MATA-LL User's Manual

A Haskell-Inspired Language Targeting Lua

MATA-LL Project

July 2026

Contents

1 Getting Started

1.1 What is MATA-LL?

MATA-LL (Modest Attempt at Typesystem Augmenting the Lua Language) is a statically-typed, non-strict functional language that compiles to standalone Lua files. If you know Haskell, you will feel at home immediately. The generated Lua requires no runtime library and runs on Lua 5.4 or LuaJIT.

1.2 Installation

Install from source with Cargo:

```
cargo install --path .
```

This builds an optimized binary and places it in `~/.cargo/bin/`, which is normally already on your `PATH`.

1.3 Hello World

Create `hello.mll`:

```
main :: IO ()
main = putStrLn "Hello, world!"
```

Compile and run:

```
mll hello.mll --run
```

Or emit a Lua file:

```
mll hello.mll --emit-lua      # writes hello.lua
lua5.4 hello.lua               # runs standalone
```

1.4 Command-Line Usage

Command	Effect
`mll file.mll`	Compile, write `file.lua`
`mll file.mll --run`	Compile and execute immediately (no `.lua` written)
`mll file.mll --emit-lua`	Write `file.lua` (also with `--run`)
`mll file.mll -L ./lib`	Add library search path
`mll file.mll --embed-source MODE`	Embed the source in the emitted Lua (`comments`, `var`, `none`)
`mll --recompile file.lua`	Recompile a `.lua` from its embedded source, rewriting it in place
`mll -v`	Print license, version, and git commit

1.5 The REPL

Start with `mll-repl`:

```
mll> x :: Int \
...> x = 42
```

```
mll> x * 2 + 1
85
mll> map (*2) [1, 2, 3]
[2, 4, 6]
```

REPL commands: `/clear`, `/decls`, `/lua EXPR`, `/source`, `/drop`, `/help`, `/quit`. Multi-line input: end a line with `\`.

2 Basic Syntax

2.1 Comments

```
-- Single-line comment
{- Block comment
   can span lines
   {- and nest -}
-}
```

2.2 Layout Rule

MATA-LL uses indentation-sensitive layout, like Haskell or Python. When a definition spans more than one line, each wrapped line must be indented past the definition's start column:

```
longExpression :: Int
longExpression = 1 + 2
    + 3 + 4    -- wrapped line: indented past 'l'
```

2.3 Type Signatures

All top-level definitions require explicit type signatures. Local bindings (`let`, `where`) are inferred automatically.

```
double :: Int -> Int
double x = x * 2
```

2.4 Identifiers

Names follow Haskell's rules: values and functions start with a lowercase letter, types and constructors with an uppercase letter. Because MATA-LL compiles to Lua, a name may happen to collide with a Lua reserved word (`until`, `repeat`, `local`, `nil`, `function`, `while`, …). This is fine — such names are escaped automatically in the generated code, so you can use them for variables, parameters, functions, and record fields:

```
until :: Int -> Bool
until n = n > 100
```

3 Types and Data

3.1 Primitive Types

Type	Lua type	Examples
`Int`	integer	`42`, `-7`, `0`
`Number`	float	`3.14`, `-0.5`
`String`	string	`"hello\n"`
`Bool`	boolean	`True`, `False`
`ByteString`	string	`bsFromString "hi"`

Note: There is no `Char` type. Strings are native Lua strings, not `[Char]`. Use `<>` for string concatenation (not `++`).

Note: `ByteString` is an intrinsic type with the same runtime representation as `String` (a Lua string) but a distinct type with explicit, encoding-agnostic byte semantics and 0-based indexing. It has no literal syntax; build one with `bsFromString` or `bsPack`, and `import ByteString` for its operations. Only strict `ByteString`s are supported.

3.2 Lists

Lists are cons-cell linked lists (not Lua arrays):

```
nums :: [Int]
nums = [1, 2, 3, 4, 5]

-- Equivalent to:
nums' :: [Int]
nums' = 1 : 2 : 3 : 4 : 5 : []
```

3.3 Tuples

```
pair :: (String, Int)
pair = ("age", 30)

triple :: (Bool, Int, String)
triple = (True, 42, "hello")
```

The tuple constructor is also available as a prefix function: `(,)` has type `a -> b -> (a, b)`, `(,,)` has type `a -> b -> c -> (a, b, c)`, and so on. Like any function it can be partially applied or passed higher-order:

```
-- pair every element with a common key
tagged = map ((,) "row") [1, 2, 3]
-- => [("row", 1), ("row", 2), ("row", 3)]

-- zip two lists into pairs
pairs = zipWith (,) [1, 2] [10, 20]
-- => [(1, 10), (2, 20)]
```

3.4 Algebraic Data Types

```
data Shape = Circle Number
           | Rectangle Number Number
           | Point

area :: Shape -> Number
area (Circle r)      = 3.14159 * r * r
area (Rectangle w h) = w * h
area Point           = 0.0
```

3.5 Records

Named fields with accessor functions:

```
data Person = Person { name :: String
                     , age  :: Int }

greet :: Person -> String
greet p = "Hello, " <> name p <> "!"

-- Record construction
alice :: Person
alice = Person { name = "Alice", age = 30 }

-- Record update (creates a copy)
olderAlice :: Person
olderAlice = alice { age = 31 }
```

The braces follow the ordinary layout rule, so they may open on a following line when that line is indented past the enclosing block — in construction and in update alike:

```
bob :: Person
bob = Person
    { name = "Bob"
    , age  = 44
    }

olderBob :: Person
olderBob = bob
    { age = 45 }
```

3.6 Newtypes

Zero-cost wrappers:

```
newtype Metres = Metres Number
newtype UserId = UserId Int
```

3.7 Type Aliases

```
type Pair a = (a, a)
type Name = String
```

3.8 Maybe and Either

```
safeDivide :: Int -> Int -> Maybe Int
safeDivide _ 0 = Nothing
safeDivide a b = Just (a `div` b)

parseAge :: String -> Either String Int
parseAge s = let n = read s
             in if n >= 0 then Right n
                else Left "negative age"
```

4 Functions

4.1 Definition and Application

```
add :: Int -> Int -> Int
add a b = a + b

-- Application (no parentheses needed)
result :: Int
result = add 3 4    -- 7

-- Partial application
inc :: Int -> Int
inc = add 1
```

4.2 Lambda Expressions

```
double :: Int -> Int
double = \x -> x * 2

-- Multi-parameter
add :: Int -> Int -> Int
add = \a b -> a + b

-- Pattern matching in lambdas
fstOf :: (a, b) -> a
fstOf = \(x, _) -> x
```

4.3 Operators

```
-- Operators are binary functions
result = 1 + 2 * 3

-- Prefix form with parentheses
result = (+) 1 2

-- Functions as infix with backticks
result = 10 `div` 3

-- Sections
double = (* 2)
```

```
inc    = (+ 1)
halve  = (`div` 2)
```

4.3.1 Operator Precedence

Declare fixity with `infixl`, `infixr`, or `infix` (levels 0–9):

```
infixl 6 +
infixl 6 -
infixl 7 *
infixr 5 :
infixr 0 $
infix  4 ==
```

Standard Haskell fixities are the defaults; an undeclared operator is `infixl 9`. A fixity declaration governs the whole module, including uses above it, and travels with an import: an operator imported from another module keeps the fixity its defining module declared, as in GHC.

Operators declared `infix` are non-associative, so `a == b == c` is a parse error — parenthesize one side, or chain comparisons with `&&`. Mixing an `infixl` and an `infixr` operator of the same precedence without parentheses is rejected the same way, exactly as in GHC.

Prefix minus has the fixity of binary subtraction (`infixl 6`), exactly as in GHC: `a + -b` is a parse error (write `a + (-b)`), `-a * b` parses as `negate (a * b)` because the operand takes everything binding tighter than precedence 6, and `-a + b` is `negate a + b`. `(-x)` is negation, never a right section; `(-)` is the subtraction function.

