This Is Aiki

The Aiki-1 Alpha Release

William D. Senn – will.senn@gmail.com

August 2026

Table of Contents

About This Document

The implementation described in this document corresponds to the initial alpha release of Aiki, commit XXX. Development after that point may change the language, libraries, tooling, or implementation described here.

Aiki is implemented in Go and was developed on Linux for Linux. The implementation is structured so that much of the language is separated from the host through a runtime abstraction layer, and there is no obvious architectural reason it could not be ported to macOS or Windows. That work has not been done. The current implementation, tooling, filesystem assumptions, graphics support, and host integration should therefore be regarded as Linux-specific.

It introduces the language from the outside in: first the visible programming model, then values, control, functions, data, modules, errors, concurrency, and graphics; after that, it describes the features Aiki deliberately omits, the architecture beneath the language, the current state of the implementation, and the role of generative AI in its development.

Because Aiki is a research artifact as well as a programming language, this document also records the current shape of the system at a particular stage of its development. Examples should therefore be read as descriptions of the alpha implementation documented here rather than as permanent guarantees about future releases.

1. Aiki in Brief

Aiki is a small programming language and interactive programming environment. It has a deliberately compact language surface: fourteen keywords, seven basic types, eleven symbolic operators, a small set of control structures, functions, lists, shapes, and a standard prelude. Additional facilities such as files, strings, mathematics, regular expressions, time, bit operations, explicit stores, testing, graphics, and turtle programming are provided through modules.

An Aiki program is made from values and expressions. Values can be bound to names, and bindings can subsequently be changed:

```
let x = 10
let greeting = "hello"

x = 20
```

The seven basic types are `number`, `boolean`, `rune`, `string`, `symbol`, `list`, and `function`. A `rune` is Aiki's fancy name for a character. Like Go, Aiki is Unicode-native, so a rune represents one Unicode character. Several basic types have direct literal forms:

```
42          # number
true        # boolean
'a'         # rune
"hello"     # string
:ready      # symbol
```

Lists provide Aiki's principal compound data form:

```
let numbers = [1, 2, 3]
```

A list can also be given a shape, which names positions within the list:

```
let @point [x, y]
let dot = [@point, 10, 20]

dot.x
dot.y
```

The value remains list data, but its components can be addressed by name.

Functions are values too. A function is ordinarily given a name by binding it:

```
let square = (x) {
    return x * x
}

square(5)
```

Expressions evaluate from left to right. Aiki does not impose the usual arithmetic hierarchy in which multiplication automatically takes precedence over addition. Thus:

```
2 + 3 * 4
```

means:

```
(2 + 3) * 4
```

Parentheses make a different grouping explicit:

```
2 + (3 * 4)
```

The same preference for visible relationships appears throughout the language. Conditional execution uses `if`, with an optional `else`; iteration uses `while`; `match` provides pattern-directed control; and `return` returns a value from a function.

```
if x > 10 {
    println("large")
} else {
    println("small")
}

while x > 0 {
    println(x)
    x = x - 1
}
```

Operations can also be combined with a pipeline:

```
let double = (x) {
    return x * 2
}

let add_one = (x) {
    return x + 1
}

5 |> double |> add_one
```

The result flows from left to right through the functions in the pipeline. Here, `5` becomes `10`, then `11`.

A small prelude is available automatically. It provides basic input and output, list operations, type inspection and conversion, concurrency, loading, and interaction with the live environment:

```
println("Hello")
first([1, 2, 3])
type(42)
to_str(42)
```

More specialized facilities are supplied as modules. A program can import a module as a value, import selected names, or bring all of a module's exported names into the current environment. For example:

```
use("turtle/simple")

new(800)
forward(50)
```

new(800) opens an 800 × 800 turtle window, and forward(50) moves the turtle forward and draws a line. The supplied modules provide facilities including files, strings, mathematics, random numbers, regular expressions, hashing, bit operations, bytes, time, explicit stores, testing, canvas graphics, and turtle graphics.

