getField ({a = 1}) "a"
--   -- or bind first
--   r = {a = 1}
--   getField r "a"
```

ADTs (Algebraic Data Types)

Uppercase bindings with braces declare tagged constructors:

```
Shape = {Circle radius; Rect width height}

-- Creates constructors:
c = Circle 5              -- {radius = 5} tagged "Circle"
r = Rect 3 4              -- {width = 3, height = 4} tagged "Rect"

-- Named fields also work:
```

```
Shape = {Circle {radius}; Rect {width; height}}
```

Pattern Matching

The -> operator introduces match alternatives:

```
-- Inline alternatives (separated by ;)
f x = x -> 0 = "zero"; 1 = "one"; _ = "other"

-- Indented alternatives
f x = x ->
  0 = "zero"
  1 = "one"
  _ = "other"

-- Pattern types
42                        -- literal match
x                         -- variable (binds anything)
_                         -- wildcard (match, don't bind)
Circle                    -- constructor tag match
Rect                    -- constructor tag match (fields accessible via .field)
[a, b, c]                 -- list pattern (exact length)
[first, ...rest]          -- list pattern with spread

-- Guards (| condition = body)
abs x = x ->
  n | n >= 0 = n
  n = 0 - n

-- Pattern matching in case expressions
area s = s ->
  Circle = 3.14 * s.radius * s.radius
  Rect = s.width * s.height
```

Scopes & Multi-line Bodies

Indented lines under a binding form a scope:

```
-- With explicit body expression (body = the expression after =)
compute x = result
  doubled = x * 2
  result = doubled + 1
-- Returns: value of `result` (15 when x=7)

-- Without body expression (scope returns implicit record)
makeVec x y =
  dx = x ** 2
  dy = y ** 2
  sumSquares = dx + dy
-- Returns: {dx = 49, dy = 9, sumSquares = 58}

-- Bare expressions in scopes evaluate for effect, not included in record
main world =
  world.io.println "hello"       -- effect: prints, result discarded
```

```
  world.io.println "world"       -- effect: prints, result discarded
```

Inline scopes use braces:

```
f x = result { doubled = x * 2; result = doubled + 1 }
```

IO & the World

IO uses capability-based design. `main` receives `world`:

```
main world =
  world.io.println "hello"              -- print line
  world.io.print "no newline"           -- print without newline
  line = world.io.readLine              -- read line from stdin
  contents = world.fs.read.file "f"     -- read file
  world.fs.write.file "f" "data"        -- write file
  world.fs.write.append "f" "more"      -- append to file
  exists = world.fs.read.exists "f"     -- check file exists
  world.fs.write.remove "f"             -- delete file
  result = world.process.exec "ls"      -- run shell command
  world.process.exit 1                  -- exit with code
  args = world.argv                     -- command-line args (list)
  val = world.getEnv "PATH"             -- environment variable
  0

-- Pass restricted capabilities to helpers
greet io = io.println "hello"
main world = greet world.io          -- only give IO, not process/fs
```

```
hello
```

`main`'s return value is the process exit code (int -> exit code, non-int -> 0).

Imports

```
-- Local file import (result is a record of all top-level bindings)
math = import "lib/math.mi"
x = math.square 5

-- URL import (cached locally)
lib = import "https://example.com/lib.mi"

-- URL import with sha256 pinning (required for reproducibility)
lib = import' "https://example.com/lib.mi" ({sha256 = "a1b2c3..."})

-- C header import (auto-parses function signatures)
m = import "/usr/include/math.h"
x = m.sin 1.0

-- C header with source file linking
lib = import' "mylib.h" ({src = "mylib.c"})

-- C header with extended options
lib = import' "mylib.h" ({
  sources = ["a.c", "b.c"]
```

```
  flags = "-O2"
  include = "include"
  pkg = "libpng"
})
```

Binding names are always lowercase. Uppercase names are reserved for type declarations (union types, record constructors, type annotations).

Use `milang pin <file>` to auto-discover URL imports and add sha256 hashes.

Annotation Domains

Milang has five binding domains, each with its own operator:

```
-- Value domain (=) — what it computes
add a b = a + b

-- Type domain (::) — structural type annotation
add :: Num : Num : Num

-- Sized numeric types: Int', UInt', Float' take a bit width
-- Prelude aliases: Int = Int' 64, UInt = UInt' 64, Float = Float' 64, Byte =
 ↪  UInt' 8
add8 :: Int' 8 : Int' 8 : Int' 8

-- Traits domain (:~) — computational attributes / effect sets
add :~ pure                          -- pure = [] (no effects)
greet :~ [console]                   -- needs console capability
server :~ [console, fs.read, fs.write]

-- Documentation domain (:?) — human-readable docs
add :? "Add two numbers"
add :? {summary = "Add two numbers"; params = {a = "First"; b = "Second"}}
add :? """
  Add two numbers together.
  Returns their sum.
  """

-- Parse domain (:!) — operator precedence/associativity
(+) :! {prec = 6; assoc = Left}

-- All domains can coexist on one binding:
distance :? "Euclidean distance"
distance :: Point : Point : Num
distance :~ pure
distance p1 p2 = (p2.x - p1.x)**2 + (p2.y - p1.y)**2
```

Thunks & Laziness

```
~expr                  -- thunk: delays evaluation
x := expensive         -- lazy binding: creates thunk, evaluates once on use
```

Metaprogramming

```
#expr                   -- quote: capture AST as a record
$expr                   -- splice: evaluate quoted AST back to code
f #param = $param       -- auto-quote param: compiler quotes arg at call site
```

Comments

```
-- line comment
/* block comment (nestable) */
```

Built-in Functions

Core

- `if cond then else` — conditional (auto-quotes branches via #-params)
- `len xs` — length of string or list
- `toString x` — convert to string
- `toInt s` — parse string to int; returns `Just` on success, `Nothing` on failure
- `toFloat s` — parse string to float; returns `Just` on success, `Nothing` on failure

String

- `charAt i s` — character at index; returns `Just` character when index valid, otherwise `Nothing`
- `slice start end s` — substring
- `indexOf needle haystack` — find substring (-1 if not found)
- `split sep s` — split string by separator
- `trim s` — strip whitespace
- `toUpper s` / `toLower s` — case conversion
- `replace old new s` — string replacement

List (prelude)

- `head xs` / `tail xs` / `last xs` / `init xs` — return `Maybe` (`Just` value or `Nothing`)
- `map f xs` / `filter f xs` / `fold f acc xs`
- `concat xs ys` / `push xs x` / `reverse xs`
- `take n xs` / `drop n xs` / `slice start end xs`
- `zip xs ys` / `enumerate xs` / `range start end`
- `sum xs` / `product xs` / `join sep xs`
- `any f xs` / `all f xs` / `contains x xs`
- `at lst i` / `at' i lst` — element at index (returns `Maybe`)
- `sort xs` / `sortBy f xs`

Record introspection

- `fields r` — list of `{name, value}` records
- `fieldNames r` — list of field name strings
- `tag r` — constructor `tag` string (or "")
- `getField r name` — dynamic field access; returns `Just value` if present, otherwise `Nothing`.
- `setField r name val` — return new record with field set

Utility

- `id x` / `const x y` / `flip f x y`
- `abs x` / `min a b` / `max a b`

Compiler Modes

```
milang run file.mi          # compile + run
milang compile file.mi o.c  # emit C code
milang dump file.mi         # show parsed AST
milang reduce file.mi       # show partially-evaluated AST
milang repl                 # interactive REPL
```

Chapter 5

Values & Literals

This chapter covers the literal forms you can write directly in source code.

Integers

Integer literals are written as decimal digits. Negative integers use a leading minus sign attached to the literal. At compile time integers have arbitrary precision; at runtime they default to `int64_t` (signed 64-bit).

The type system supports sized integers via `Int'` (signed) and `UInt'` (unsigned) type constructors that take a bit width: `Int' 8`, `Int' 32`, `UInt' 64`, etc. The prelude provides aliases: `Int = Int' 64`, `UInt = UInt' 64`, `Byte = UInt' 8`.

```
small = 42
zero = 0
negative = -3
big = 2 ** 32
```

```
small = 42
zero = 0
negative = -3
big = 4294967296
```

Floats

Floating-point literals require digits on both sides of the decimal point. They default to C `double` (64-bit). Negative floats use a leading minus sign. Sized floats are available via `Float'`: `Float' 32` for single precision, `Float' 64` for double precision. The prelude alias `Float = Float' 64`.

```
pi = 3.14
half = 0.5
neg = -2.718
```

```
pi = 3.14
half = 0.5
```

Strings

Strings are double-quoted and support the escape sequences `\n`, `\t`, `\\`, and `\"`.

```
greeting = "hello, world"
escaped = "line one\nline two"
length = len greeting
```

```
greeting = hello, world
escaped = line one
line two
length = 12
```

Triple-Quoted Strings

Triple-quoted strings span multiple lines with automatic margin stripping (Swift-style). The indentation of the closing `"""` defines the margin — everything to the left of that column is removed.

```
msg = """
  Hello, world!
    indented line
  """
```

```
msg = Hello, world!
  indented line
```

Booleans

There is no dedicated boolean type. Milang uses integers: `1` is true, `0` is false. Comparison and logical operators return `1` or `0`, and `if` treats `0` as false and any non-zero value as true.

```
yes = 1
no = 0
check = 3 > 2
```

```
yes = 1
no = 0
check = 1
```

Lists

Lists are linked-list cons cells declared in the prelude as `List = {Nil; Cons head tail}`. The literal syntax `[1, 2, 3]` desugars into a chain of `Cons`/`Nil` records. The cons operator `:` is right-associative.

```
nums = [1, 2, 3]
empty = []
consed = 10 : 20 : 30 : []
```

```
nums = [1, 2, 3]
empty = []
consed = [10, 20, 30]
```

See the Lists chapter for the full prelude API.

Records

A record is a set of named `fields` enclosed in braces and separated by `;` or newlines.

```
point = {x = 3; y = 4}
access = point.x + point.y
```

```
point =  {x = 3, y = 4}
access = 7
```

See the Records & ADTs chapter for updates, destructuring, and algebraic data types.

Constructors

An uppercase name applied to arguments creates a tagged record. Tags are introduced by ADT declarations or used ad-hoc.

```
Shape = {Circle radius; Rect width height}
c = Circle 5
r = Rect 3 4
```

```
Shape = _module_ {Circle = <closure>, Rect = <closure>}
Circle = <closure>
Rect = <closure>
c = Circle {radius = 5}
r = Rect {width = 3, height = 4}
```

Functions as Values

Functions are first-class values. They can be bound to names, passed as arguments, and returned from other functions. A function that has not received all of its arguments displays as `<closure>`.

```
add x y = x + y
inc = add 1
result = inc 10
```

```
add = <closure>
inc = <closure>
result = 11
```

See the Functions chapter for lambdas, pipes, and more.

Chapter 6

Functions

Functions are defined with a name, parameters, =, and a body. All functions are first-class, automatically curried, and can be used anywhere a value is expected.

Definition

A function binding lists its parameters before =. The body is a single expression or an indented scope.