4.4 Guards

```
abs :: Int -> Int
abs n | n < 0     = -n
      | otherwise = n

classify :: Int -> String
classify n
    | n == 0    = "zero"
    | n > 0     = "positive"
    | otherwise = "negative"
```

4.5 Where Clauses

```
circleArea :: Number -> Number
circleArea r = pi * r * r
    where pi = 3.14159265

bmi :: Number -> Number -> String
bmi weight height
    | b < 18.5  = "underweight"
    | b < 25.0  = "normal"
    | otherwise = "overweight"
    where b = weight / (height * height)
```

4.6 Let/In Expressions

```
cylinderVolume :: Number -> Number -> Number
cylinderVolume r h =
    let pi = 3.14159265
        baseArea = pi * r * r
    in baseArea * h
```

5 Pattern Matching

5.1 Function Definitions

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)

head :: [a] -> a
head (x:_) = x

fst :: (a, b) -> a
fst (a, _) = a
```

5.2 Case Expressions

```
describe :: [a] -> String
describe xs = case xs of
    []    -> "empty"
    [_]   -> "singleton"
    _:_:_ -> "multiple"
```

5.3 Guards in Case

```
eval :: Expr -> Int
eval e = case e of
    Lit n      -> n
    Add a b    -> eval a + eval b
    Neg n | n < 0 -> eval (Neg (Lit (-n)))
          | otherwise -> -(eval n)
```

6 Typeclasses

6.1 Defining a Class

```
class Printable a where
    display :: a -> String
```

6.2 Instances

```
instance Printable Int where
    display n = show n

instance Printable Bool where
    display True  = "yes"
    display False = "no"
```

6.3 Superclass Constraints

```
instance Show a => Printable [a] where
    display xs = "[" <> intercalate ", " (map show xs) <> "]"
```

6.4 Deriving

Automatic instance generation for common classes:

```
data Color = Red | Green | Blue
    deriving (Show, Eq, Ord, Enum, Bounded)

data Tree a = Leaf a | Branch (Tree a) (Tree a)
    deriving (Show, Eq, Functor)
```

Supported: `Show`, `Eq`, `Ord`, `Enum`, `Bounded`, `Functor`, `ToJSON`, `FromJSON`, and `LuaDict`. Derived `show` output matches GHC exactly: strings are quoted and escaped by GHC's rules, lists and tuples separate with `,` and no space (`show [1,2]` is `"[1,2]"`), records show in record syntax (`P {px = 1}`), and `Number` uses GHC's shortest-identifying `Double` format (`show 3.0` is `"3.0"`, `show 0.01` is `"1.0e-2"`). `ToJSON` and `FromJSON` generate a JSON encoder and decoder for the type; they require `import JSON` — see Section 12.12. `LuaDict` is special: it generates no methods but changes the runtime representation of a record or an all-nullary enum for Lua interop — see Section 12.11.

6.5 Built-in Classes

Class	Methods
`Show`	`show :: a -> String`
`Eq`	`(==), (/=) :: a -> a -> Bool`
`Ord`	`compare, (<), (>), (<=), (>=)`
`Enum`	`succ, pred, toEnum, fromEnum, enumFrom, …`
`Bounded`	`minBound, maxBound`
`Read`	`read :: String -> a`
`Num`	`(+), (-), (*), negate, abs, signum, fromInteger`
`Fractional`	`(/), recip, fromRational`
`Real`	(superclass marker: `(Num a, Ord a)`)
`Integral`	`quot, rem, div, mod, quotRem, divMod, toInteger`
`Functor`	`fmap :: (a -> b) -> f a -> f b`
`Applicative`	`pure, (<*>), liftA2`
`Monad`	`(>>=), return`

6.6 Kinds and Higher-Kinded Classes

Kinds classify types the way types classify values: a complete type such as `Int` or `Maybe String` has kind `Type`, while a type constructor that still needs arguments has an arrow kind (`Maybe`

is `Type -> Type`, `Either` is `Type -> Type -> Type`). There is no syntax for kinds — the compiler infers every kind from how a type or class variable is used, and anything unconstrained defaults to `Type`.

A class variable's kind comes from the method signatures. In

```
class Collapse t where
    collapse :: t Int -> Int
```

the use `t Int` forces `t` to kind `Type -> Type`, so instances must supply a type *constructor*, not a complete type. The bare, unapplied list constructor is written `[]`, and constructors may be partially applied:

```
instance Collapse [] where ...          -- lists
instance Collapse Maybe where ...
instance Collapse (Either c) where ...  -- Either, one argument short

instance Collapse [a] where ...       -- KIND ERROR: [a] is complete
instance Collapse Int where ...   -- KIND ERROR
```

Data types may take higher-kinded parameters the same way — `data Wrap f = Wrap (f Int)` infers `f` at `Type -> Type` — and every type you write (signatures, fields, ascriptions) is kind-checked, so applying a complete type to an argument (`Maybe Int Bool`) or using an unsaturated constructor where a complete type is required (`f :: Maybe -> Int`) is a compile-time error.

7 Monads and IO

7.1 IO Actions

Programs with a `main :: IO ()` are standalone executables:

```
main :: IO ()
main = do
    putStrLn "What is your name?"
    name <- getLine
    putStrLn ("Hello, " <> name <> "!")
```

7.2 Do-Notation

Do-notation desugars to monadic bind:

```
-- These are equivalent:
example1 :: IO ()
example1 = do
    x <- return "hello"
    let y = x <> "!"
    putStrLn y

example2 :: IO ()
example2 = return "hello" >>= \x ->
           let y = x <> "!" in
           putStrLn y
```

7.3 Common IO Functions

```
putStrLn :: String -> IO ()      -- print with newline
putStr   :: String -> IO ()      -- print without newline
getLine  :: IO String            -- read a line from stdin
getArgs  :: IO [String]          -- command-line arguments
exit     :: ExitValue -> IO ()   -- exit process
error    :: String -> a          -- abort with message
```

`getLine` is available in the Prelude with no import, GHC-style. It reads one line from stdin without the trailing newline; at end of input it raises a catchable error (`Prelude.getLine: end of input`) that `try` and `catch` capture — mata-ll's analog of GHC's `getLine` throwing an `isEOFError` exception. The `LIO` library module offers further input functions: `readLine :: IO String` (reads a line the same way and raises the same kind of catchable end-of-input error, named `LIO.readLine: end of input`), and `fileLines :: String -> IO [String]` reads a whole file as a list of lines.

7.4 Exception Handling

```
import LIO  -- fileLines :: String -> IO [String]

main :: IO ()
main = do
    result <- try (fileLines "data.txt")
    case result of
        Right ls -> mapM_ putStrLn ls
        Left err -> putStrLn ("Error: " <> err)

-- Or with catch:
safeRead :: String -> IO [String]
safeRead path = catch (fileLines path)
                      (\err -> return [])
```

8 Evaluation Strategy

MATA-LL is non-strict: function arguments are not evaluated until needed. Values are memoized once forced.

```
-- Infinite list -- only computed as needed
nats :: [Int]
nats = [1..]

fibs :: [Int]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)

-- take 10 fibs evaluates only 10 elements
first10 :: [Int]
first10 = take 10 fibs
```

8.1 Explicit Forcing

Use `seq` to force evaluation:

```
seq :: a -> b -> b
-- Forces first argument, returns second

strictSum :: [Int] -> Int
```

```
strictSum = go 0
    where go acc []     = acc
          go acc (x:xs) = seq acc (go (acc + x) xs)
```

Note: The compiler avoids thunks where it can prove that is safe, under one normative rule: *bottom is never evaluated eagerly.* An expression is evaluated eagerly only when the consumer is proven to force it (demand analysis, which recognizes tail accumulators as strict), or when it is provably total — a literal, an already-forced variable, a constructor or tuple of such, or non-trapping arithmetic over such. Everything else stays a thunk. Most programs do not need manual `seq`.

9 List Comprehensions

```
-- Generators and guards
evens :: [Int]
evens = [x | x <- [1..20], x `mod` 2 == 0]

-- Multiple generators (nested loops)
pairs :: [(Int, Int)]
pairs = [(x, y) | x <- [1..3], y <- [1..3], x /= y]

-- Pattern-matching generators
successes :: [Result a] -> [a]
successes results = [v | Ok v <- results]
```

10 Modules

10.1 Importing

```
-- Import everything from a module
import Data.List

-- Import specific items
import Data.List (sortBy, groupBy)

-- Import hiding specific items
import Data.List hiding (head, tail)