Aiki is also an interactive environment. Running `aiki` starts the REPL, where expressions can be entered and evaluated immediately. A source file is run with:

```
aiki program.ai
```

and an expression can be evaluated directly with `-e`. The live environment provides `help()` and `doc()` for discovering the language and its modules, along with operations such as `reset()`, `delete()`, and `quit()`.

1.1 Exploring the REPL

Aiki includes built-in help and documentation.

```
> help()
Aiki Help

Syntax:
Statements: let, if, while, match, select, return
Expressions: +, -, *, /, <, >, <=, >=, and, or, not, |>
Values: number, string, rune, symbol, list, function

Prelude Functions:
IO: input, print, println, read
List: append, empty, first, length, prepend, rest
Type: chr, equal, inspect, is_error, ord, shape, to_decimal, to_number, to_str,
type
Concurrency: channel, recv, send, spawn
REPL: apply, delete, doc, help, load, quit, reset, stack_limit
Other: abs, truncate

Modules:
bits, bytes, bytes/ffi, bytes/native, canvas, file, hash, hash/ffi, hash/native,
list, math, math/ffi, math/native, random, regex/ffi, store, string, test, time,
turtle, turtle/simple

Use help("name") for prelude functions, help("module") for module info,
help("module.func") for module functions.
Use doc("name") for full documentation.
```

`help()` expects a string when asking about a particular name:

```
> help(first)
:148:in 'help': help: expected string, got function
_help(first(args))
from :1:in ''
```

Use:

```
> help("first")
first
```

```
  Returns first element of list.

Syntax: first(list)
```

For fuller documentation:

```
> doc("first")
first

Returns the first element of a list.

first(list)

first([1, 2, 3])      # 1
first(["a", "b"])     # "a"
```

Long help and documentation output is paged when Aiki is running interactively in a terminal; short output and redirected output are written directly.

Bindings can be removed from the current environment with `delete()`:

```
> let temp = 123
> temp
123
> delete("temp")
true
> temp
:1:in '': undefined: temp
```

`reset()` clears the current user environment and restores a fresh session:

```
> reset()
Environment reset.
> x
:1:in '': undefined: x
```

1.2 Tooling

Aiki comes with its own supporting tools for writing, checking, and exercising programs. The command structure is deliberately simple: aiki fmt, aiki lint, aiki test, aiki smoke, and aiki enginesmoke. This organization is inspired by Go's command-line tooling, where a single language command provides a consistent entry point to the tools used with the language.

`aiki fmt` rewrites Aiki source into the language's canonical format. `aiki lint` checks source structurally and can report problems such as undefined names and naming violations without executing the program.

aiki test runs Aiki-native tests using the supplied test module. aiki smoke runs Aiki programs against expected transcripts, including expected input, output, errors, exit status, and, where needed, a display check. aiki enginesmoke works one level lower, exercising the lexer, parser, evaluator, and grammar independently; its coverage mode also verifies that every EBNF production is exercised by at least one structural specimen.

The same `aiki` command therefore starts the interactive environment, runs source files, evaluates expressions, formats and checks source, and tests behavior at both the program and language-engine levels.

1.3 How to Read Aiki

Aiki has few syntactic forms. Values are bound to names. Functions are values. Lists are the principal compound data form. Expressions proceed from left to right, and parentheses state grouping explicitly. The prelude supplies the small default vocabulary; modules extend it with additional capabilities. Most of the language can therefore be approached by learning a small vocabulary and then combining its parts.

1.4 Aiki Language Card

```
Keywords (14) - let if else while match select default return true false and or
not package
Basic Types (7) - number boolean rune string symbol list function
System Types (6) - bytes file channel module canvas store
Literals - 42  true  false  'a'  "abc"  :foo
Bindings - let x = expr | x = expr
Operators (11) - + - * / < > <= >= |> . =
Expressions - literals names calls indexing field access unary binary pipeline
grouping
Evaluation - left-to-right; parentheses group
Control - if expr {...} else {...} | while expr {...} | match expr {...} | select
{...} | return expr
Functions - let f = (args) {...} | let f = (first, ...rest) {...} | f(args)
Data - list: [a, b, c] | index: xs[i] | shape: let @point [x, y] |
       shaped: [@point, x, y] | field: p.x
Primitives (3) - import use export
Prelude (34) - apply load stack_limit | spawn channel send recv |
               print println read input |
               first rest length prepend append empty |
               type inspect equal ord chr shape is_error |
               to_str to_decimal to_number |
               abs truncate | help doc reset delete quit
Modules - package "name" | import("name") | use("name") | export(:name)
Errors - error values | is_error(value)
Interaction - help("name") | doc("name") | reset() | delete("name") | quit()
Additional Capabilities - files | strings | math | random | regex | hash | bits |
store |
bytes | time | canvas | turtle | testing
Tooling - aiki | -e | fmt | lint | test | smoke | enginesmoke
```