```
add x y = x + y
result = add 3 4
```

```
add = <closure>
result = 7
```

When the body needs local bindings, indent them under an explicit result expression:

```
distance x1 y1 x2 y2 = result
  dx = (x2 - x1) ** 2
  dy = (y2 - y1) ** 2
  result = dx + dy
a = distance 0 0 3 4
```

```
distance = <closure>
a = 25
```

Application

Function application is juxtaposition (space-separated), and it is left-associative: `f a b` means `(f a) b`.

```
add 3 4          -- 7
(add 3) 4        -- same thing
```

Lambdas

Anonymous functions use `\params -> body`.

```
double = \x -> x * 2
add = \x y -> x + y
a = double 5
b = add 3 4
```

```
double = <closure>
add = <closure>
a = 10
b = 7
```

Lambdas are ordinary values and appear frequently as arguments to higher-order functions.

Currying & Partial Application

Every function is automatically curried. Supplying fewer arguments than a function expects returns a new function that waits for the remaining ones.

```
add x y = x + y
add5 = add 5
result = add5 10
```

```
add = <closure>
add5 = <closure>
result = 15
```

This makes it natural to build specialised functions on the fly:

```
doubled = map (\x -> x * 2) [1, 2, 3, 4]
evens = filter (\x -> x % 2 == 0) [1, 2, 3, 4, 5, 6]
total = fold (+) 0 [1, 2, 3, 4, 5]
```

```
doubled = [2, 4, 6, 8]
evens = [2, 4, 6]
total = 15
```

Pipes & Composition

The pipe operator `|>` passes a value as the `last` argument to a function, reading left-to-right:

```
result = [1, 2, 3, 4, 5] \
  |> map (\x -> x * 2) \
  |> filter (\x -> x > 4) \
  |> sum
```

```
result = 24
```

Composition operators combine functions without naming an intermediate value. `>>` composes left-to-right and `<<` composes right-to-left:

```
double x = x * 2
inc x = x + 1
double_then_inc = double >> inc
inc_then_double = inc >> double
a = double_then_inc 5
b = inc_then_double 5
```

```
double = <closure>
inc = <closure>
```

```
double_then_inc = <closure>
inc_then_double = <closure>
a = 11
b = 12
```

Recursion & Tail-Call Optimisation

Functions can reference themselves by name. Milang detects self-calls (and mutual calls) in `tail` position and compiles them with goto-based trampolining, so they run in constant stack space.

```
factorial n = if (n == 0) 1 (n * factorial (n - 1))
result = factorial 10
```

```
factorial = <closure>
result = 3628800
```

A `tail`-recursive accumulator style avoids building up a chain of multiplications:

```
fac_acc acc n = if (n == 0) acc (fac_acc (acc * n) (n - 1))
result = fac_acc 1 20
```

```
fac_acc = <closure>
result = 2432902008176640000
```

Higher-Order Functions

A higher-order function accepts or returns another function.

```
twice f x = f (f x)
inc x = x + 1
a = twice inc 3
b = twice (\x -> x * 2) 3
```

```
twice = <closure>
inc = <closure>
a = 5
b = 12
```

`if` Is a Function

Milang has zero keywords. `if` is an ordinary user-defined function in the prelude. It uses **auto-quote parameters** (`#param`) so the compiler automatically delays evaluation of each branch — only the chosen one runs:

```
if (x > 0) "positive" "non-positive"
```

No special syntax is needed at the call site. The `if` definition uses `#t` and `#e` parameters which trigger automatic quoting; inside the body, `$t` and `$e` splice (evaluate) only the selected branch. See the Metaprogramming chapter for details on auto-quote params, and Thunks & Laziness for the older ~ approach.

Chapter 7

Operators

Operators in milang are ordinary functions with special syntax. Every operator can be used in prefix form by wrapping it in parentheses, and any function can be used infix with backtick syntax.

Arithmetic

Operator	Meaning
`+`	Addition (also string concatenation)
`-`	Subtraction
`*`	Multiplication
`/`	Division (integer for ints, float for floats)
`%`	Modulo (integers only)
`**`	Exponentiation (integer exponent)

```
a = 2 + 3
b = 10 - 4
c = 3 * 7
d = 10 / 3
e = 10 % 3
f = 2 ** 10
```

```
a = 5
b = 6
c = 21
d = 3
e = 1
f = 1024
```

Float division produces a decimal result:

```
a = 7.0 / 2.0
b = 3.14 * 2.0
```

```
a = 3.5
b = 6.28
```

Comparison

Comparison operators return `1` (true) or `0` (false). `==` and `/=` work structurally on records, lists, and strings.

Operator	Meaning
`==`	Equal
`/=`	Not equal
`<`	Less than
`>`	Greater than
`<=`	Less than or equal
`>=`	Greater than or equal

```
a = 3 == 3
b = 3 /= 4
c = 5 > 2
d = [1, 2] == [1, 2]
e = "hello" == "hello"
```

```
a = 1
b = 1
c = 1
d = 1
e = 1
```

Logical

Logical operators short-circuit and return `1` or `0`. `not` is a function, not an operator.

```
a = 1 && 1
b = 1 && 0
c = 0 || 1
d = 0 || 0
e = not 0
f = not 1
```

```
a = 1
b = 0
c = 1
d = 0
e = 1
f = 0
```

Short-circuit evaluation means the right-hand side is never forced when the left side determines the result:

```
safe = 0 && (1 / 0)   -- 0, right side never evaluated
```

String Concatenation

The `+` operator also concatenates strings:

```
greeting = "hello" + " " + "world"
```

```
greeting = hello world
```

Cons

The `:` operator prepends an element to a list. It is right-associative.

```
xs = 1 : 2 : 3 : []
```

```
xs = [1, 2, 3]
```

Pipe

`x |> f` is syntactic sugar for `f x`, enabling left-to-right data flow:

```
double x = x * 2
result = 5 |> double
```

```
double = <closure>
result = 10
```

Composition

`f >> g` composes left-to-right (`\x -> g (f x)`). `f << g` composes right-to-left (`\x -> f (g x)`).

```
double x = x * 2
inc x = x + 1
pipeline = double >> inc
a = pipeline 5
```

```
double = <closure>
inc = <closure>
pipeline = <closure>
a = 11
```

Record Merge

`a <- b` produces a new record with all `fields` from `a`, overwritten by `fields` from `b`:

```
base = {x = 1; y = 2; z = 3}
updated = base <- {x = 10; z = 30}
```

```
base =  {x = 1, y = 2, z = 3}
updated =  {x = 10, y = 2, z = 30}
```

Operators as Functions

Wrap any operator in parentheses to use it in prefix (function) position:

```
a = (+) 3 4
b = (*) 5 6
total = fold (+) 0 [1, 2, 3, 4, 5]
```

```
a = 7
b = 30
total = 15
```

Functions as Infix Operators

Surround a function name with backticks to use it as an infix operator:

```
bigger = 3 `max` 7
smaller = 3 `min` 7
```

```
bigger = 7
smaller = 3
```

User-Defined Operators

You can define custom operators just like functions. Precedence and associativity are set with the parse domain `:!`. See the Parse Declarations and User Operators chapters for details.

```
(<=>) a b = if (a == b) 0 (if (a > b) 1 (0 - 1))
(<=>) :! {prec = 30; assoc = Left}
```

Chapter 8

Records & ADTs

Records are milang's primary data structure. They hold named `fields`, support structural updates, and form the basis of algebraic data types (ADTs).

Record Literals

A record is a set of `field = value` pairs inside braces, separated by `;` or newlines:

```
point = {x = 3; y = 4}
person = {name = "Alice"; age = 30}
```

```
point =  {x = 3, y = 4}
person =  {name = Alice, age = 30}
```

Field Access

Use dot notation to read a field. Dots chain for nested records.

```
point = {x = 3; y = 4}
a = point.x
b = point.y
```

```
point =  {x = 3, y = 4}
a = 3
b = 4
```

Positional Access

Fields can also be accessed by declaration order using _0, _1, etc.:

```
pair = {first = "hello"; second = "world"}
a = pair._0
b = pair._1
```

```
pair =  {first = hello, second = world}
a = hello
b = world
```

Record Update

The `<-` operator creates a new record with selected `fields` replaced. Fields not mentioned are carried over unchanged.

```
base = {x = 1; y = 2; z = 3}
moved = base <- {x = 10; z = 30}
```

```
base =  {x = 1, y = 2, z = 3}
moved =  {x = 10, y = 2, z = 30}
```

Destructuring

Bind `fields` from a record directly into the current scope. Use `{field}` for same-name bindings, or `{local = field}` to rename:

```
point = {x = 3; y = 4}
{x; y} = point
sum = x + y
```

```
point =  {x = 3, y = 4}
_destruct_23 =  {x = 3, y = 4}
x = 3
y = 4
```

Renaming during destructuring:

```
point = {x = 3; y = 4}
{myX = x; myY = y} = point
result = myX + myY
```

```
point =  {x = 3, y = 4}
_destruct_23 =  {x = 3, y = 4}
myX = 3
myY = 4
result = 7
```

Scope-as-Record

When a function body has no explicit result expression — just indented bindings — the named bindings are collected into an implicit record:

```
makeVec x y =
  magnitude = x + y
  product = x * y
v = makeVec 3 4
```

```
makeVec = <closure>
v =  {magnitude = 7, product = 12}
```

Bare expressions (not bound to a name) execute for their side effects and are **not** included in the returned record. This is how `main` works — see the Scopes chapter.

ADTs (Algebraic Data Types)

An uppercase name bound to braces containing uppercase constructors declares a tagged union:

```
Shape = {Circle radius; Rect width height; Point}
c = Circle 5
r = Rect 3 4
p = Point
```

```
Shape = _module_ {Circle = <closure>, Rect = <closure>, Point = Point {}}
Circle = <closure>
Rect = <closure>
Point = Point {}
c = Circle {radius = 5}
r = Rect {width = 3, height = 4}
p = Point {}
```

Each constructor becomes a function that produces a tagged record. Zero-field constructors (like `Point` above) are plain tagged records with no arguments.

Constructors are also available namespaced under the type name (e.g. `Shape.Circle`).

Constructors as Functions

Because constructors are just functions, they work naturally with `map` and other higher-order functions:

```
values = map (\x -> Just x) [1, 2, 3]
```

```
values = [Just {val = 1}, Just {val = 2}, Just {val = 3}]
```

Pattern Matching on Tags

Use the `->` operator to match on a value's constructor `tag`. After a `tag` matches, the record's `fields` are accessible via dot notation or destructuring:

```
Shape = {Circle radius; Rect width height}
area shape = shape ->
  Circle = 3.14 * shape.radius * shape.radius
  Rect = shape.width * shape.height
a = area (Circle 5)
b = area (Rect 3 4)
```

```
Shape = _module_ {Circle = <closure>, Rect = <closure>}
Circle = <closure>
Rect = <closure>
area = <closure>
a = 78.5
b = 12
```

Named-field destructuring in alternatives:

```
Shape = {Circle radius; Rect width height}
area shape ->
  Circle {radius} = 3.14 * radius * radius
  Rect {width; height} = width * height
a = area (Circle 5)
b = area (Rect 3 4)
```