-- Qualified import
import qualified Data.Map as M
```

Modules map to file paths: `Data.List` resolves to `Data/List.mll` relative to source or library directories.

10.2 Export Lists

A module header may carry a Haskell-style export list:

```
module Geometry (area, Shape(..)) where
```

The list controls *import visibility within mata-ll*: other `.mll` modules can import only the listed names; everything else is private to the module. Unlike in GHC, the list does *not* export anything to the Lua host — the compiled module's `return` table is populated exclusively by `export` declarations (see *Exporting to Lua*). A module with neither a `main` nor any `export` compiles, standalone, to a Lua file with no runnable or callable code (dead-code elimination keeps only what `main` or an `export` can reach); the compiler warns when that happens. Such library modules are meant to be imported by the module you compile.

The library modules that ship with the compiler are catalogued in the next section.

11 Standard Library

The standard library ships inside the compiler: each module's `.mll` source is compiled into the `mllc` binary with Rust's `include_str!`, so an installed `mllc` is self-contained and needs no library directory on disk. A `-L` search path still takes precedence over the embedded copy, which lets you iterate on a module without rebuilding the compiler. The Prelude is imported into every module automatically; the rest are brought in with `import`.

Import	Provides
`Prelude`	Core types, classes, and functions (auto-imported)
`ByteString`	Byte sequences backed by Lua strings
`LBit`	Bitwise operations (Lua semantics)
`LMath`	`math.*` bindings
`LOS`	`os.*` bindings
`LString`	`string.*` bindings
`LIO`	File and stream I/O
`LIOLinear`	Linear exactly-once write handles
`Regex`	Backtracking regular-expression matcher
`JSON`	JSON codec and `ToJSON`/`FromJSON`
`Data.List`	List functions beyond the Prelude
`Data.Maybe`	`Maybe` helpers
`Data.Map`	Key/value maps over the HashMap type
`Data.Foldable`	Foldable operations
`Data.Traversable`	Traversable operations
`Control.Monad`	Monadic control helpers

The `L`-prefixed modules are FFI bindings to the Lua standard library; each function reduces to a Lua call (see *Foreign Function Interface*).

11.1 Prelude

Imported into every module. It defines the core types `Either`, `Ordering`, `ExitValue`, and `Any`; the `Semigroup` and `Monoid` classes; and the standard list, folding, and IO functions — `map`, `filter`, `foldr`, `zip`, `takeWhile`, `putStrLn`, `print`, `getLine`, `mapM`, `sequence`, `when`, `assert`, and more. The full catalogue is in the *Prelude Reference*.

11.2 ByteString

Byte sequences backed by an immutable Lua string, indexed from 0. The functions are compiler builtins, so they work without the import; import the module for its documentation. Construction: `bsEmpty`, `bsPack`, `bsConcat`, `bsConcatList`, `bsReplicate`. Access: `bsLength`, `bsIndex`, `bsSub`, `bsNull`. Transformation: `bsMap`, `bsFoldl`, `bsXor`, `bsZipWith`. Conversion: `bsToString`, `bsFromString`, `bsUnpack`.

11.3 The L-family: Lua bindings

These modules bind the Lua standard library through the FFI.

`LBit` — `band`, `bor`, `xor`, `bnot`, `shiftL`, `shiftR`. The shifts follow Lua semantics: `shiftR` is logical (it fills the vacated high bits with zero) rather than sign-extending, and a shift count at or beyond 64 gives 0.

`LMath` — the constants `pi`, `huge`, `maxinteger`, and `mininteger`; trigonometric, exponential, and rounding functions such as `sin`, `cos`, `sqrt`, `exp`, `log`, `floor`, and `ceil`; and the effectful `random`, `randomInt`, and `randomseed`.

`LOS` — `clock`, `time`, `difftime`, `date`, `getenv`, `remove`, `rename`, `execute`, `exit`, and `tmpname`.

`LString` — `strByte`, `strLen`, `strSub`, `strChar`, and `strToInts`, which turns a `String` into the list of its character codes so it can be processed with ordinary list functions.

11.4 LIO and LIOLinear

`LIO` provides file and stream I/O over an opaque `FileHandle`: `fOpen` and `fClose`; `fRead`, `fReadN` (a fixed byte count), `fReadLine`, and `fWrite`; and the stream operations `readLine`, `readStdin`, `writeStdout`, and `flushStdout`. `fOpen` returns `Either String FileHandle`, so an open failure is a value rather than a raised error.

`LIOLinear` provides a write handle, `WHandle`, under the linear exactly-once discipline. Every operation consumes the handle: `hPut` hands back a fresh handle to thread onward, and `hClose` ends the chain. The type checker proves the handle is used and closed once — dropping it (a leak) or mentioning it twice (a write or close after close) is a compile error. The entry point is `withOutFile`, whose callback receives the handle at `%1` (see *Linear Types*); `openOut` opens one directly.

11.5 Regex

A backtracking regular-expression matcher written in MATA-LL, with a compiled pattern type `RE` and a `Match` result. `compile` turns a pattern string into `Either String RE`; `test` reports whether a pattern occurs, `find` and `findStr` return the first match, and `matchFull` requires the whole string to match. The syntax covers `. * + ? |`, groups `()`, character classes, the anchors `^` and `$`, and the escapes `\d`, `\w`, and `\s` (with their negations).

11.6 JSON

`import JSON` provides a JSON codec written in MATA-LL: `parseJSON` and `encodeJSON` over the `Json` data type, plus `decodeJSON` and `encodeToJSON` driven by the `FromJSON` and `ToJSON` classes. It is covered in Section 12.12.

11.7 Data and Control modules

`Data.List` adds list functions the Prelude omits: `sortBy`, `foldl'`, `nubBy`, `groupBy`, `partition`, `intersperse`, `intercalate`, `unfoldr`, `scanl`, `scanr`, `find`, and `drop`.

`Data.Maybe` — `maybe`, `fromMaybe`, `isJust`, `isNothing`, `fromJust`, `listToMaybe`, `maybeToList`, `catMaybes`, and `mapMaybe`.

`Data.Map` — key/value maps aliased to the HashMap type (`type Map k v = HashMap k v`). Import it `qualified`, since names such as `lookup`, `insert`, and `map` shadow the Prelude. It provides `empty`, `singleton`, `insert`, `delete`, `lookup`, `member`, `size`, `keys`, `values`, `toList`, `fromList`, `map`, `filter`, the `foldlWithKey` and `foldrWithKey` folds, and `union`, `intersection`, and `difference`. Keys must be primitive (see *HashMap*).

`Data.Foldable` and `Data.Traversable` re-export the Foldable and Traversable operations — `foldr`, `foldMap`, `toList`, `sum`, `traverse`, `sequenceA`, `mapM`, and `sequence` — so GHC-style selective imports resolve.

`Control.Monad` — `forM`, `forM_`, `sequence_`, `void`, `join`, `guard`, and `unless`.

12 Foreign Function Interface

MATA-LL interoperates with Lua through type families.

12.1 Pure FFI

For Lua functions with no side effects:

```
sin :: Number -> LuaPure "math.sin" Number
cos :: Number -> LuaPure "math.cos" Number
strlen :: String -> LuaPure "string.len" Int
```

`LuaPure "name" T` reduces to `T` — the function is callable as a pure expression.

12.2 Effectful FFI

For Lua functions that perform I/O or have side effects:

```
rnd :: Int -> Int -> LuaIO "math.random" Int
clock :: LuaIO "os.clock" Number
```

`LuaIO "name" T` reduces to `IO T`.

12.3 Method-Call Syntax

Colon prefix generates Lua method calls (`obj:method()`):

```
newtype Handle = Handle LuaUserData

fileWrite :: Handle -> String -> LuaIO ":write" ()
fileClose :: Handle -> LuaIO ":close" ()
```

The receiver is opaque: `LuaUserData` is the builtin type for a Lua value (such as a file handle) that MATA-LL holds but never inspects, and wrapping it in a `newtype` gives each kind of handle its own distinct type. This is the idiom the standard library uses (`newtype FileHandle = FileHandle LuaUserData` in `LIO`).

12.4 Optional Parameters

`Maybe` arguments become optional Lua parameters:

```
rnd :: Number -> Maybe Number -> LuaIO "math.random" Number
-- rnd 1.0 Nothing     => math.random(1.0)
-- rnd 1.0 (Just 6.0)  => math.random(1.0, 6.0)
```

12.5 Passing Structured Values to Lua

When you call a host function, MATA-LL marshals each argument into its natural Lua representation — the mirror image of the conversion applied to exported return values (see *Exporting to Lua*). The conversion is structural and recursive, at every depth:

- A list `[a]` becomes a plain 1-based Lua array, each element marshalled by its type.
- A `LuaDict` record becomes a name-keyed Lua table; a tuple becomes a positional table.
- A `HashMap k v` becomes a string-keyed Lua table (a dictionary), each value marshalled by `v`.
- A `Maybe` *inside* a record, list, or tuple is unwrapped: `Just x` passes the bare (marshalled) `x`, and `Nothing` passes `nil` (an absent field). This is distinct from a top-level `Maybe` *parameter*, which is an optional positional argument (previous subsection).
- A `String` always arrives as a native Lua string, however it was built — a literal, a concatenation, or a JSON-decoded value.
- Opaque values — a type variable, a plain (non-`LuaDict`) ADT, a function, `LuaUserData` — pass through unchanged, so a value the host only stores and hands back (such as a fold's threaded state) round-trips intact.

So a `HashMap String [Int]` reaches the host as a dictionary of arrays, and a record whose field is a list of records is fully converted. Marshalling is non-destructive: the argument is copied into fresh Lua values, so the original remains usable in your program after the call. An FFI call is strict in its arguments — everything the host will read is forced at the call — but this evaluates nothing the call does not already demand.

12.6 Receiving Values from Lua

A result crossing back *in* is decoded by the exact dual of the argument marshaller: the declared result type drives a type-directed decoder that descends into the same containers, so a value passed out to an echoing host and handed back arrives identical at every nesting depth.

- A Lua array decodes to a list `[a]`, each element recursively decoded. An empty or absent (`nil`) array decodes to `[]`.
- A `LuaDict` record is read from a Lua table by its (possibly `as`-renamed) field keys; a tuple *nested* inside a structure is read from a positional table.
- A *top-level* tuple result is the one exception to "tables in, tables out": it is received as Lua's multiple return values (`return a, b`), not as a single table.
- A `HashMap k v` is read from a keyed table; each value is decoded by `v`. Keys are validated against `k` (string vs. numeric) but kept as they are — the argument marshaller likewise never converts keys.
- A `Maybe` reached through this descent inverts the unwrapping on the way out: a present value decodes to `Just` (its payload recursively decoded), `nil` to `Nothing`.
- Opaque values — a type variable, a plain (non-`LuaDict`) ADT, a function, `LuaUserData` — pass through untouched, so a mata-ll value the host merely held and handed back round-trips intact.

Every declared scalar is validated against its Lua runtime type as it crosses. If the host returns the wrong shape — a missing record field, a string where a number was declared, a scalar where an array was declared — the call fails immediately with an error naming what was declared, what actually arrived, and where (the field or element position, and the host function), instead of surfacing later as an arbitrary Lua error deep in your program:

```
FFI result: declared Int but the host returned nil
  (field 'certPort' of record Cert; in the result of host.cert)
```

12.7 Passing Callbacks to Lua

An FFI argument may itself have a function type: mata-ll hands one of its own functions to the host. (The opposite direction — Lua handing a function to mata-ll — is `engage`; see *Scoped Lua Callbacks*.) The motivating case is a fold-style host API that calls the callback once per item and threads its return value as the next state, such as a SQL driver folding over result rows:

```
foldRows :: String
         -> (Int -> acc -> acc)          -- pure callback
         -> acc
         -> LuaPure "db.fold" acc

foldRowsIO :: String
           -> (Int -> acc -> LuaIO s acc)  -- effectful callback
           -> acc
           -> LuaIO "db.fold" acc
```

The compiler wraps each function-typed argument so the Lua host can call it with positional arguments. The wrapper applies the mata-ll callback to all of the host's arguments at once (mata-ll functions are n-ary, not curried); marshals an argument across the boundary when its type is a list or a nested function, and otherwise passes it raw; runs the returned action for an effectful callback (result `LuaIO s acc`), or takes the value directly for a pure one (result `acc`); and returns the result to the host, marshalling it unless it is the opaque threaded state.

The threaded state (`acc`) is a polymorphic type variable, so the host cannot inspect it — and the wrapper passes it through *raw* in both directions rather than marshalling it. This is what lets *any* mata-ll value, including tuples and ADTs, be used as the state and round-trip intact: marshalling would flatten a tuple `(a, b)` to a Lua array and rebuild it as the cons list `a : b : []`, corrupting it.

The compiler type-checks this soundness: when a callback threads a polymorphic state, it must be *one shared type variable* across four positions — the callback's accumulator argument, the callback's result, the FFI's initial-state argument, and the FFI's return type. Effectful callbacks must use `LuaIO s acc`, not `IO acc`. A callback with no type variables (say `String -> String` for `string.gsub`) threads no opaque state and is accepted without these constraints.

12.8 Iterators

Wrap Lua iterator protocols. The type argument to `LuaIterator` names the RESULT list — the iterator yields its ELEMENTS, one per step, each decoded as the element type (a structured element such as a list is decoded like any other FFI result):

```
-- host yields plain strings -> a list of strings
gmatch :: String -> String -> LuaIterator "string.gmatch" [String]

-- host yields plain integers -> a list of integers
counter :: LuaIterator "count" [Int]

-- host yields Lua arrays of ints -> a list of lists
rows :: LuaIterator "rows" [[Int]]
```

The result is always written as an explicit list. A bare element type (`LuaIterator "string.gmatch" String`) is rejected: because a list argument such as `[[Int]]` already means "an iterator yielding `[Int]`", a bare element would make the argument ambiguous, so it must always name the result list directly.

12.8.1 Memory Behaviour of Iterator Lists

An iterator list follows the same rules as any other lazy list (see *Evaluation Strategy*), and those rules decide when its realized cells can be reclaimed. What matters is not whether the iterator takes an argument — it is whether the realized list is rooted by a nullary top-level binding.

A nullary top-level binding such as `counter` above compiles to a memoized module-lifetime thunk (a constant applicative form). Once forced, the head cell is memoized into that slot and every realized tail into its predecessor cell, so the entire realized spine stays reachable from a permanent root. It is retained until the host drops the module from `package.loaded`, which normal programs never do — garbage collection cannot reclaim it.

A binding with parameters compiles to a plain function and is never memoized. Each application builds a fresh, independent list — calling it twice runs the Lua factory twice, with no sharing between the calls. When the result is consumed directly at the application site, the realized cells are reclaimed on the next collection cycle after consumption.

Re-binding an applied result to a nullary top-level name (`events = mkEvents 0`) recreates the module-lifetime root and retains the full spine exactly as the nullary form does. A `let`-bound local also roots the head, but only for the lifetime of its enclosing frame.

The consumer decides peak memory:

- A strict tail-recursive walker compiles to a proper Lua tail call and streams in constant space: consumed cells become garbage as the cursor advances, provided nothing holds the head. Measured at about 75 KB, flat, over two million elements.
- `take n` realizes exactly `n` cells.
- `mapM_` — and IO sequencing generally — also streams: each sequencing step is emitted as a proper Lua tail call through the runtime's forwarding runner, so the walk runs in constant stack and consumed cells become garbage as the cursor advances. Measured over one million elements: constant stack on both LuaJIT and Lua 5.5, with the live set flat (about 73 KB LuaJIT, 55 KB Lua 5.5) for the whole walk.

For an endless iterator these effects compound: with a tail-call consumer — pure walker or `mapM_` — and no head root it streams forever in constant space; with a head root, growth is unbounded until memory exhaustion.

```
-- Retains every realized cell for the life of the module:
events :: LuaIterator "events" [Int]

-- Streams in constant space: parameterized binding, applied
-- at the use site, consumed by a strict tail-call walker.
mkEvents :: Int -> LuaIterator "events" [Int]

walk :: Int -> [Int] -> Int
walk n []     = n
walk n (_:ys) = seq n (walk (n + 1) ys)

main :: IO ()
main = print (walk 0 (mkEvents 0))
```

For large or endless iterators: give the binding a parameter and apply it at the use site; do not re-bind the result to a top-level name; bound the list with `take` or consume it with a strict tail-recursive walker or `mapM_` — both stream in constant space.

12.9 Error-Capturing FFI

Many Lua functions signal failure by *raising* an error (`error(...)`). Called through a plain `LuaPure`/`LuaIO` binding, such an error propagates as a runtime crash. `LuaCatch` and `LuaIOCatch` instead run the call under Lua's `pcall` and hand back an `Either`: a raised error becomes `Left` (carrying the message), a normal return becomes `Right`.