2. Values, Names, and Evaluation

Aiki evaluates expressions to produce values. Values are the things a program computes with: numbers, Boolean values, characters, strings, symbols, lists, and functions. A value can be used directly, passed to a function, placed in a list, returned from a function, or bound to a name.

2.1 Values

Aiki has seven basic language types:

```
number
boolean
rune
string
symbol
list
function
```

The running system also works with values needed by facilities outside the basic language, including bytes, files, channels, modules, canvases, and stores. Recoverable errors are represented separately as shaped list values. These do not enlarge the basic seven-type language surface; they are supplied by the runtime and its modules.

2.2 Exact Numbers

Aiki has one numeric type, `number`. Numbers are represented as arbitrary-precision rational values, so ordinary arithmetic is exact rather than floating-point.

Integers, decimals, and fractions are simply different ways of writing numbers:

```
42
0.1
3.14
1/3
```

At the REPL, Aiki shows their exact values:

```
> 0.1
1/10
> 0.1 + 0.2
3/10
> 3.14
157/50
> 1/3
1/3
```

For comparison, Python uses binary floating-point numbers for ordinary decimal arithmetic:

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

Aiki keeps the same computation exact:

```
> 0.1 + 0.1 + 0.1
3/10
```

The difference is not formatting. In Aiki, `0.1` is represented as the exact rational value `1/10`, so adding it three times produces exactly `3/10`.

A fraction written without spaces, such as `1/3`, is a single number literal. This matters because `/` is also an operator. For example:

```
1 / 3 + 1 / 3 + 1 / 3
```

contains division and addition operators. Because Aiki evaluates expressions from left to right, it produces:

```
13/27
```

By contrast:

```
1/3 + 1/3 + 1/3
```

contains three rational literals and produces:

```
1
```

Both use exact arithmetic, but only the first expression participates in Aiki's normal left-to-right operator evaluation.

There is no ordinary fixed-width integer limit in Aiki's core numeric representation. Numerators and denominators use arbitrary-precision integers, so arithmetic is limited by available resources rather than by a 32- or 64-bit integer boundary.

Some mathematical operations cannot in general produce rational results. Operations such as square roots of nonsquares and trigonometric functions therefore require approximation. The math/ffi module obtains such approximations through host floating-point mathematics. A returned float64 is converted at the FFI boundary to the exact rational representation of that IEEE-754 binary64 value; thereafter it is an ordinary Aiki number and carries no retained approximation or provenance flag. math/native instead computes approximations using

Aiki rational arithmetic with finite iterative methods. Exact rational arithmetic remains Aiki's normal numeric model; the module chosen identifies where approximation occurred.

2.3 Booleans

Aiki's Boolean values are `true` and `false`. Boolean expressions use `and`, `or`, and `not`:

```
true and false
true or false
not true
```

Boolean contexts use truth rather than requiring a Boolean value. `false`, the empty list, and shaped `@error` values are falsy; all other values are truthy. In particular, `0` and the empty string are truthy. Truth interpretation does not change the value's type.