```
Shape = _module_ {Circle = <closure>, Rect = <closure>}
Circle = <closure>
```

```
Rect = <closure>
area = <closure>
a = 78.5
b = 12
```

See the Pattern Matching chapter for the full range of patterns, guards, and list matching.

Record Introspection

Several built-in functions let you inspect records dynamically:

Function	Returns
`fields r`	List of `{name, value}` records (`[]` for non-records)
`fieldNames r`	List of field-name strings
`tag r`	Constructor `tag` string, or `""` for untagged values
`getField r "name"`	`Just value` if present, `Nothing` if missing
`setField r "name" val`	New record with field set

Chapter 9

Pattern Matching

Pattern matching in milang uses the `->` operator to dispatch on a value's shape. There are no keywords — `->` is an expression that evaluates the first alternative whose pattern matches.

Basic Syntax

Write `expr ->` followed by alternatives. Each alternative is `pattern = body`. Alternatives can appear inline (separated by `;`) or indented on separate lines.

Inline:

```
classify x = x -> 0 = "zero"; 1 = "one"; _ = "other"
a = classify 0
b = classify 1
c = classify 42
```

```
classify = <closure>
a = zero
b = one
c = other
```

Indented:

```
classify x = x ->
  0 = "zero"
  1 = "one"
  _ = "other"
a = classify 0
b = classify 1
c = classify 42
```

```
classify = <closure>
a = zero
b = one
c = other
```

Literal Patterns

Integers and strings match by exact value:

```
describe n = n ->
  0 = "zero"
  1 = "one"
  _ = "many"
a = describe 0
b = describe 1
c = describe 99
```

```
describe = <closure>
a = zero
b = one
c = many
```

Variable Patterns

A lowercase name matches any value and binds it for use in the body:

```
myAbs x = x ->
  n | n >= 0 = n
  n = 0 - n
a = myAbs 5
b = myAbs (0 - 3)
```

```
myAbs = <closure>
a = 5
b = 3
```

Wildcard

_ matches any value without binding it. Use it for catch-all branches:

```
isZero x = x ->
  0 = 1
  _ = 0
a = isZero 0
b = isZero 7
```

```
isZero = <closure>
a = 1
b = 0
```

Constructor Tag Patterns

Match on a tagged record's constructor. After matching, the scrutinee's `fields` are accessible through dot notation:

```
Shape = {Circle radius; Rect width height}
area shape = shape ->
  Circle = 3.14 * shape.radius * shape.radius
  Rect = shape.width * shape.height
a = area (Circle 5)
b = area (Rect 3 4)
```

```
Shape = _module_ {Circle = <closure>, Rect = <closure>}
Circle = <closure>
```

```
Rect = <closure>
area = <closure>
a = 78.5
b = 12
```

With named-field destructuring in the pattern, `fields` are bound directly:

```
Shape = {Circle radius; Rect width height}
area shape ->
  Circle {radius} = 3.14 * radius * radius
  Rect {width; height} = width * height
a = area (Circle 5)
b = area (Rect 3 4)
```

```
Shape = _module_ {Circle = <closure>, Rect = <closure>}
Circle = <closure>
Rect = <closure>
area = <closure>
a = 78.5
b = 12
```

List Patterns

Match a list by its elements. `[a, b, c]` matches a list of exactly three elements. `[first, ...rest]` matches one or more elements, binding the `tail`:

```
xs = [10, 20, 30, 40]
result = xs ->
  [a, b, ...rest] = {first = a; second = b; rest = rest}
  [] = {first = 0; second = 0; rest = []}
```

```
xs = [10, 20, 30, 40]
result =  {first = 10, second = 20, rest = [30, 40]}
```

An empty-list pattern matches `[]` (`Nil`):

```
isEmpty xs = xs ->
  [] = "empty"
  _ = "non-empty"
a = isEmpty []
b = isEmpty [1]
```

```
isEmpty = <closure>
a = empty
b = non-empty
```

Guards

A guard adds a condition to an alternative using `| condition` before the `=`. The alternative only matches when both the pattern and the guard are satisfied:

```
classify x = x ->
  n | n < 0 = "negative"
  n | n == 0 = "zero"
  _ = "positive"
a = classify (0 - 5)
b = classify 0
```

```
c = classify 10
```

```
classify = <closure>
a = negative
b = zero
c = positive
```

Guard-Only Matching

When every alternative uses only a guard (no structural pattern), you can write guards directly after `->`:

```
classify x = x ->
  | x < 0 = "negative"
  | x == 0 = "zero"
  | _ = "positive"
a = classify (0 - 5)
b = classify 0
c = classify 10
```

```
classify = <closure>
a = negative
b = zero
c = positive
```

Combined Pattern + Guard

A constructor or literal pattern can be paired with a guard:

```
Shape = {Circle radius; Rect width height}
safeArea shape ->
  Circle {radius} | radius > 0 = 3.14 * radius * radius
  _ = 0
a = safeArea (Circle 5)
b = safeArea (Circle 0)
c = safeArea (Rect 3 4)
```

```
Shape = _module_ {Circle = <closure>, Rect = <closure>}
Circle = <closure>
Rect = <closure>
safeArea = <closure>
a = 78.5
b = 0
c = 0
```

Match in Function Bindings

The `f param ->` sugar defines a function that immediately matches its `last` parameter, avoiding an extra `= param ->` layer:

```
Shape = {Circle radius; Rect width height}
describe label shape ->
  Circle = label + ": circle"
  Rect = label + ": rect"
```

```
  _ = label + ": unknown"
a = describe "shape" (Circle 5)
b = describe "shape" (Rect 3 4)
```

```
Shape = _module_ {Circle = <closure>, Rect = <closure>}
Circle = <closure>
Rect = <closure>
describe = <closure>
a = shape: circle
b = shape: rect
```

Exhaustiveness

When the compiler can determine the type of a scrutinee (e.g., from a `::` type annotation), it checks that all constructors of a union type are covered. If any constructor is missing and there is no wildcard `_` catch-all, the compiler emits a warning:

```
warning: non-exhaustive patterns for Shape — missing: Rect
```

To silence the warning, either cover all constructors explicitly or add a wildcard branch:

```
area s = s ->
  Circle = 3.14 * s.radius * s.radius
  _ = 0  -- catch-all for all other shapes
```

Exhaustiveness checking only triggers when the scrutinee type is a known union type from a `::` annotation. Unannotated scrutinees without a catch-all will compile without warning but may fail at runtime if an unmatched constructor is encountered.

Matching Maybe

```
matchMaybe m = m ->
  Just {val} = "Just(" + toString val + ")"
  Nothing = "Nothing"

main world =
  world.io.println (matchMaybe (Just 5))
  world.io.println (matchMaybe Nothing)
```

```
Just(5)
Nothing
```

Chapter 10

Lists

Lists in milang are singly-linked cons cells, declared in the prelude as `List = {Nil; Cons head tail}`. The bracket syntax is sugar that desugars into this representation.

Constructing Lists

```
nums = [1, 2, 3, 4, 5]
empty = []
consed = 10 : 20 : 30 : []
```

```
nums = [1, 2, 3, 4, 5]
empty = []
consed = [10, 20, 30]
```

`[]` is `Nil`, and `[1, 2, 3]` desugars to `Cons 1 (Cons 2 (Cons 3 Nil))`. The `:` operator (cons) is right-associative.

Use `range` to generate a sequence:

```
a = range 1 6
b = range 1 11
```

```
a = [1, 2, 3, 4, 5]
b = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Accessing Elements

`head`, `tail`, `last`, and `init` all return `Maybe` values — `Just x` on success, `Nothing` on an empty list. `at` returns `Maybe` for index access.

```
xs = [10, 20, 30]
a = head xs
b = tail xs
c = last xs
d = init xs
e = at xs 1
f = head []
```

```
xs = [10, 20, 30]
a = Just {val = 10}
```

```
b = Just {val = [20, 30]}
c = Just {val = 30}
d = Just {val = [10, 20]}
e = Just {val = 20}
f = Nothing {}
```

`len` returns the number of elements:

```
a = len [1, 2, 3]
b = len []
```

```
a = 3
b = 0
```

Transforming

map

Apply a function to every element:

```
doubled = map (\x -> x * 2) [1, 2, 3, 4, 5]
```

```
doubled = [2, 4, 6, 8, 10]
```

filter

Keep elements satisfying a predicate:

```
evens = filter (\x -> x % 2 == 0) [1, 2, 3, 4, 5, 6]
```

```
evens = [2, 4, 6]
```

fold

Left-fold with an accumulator:

```
total = fold (+) 0 [1, 2, 3, 4, 5]
```

```
total = 15
```

reverse

```
backwards = reverse [1, 2, 3, 4, 5]
```

```
backwards = [5, 4, 3, 2, 1]
```

take / drop

```
front = take 3 [1, 2, 3, 4, 5]
back = drop 3 [1, 2, 3, 4, 5]
```

```
front = [1, 2, 3]
back = [4, 5]
```

zip

Pair up elements from two lists:

```
pairs = zip [1, 2, 3] [10, 20, 30]
```

```
pairs = [[1, 10], [2, 20], [3, 30]]
```

enumerate

Produce `[index, value]` pairs:

```
indexed = enumerate ["a", "b", "c"]
```

```
indexed = [[0, a], [1, b], [2, c]]
```

Combining Lists

```
joined = concat [1, 2] [3, 4]
appended = push [1, 2, 3] 4
```

```
joined = [1, 2, 3, 4]
appended = [1, 2, 3, 4]
```

`join` concatenates a list of strings with a separator:

```
csv = join ", " ["alice", "bob", "carol"]
```

```
csv = alice, bob, carol
```

Querying

```
xs = [1, 2, 3, 4, 5]
a = sum xs
b = product xs
c = any (\x -> x > 4) xs
d = all (\x -> x > 0) xs
e = contains xs 3
f = contains xs 99
```

```
xs = [1, 2, 3, 4, 5]
a = 15
b = 120
c = 1
d = 1
e = 1
f = 0
```

Pipelines

Lists work naturally with the pipe operator for readable data processing:

```
result = range 1 11 \
  |> filter (\x -> x % 2 == 0) \
  |> map (\x -> x * x) \
  |> sum
```

```
result = 220
```

Pattern Matching on Lists

Match by exact length with `[a, b, c]`, or match head and `tail` with `[first, ...rest]`:

```
xs = [10, 20, 30, 40]
result = xs ->
  [a, b, ...rest] = a + b
  [] = 0
```

```
xs = [10, 20, 30, 40]
result = 30
```

Recursive functions often pattern-match to walk a list:

```
mySum xs = xs ->
  [x, ...rest] = x + mySum rest
  [] = 0
```

Chapter 11

Scopes & Bindings

Scopes are the backbone of milang's structure. Every indented block and every brace-delimited block creates a new scope with its own bindings.

Basic Bindings

```
name = expr            -- eager binding
name := expr           -- lazy binding (thunk, evaluated at most once)
name params = expr     -- function binding
```

```
x = 42
double x = x * 2
result = double x
```

```
x = 42
double = <closure>
result = 84
```

Indentation-Sensitive Scoping

Indented lines beneath a binding form a scope. There are two modes depending on whether an explicit result expression appears after =.