```
-- pure call, result Either String a
charOf :: Int -> LuaCatch "string.char" (Either String String)

-- effectful call, result IO (Either String a)
readAll :: Handle -> LuaIOCatch ":read" (Either String String)
```

`LuaCatch "name" (Either String T)` reduces to `Either String T`; `LuaIOCatch "name" (Either String T)` reduces to `IO (Either String T)`. The result *must* be written as `Either String a` — the compiler rejects any other shape. The success payload is marshalled as usual, inside the `Right`:

```
main :: IO ()
main = case charOf 65 of
    Right c  -> putStrLn c              -- "A"
    Left msg -> putStrLn ("failed: " <> msg)
```

Use these when a Lua function fails by raising. For the `(nil, errmsg)` return convention (`io.open`-style) use `LuaTry` instead; to catch errors from an ordinary `LuaIO` action, wrap it in `try` (see *Exception Handling*).

12.10 Exporting to Lua

Use `export` to make functions callable from Lua:

```
fib :: [Int]
fib = 1 : 1 : zipWith (+) fib (tail fib)

export fibonacci :: Int -> [Int]
fibonacci = flip take fib
```

From Lua:

```
local mll = require("fib")
local fibs = mll.fibonacci(10)
-- fibs is a plain Lua array: {1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

Exported functions marshal in both directions. Arguments the host passes are converted to mata-ll representations: Lua arrays become cons lists (recursively), Lua functions become callable callbacks, and scalars pass as-is. The result is deep-converted to plain Lua data: thunks are forced, cons lists become 1-based arrays, `LuaDict` records stay name-keyed tables, tuples and plain ADTs become positional tables, `Just x` is unwrapped to the bare `x` and `Nothing` becomes `nil`.

Note: Lua's `nil` carries less structure than mata-ll values. An empty list crosses to the host as an empty table at every edge — exported results included — so a host can `ipairs` a list result without a nil check; the declared type is what distinguishes it from `Nothing`, which crosses as `nil`. And because `nil` cannot nest, `Just Nothing` flattens to `nil` and `Just (Just x)` to `x` at the boundary.

An export need not be a function. A plain value — a scalar, tuple, record, ADT, `Maybe`, or finite list — appears in the module's return table as its forced, marshalled value, following the same conversions as a function result: a scalar crosses as a Lua number or string, a tuple or plain record as a positional table, a `LuaDict` record as a name-keyed table. An exported `IO` or `LuaIO` action appears as a Lua function that runs the action when called and returns its marshalled result.

```
export answer :: Int
answer = 42

export tick :: IO Int
tick = return 99
```

From Lua, `mll.answer` *is* the number `42`, and `mll.tick()` runs the action and returns `99`.

Not every signature can cross the boundary. An export whose type uses something the marshaller cannot represent in Lua is rejected at compile time, with the offending sub-type named in the error:

- a polymorphic type variable (`export ident :: a -> a`) — Lua has no representation for a polymorphic value;
- a class constraint (`Num a => …`) — a typeclass dictionary cannot cross the boundary;
- a region-scoped `ST`/`STArray` handle — it must not outlive its `runST`;
- an `IO` or `LuaIO` action in *argument* position — a Lua caller has no action to hand in;
- a function type anywhere except as a direct top-level argument of the export itself. That one position is a *callback*, and it must return `LuaIO s r` — Lua functions are untrusted and assumed effectful.

12.11 LuaDict

Many Lua APIs take or return *dictionaries* — tables keyed by name, such as `{width = 80, height = 25}`. Deriving `LuaDict` on a record makes its runtime representation exactly such a table. This is not a marshalling layer that converts at the FFI boundary: the record value *is* a name-keyed Lua table from the moment it is constructed, so it can be handed to (or received from) Lua functions with no conversion cost.

```
data Config = Config { width  :: Int
                     , height :: Int
                     , title  :: String
                     } deriving (Show, Eq, LuaDict)

-- a Lua function expecting a dictionary
setup :: Config -> LuaIO "display.setup" ()

main :: IO ()
main = do
    let c = Config { width = 80, height = 25, title = "hi" }
    -- at runtime, c is {width = 80, height = 25, title = "hi"}
    setup c
```

Without `LuaDict`, records are stored as positional Lua arrays; with it, every field becomes a named key. On the MATA-LL side nothing changes: accessor functions, record construction (named or positional), record update, pattern matching, and derived `Show`/`Eq` all work as usual.

The direction is symmetric — an FFI function declared to *return* a `LuaDict` type accepts a Lua table with the corresponding keys:

```
currentConfig :: LuaIO "display.get_config" Config
```

`LuaDict` accepts two shapes of data type, each with a natural name-based Lua representation:

- A *record* — exactly one constructor, record syntax (named fields, not positional), at least one field, no GADT or existential. Its value is the name-keyed table shown above.
- An *all-nullary sum type* — an enumeration whose every constructor takes no fields. Its value is a *string*, described next.

Violations (for instance a multi-constructor type with fields, which fits neither shape) are compile errors with an explanation of why.

12.11.1 LuaDict Enums

An all-nullary sum type — an enumeration, every constructor without fields — derives `LuaDict` to cross the Lua boundary as a plain *string*. As with a record, no marshalling runs at the FFI boundary: the runtime value *is* that string — the constructor's `as "tag"` rename when present, otherwise the constructor's own name — so such an enum passes to and from Lua functions with no conversion cost.

```
data Perm = Anonymous as "anonymous" | User | Admin
    deriving (Show, Eq, Ord, Enum, Bounded, LuaDict)

-- at the Lua boundary:
--   Anonymous  is  "anonymous"   (the as rename)
--   User       is  "User"        (the source name)
--   Admin      is  "Admin"
```

The string is a boundary-only representation. On the MATA-LL side nothing changes: construction, pattern matching, and the derived `Ord`, `Enum`, `Bounded`, and `Show` all follow declaration order and the source constructor names, exactly as for an enum without `LuaDict`. So `fromEnum Anonymous == 0`, `Anonymous < User`, and `show User == "User"` hold regardless of any `as` rename — `as` moves only the external string. Effective tags (the rename when present, the source name otherwise) must be unique and non-empty, so two constructors cannot collide on one Lua string.

12.11.2 Renaming Keys with `as`

By default the Lua key is the field name. Since record accessors share a single global function namespace (as in Haskell), fields are conventionally prefixed (`cfgWidth`, `cfgHeight`) — but a Lua API will expect `width`, not `cfgWidth`. The `as` annotation renames the Lua table key:

```
data Config = Config { cfgWidth  as "width"  :: Int
                     , cfgHeight as "height" :: Int
                     , cfgTitle  as "title"  :: String
                     } deriving (Show, Eq, LuaDict)

area :: Config -> Int
area c = cfgWidth c * cfgHeight c    -- Haskell name, unchanged

resize :: Config -> Int -> Config
resize c w = c { cfgWidth = w }      -- updates the "width" key
```

`as "key"` sets the field's *external name* — the name the field carries at the language boundary. In MATA-LL source the field keeps its declared name everywhere: accessors, record construction, record update, and pattern matching all use `cfgWidth`. The rename applies in both directions

across the FFI boundary — tables you pass to Lua carry the renamed keys, and tables returned from Lua are read by the renamed keys.