`and` and `or` operate on those same truth rules and return one of their operand values rather than converting the result to Boolean. Both operands are evaluated before the operator is applied: `and` returns its left operand when that value is falsy and otherwise returns its right operand; `or` returns its left operand when that value is truthy and otherwise returns its right operand. They do not short-circuit operand evaluation. `not`, by contrast, always produces true or false.

```
0 and 42       # 42
[] or 42       # 42
not 0          # false
```

2.4 Runes, Strings, and Symbols

A `rune` is Aiki's fancy name for a character. Like Go, Aiki is Unicode-native. A rune represents a Unicode character rather than simply an 8-bit character:

```
'a'
'λ'
'日'
```

Strings are written with double quotes:

```
"hello"
"Hello, 世界"
```

Runes and strings are different types. `'a'` is a rune; `"a"` is a string.

Symbols are names that are themselves values. They begin with `:`:

```
:ready
:red
:north
```

Given:

```
let color = :red
```

`color` is a name bound to a value, while `:red` is the value itself. Symbols are useful wherever a program needs a small, readable value that identifies something without treating it as text.

2.5 Lists and Functions Are Values Too

A list collects values and may contain values of different types:

```
[1, 2, 3]
["red", "green", "blue"]
[1, "two", :three]
```

Lists are Aiki's principal compound data structure and receive fuller treatment later.

A function is also a value:

```
(x) {
    return x * x
}
```

It can be given a name using the same binding mechanism as any other value:

```
let square = (x) {
    return x * x
}
```

There is no separate named-function declaration. `let` simply binds the function value to `square`, just as it can bind `42` to answer or `:red` to `color`.

2.6 Names and Bindings

`let` creates a binding:

```
let answer = 42
let message = "hello"
```

The name can then be used wherever its value is needed:

```
answer + 8
println(message)
```

Assignment changes an existing binding:

```
answer = 43
```

Thus:

```
let x = 10
x = 20
```

first creates `x`, then changes its value.

Names live in lexical environments. A function creates an enclosed environment for its parameters and local bindings. When Aiki looks up a name, it first looks in the current environment and then outward through enclosing environments.

This allows a function to use a value from the environment in which it was created:

```
let amount = 10

let add_amount = (x) {
    return x + amount
}
```

```
add_amount(5)
```

The result is `15`.

The interactive user environment is enclosed by the prelude environment. This is why names such as `println`, `first`, and `type` are available without being defined by the user.

New vocabulary does not require new syntax. Because functions are values, existing operations may be renamed, captured, passed, and composed using ordinary bindings.

2.7 Shadowing and the Live Environments

Aiki permits ordinary bindings to shadow names from enclosing environments, including names supplied by the prelude and by modules brought into the current environment with `use`. Shadowing produces a warning for prelude functions but is not prohibited.

```
let original_type = type

let type = (x) {
    return :whatever
}

type(42)
:whatever

original_type(42)
:number
```

The original binding remains available if it was captured before being shadowed. Removing the local binding exposes the enclosing binding again:

```
delete("type")
type(42)
:number
```

Shadowing is lexical. A binding introduced inside a function affects only that function environment and environments enclosed by it.

```
let f = () {
    let type = (x) {
        return :local
    }

    return type(42)
}

f()
:local

type(42)
:number
```

Names that resemble HAL primitives are not reserved in user code. A user may define `_first`, for example, but that binding does not replace the HAL primitive used internally by the prelude:

```
let _first = (xs) {
    return :mine
}

_first([9, 8, 7])
:mine
```

```
first([9, 8, 7])
9
```

The distinction is environmental rather than lexical. Much of Aiki's visible vocabulary is therefore negotiable within ordinary lexical scope, while the runtime boundary beneath the prelude remains protected.

2.8 Expressions

An expression is something Aiki can evaluate to obtain a value. Expressions include literals and names, operations, function calls, indexing, field access, pipelines, and explicit grouping:

```
42
x
x + 1
square(5)
xs[0]
dot.x
5 |> double
(x + 1)
```

These forms can be combined wherever a value-producing computation is expected.

Unary operations are `-` and `not`:

```
-10
not true
```

Binary operations include arithmetic, comparison, and Boolean operations:

```
10 + 5
10 < 20
true and false
```

2.9 Left-to-Right Evaluation

Aiki evaluates binary expressions from left to right. There is no arithmetic precedence hierarchy.

Thus:

```
2 + 3 * 4
```

means:

```
(2 + 3) * 4
```

and produces `20`.