Explicit Body

When a binding has an expression directly after =, that expression is the scope's return value. The indented children are local definitions visible only inside that scope:

```
compute x = result
  doubled = x * 2
  result = doubled + 1
a = compute 7
```

```
compute = <closure>
a = 15
```

Here `doubled` and `result` are local to `compute`. The value of `compute 7` is the expression after =, which is `result` (15).

Implicit Record (Scope-as-Record)

When a binding has **no** expression after `=` — only indented children — the named bindings are collected into a record and returned automatically:

```
makeVec x y =
  sum = x + y
  product = x * y
v = makeVec 3 4
```

```
makeVec = <closure>
v =  {sum = 7, product = 12}
```

`makeVec 3 4` returns `{sum = 7, product = 12}`. This is milang's lightweight alternative to explicit record construction.

Inner Scopes Shadow Outer Names

A binding in an inner scope shadows any identically-named binding from an enclosing scope. The outer binding is unaffected:

```
x = 10
f = result
  x = 99
  result = x + 1
outer = x
inner = f
```

```
x = 10
f = 100
outer = 10
inner = 100
```

Inline Scopes (With Blocks)

Braces create an inline scope on a single line. The expression before the braces is the return value, and the bindings inside are local:

```
f x = result { doubled = x * 2; result = doubled + 1 }
a = f 7
```

```
f = <closure>
a = 15
```

Bare Expressions (Effect Statements)

A bare expression in a scope — one not bound to a name — is evaluated for its side effect. Its result is discarded and **not** included in any implicit record:

```
main world =
  world.io.println "hello"    -- effect, result discarded
  world.io.println "world"    -- effect, result discarded
  0                           -- explicit body (exit code)
```

The first two lines run `println` for their side effects. The final `0` is the return value of `main`.

The `main` Function Pattern

A typical `main` combines all three concepts — local bindings, bare effect expressions, and an explicit result:

```
main world =
  name = "milang"                       -- local binding
  world.io.println ("Hello, " + name)  -- bare effect
  0                                     -- return value (exit code)
```

Binding Order

- Bindings evaluate top-to-bottom (left-to-right in brace scopes).
- Later bindings may reference earlier ones.
- The compiler tracks impure (`world`-tainted) bindings and guarantees they execute in declaration order via an auto-monad spine.
- Pure bindings can theoretically be reordered by the optimiser.

Chapter 12

Imports & Modules

Every `.mi` file is a module. Importing a module evaluates it and returns a record containing all of its top-level bindings. You bind that record to a name and access its members with dot notation — no special export lists or visibility modifiers.

Local Imports

Use `import` with a file path (relative to the importing file's directory):

```
math = import "lib/mymath.mi"

area = math.circle_area 5
```

The result of `import` is a record, so `math.circle_area` and `math.pi` access individual bindings from the imported file.

A Complete Example

```
-- Suppose lib/mymath.mi contains:
--   pi = 3.14159
--   circle_area r = pi * r * r

-- We can inline the same definitions here to demonstrate:
pi = 3.14159
circle_area r = pi * r * r

circumference r = 2 * pi * r

pi = 3.14159
circle_area = <closure>
circumference = <closure>
```

URL Imports

Remote modules are imported the same way — just use a URL:

```
collections = import "https://example.com/milang-stdlib/collections.mi"
```

```
total = collections.sum [1, 2, 3]
```

The compiler downloads the file and caches it locally. On subsequent runs the cached version is used.

Pinned Imports with `import'`

URL imports must be pinned by their SHA-256 hash using the `import'` form:

```
lib = import' "https://example.com/lib.mi" ({sha256 = "a1b2c3..."})
```

If the downloaded content does not match the hash, compilation fails. The `milang pin` command computes the hash for you:

```
$ milang pin https://example.com/lib.mi
sha256 = "a1b2c3d4e5f6..."
```

C Header Imports

When the path ends in `.h`, the compiler parses the C header and exposes its functions as milang bindings. See the C FFI chapter for details.

```
m = import "math.h"
result = m.sin 1.0
```

You can also associate C source files and compiler flags with `import'`:

```
lib = import' "mylib.h" ({src = "mylib.c"})
answer = lib.add_ints 3 4
```

Circular Imports

Milang supports circular imports. When module A imports module B and B imports A, the resolver detects the cycle and marks the circular bindings as lazy (thunks) to break the dependency. Both modules load correctly and can reference each other's bindings.

Diamond Imports

If two modules both import the same third module, it is loaded and evaluated only once. The two importers share the same record, so there is no duplication or inconsistency.

Visibility

All top-level bindings in a `.mi` file are exported — there is no private/public distinction. If you want to signal that a binding is an internal helper, use a naming convention such as an underscore prefix (`_helper`), but the compiler does not enforce this.

Chapter 13

IO & the World

Milang uses a capability-based IO model. Side effects are not global — they flow through an explicit `world` record that the runtime passes to `main`. If a function never receives `world` (or a sub-record of it), it cannot perform IO.

Hello World

```
main world =
  world.io.println "Hello, world!"
```

```
Hello, world!
```

`main` is the program entry point. It receives `world` and its return value becomes the process exit code.

The World Record

`world` is a record containing sub-records for different capabilities:

Path	Description
`world.io`	Console IO (println, print, readLine)
`world.fs.read`	Read-only filesystem (file, exists)
`world.fs.write`	Write filesystem (file, append, remove)
`world.fs`	Full filesystem access (read + write)
`world.process`	Process execution and exit
`world.argv`	Command-line arguments (pure — no effect)
`world.getEnv`	Read environment variables

Console IO

```
world.io.println msg          -- print with trailing newline
world.io.print msg            -- print without newline
line = world.io.readLine      -- read one line from stdin
```

File IO

File operations are split by capability: `world.fs.read` for reading and `world.fs.write` for writing. This enables fine-grained trait annotations.

```
content = world.fs.read.file "data.txt"
world.fs.write.file "out.txt" content
world.fs.write.append "log.txt" "new line\n"
exists = world.fs.read.exists "data.txt"     -- returns 1 or 0
world.fs.write.remove "tmp.txt"
```

Process

```
output = world.process.exec "ls -la"      -- run shell command, return output
world.process.exit 1                       -- exit immediately with status code
```

Command-Line Arguments and Environment

`world.argv` is an inert list — it does not perform IO, so it is always available:

```
main world =
  world.io.println (len world.argv)
```

`world.getEnv` reads an environment variable by name:

```
home = world.getEnv "HOME"
```

Capability Restriction

Because capabilities are just record `fields`, you can restrict what a helper function can do by passing only the sub-record it needs:

```
greet io = io.println "hello from restricted IO"

main world =
  greet world.io
```

```
hello from restricted IO
```

`greet` receives `world.io` and can print, but it structurally cannot access `world.process` — there is no way for it to execute shell commands or exit the process.

Exit Code

The return value of `main` is used as the process exit code. An integer is used directly; any non-integer value (record, string, etc.) defaults to exit code 0.

```
main world =
  world.io.println "exiting with code 0"
```

```
exiting with code 0
```

Script Mode

When a file has no `main` binding that takes a parameter, milang runs in script mode: every top-level binding is evaluated and printed.

```
x = 6 * 7
y = x + 1
greeting = "hello"
```

```
x = 42
y = 43
greeting = hello
```

This is useful for quick calculations and exploring the language without writing a full `main` function.

Auto-Monad Spine

You do not need monads or do-notation in milang. The compiler automatically tracks which expressions are `world`-*tainted* (they transitively reference `world`). Impure expressions are guaranteed to execute in the order they appear in the source. Pure expressions can float freely, opening the door for future optimizations. The result is imperative-looking code that is safe and predictable.

Chapter 14

C FFI

Milang can call C functions directly by importing a `.h` header file. The compiler parses the header, extracts function signatures, and maps C types to milang types (`int`/`long` -> `Int`, `double` -> `Float`, `float` -> `Float' 32`, `char*` -> `Str`). At code generation time the header is `#included` and calls are emitted inline — no wrapper overhead.

Importing C Headers

Import a system header the same way you import a `.mi` file:

```
m = import "math.h"

result = m.sin 1.0
root = m.sqrt 144.0
```

The result is a record whose `fields` are the C functions declared in the header. Use dot notation to call them.

Selective Import with `import'`

If you only need a few functions, or need to attach compilation options, use the `import'` form:

```
m = import' "math.h" ({})
result = m.cos 0.0
```

Associating C Source Files

For your own C libraries, tell the compiler which source files to compile alongside the generated code:

```
lib = import' "mylib.h" ({src = "mylib.c"})
answer = lib.add_ints 3 4
```

The `src` field takes a single source file path (relative to the importing `.mi` file).

Advanced Options

The options record passed to `import'` supports several `fields`:

Field	Type	Description
`src`	`Str`	Single C source file to compile
`sources`	`List`	Multiple source files: `["a.c", "b.c"]`
`flags`	`Str`	Additional compiler flags (e.g. `"-O2 -Wall"`)
`include`	`Str`	Additional include directory
`pkg`	`Str`	pkg-config package name — auto-discovers flags and includes

Example with multiple options:

```
lib = import' "mylib.h" ({
  sources = ["mylib.c", "helpers.c"]
  flags = "-O2"
  include = "vendor/include"
})
```

Using pkg-config for a system library:

```
json = import' "json-c/json.h" ({pkg = "json-c"})
```

How It Works

1. The import resolver reads the `.h` file and extracts function declarations.
2. Each C function becomes an internal `CFunction` AST node with its milang type signature.
3. During C code generation the header is `#included` and calls are emitted as direct C function calls.
4. Any associated source files are compiled and linked automatically.

Security Considerations

C code bypasses milang's capability model — a C function can perform arbitrary IO, allocate memory, or call system APIs regardless of what capabilities were passed to the milang caller. Use the following flags to restrict FFI access:

- `--no-ffi` — disallow all C header imports. Any `import "*.h"` will fail.
- `--no-remote-ffi` — allow local `.mi` files to use C FFI, but prevent URL-imported modules from importing C headers. This stops remote code from escaping the capability sandbox through native calls.

These flags are especially important when running untrusted or third-party milang code.

Chapter 15

Standard Library Reference

This page documents all functions available in the milang prelude, C builtins, and operators. Functions marked ”extensible” can be extended for user-defined types via open function chaining.

Types

```
Bool = {True; False}
List = {Nil; Cons head tail}
Maybe = {Nothing; Just val}

-- Sized numeric type aliases (Int', UInt', Float' are primitive constructors)
Int = Int' 64       -- signed 64-bit integer
UInt = UInt' 64     -- unsigned 64-bit integer
Float = Float' 64   -- 64-bit floating-point
Byte = UInt' 8      -- unsigned 8-bit integer
```

Extensible Functions

These functions are designed to be extended via open function chaining for user-defined types.