The external name is shared by everything that puts the field outside MATA-LL: it is the key in the runtime Lua table of a `LuaDict` record *and* the JSON object key of a derived `ToJSON`/`FromJSON` codec (Section 12.12). One rename drives both boundaries:

```
data Acct = Acct { acctName  as "name" :: String
                 , acctScore :: Int
                 } deriving (Eq, LuaDict, ToJSON, FromJSON)

-- at runtime the value is the Lua table
--   {name = "zoe", acctScore = 5}
-- and the derived codec uses the same keys:
--   encodeToJSON (Acct "zoe" 5)
--     == "{\"name\":\"zoe\",\"acctScore\":5}"
-- decodeJSON reads the renamed keys back; the Haskell
-- field name acctName appears at neither boundary.
```

Besides dropping prefixes, `as` is useful for `snake_case` keys expected by Lua APIs, and for keys that are Lua reserved words:

```
data Creds = Creds { credsUser as "user_name" :: String
                   , credsNote as "function"  :: String
                   } deriving (LuaDict)
```

Reserved-word keys are emitted with bracket syntax (`{["function"] = ...}`), so any string is a valid key.

The compiler enforces two rules on renames:

- Effective keys must be unique and non-empty. Each field becomes one key of a single table (and of a single JSON object), so two fields mapping to the same key (or a key of `""`) are rejected.
- `as` requires that the type derive at least one of `LuaDict`, `ToJSON`, or `FromJSON`. Without one of those the rename has nothing to apply to — there is no name-keyed table and no JSON codec — and the annotation is rejected rather than silently ignored.

Sum-type *constructors* have a parallel rename — `Con field-types as "name"` — described with the JSON encoding conventions (Section 12.12.2). For a fielded constructor it drives only the JSON tag, since a fielded variant is a positional table at the Lua boundary with no name to change; for the nullary constructors of a `LuaDict` enum it also sets the Lua boundary string (Section 12.11.1 above).

Note: `as` renaming is a MATA-LL extension with no GHC equivalent. In GHC, `as` is only a keyword in import lists; here it is additionally recognized inside record declarations (after a field name) and data declarations (after a constructor's field types).

12.12 JSON

JSON is the lingua franca on the far side of the FFI — Lua web frameworks, configuration files, and HTTP APIs all speak it. `import JSON` provides a complete codec: a strict JSON parser, a serializer, and the `ToJSON`/`FromJSON` classes, written entirely in MATA-LL (no Lua JSON library is needed).

At the base sits an ordinary algebraic data type for JSON values, with a parser and a serializer:

```
data Json = JNull | JBool Bool | JNum Number | JStr String
```

```
          | JArr [Json] | JObj [(String, Json)]

parseJSON  :: String -> Either String Json
encodeJSON :: Json -> String
```

You can pattern-match `Json` values directly, but usually you will not need to: deriving `ToJSON` and `FromJSON` generates the codec for your own types, and one function each way does the whole job:

```
encodeToJSON :: ToJSON a   => a -> String
decodeJSON   :: FromJSON a => String -> Either String a
```

```
import JSON

data Person = Person { name :: String, age :: Int }
    deriving (Eq, ToJSON, FromJSON)

main :: IO ()
main = do
    putStrLn (encodeToJSON (Person "Ann" 30))
    -- {"name":"Ann","age":30}
    case (decodeJSON "{\"age\":7,\"name\":\"Bo\"}"
              :: Either String Person) of
        Right p  -> putStrLn ("hello, " <> name p)  -- hello, Bo
        Left err -> putStrLn err
```

Decoding never crashes on bad input — it returns `Either String a`, with the parse or shape error on the `Left`. Like `read`, `decodeJSON` dispatches on its *result* type; when the surrounding context does not pin that type, ascribe it at the call site as above. Field order in the input is irrelevant, and unknown object keys are ignored.

12.12.1 Encoding Conventions

The derived codec mirrors the layout of GHC's `aeson` under its default options. `deriving (ToJSON)` and `deriving (FromJSON)` are exact mirrors: everything the encoder writes, the decoder reads back, so `decodeJSON (encodeToJSON x) == Right x`.

A single-constructor record maps to an object keyed by the field names (`Person` above). When a field is renamed with `as "key"`, the JSON key is the renamed key — the same shared external name that `LuaDict` uses as the Lua table key (Section 12.11.2).

A multi-constructor type is *tagged*, so the decoder can tell the constructors apart:

```
data Shape = Circle { radius :: Number }
           | Rect Int Int
           | Point0
    deriving (Eq, ToJSON, FromJSON)

-- encodeToJSON (Circle 2.5)  record fields inline next to the tag:
--   {"tag":"Circle","radius":2.5}
-- encodeToJSON (Rect 3 4)    positional arguments under "contents":
--   {"tag":"Rect","contents":[3,4]}
-- encodeToJSON Point0        nullary: the bare constructor name:
--   "Point0"
```

The decoder accepts a nullary constructor in either spelling — the bare string `"Point0"` or the tagged object `{"tag":"Point0"}`.

A type with a single *positional* constructor is untagged: one argument encodes as the value

itself, several as an array.

```
data Wrap  = Wrap Int          deriving (ToJSON, FromJSON)
data Pair2 = Pair2 Int String  deriving (ToJSON, FromJSON)

-- encodeToJSON (Wrap 7)        ==  7
-- encodeToJSON (Pair2 7 "a")   ==  [7,"a"]
```

Lists encode as arrays, `Maybe` maps to null (`Nothing` encodes as `null` with the key kept present; a missing key, an explicit `null`, and the plain value all decode), and fields of other `ToJSON`/`FromJSON` types nest — including self- and mutually-recursive types:

```
data Team = Team { leader  :: Person
                 , members :: [Person]
                 , motto   :: Maybe String
                 } deriving (Eq, ToJSON, FromJSON)

-- encodeToJSON (Team (Person "A" 1) [Person "B" 2] Nothing) ==
-- {"leader":{"name":"A","age":1},
--  "members":[{"name":"B","age":2}],"motto":null}

data JTree = JLeaf Int | JNode [JTree]
    deriving (Eq, ToJSON, FromJSON)

-- encodeToJSON (JNode [JLeaf 1]) ==
-- {"tag":"JNode","contents":[{"tag":"JLeaf","contents":1}]}
```

12.12.2 Renaming Constructor Tags with `as`

By default the tag is the constructor's source name. Like a record field (Section 12.11.2), a constructor can be renamed with `as`: `Con field-types as "name"` — after the field types (or the record braces), before the next `|` or `deriving` — sets the constructor's *external tag*, the string the derived codec writes and reads to tell the constructors apart. A nullary constructor encodes as the bare external string, a fielded one carries it as the `"tag"` value:

```
data Suit = Clubs as "clubs" | Spades as "spades"
    deriving (Show, Eq, ToJSON, FromJSON)

data Outcome = Ok Int as "ok" | Err String as "error"
    deriving (Show, Eq, ToJSON, FromJSON)

-- encodeToJSON Spades     == "spades"
-- encodeToJSON (Err "x") == {"tag":"error","contents":"x"}
-- decodeJSON reads the external tags back; a constructor
-- without `as` keeps its source name as before.
```

For the JSON codec the rename changes only the tag. `show` still prints the source constructor name (`show Spades == "Spades"` — `Show` is for debugging, as in Haskell), and construction, pattern matching, and the in-language representation are untouched.

Whether the rename also reaches the Lua boundary depends on the representation. A fielded variant is a positional table there, and a plain enum without `LuaDict` a positional integer, neither with a name for `as` to change — so the JSON tag is the only surface those constructors are renamed on. The exception is an all-nullary constructor of a type deriving `LuaDict`, which is a plain string at the boundary (Section 12.11.1): there one `as` sets both the JSON tag and the Lua string.

The compiler enforces the analogous rules. Effective tags (the rename when present, the

source name otherwise) must be unique and non-empty within one type: the tag is the only thing the decoder has to tell the constructors apart, so two constructors sharing one — including a rename that collides with another constructor's unrenamed name — would encode identically and decode ambiguously, and are rejected. The rename requires the type to derive `ToJSON`, `FromJSON`, or `LuaDict` — one of the derivings that names constructors externally, as a JSON tag or, for an all-nullary `LuaDict` enum, a Lua string. Without one it is rejected rather than silently ignored, as is a rename on the constructor of an *untagged* type — a single non-nullary constructor, whose JSON carries no tag anywhere.