If a different grouping is intended, it is written explicitly:

```
2 + (3 * 4)
```

which produces `14`.

This rule applies across Aiki's binary operators. The written sequence determines the evaluation sequence unless parentheses explicitly group part of the expression. Parentheses therefore do not override a hidden precedence table; there is no precedence table to remember.

2.10 Pipelines

The pipeline operator, `|>`, provides another explicitly left-to-right form of evaluation. The value on the left becomes the first argument to the function on the right:

```
let double = (x) {
    return x * 2
}

let add_one = (x) {
    return x + 1
}

5 |> double |> add_one
```

The computation reads in the same direction it executes: start with `5`, pass it to `double`, then pass that result to `add_one`. The final value is `11`.

Values, names, expressions, and left-to-right evaluation form the basic computational vocabulary of Aiki.

3. Control, Functions, and Data

Aiki has a small set of control structures, one function syntax, and one principal compound data structure. The same values and expressions introduced in the previous section are used throughout.

3.1 Conditional Control

`if` evaluates an expression and executes a block when the result is true:

```
let temperature = 72

if temperature > 70 {
    println("warm")
}
```

An optional `else` supplies the alternative:

```
if temperature > 70 {
    println("warm")
} else {
    println("cool")
}
```

An `if` can be nested inside another when one decision genuinely depends on another:

```
let logged_in = true
let is_admin = true

if logged_in {
    if is_admin {
        println("admin")
    }
}
```

Here, checking `is_admin` only makes sense after `logged_in` is known to be true. Nesting expresses that dependency directly.

3.2 Matching

For choosing among alternative cases, Aiki provides `match`. There is no separate `switch` construct`;` `match` fills that role and also supports pattern-directed selection.

```
let @ok [value]
let @error [message]

let result = [@ok, 42]

match result {
    [@ok, value] {
        println(value)
    }

    [@error, message] {
        println(message)
    }

    _ {
        println("unknown")
    }
}
```

This prints:

```
42
```

A pattern can be a wildcard (`_`), a name that binds the matched value, a literal value, a list pattern, or a shaped-list pattern. match can therefore select a case and take structured data apart at the same time.

`if` and `match` serve different purposes. Use `if` when the question is whether a condition is true. Use `match` when the question is which value or structure has been received.

3.3 Iteration

`while` repeats a block while its condition remains true:

```
let n = 3

while n > 0 {
    println(n)
    n = n - 1
}
```

This prints:

```
3
2
1
```

Aiki has no `for` statement. Repetition can be expressed directly with `while`, recursively with functions, or through higher-order operations over lists.

3.4 Functions

As established earlier, functions are values. A function is written:

```
(x) {
    return x * x
}
```

and can be bound to a name with `let`:

```
let square = (x) {
    return x * x
```

```
}
```

Functions are called with parentheses:

```
square(5)
```

Arguments are expressions, and functions may accept several parameters:

```
let add = (a, b) {
    return a + b
}

add(10, 20)
```

Because functions are values, they can also be passed as arguments, placed in lists, returned from other functions, and called recursively. Recursion requires no special syntax; a function simply calls its own bound name like any other function.

3.5 Rest Parameters

A function can collect additional arguments into a list with a rest parameter:

```
let gather = (first, ...rest) {
    return rest
}
```

Calling:

```
gather(1, 2, 3, 4)
```

binds `first` to `1` and `rest` to:

```
[2, 3, 4]
```

3.6 Return and Guard Clauses

`return` ends the current function and supplies its result:

```
let larger = (a, b) {
    if a > b {
        return a
    }

    return b
}
```

When a function needs to consider several independent conditions, successive `if` statements with early `return` can be clearer than deep nesting:

```
let sign = (x) {
    if x < 0 {
        return :negative
    }

    if x > 0 {
        return :positive
    }

    return :zero
```

```
}
```

Nesting is useful when one decision depends on another. Guard clauses are useful when several cases can be disposed of in sequence.