Function	Signature	Description
`truthy`	`a : Num`	Boolean coercion. Falsy: `0`, `0.0`, `""`, `False`, `Nil`. Truthy: everything else. Used by `if`, guards, `not`, `&&`, `||`.
`toString`	`a : Str`	String conversion. Handles `True`, `False`, `Nil` symbolically; delegates to `_toString` for primitives (int, float, string).
`eq`	`a : a : Num`	Equality. Default falls through to structural `==`. Used by `contains`.

Core Functions

Function	Signature	Description
`id`	`a : a`	Identity function.
`const`	`a : b : a`	Returns first argument, ignores second.
`flip`	`(a : b : c) : b : a : c`	Flips the first two arguments of a function.
`not`	`a : Num`	Logical negation via `truthy`.

List Functions

Function	Signature	Description
`null`	`List : Num`	Returns 1 if list is `Nil`, 0 otherwise.
`head`	`List : Maybe`	First element wrapped in `Maybe` (`Nothing` if empty).
`tail`	`List : Maybe`	Tail wrapped in `Maybe` (`Nothing` if empty).
`fold`	`(a : b : a) : a : List : a`	Left fold over a list.
`map`	`(a : b) : List : List`	Apply function to each element.
`filter`	`(a : Num) : List : List`	Keep elements where predicate is `truthy`.
`concat`	`List : List : List`	Concatenate two lists.
`push`	`List : a : List`	Append element to end of list.
`at`	`List : Num : Maybe`	Get element at index (zero-based); returns `Nothing` if out of bounds. `at'` takes index first.
`sum`	`List : Num`	Sum of numeric list.
`product`	`List : Num`	Product of numeric list.
`any`	`(a : Num) : List : Num`	1 if predicate is `truthy` for any element.
`all`	`(a : Num) : List : Num`	1 if predicate is `truthy` for all elements.
`contains`	`List : a : Num`	1 if list contains element (via `eq`).
`range`	`Num : Num : List`	Integer range `[start, end)`.
`zip`	`List : List : List`	Pair corresponding elements into 2-element lists.
`last`	`List : Maybe`	Last element wrapped in `Maybe` (`Nothing` if empty).
`init`	`List : Maybe`	All elements except the `last` wrapped in `Maybe` (`Nothing` if empty).
`reverse`	`List : List`	Reverse a list.
`take`	`Num : List : List`	First `n` elements.
`drop`	`Num : List : List`	Drop first `n` elements.
`enumerate`	`List : List`	Pair each element with its index: `[[0, a], [1, b], ...]`.
`join`	`Str : List : Str`	Join string list with separator.

Numeric Functions

Function	Signature	Description
`abs`	`Num : Num`	Absolute value.
`neg`	`Num : Num`	Negation (`0 - x`).
`min`	`Num : Num : Num`	Minimum of two numbers.
`max`	`Num : Num : Num`	Maximum of two numbers.

String Builtins

String operations provided by the C runtime:

Function	Signature	Description
`len`	`a : Num`	Length of a string or list; returns `0` for non-iterable values.
`strlen`	`Str : Num`	Length of a string (alias for `len`).
`charAt`	`Str : Num : Maybe`	Character at index; returns `Just` a single-char string when index is valid, or `Nothing` when out of range.
`indexOf`	`Str : Str : Num`	Index of first occurrence of substring (`-1` if not found).
`slice`	`Str : Num : Num : Str`	Substring from start index to end index.
`split`	`Str : Str : List`	Split string by delimiter.
`trim`	`Str : Str`	Remove leading/trailing whitespace.
`toUpper`	`Str : Str`	Convert to uppercase.
`toLower`	`Str : Str`	Convert to lowercase.
`replace`	`Str : Str : Str : Str`	Replace all occurrences: `replace old new str`.

Type Conversion Builtins

Function	Signature	Description
`toString`	`a : Str`	Convert to string (extensible — see above).
`toInt`	`a : Maybe`	Convert to integer; returns `Just` on success (parsing or conversion), `Nothing` on failure.
`toFloat`	`a : Maybe`	Convert to float; returns `Just` on success, `Nothing` on failure.

Record Introspection Builtins

Functions for inspecting and modifying record structure at runtime:

Function	Signature	Description
`tag`	`Record : Str`	Constructor tag name (e.g., `tag (Just 1)` -> `"Just"`).
`fields`	`Record : List`	List of field values; returns `[]` for non-record values.
`fieldNames`	`Record : List`	List of field names; returns `[]` for non-record values.
`getField`	`Record : Str : Maybe`	Dynamic field access by name; returns `Just value` if present, `Nothing` otherwise.
`setField`	`Record : Str : a : Record`	Return copy with field updated; on non-record values returns the original value unchanged.

Operators

Operator	Signature	Description
`|>`	`a : (a : b) : b`	Pipe forward: `x |> f = f x`.
`>>`	`(a : b) : (b : c) : a : c`	Forward composition.
`<<`	`(b : c) : (a : b) : a : c`	Backward composition.
`<-`	`Record : Record : Record`	Record merge: `base <- overlay`.
`&&`	`a : a : Num`	Short-circuit logical AND (via `truthy`).
`||`	`a : a : Num`	Short-circuit logical OR (via `truthy`).
`:`	`a : List : List`	Cons (prepend element to list).
`+ - * / % **`	`Num : Num : Num`	Arithmetic (+ also concatenates strings; ** takes an integer exponent).
`== /= < > <= >=`	`a : a : Num`	Comparison (structural equality for records).

Maybe examples

```
-- Maybe-returning stdlib usage
p1 = toInt "123"
p2 = toInt "abc"
p3 = toFloat "3.14"
r = {a = 1}

show mi = mi -> Just {val} = toString val; Nothing = "Nothing"

main world =
  world.io.println (show p1)
  world.io.println (show p2)
  world.io.println (toString p3)
  world.io.println (show (getField r "a"))
  world.io.println (show (getField r "b"))
```

```
123
Nothing
Just(3.14)
1
Nothing
```

Part III

Annotation Domains

Chapter 16

Type Annotations (: :)

Type annotations in milang are optional — the compiler infers types. When you do annotate, you use the `::` domain to attach a type expression to a binding. Annotations are separate lines that merge with the corresponding value binding.

Syntax

```
name :: typeExpr
name args = body
```

Inside a type expression, `:` means "function type" and is right-associative. So `Num : Num : Num` means "a function that takes a `Num`, then a `Num`, and returns a `Num`."

Primitive Types

Type	Description
`Num`	Alias for `Int` (backward compatibility)
`Int`	Signed 64-bit integer (alias for `Int' 64`)
`UInt`	Unsigned 64-bit integer (alias for `UInt' 64`)
`Float`	64-bit floating-point (alias for `Float' 64`)
`Byte`	Unsigned 8-bit integer (alias for `UInt' 8`)
`Str`	String
`List`	Linked list (`Cons`/`Nil`)

Sized Numeric Types

The primitive type constructors `Int'`, `UInt'`, and `Float'` take a compile-time bit width:

```
add8 :: Int' 8 : Int' 8 : Int' 8
add8 a b = a + b

compact :: Float' 32 : Float' 32
compact x = x * 1.0
```

The prelude defines convenient aliases:

```
Int = Int' 64
UInt = UInt' 64
Float = Float' 64
Byte = UInt' 8
```

You can define your own aliases:

```
Short = Int' 16
Word = UInt' 32
```

Details on Sized Numeric Types

The primitive constructors `Int'`, `UInt'`, and `Float'` take a compile-time bit-width argument and provide fixed-width numeric types. The language treats these as distinct primitive types rather than mere annotations:

- Signed integers (`Int' n`) use two's-complement semantics; arithmetic on signed integers is performed modulo 2^n and results are interpreted in two's complement when read as signed values. Overflow wraps around (modular arithmetic).
- Unsigned integers (`UInt' n`) are arithmetic modulo 2^n with values in the range [0, 2^n-1]. Mixing signed and unsigned operands follows a conservative promotion model: the operands are first promoted to the wider bit-width and if any operand is unsigned the operation is performed in the unsigned domain of that width.
- Floating-point types (`Float' 32`, `Float' 64`) correspond to standard IEEE-like single- and double-precision floats. Arithmetic on mixed-width floats promotes to the wider precision before performing the operation.

Promotion and Result Width

- For integer arithmetic, the result width is the maximum of the operand widths after promotion; the resulting value is wrapped/clamped to that width as described above.
- For mixed signed/unsigned arithmetic the operation is performed in the unsigned interpretation of the promoted width.

Compile-time Requirements and Partial Evaluation

- The bit-width argument (the `n` in `Int' n`) must be a compile-time constant. The reducer treats sized-type aliases (for example `Int = Int' 64`) as syntactic sugar and reduces type aliases away.
- Note: currently the compiler treats sized types primarily as type-level annotations and for FFI/representation purposes. Constant arithmetic is evaluated by the reducer using Milang's unbounded numeric semantics (or the platform default) and is not automatically wrapped/clamped to a target bit width. If exact width-limited arithmetic is required, use explicit conversion primitives or perform the operation in C via the FFI.

Practical Notes

- Use sized types when you need explicit control over representation and ABI compatibility (FFI interop, binary formats, embedded targets).
- The prelude exposes convenient aliases (`Int`, `UInt`, `Float`, `Byte`) for common widths; you can define your own aliases like `Short = Int' 16`.

Basic Examples

```
double :: Num : Num
double x = x * 2

add :: Num : Num : Num
add a b = a + b

greeting :: Str : Str
greeting name = "Hello, " + name

result = add (double 3) 4
message = greeting "milang"
```

```
double = <closure>
add = <closure>
greeting = <closure>
result = 10
message = Hello, milang
```

Record Types

Record types describe the shape of a record — field names and their types:

```
Point :: {x = Num; y = Num}
```

You can use a named record type in function signatures:

```
Point :: {x = Num; y = Num}

mkPoint :: Num : Num : Point
mkPoint x y = {x = x; y = y}

p = mkPoint 3 4
```

```
mkPoint = <closure>
p =  {x = 3, y = 4}
```

Polymorphism (Type Variables)

Any unbound identifier in a type expression is automatically a type variable. There is no `forall` keyword — just use lowercase names:

```
apply :: (a : b) : a : b
apply f x = f x
```

Here `a` and `b` are type variables. `apply` works for any function type `a : b` applied to an `a`, producing a `b`.

```
apply :: (a : b) : a : b
apply f x = f x

double x = x * 2
result = apply double 21
```

```
apply = <closure>
double = <closure>
```

```
result = 42
```

ADT Types

You can annotate functions that operate on algebraic data types:

```
Shape = {Circle radius; Rect width height}

area :: Shape : Num
area s = s ->
  Circle = 3 * s.radius * s.radius
  Rect = s.width * s.height

a = area (Circle 5)
b = area (Rect 3 4)
```

```
Shape = _module_ {Circle = <closure>, Rect = <closure>}
Circle = <closure>
Rect = <closure>
area = <closure>
a = 75
b = 12
```

The Dual Meaning of `:`

The `:` symbol is overloaded depending on context:

- **Value domain:** cons operator — `1 : [2, 3]` builds a list
- **Type domain:** function arrow — `Num : Num : Num` describes a function

This works because `::` on its own line clearly marks the boundary between value code and type code. There is never ambiguity.