12.12.3 Decode Errors

A failed decode names the type being decoded and the full path to the offending value — fields, array indices, constructor arguments — before saying what was expected and what was found:

```
decodeJSON "{\"name\":\"Ann\",\"age\":\"old\"}"
  => Left "while decoding Person: in field 'age': expected an integer, but found a
      string"

decodeJSON "{...,\"members\":[{\"name\":\"B\",\"age\":2},{\"name\":\"C\"}]}"
  => Left "while decoding Team: in field 'members': at array index 1: while decoding
       Person: the required field 'age' is missing"
```

Numbers are handled exactly: integers round-trip with full 64-bit precision (beyond the 2^{53} range of a double), and decoding an `Int` field rejects non-integral numbers (`3.5`) and numbers outside the 64-bit range instead of silently rounding.

12.12.4 What Can Be Derived

A derived codec resolves one concrete encoder or decoder per field at compile time, which places requirements on the type:

- `import JSON` must be present — the classes and the combinators the generated code calls live there.
- The type must have no type parameters. (GHC's `aeson` handles `Box a` with a constraint on the instance; MATA-LL does not derive constrained codecs, so derive for concrete types and wrap each instantiation you need in its own data type.)
- No GADT or existential constructors.
- Every field must be `Int`, `Number`, `String`, `Bool`, `Json` (kept verbatim), a list or `Maybe` of a supported type, or a type that itself has a `ToJSON`/`FromJSON` instance. Function, tuple, and `IO` fields are rejected.
- In a multi-constructor type, no field's JSON key may be `"tag"` — it would collide with the constructor tag. (Renaming such a field with `as` resolves the collision.)

Violations are compile errors with an explanation of why. For a type the derive rejects, write the instance by hand: the JSON module exports the decoder combinators the derived code itself uses (`jFieldWith`, `jOptFieldWith`, `jContext`, `jBind`, …), so a manual instance composes the same pieces.

Note: Two deliberate deviations from GHC's `aeson`. First, `aeson` encodes nullary constructors as bare strings only in *all*-nullary sums and emits `{"tag":"Point0"}` in a mixed sum; MATA-LL emits the bare string for every nullary constructor. The derived decoder accepts both spellings, so MATA-LL round-trips and `aeson`-encoded data both decode. Second, `Maybe (Maybe a)` cannot round-trip under the null-is-Nothing convention: `Just Nothing` has no JSON form distinct from `Nothing`.

13 Advanced Types

13.1 GADTs

Generalized Algebraic Data Types allow constructors to refine type parameters:

```
data Expr a where
    LitI :: Int -> Expr Int
    LitB :: Bool -> Expr Bool
    Add  :: Expr Int -> Expr Int -> Expr Int
    If   :: Expr Bool -> Expr a -> Expr a -> Expr a

eval :: Expr a -> a
eval (LitI n)     = n
eval (LitB b)     = b
eval (Add x y)    = eval x + eval y
eval (If c t e)   = if eval c then eval t else eval e
```

13.2 Existential Types

Hide a type behind an interface:

```
data ShowBox = forall a. MkShowBox a (a -> String)

showIt :: ShowBox -> String
showIt (MkShowBox x f) = f x

boxes :: [ShowBox]
boxes = [MkShowBox 42 show, MkShowBox "hi" id]
```

The hidden type is *rigid* inside a match: it was erased when the value was packed, so the compiler rejects any use that would need to know it — returning the unpacked value, comparing it to a concrete type, or letting it reach any type that outlives the match. The error notes name the constructor that hid the type. Two matches on two boxes see two *different* hidden types.

A constructor can promise classes for the hidden type; packing checks the instance, and inside a match exactly those classes (and their superclasses) are usable:

```
data Showable = forall a. Show a => Showable a

describe :: Showable -> String
describe (Showable x) = show x    -- Show is declared: fine
-- (Showable x) -> x + 1          -- rejected: Num was never promised
```

GADT syntax hides implicitly: a signature variable that does not reach the constructor's result type is existential, context included (`MkBox :: Show a => a -> Box` in `where`-style declarations). Record fields whose type mentions the hidden variable have no selector function and cannot be record-updated — pattern-match positionally and rebuild with the constructor instead.

13.3 DataKinds

Promote data constructors to the type level for compile-time safety:

```
data AgendaState = Empty | NonEmpty

data Agenda a (s :: AgendaState) where
    Nil   :: Agenda a 'Empty
```

```
    One   :: a -> Agenda a 'NonEmpty
    Cons  :: a -> Agenda a 'NonEmpty -> Agenda a 'NonEmpty

-- Only non-empty agendas can be submitted
submit :: Agenda String 'NonEmpty -> IO ()
submit agenda = putStrLn "Submitted!"
```

13.4 Type Families

User-defined type-level computation:

```
type family Element container where
    Element [a]           = a
    Element (HashMap k v) = v
```

13.5 Rank-2 Types and Scoped Mutation

The `ST` monad provides mutable arrays inside pure computations:

```
import Data.List (replicate)

sumList :: [Int] -> Int
sumList xs = runST (do
    acc <- newSTArrayFromList [0]
    go acc xs
    readSTArray acc 0)

go :: STArray s -> [Int] -> ST s ()
go acc []     = return ()
go acc (x:rest) = do
    cur <- readSTArray acc 0
    writeSTArray acc 0 (cur + x)
    go acc rest
```

The `forall s.` in `runST`'s type prevents mutable state from escaping.

13.6 Scoped Lua Callbacks

When exporting functions that receive Lua callbacks:

```
export processEvent :: forall s. (Int -> Int -> LuaIO s Int)
                    -> Int -> LuaIO s Int
processEvent f n = do
    (liftIO . putStrLn) $ "Called with " <> show n
    f n (n + 1)
```

The `forall s.` scope parameter prevents the callback from being stored or used outside its invocation.

13.7 Linear Types

A function arrow can carry a *multiplicity*: `a %1 -> b` promises that the function consumes its argument *exactly once*. A plain `->` is unrestricted (spelled explicitly as `%Many` or `%'Many`). This is GHC's `LinearTypes`: a second use of a `%1` value is the double-close/double-free class of bug, and *zero* uses leak the resource — both are compile errors:

```
data Conn = Conn Int

close :: Conn %1 -> IO ()
close (Conn fd) = putStrLn ("closed " <> show fd)

good :: Conn %1 -> IO ()
good c = close c            -- fine: exactly one use

bad :: Conn %1 -> IO ()
bad c = do
    close c
    close c                 -- rejected: c used twice

leak :: Conn %1 -> IO ()
leak c = putStrLn "bye"     -- rejected: c never consumed
```

The compiler counts uses along every evaluation path: sequential statements add up, the branches of a `case`/`if` are alternatives (one use in each branch is one use — and every branch *must* consume the value, since a branch that skips it would leak it), and aliases inherit the obligation — a pattern binder from a match on a `%1` value, a `<-` binder from an action that consumed one, or a `let` binding whose right-hand side did. Because evaluation is lazy, a `%1` value parked in a binding that is never forced is *not* consumed and is rejected, as is matching part of one with a wildcard (`_`), or discarding a non-`()` result built from one. A scalar (`Int`, `Number`, `Bool`, `String`) derived from a `%1` value is tracked *exactly once* like any other alias — there is no memoization exemption, in strict parity with GHC, so reading such a scalar twice (`go + go` where `go = useOnce t`) is a leak-or-double-use error just as reading the linear value itself twice would be. `()`-typed results are exempt: the run-for-effect idiom (`close c >> …`) discards them by design.

A signature may also be *multiplicity-polymorphic*: `apply :: (a %m -> b) -> a %m -> b` lets each caller choose `m`; applied to a `%1` function the argument stays linear through the helper, and inside `apply` itself the `%m` binder is held to exactly-once (a caller may pick `m = 1`). Local `where`/`let` *functions* that merely forward a value are inferred to consume it once, so forwarding through them is fine.

Passing a `%1` value to a function whose own arrow is a plain `->` is rejected — the callee promises neither exactly-once nor even at-most-once. Conversely a `%1` function is not interchangeable with an unrestricted one (`map close conns` is a type error, as in GHC). Returning the value, storing it in a constructor or tuple exactly once, and consuming it through another `%1` function are all single uses. Capturing it in a lambda (the closure may run any number of times — or never) is rejected. In a `Maybe` do-block, consume the value in a bind's *action*, not its continuation: the continuation is skipped on `Nothing`, which would leak the value (GHC rejects this too). List and user-defined monads reject linear values in their continuations outright, since their binds may run them any number of times.