Type Checking Behavior

The type checker is bidirectional: it pushes `::` annotations downward and infers types bottom-up. Type errors are reported as errors. Checking is structural — records match by shape (field names and types), not by name. Any record with the right `fields` satisfies a record type.

Chapter 17

Traits & Effects (`:~`)

The `:~` annotation domain attaches trait or effect information to a binding. It describes *what capabilities* a function uses — console IO, file reads, process execution, and so on. Traits annotations are orthogonal to type annotations (`::`) and can coexist on the same binding.

Syntax

```
name :~ traitsExpr
```

The traits expression is typically a list of effect names:

```
greet :~ [console]
greet world = world.io.println "hello"
```

Effect Names

Effect	Capabilities covered
`console`	`println`, `print`, `readLine`
`fs.read`	`readFile`, `exists`
`fs.write`	`writeFile`, `appendFile`, `remove`
`exec`	`process.exec`
`env`	`getEnv`

Use `[]` (empty list) or define a name bound to `[]` to declare a function as pure:

```
pure :~ []

add :~ pure
add a b = a + b
```

Defining Effect Groups

You can define reusable groups of effects:

```
readonly :~ [console, fs.read]
readwrite :~ [console, fs.read, fs.write]
```

Then reference those groups in other annotations.

Example

```
distance :~ []
distance x1 y1 x2 y2 = (x2 - x1)**2 + (y2 - y1)**2

main world =
  world.io.println (distance 0 0 3 4)
```

```
25
```

Combining with Other Domains

All annotation domains can coexist on a single binding:

```
add :? "Add two numbers"
add :: Num : Num : Num
add :~ []
add a b = a + b
```

Current Status

Trait annotations are parsed, stored, and **enforced** by the compiler. The compiler performs taint analysis: it tracks the `world` value and any names that transitively reference it (via aliasing or closures), then infers the effect set of every binding. If a function's inferred effects are not a subset of its declared traits, the compiler emits an error.

Functions without a `:~` annotation are assumed pure (`:~ []`). This means any function that performs IO must declare its effects. The only exception is `main`, which is implicitly granted all capabilities.

For example, declaring `:~ []` (pure) but calling `world.io.println` inside the body is a compile error — and so is omitting the annotation entirely:

```
-- This is an error: no annotation, so assumed pure, but uses console
helper world = world.io.println "oops"

-- Fix: add trait annotation
helper :~ [console]
helper world = world.io.println "ok"
```

Chapter 18

Documentation (`:?`)

The `:?` annotation domain attaches documentation to a binding. Doc expressions are ordinary milang expressions — usually strings or structured records — that the compiler stores as compile-time metadata. They do not affect runtime behavior.

Simple String Docs

The most common form is a short description string:

```
add :? "Add two numbers"
add a b = a + b
```

Structured Docs

For richer documentation, use a record with `fields` like `summary`, `params`, and `returns`:

```
distance :? {
  summary = "Squared distance between two points"
  params = {
    x1 = "First x coordinate"
    y1 = "First y coordinate"
    x2 = "Second x coordinate"
    y2 = "Second y coordinate"
  }
  returns = "The squared distance as a number"
}
distance x1 y1 x2 y2 = (x2 - x1)**2 + (y2 - y1)**2
```

The field names are not enforced — you can use whatever structure makes sense for your project.

Triple-Quoted String Docs

For multi-line documentation, use triple-quoted strings. Margin stripping (based on the closing `"""` indentation) keeps the source tidy:

```
greet :? """
  Greet a person by name.
  Prints a friendly message to the console.
```

```
  """
greet world name = world.io.println ("Hello, " + name + "!")
```

Example

```
add :? "Add two numbers"
add :: Num : Num : Num
add a b = a + b

distance :? {summary = "Squared distance"; returns = "Num"}
distance x1 y1 x2 y2 = (x2 - x1)**2 + (y2 - y1)**2

main world =
  world.io.println (add 3 4)
  world.io.println (distance 0 0 3 4)
```

```
7
25
```

Doc annotations do not change execution — the output above is the same with or without `:?` lines.

Combining All Five Domains

Every annotation domain can coexist on a single binding:

```
distance :? "Squared Euclidean distance"
distance :: Num : Num : Num : Num : Num
distance :~ []
distance x1 y1 x2 y2 = (x2 - x1)**2 + (y2 - y1)**2
```

The domains are `=` (value), `::` (type), `:~` (traits), `:?` (docs), and `:!` (parse). They are independent and can appear in any order before the value binding.

Future: `milang doc`

A planned `milang doc` command will extract `:?` annotations from source files and generate reference documentation automatically.

Chapter 19

Parse Declarations (`:!`)

The `:!` annotation domain declares how the parser should handle a user-defined operator — specifically its precedence and associativity. The parser pre-scans the source for `:!` declarations before parsing expressions, so they take effect immediately.

Syntax

```
(op) :! {prec = N; assoc = Left}
```

- `prec` — an integer precedence level. Higher values bind more tightly.
- `assoc` — one of `Left`, `Right`, or `None`. Determines how chains of the same operator group:
  - `Left`: `a op b op c` parses as `(a op b) op c`
  - `Right`: `a op b op c` parses as `a op (b op c)`
  - `None`: chaining is a parse error; explicit parentheses are required.

Example

```
(<=>) :! {prec = 30; assoc = Left}
(<=>) a b = if (a == b) 0 (if (a > b) 1 (0 - 1))

cmp1 = 5 <=> 3
cmp2 = 3 <=> 3
cmp3 = 1 <=> 5
```

```
<=> = <closure>
cmp1 = 1
cmp2 = 0
cmp3 = -1
```

Built-in Operator Precedences

For reference, the approximate precedence levels of built-in operators:

Precedence	Operators	Associativity
1	`|>` `>>` `<<`	Left
2	`||`	Left

Precedence	Operators	Associativity
3	&&	Left
4	== /= < > <= >=	Left
5	:	Right
6	+ - ++	Left
7	* / %	Left
8	**	Right

User-defined operators without a `:!` declaration receive a default precedence. Define `:!` to override this and integrate your operator naturally with built-in ones.

Metaprogramming Hook

Because `:!` declarations are processed during parsing (before evaluation), they serve as a metaprogramming hook — they let you reshape how the parser reads subsequent expressions. Combined with user-defined operators, this gives you control over the syntactic structure of your programs.

Part IV

Advanced Topics

Chapter 20

Open Function Chaining

Milang supports **open function chaining** — when you redefine a function that uses pattern matching (the `->` arrow), your new alternatives are automatically prepended to the existing definition. The previous definition becomes the fallback for values that don't match your new patterns.

This is milang's answer to typeclasses: no new syntax, no special declarations. `Just` define the same function again with new patterns.

How It Works

When a binding is redefined in the same scope and the new definition uses pattern matching (`->`) **without a catch-all wildcard** (`_`), the compiler chains the two definitions together. The new alternatives are tried first; if none match, the old definition handles the value.

If the new definition **includes** a catch-all wildcard, it fully replaces the old definition — the catch-all means "I handle everything."

```
-- Base: has catch-all
describe val = val -> _ = "unknown"

-- Extension: no catch-all — chains with base
describe val = val -> Circle = "a circle"; Rect = "a rectangle"
```

Now `describe (Circle 5)` returns `"a circle"` and `describe 42` falls through to the base, returning `"unknown"`.

Extensible Builtins

Three core prelude functions are designed to be extended this way:

`truthy`

`truthy` is the universal boolean coercion point. It is called internally by `if`, guards, `not`, `&&`, and `||`. The prelude default treats `0`, `0.0`, `""`, `False`, `Nil`, and `Nothing` as falsy (returns 0); everything else is `truthy` (returns 1).

Extend `truthy` to teach the language how your types behave in boolean contexts:

```
Result = {Err msg; Ok val}
truthy val = val -> Err = 0; Ok = 1

main world =
  world.io.println (toString (if (Ok 42) "yes" "no"))
  world.io.println (toString (if (Err "oops") "yes" "no"))
  world.io.println (toString (not (Err "fail")))
```

```
yes
no
1
```

toString

`toString` converts values to their string representation. The prelude handles `True`, `False`, `Nil`, `Nothing`, and `Just` symbolically, then falls through to `_toString` (the C-level primitive) for ints, floats, and strings. Extend it for your own types:

```
Pair = {Pair fst snd}
toString val = val -> Pair = "(" + toString val.fst + ", " + toString val.snd +
 ↪  ")"

main world =
  world.io.println (toString (Pair 1 2))
  world.io.println (toString (Pair "hello" True))
```

```
(1, 2)
(hello, True)
```

eq

`eq` is the extensible equality function. The prelude default falls through to structural `==`. The `contains` function uses `eq`, so extending `eq` automatically affects list membership checks:

```
Card = {Card rank suit}
eq a b = a -> Card = a.rank == b.rank

main world =
  world.io.println (toString (eq (Card 10 "H") (Card 10 "S")))
  world.io.println (toString (contains [Card 10 "H", Card 5 "D"] (Card 10
 ↪  "S")))
```

```
1
0
```

Scope Chaining

Open chaining works across scopes. A redefinition inside a function body (a `With` block) chains with the outer scope's definition, not just same-scope duplicates. Multiple levels of chaining compose naturally:

```
Result = {Err msg; Ok val}
truthy val = val -> Err = 0; Ok = 1

main world =
  Severity = {Low; High}
```

```
  truthy val = val -> Low = 0; High = 1
  -- truthy now handles Result, Severity, AND all prelude types
  world.io.println (toString (truthy (Ok 1)))
  world.io.println (toString (truthy (Err "x")))
  world.io.println (toString (truthy High))
  world.io.println (toString (truthy Low))
  world.io.println (toString (truthy Nothing))
```

```
1
0
1
0
0
```

Writing Extensible Functions

To make your own functions extensible, follow this pattern:

1. **Define a base** with a catch-all wildcard — this provides default behavior.
2. **Extend without a catch-all** — your new alternatives are prepended; the base stays as fallback.

```
-- Base definition (has catch-all)
describe val = val -> _ = "something"

-- Extension (no catch-all — chains)
Shape = {Circle radius; Rect width height}
describe val = val -> Circle = "a circle"; Rect = "a rectangle"

main world =
  world.io.println (describe (Circle 5))
  world.io.println (describe (Rect 3 4))
  world.io.println (describe 42)
```

```
a circle
a rectangle
something
```

If you include a catch-all in an extension, it fully replaces the base — use this when you genuinely want to override all behavior.

Chapter 21

Partial Evaluation

Partial evaluation is milang's core compilation model. There is no separate optimisation pass — the compiler itself evaluates every expression whose inputs are known at compile time and emits C code only for what remains. The result is that high-level abstractions (helper functions, configuration records, computed constants) often carry **zero runtime cost**.