Multiplicities are a type-checking discipline only: they erase after checking, and the compiled Lua is byte-identical with or without the annotations.

Boundary. The checker's approximations err toward rejection, never toward accepting a double use or a leak, and scalars now get no exemption, so there is no remaining accept-direction gap. Two deviations from GHC's verdict remain. The Lua side of a `%1` FFI signature is *trusted*: mata-ll charges the argument once per call and cannot check what the host does with it — the `%1` is your assertion about the foreign function. And an unannotated `let`/`where` binding that is never forced consumes nothing (its thunk never runs under lazy evaluation), so mata-ll accepts `let u = t in useOnce t` where GHC's stricter typing rule rejects it — more permissive than GHC, but it cannot leak or double-use at run time. Conservative rejections you may hit: a wildcard over a tainted scalar scrutinee (use a variable pattern instead), a discarded non-`()`

result, and a record update over a record built from a linear value.

14 HashMap

A built-in dictionary type backed by Lua tables. The `hm*` builtins need no import:

```
main :: IO ()
main = do
    let m = hmFromList [("a", 1), ("b", 2), ("c", 3)]
    putStrLn (show (hmLookup "b" m))  -- Just 2
    let m2 = hmInsert "d" 4 m
    putStrLn (show (hmKeys m2))       -- [a, b, c, d]
    putStrLn (show (hmMember "x" m))  -- False
```

Operations: `hmEmpty`, `hmInsert`, `hmLookup`, `hmDelete`, `hmSize`, `hmKeys`, `hmValues`, `hmMember`, `hmFromList`, `hmToList`. Keys must be primitive (a `String` or a numeric type).

The `Data.Map` library module wraps the same type under the Haskell-compatible names (`empty`, `insert`, `lookup`, `fromList`, …, plus `union`, `filter`, `foldrWithKey` and friends). Import it *qualified*, as in Haskell: several of its names (`lookup`, `null`, `map`, `filter`) deliberately shadow the Prelude, and because mata-ll merges every import into a single namespace, an unqualified `import Data.Map` is rejected with a name-conflict error.

```
import qualified Data.Map as M

main :: IO ()
main = do
    let m = M.fromList [("a", 1), ("b", 2), ("c", 3)]
    putStrLn (show (M.lookup "b" m))  -- Just 2
    putStrLn (show (M.keys (M.insert "d" 4 m)))
```

15 Ranges and Enumerations

Range syntax desugars to `Enum` methods:

```
[1..10]        -- enumFromTo 1 10        => [1,2,...,10]
[1,3..10]      -- enumFromThenTo 1 3 10  => [1,3,5,7,9]
[1..]          -- enumFrom 1             => [1,2,3,...] (infinite)
[1,3..]        -- enumFromThen 1 3       => [1,3,5,...] (infinite)
```

Works with any type that has an `Enum` instance, including user-defined types with `deriving Enum`.

16 Differences from Haskell

If you are coming from Haskell, keep these differences in mind:

1. **String is not `[Char]`.** It is a native Lua string. Use `<>` for concatenation; `++` is list-append only.
2. **Numeric classes present.** `Num`, `Fractional`, `Real`, and `Integral` are built in with GHC's signatures, and numeric literals are polymorphic (`Num a => a` / `Fractional a => a`) with GHC defaulting. The deviations are that `fromRational` takes a `Number` (mata-ll has no `Rational`), `Real` has no `toRational`, and `Floating`/`RealFrac` are not yet classes.
3. **No `Char` type.** Character operations use string functions from `LString`.
4. **No multi-parameter typeclasses.**

5. **No lazy I/O**. File operations read eagerly.
6. **No** `IORef`/`MVar`/`STRef`. Use `STArray` for scoped mutation.
7. **All top-level bindings need type signatures**.
8. **Single-file modules**. Each `.mll` file is one module.

17 Complete Example: AVL Dictionary

```
data Dict k v = Empty
              | Node Int k v (Dict k v) (Dict k v)

empty :: Dict k v
empty = Empty

height :: Dict k v -> Int
height Empty = 0
height (Node h _ _ _ _) = h

node :: k -> v -> Dict k v -> Dict k v -> Dict k v
node k v l r = Node (1 + max (height l) (height r)) k v l r

insert :: k -> v -> Dict k v -> Dict k v
insert k v Empty = node k v Empty Empty
insert k v (Node h nk nv left right)
    | k < nk    = rebalance (node nk nv (insert k v left) right)
    | k > nk    = rebalance (node nk nv left (insert k v right))
    | otherwise = Node h k v left right

lookup' :: k -> Dict k v -> Maybe v
lookup' _ Empty = Nothing
lookup' k (Node _ nk nv left right)
    | k < nk    = lookup' k left
    | k > nk    = lookup' k right
    | otherwise = Just nv

main :: IO ()
main = do
    let d = insert "c" 3 $ insert "b" 2 $ insert "a" 1 $ empty
    putStrLn (show (lookup' "b" d))   -- Just 2
    putStrLn (show (lookup' "z" d))   -- Nothing
```

(Full rebalancing omitted for brevity — see `experiments/dict.mll`.)

18 Complete Example: Lazy Fibonacci Export

A library module that exports an efficient Fibonacci function to Lua:

```
fib :: [Int]
fib = 1 : 1 : zipWith (+) fib (tail fib)

export fibonacci :: Int -> [Int]
fibonacci = flip take fib
```

Usage from Lua:

```
local m = require("fib")
for i, v in ipairs(m.fibonacci(20)) do
```

```
    print(i, v)
end
```

The infinite list is computed lazily; only the requested elements are evaluated.

19 Prelude Reference

19.1 Functions

```
-- Output
putStrLn :: String -> IO ()
putStr   :: String -> IO ()

-- Conversion
show :: Show a => a -> String
read :: Read a => String -> a

-- Combinators
id    :: a -> a
const :: a -> b -> a
flip  :: (a -> b -> c) -> b -> a -> c
(.)   :: (b -> c) -> (a -> b) -> a -> c
($)   :: (a -> b) -> a -> b

-- Arithmetic (numeric-class methods)
(+), (-), (*)      :: Num a => a -> a -> a
negate, abs, signum :: Num a => a -> a
(/), recip         :: Fractional a => a -> a  -- Number, not Int
div, mod, quot, rem :: Integral a => a -> a -> a  -- Int

-- Comparison (require Eq/Ord instances)
(==), (/=) :: Eq a  => a -> a -> Bool
(<), (>), (<=), (>=) :: Ord a => a -> a -> Bool
max, min   :: Ord a => a -> a -> a
compare    :: Ord a => a -> a -> Ordering

-- Boolean
(&&), (||) :: Bool -> Bool -> Bool
not        :: Bool -> Bool
otherwise  :: Bool    -- = True

-- List operations
map       :: (a -> b) -> [a] -> [b]
filter    :: (a -> Bool) -> [a] -> [a]
foldl     :: (b -> a -> b) -> b -> [a] -> b
foldr     :: (a -> b -> b) -> b -> [a] -> b
head      :: [a] -> a
tail      :: [a] -> [a]
take      :: Int -> [a] -> [a]
drop      :: Int -> [a] -> [a]
length    :: [a] -> Int
reverse   :: [a] -> [a]
elem      :: Eq a => a -> [a] -> Bool
zipWith   :: (a -> b -> c) -> [a] -> [b] -> [c]
concatMap :: (a -> [b]) -> [a] -> [b]
(++)      :: [a] -> [a] -> [a]
```

```
-- String
(<>)      :: String -> String -> String

-- Tuple
fst :: (a, b) -> a
snd :: (a, b) -> b

-- Control
seq       :: a -> b -> b
error     :: String -> a
undefined :: a

-- Monadic
when     :: Bool -> IO () -> IO ()
mapM_    :: (a -> IO ()) -> [a] -> IO ()
mapM     :: Monad m => (a -> m b) -> [a] -> m [b]
sequence :: Monad m => [m a] -> m [a]
```

`mapM`/`sequence` (and `forM = flip mapM` in `Control.Monad`) collect their results and work in any monad — `IO`, `Maybe`, `Either`, or lists. The underscore variants (`mapM_`, `sequence_`, `forM_`) discard the results and run in `IO`.

19.2 Types

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
data Ordering = LT | EQ | GT
data ExitValue = Normal | Err Int
```