How it works

When the compiler processes a binding it walks the expression tree with a recursive reducer (`reduceD`). At each node it checks whether the operands are *concrete* — literal integers, floats, strings, lambdas, or records whose `fields` are themselves concrete. If they are, the expression is evaluated immediately and replaced by its result. If any operand is unknown (a function parameter, an IO result, etc.) the expression is left as *residual code* for the C back-end to emit.

```
-- Fully reduced at compile time:
x = 6 * 7              -- becomes: x = 42
f a = a * 2
y = f 21                -- becomes: y = 42

-- Stays as residual code (parameter unknown):
double a = a * 2        -- emitted as a C function
```

SCC dependency analysis

Bindings are sorted into *strongly connected components* so that each group is reduced in dependency order. Mutually-recursive bindings land in the same SCC and are handled together.

Depth-limited recursion

Recursive functions are unrolled only when every argument is concrete, and reduction is capped at a fixed depth (128 steps). This prevents the compiler from looping on unbounded recursion while still collapsing finite recursive computations at compile time.

Zero-cost abstractions

Because the reducer runs before code generation, any abstraction that is fully known at compile time disappears entirely from the output:

```
-- Configuration record — reduced away at compile time
config = {width = 800; height = 600}
pixels = config.width * config.height
```

```
config =  {width = 800, height = 600}
pixels = 480000
```

The binding `pixels` is reduced to the integer 480000 before any C code is generated. No record allocation, no field lookup — just a constant.

Inspecting the reducer output

milang ships two commands for inspecting what the compiler sees:

```
milang dump file.mi      -- parsed AST (before reduction)
milang reduce file.mi    -- AST after partial evaluation (what codegen sees)
```

Comparing the two on the same file shows exactly which expressions were collapsed and which remain as residual code. This is the primary tool for understanding compile-time behaviour.

What stays as residual code

Anything that depends on a value unknown at compile time is left for the C back-end:

```
main world =
  line = world.io.readLine    -- runtime IO — cannot reduce
  world.io.println line       -- emitted as C call
```

Function parameters, IO results, and any expression transitively depending on them are residual. Everything else is reduced.

Chapter 22

Thunks & Laziness

Milang is **eager by default** — every expression is evaluated as soon as it is bound. The tilde operator `~` lets you opt into delayed evaluation where you need it.

Creating thunks with `~`

Prefixing an expression with `~` wraps it in a *thunk*: a suspended computation that is not executed until its value is actually needed.

```
eager = 1 + 2       -- evaluated immediately
delayed = ~(1 + 2)  -- wrapped in a thunk; not yet evaluated
result = delayed     -- forced here: evaluates to 3
```

```
eager = 3
delayed = 3
result = 3
```

When a thunk is used in a context that needs its value (passed to an operator, printed, pattern-matched, etc.) it is *forced* automatically — you never call a thunk explicitly.

`if` and auto-quote parameters

In earlier versions of milang, `if` required explicit thunks on both branches to prevent eager evaluation:

```
-- Old style (still works, but no longer necessary):
result = if (x > 5) (x * 2) (x * 3)
```

The `if` conditional quotes its branches implicitly; write conditionals like this:

```
x = 10
result = if (x > 5) (x * 2) (x * 3)
```

```
x = 10
result = 20
```

Both styles work — if you pass a thunk to an auto-quote parameter, the thunk is forced after splicing. See the Metaprogramming chapter for details on #-params.

Nested conditionals

Conditionals compose naturally:

```
z = 7
result = if (z > 10) 100 (if (z > 5) 50 0)
```

```
z = 7
result = 50
```

Each inner `if` is only evaluated when its enclosing branch is selected.

Lazy bindings with `:=`

The `:=` operator creates a *lazy binding* — syntactic sugar for a thunk that caches its result after the first force:

```
x = 3
y := x + 10   -- not evaluated until y is used
z = y * 2     -- forces y here; y becomes 13, z becomes 26
```

```
x = 3
y = <closure>
z = 26
```

Lazy bindings are useful for expensive computations that may never be needed, or for establishing declaration-order dependencies without paying upfront cost.

When to use thunks

Situation	Mechanism
Conditional branches (`if`)	Auto-quoted branch parameters handle this
Short-circuit logic (`&&`, `||`)	Auto-quote params handle the lazy operand
Deferred expensive work	Lazy binding `:=`
Controlling IO ordering	Thunks delay side effects until forced

The general rule: reach for ~ whenever you need to **control when** an expression is evaluated rather than relying on milang's default left-to-right eager order.

Chapter 23

Metaprogramming

Milang provides two complementary operators for working with syntax at compile time: **quote** (`#`) captures an expression as a data structure, and **splice** (`$`) evaluates a data structure back into code. Combined with partial evaluation these give you compile-time code generation without a separate macro system.

Quote: `#expr`

The `#` operator captures the *abstract syntax tree* of its operand as a tagged record. The expression is **not** evaluated — only its structure is recorded.

```
q_int = #42
q_op  = #(1 + 2)
```

```
q_int = Int {val = 42}
q_op = Op {op = +, left = Int {val = 1}, right = Int {val = 2}}
```

Each syntactic form maps to a specific record `tag`:

Syntax	Quoted form
`#42`	`Int {val = 42}`
`#"hello"`	`Str {val = "hello"}`
`#x`	`Var {name = "x"}`
`#(f x)`	`App {fn = Var {name = "f"}; arg = Var {name = "x"}}`
`#(a + b)`	`Op {op = "+"; left = ...; right = ...}`
`#(\x -> x)`	`Fn {param = "x"; body = ...}`

Because quoted ASTs are ordinary records you can inspect their `fields`, pass them to functions, and build new ASTs by constructing records directly.

Splice: `$expr`

The `$` operator takes a record that represents an AST node and evaluates it as code:

```
ast = #(1 + 2)
result = $ast
```

```
ast = Op {op = +, left = Int {val = 1}, right = Int {val = 2}}
result = 3
```

Splicing a quoted literal round-trips back to its value. More usefully, you can build AST records by hand and splice them:

```
ast = Op {op = "*"; left = Int {val = 6}; right = Int {val = 7}}
answer = $ast
```

```
ast = Op {op = *, left = Int {val = 6}, right = Int {val = 7}}
answer = 42
```

Writing macros

A macro in milang is just a function that takes and returns AST records. Because the partial evaluator runs at compile time, macro expansion happens before code generation — there is no runtime cost.

```
-- Macro: double an expression (x + x)
double_ast expr = Op {op = "+"; left = expr; right = expr}
r1 = $(double_ast #5)

-- Macro: negate (0 - x)
negate_ast expr = Op {op = "-"; left = Int {val = 0}; right = expr}
r2 = $(negate_ast #42)
```

```
double_ast = <closure>
r1 = 10
negate_ast = <closure>
r2 = -42
```

Macros compose — you can pass one macro's output as another's input:

```
double_ast expr = Op {op = "+"; left = expr; right = expr}
negate_ast expr = Op {op = "-"; left = Int {val = 0}; right = expr}
r = $(double_ast (negate_ast #7))
```

```
double_ast = <closure>
negate_ast = <closure>
r = -14
```

Pattern matching on ASTs

Because quoted expressions are records, you can pattern-match on them to transform code structurally:

```
-- Swap the arguments of a binary operator
swap_op ast = ast ->
  Op {op = op; left = l; right = r} = Op {op = op; left = r; right = l}
  _ = ast
```

Auto-Quote Parameters (`#param`)

When defining a function (or binding), prefixing a parameter name with # tells the compiler to **automatically quote** the corresponding argument at the call site. Inside the body, `$param` splices the captured AST back into code and evaluates it. The caller writes ordinary expressions — no sigils needed.

```
-- 'if' is defined in the prelude using auto-quote params:
--   if cond #t #e = (truthy cond) -> 0 = $e; _ = $t
-- The caller just writes:
x = 10
result = if (x > 5) (x * 2) (x * 3)
```

```
x = 10
result = 20
```

Because the `#t` and `#e` parameters auto-quote their arguments, neither branch is evaluated until the matching `$t` or `$e` splice runs. This is how milang achieves lazy branching without keywords or special forms — `if` is an ordinary user-defined function.

Auto-quote parameters work with any function, not just `if`:

```
-- A logging wrapper that only evaluates its message when enabled
log_if enabled #msg world = if enabled (world.io.println $msg) 0
```

Note that `world` must be threaded through explicitly: `world.io.println` requires the `world` value to be in scope, so any function calling IO must accept it as a parameter.

How it works

1. The function definition declares `#param` — the `#` is part of the parameter name in the source but is stripped for binding purposes.
2. At each call site, the compiler wraps the corresponding argument in `#(...)`, producing a quoted AST record.
3. In the body, `$param` splices the record back into an expression and evaluates it in the current environment.
4. If the spliced expression contains a thunk (`~expr`), the thunk is automatically forced after splicing — so old-style `~` code remains backward-compatible.

Relation to thunks

Auto-quote parameters are strictly more general than thunks (`~`). A thunk delays evaluation of a single expression; a quoted parameter captures the full AST, which can be inspected, transformed, or conditionally evaluated. For simple conditional laziness (like `if`), the effect is the same — but auto-quote opens the door to user-defined control flow, macros that inspect their arguments, and other metaprogramming patterns.

Inspection commands

Two CLI commands help you understand what the compiler sees:

```
milang dump file.mi      -- show the parsed AST (before reduction)
milang reduce file.mi    -- show the AST after partial evaluation
```

Use `dump` to verify that quoting produces the record structure you expect, and `reduce` to confirm that your macros expand correctly at compile time.

Chapter 24

User-Defined Operators

In milang operators are ordinary functions whose names are made of operator characters (`+ - * / ^ < > = ! & | @ % ? :`). You define, use, and pass them around exactly like any other function.

Defining an operator

Wrap the operator name in parentheses and define it as a normal function:

```
(<=>) a b = if (a == b) 0 (if (a > b) 1 (0 - 1))
cmp1 = 5 <=> 3
cmp2 = 3 <=> 3
cmp3 = 1 <=> 5
```

```
<=> = <closure>
cmp1 = 1
cmp2 = 0
cmp3 = -1
```

The definition `(<=>) a b = ...` creates a two-argument function. You then use it infix without parentheses: `5 <=> 3`.

Setting precedence and associativity

By default a user-defined operator gets a low precedence. Use a **parse declaration** (`:!`) to set the precedence level and associativity. The declaration must appear before the operator's first infix use:

```
(<+>) :! {prec = 6; assoc = Left}
(<+>) a b = {x = a.x + b.x; y = a.y + b.y}

result = {x=1;y=2} <+> {x=3;y=4}
```

- `prec` — an integer; higher binds tighter (e.g. `*` is 7, `+` is 6).
- `assoc` — `Left` or `Right`; controls grouping of chained uses.

Operators as first-class values

Wrapping a built-in or user-defined operator in parentheses gives you a function value you can pass to higher-order functions:

```
add = (+)
result = add 3 4
```

```
add = <closure>
result = 7
```

This is especially useful with folds and maps:

```
total = fold (+) 0 [1, 2, 3, 4, 5]
```

Functions as infix operators

The backtick syntax lets you use any two-argument function in infix position:

```
div a b = a / b
result = 10 `div` 2
```

```
div = <closure>
result = 5
```

`a `f` b` is equivalent to `f a b`. This works with any function, not just operator-named ones.

Prefix vs. infix

Every operator can be used both ways:

```
-- Infix (the usual way)
r1 = 5 <=> 3

-- Prefix (wrap in parens)
r2 = (<=>) 5 3
```

Both forms are interchangeable. Prefix is handy when you want to partially apply an operator or pass it as an argument.

Chapter 25

Security & Capabilities

Milang's security model is **structural**: if a function does not receive a capability, it physically cannot use it. There is no global mutable state, no ambient authority, and no `unsafePerformIO` escape hatch.

Capability-based IO

All side effects flow through the `world` record that the runtime passes to `main`. The record contains sub-records — `world.io`, `world.process`, etc. — each granting access to a specific class of operations.

You restrict a function's power by passing only the sub-record it needs:

```
-- greet can print but cannot access the filesystem or exec processes
greet io = io.println "hello"

main world = greet world.io
```

Because `greet` receives `world.io` and nothing else, it is structurally impossible for it to read files, spawn processes, or access environment variables. This is milang's equivalent of the principle of least privilege — enforced by the language, not by convention.

Remote import pinning

Milang supports importing modules by URL. To prevent supply-chain attacks every remote import must be *pinned* to a SHA-256 content hash:

```
milang pin file.mi
```

This command scans `file.mi` for URL imports, fetches each one, computes its hash, and records the result. On subsequent compilations the compiler verifies that the fetched content matches the pinned hash and refuses to proceed if it does not.

C FFI security

Milang can call C functions via its FFI. Native C code operates outside the capability model, so FFI is the one place where the structural guarantee can be bypassed. Two CLI flags are described to let you lock this down (note: these flags are not currently implemented in the core compiler):

- `--no-ffi` — disallow all C FFI imports. The program may only use pure milang and the built-in `world` capabilities.
- `--no-remote-ffi` — allow C FFI in local `.mi` files but forbid it in any module imported by URL (and any module transitively imported from that URL module). This lets you trust your own native code while sandboxing third-party libraries.

Trust is transitive: if your local file imports a remote module `A`, and `A` imports a relative module `B`, then `B` is also in the remote trust zone and subject to `--no-remote-ffi`.

Structural security summary

Layer	Mechanism
Function-level isolation	Pass minimal `world` sub-records
Supply-chain integrity	`milang pin` + SHA-256 verification
Native code gating	`--no-ffi`, `--no-remote-ffi`
Purity tracking	Compiler tracks `world`-tainted expressions; pure code cannot perform IO

The design principle is simple: the only way to perform a side effect is to hold the right capability, and capabilities can only travel through explicit function arguments. The FFI gating flags close the one remaining loophole by controlling access to native C code.

Part V

Tools

Chapter 26

Tooling & Build Notes

This page collects practical tips for building Milang programs and the documentation.

Running examples and the CLI

- Use the local `./milang` binary in the repository root for running and experimenting with `.mi` files; if you installed `milang` on your PATH you can omit `./`.
- Useful commands:
  - `./milang run file.mi` — compile and run
  - `./milang compile file.mi output.c` — emit standalone C
  - `./milang dump file.mi` — show parsed AST (before reduction)
  - `./milang reduce file.mi` — show partially-evaluated AST (what the codegen sees)
  - `./milang repl` — interactive REPL

C toolchain

Milang emits C and requires a working C toolchain (gcc or clang). On Debian/Ubuntu:

```
sudo apt-get install build-essential pkg-config
```

Building the docs (mdBook)

The repo includes an mdbook-style source under `docs/src` and a small preprocessor `docs/mdbook-milang.py` that executes code blocks tagged `milang,run` and appends their output. Two ways to build:

- If the project has a `Makefile`, run `make docs` (required if available).
- Or, install mdBook and run:

```
mdbook build docs/src -d docs/out
```

Ensure your mdBook configuration registers the `mdbook-milang.py` preprocessor, or run the script manually when verifying examples.

Common issues & debugging

- Parsing ambiguity with inline record literals: when passing a record literal directly as an argument, parenthesize it or bind it to a name, e.g. `getField ({a = 1}) "a"` or `r = {a = 1}` `getField r "a"`.
- `toInt` / `toFloat` return `Nothing` on parse failure — check results with pattern matching on `Just` / `Nothing`.
- `charAt` and `getField` return `Nothing` when out-of-bounds or missing.
- Division/modulo by zero is handled at the runtime/C level (implementation-defined); avoid relying on undefined behaviour in portable code.
- Use `milang dump` to inspect how the parser grouped expressions if you hit unexpected parse errors.

Quick checklist

- `./milang dump` to verify parser grouping
- `./milang reduce` to verify partial evaluation
- `mdbook build` (or `make docs`) to render the site

Chapter 27

REPL

The milang REPL (Read-Eval-Print Loop) lets you evaluate expressions and define bindings interactively.

Starting the REPL

```
./milang repl
```

You'll see a λ> prompt. Type any expression and press Enter to evaluate it:

```
λ> 2 + 3
5
λ> "hello" + " " + "world"
"hello world"
```

Type `:q` or press **Ctrl-D** to exit.

Defining Bindings

Bindings persist across inputs, so you can build up definitions incrementally:

```
λ> double x = x * 2
double = (\x -> (x * 2))
λ> double 21
42
λ> quadruple x = double (double x)
quadruple = (\x -> (double (double x)))
λ> quadruple 5
20
```

Single-line Input

The REPL reads one line at a time. Each binding must fit on a single line. Use semicolons to separate alternatives within a `->` pattern match:

```
λ> area s = s -> Circle = 3.14 * s.radius * s.radius; Rect = s.width * s.height
```

Multi-line indented definitions must be written in a `.mi` file and loaded via `milang run`.

Prelude Functions

All standard prelude functions are available immediately — no imports needed:

```
λ> map (\x = x * x) [1, 2, 3, 4, 5]
Cons {head = 1, tail = Cons {head = 4, tail = ...}}
λ> filter (\x = x > 3) [1, 2, 3, 4, 5]
Cons {head = 4, tail = Cons {head = 5, tail = Nil {}}}
λ> fold (\acc x = acc + x) 0 [1, 2, 3]
6
λ> len [10, 20, 30]
3
```

> **Note:** Lists are displayed as raw `Cons`/`Nil` record expressions — the REPL shows the partially-evaluated AST, not a pretty-printed representation.

Type Annotations

You can add type annotations to bindings:

```
λ> x :: Int
λ> x = 42
x = 42
```

The type is associated with the binding and checked when the value is defined.

Viewing Bindings

Use `:env` to show all user-defined bindings (prelude bindings are hidden):

```
λ> double x = x * 2
double = (\x -> (x * 2))
λ> :env
double = (\x -> (x * 2))
```

How It Works

Each REPL input is:

1. Parsed as either a binding (namespace) or a bare expression
2. Reduced using the same partial evaluator as `milang reduce`
3. The reduced form is printed

New bindings extend the accumulated environment for all subsequent inputs. This is a **pure partial evaluator** — there is no C compilation or gcc invocation in the REPL. Residuals (expressions that cannot be further reduced) are printed as-is.

Limitations

- **No IO** — the REPL evaluates pure expressions only. There is no `world` value available, so `world.io.println` and similar IO operations cannot be used.
- **No imports** — `import` declarations are not supported in the REPL.

- **No multi-line input** — each input must fit on a single line. Write multi-line programs in `.mi` files.
- **No command history** — the up/down arrow keys do not recall previous inputs.
- **Raw list output** — lists are printed as `Cons`/`Nil` record expressions, not `[1, 2, 3]`.

For IO and imports, write a `.mi` file and use `milang run` instead.

Chapter 28

Compiler Modes

The `milang` binary supports six commands. Each operates on a `.mi` source file and exercises different parts of the compilation pipeline.

`milang run`

```
./milang run file.mi
```

The most common command. It parses the file, resolves imports, partially evaluates, generates C, compiles with `gcc -O2`, runs the resulting binary, and cleans up temporary files.

If the file defines `main world = ...`, the program runs as a normal executable and `main`'s return value becomes the process exit code. If there is no `main` binding with a parameter, milang runs in **script mode** (see below).

Example

```
main world =
  world.io.println "running!"
```

```
running!
```

`milang compile`

```
./milang compile file.mi            # writes file.c
./milang compile file.mi output.c   # writes output.c
./milang compile file.mi -          # prints C to stdout
```

Emits a self-contained C source file. The milang runtime is embedded in the output, so the generated file has no external dependencies beyond the standard C library.

Compile it yourself:

```
gcc output.c -o program
./program
```

This is useful when you want to inspect the generated code, cross-compile, or integrate milang output into a larger C project.

milang dump

```
./milang dump file.mi
```

Prints the **parsed AST** before import resolution or partial evaluation. This shows you exactly how the parser interpreted your source, including all five annotation domains (`=`, `::`, `:~`, `:?`, `:!`).

Use this to debug syntax issues — if the parser grouped your expressions differently than you expected, dump will show it.

milang reduce

```
./milang reduce file.mi
```

Prints the AST **after partial evaluation**. This is what the code generator actually sees. Any expression that could be computed at compile time has been reduced to a literal value.

Use this to verify that the partial evaluator is doing what you expect:

```
square x = x * 2
answer = square 21
```

```
square = <closure>
answer = 42
```

Running `milang reduce` on this file would show `answer = 42` — the function call was evaluated at compile time.

If you need to see reduction without the prelude injected, use `milang raw-reduce file.mi` to view the reduced AST for the file alone.

milang pin

```
./milang pin file.mi
```

Finds all URL imports in the file, fetches them, computes a Merkle hash (SHA-256 of the content plus all transitive sub-imports), and rewrites the source file to include the hash:

Before:

```
utils = import "https://example.com/utils.mi"
```

After:

```
utils = import' "https://example.com/utils.mi" ({sha256 = "a1b2c3..."})
```

This ensures that the imported code hasn't changed since you `last` pinned it. If the content changes, the compiler will report a hash mismatch and refuse to proceed.

milang repl

```
./milang repl
```

Starts an interactive REPL where you can evaluate expressions and define bindings. See the REPL chapter for details.

Script Mode

When a `.mi` file has no `main` binding that takes a parameter, `milang run` operates in **script mode**: it evaluates every top-level binding and prints each name-value pair.

```
a = 2 + 3
b = a * a
greeting = "hello"
```

```
a = 5
b = 25
greeting = hello
```

Script mode is ideal for quick calculations and testing small snippets. Prelude definitions are automatically hidden from the output.

Security Flags

Milang supports flags to restrict what imported code can do (note: these flags are not currently implemented in the core compiler; see .github/ROADMAP.md):

Flag	Effect
`--no-ffi`	Disables all C FFI imports (no `.h` files can be imported)
`--no-remote-ffi`	Disallows C FFI for remote (URL) imports specifically

These flags are useful when running untrusted code. A URL import might try to `import "/usr/include/stdlib.h"` and call arbitrary C functions — `--no-remote-ffi` prevents this while still allowing your own local FFI usage.

Summary

Command	What it does
`run file.mi`	Compile and execute
`compile file.mi [out.c]`	Emit standalone C
`dump file.mi`	Show parsed AST
`reduce file.mi`	Show partially-evaluated AST
`pin file.mi`	Fetch URL imports, write SHA-256 hashes
`repl`	Interactive evaluation