3.7 Lists

Lists are Aiki's principal compound data structure and can contain values of different types, including other lists and functions:

```
[1, 2, 3]
[1, "two", :three]
[1, [2, 3], "four"]
```

Elements are accessed by zero-based index:

```
let xs = [10, 20, 30]

xs[0]
xs[1]
xs[2]
```

which produce `10`, `20`, and `30`.

The prelude provides the basic operations for constructing and decomposing lists:

```
first([1, 2, 3])
rest([1, 2, 3])
length([1, 2, 3])
prepend([1, 2, 3], 0)
append([1, 2, 3], 4)
empty([])
```

These produce:

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

Together, `first` and `rest` expose the recursive structure of list data directly.

Lists remain immutable. Systems programs that require explicit mutable indexed storage use the store module, which produces a distinct store value rather than changing list semantics.

3.8 Shapes

Aiki can give names to positions in a list by defining a shape:

```
let @point [x, y]
```

A shaped list is constructed by placing the shape at the beginning of the list:

```
let p = [@point, 10, 20]
```

Its fields can then be addressed by name:

```
p.x
```

```
p.y
```

which produce `10` and `20`.

The underlying value remains list data. A shape does not introduce a separate object or class hierarchy; it gives names to positions in a list. Shapes can also appear in patterns, allowing structured values to be recognized and unpacked directly by `match`.

3.9 Processing Lists

Aiki supports several ways of expressing computation over the same list data: explicit iteration, structural recursion, tail recursion, and higher-order list operations.

An iterative sum can traverse a list explicitly:

```
let xs = [1, 2, 3, 4]
let total = 0

while not empty(xs) {
    total = total + first(xs)
    xs = rest(xs)
}

total
```

The result is `10`.

The same computation can follow the recursive structure of the list directly:

```
let sum = (xs) {
    if empty(xs) {
        return 0
    }

    return first(xs) + sum(rest(xs))
}

sum([1, 2, 3, 4])
```

Again, the result is `10`. The base case handles the empty list; otherwise the first value is added to the sum of the rest.

This form is recursive, but it is not tail recursive. In:

```
return first(xs) + sum(rest(xs))
```

the addition still has to be performed after the recursive call returns. Each invocation therefore has unfinished work to retain.

The same computation can instead be written with an accumulator:

```
let sum_tail = (xs, total) {
    if empty(xs) {
        return total
    }

    return sum_tail(rest(xs), total + first(xs))
}

sum_tail([1, 2, 3, 4], 0)
```

The result is again `10`, but now the recursive call is the final action:

```
return sum_tail(rest(xs), total + first(xs))
```

Nothing remains for the caller to do after that call. Aiki recognizes calls in tail position and optimizes them so repeated tail calls do not consume additional ordinary call-stack frames.

The same sum can also be expressed with a higher-order operation from the `list` module:

```
let list = import("list")

let add = (a, b) {
    return a + b
}

list.reduce([1, 2, 3, 4], 0, add)
```

The result is again `10`.

The `list` module also supplies `map`, `filter`, and `each`, along with operations such as `sum`, `max`, `min`, `reverse`, `find`, `any`, `all`, `concat`, and `range`.

The list `[1, 2, 3, 4]` is the same data whether it is processed with `while`, recursion, tail recursion, or a higher-order operation. What changes is the relationship being expressed. Iteration and recursion are not separate facilities with separate data structures; they are different ways of describing processes over the same values.

4. Prelude and Modules

Aiki keeps the automatically available environment small. Common operations needed for ordinary computation are provided by the prelude; more specialized facilities are supplied through modules. The distinction is visible in the distribution:

```
$AIKIHOME/
├── lib/                        supplied modules
└── engine/runtime/prelude/   automatically loaded prelude
```

4.1 The Prelude

The prelude is loaded automatically, so its functions can be used without importing anything:

```
println("hello")
length([1, 2, 3])
type(42)
```

The current prelude contains 34 functions.

Basic loading and application:

`apply` `load` `stack_limit`

Concurrency:

`spawn` `channel` `send` `recv`

Input and output:

`print` `println` `read` `input`

Basic list operations:

`first` `rest` `length` `prepend` `append` `empty`

Type inspection and conversion:

`type` `inspect` `equal` `ord` `chr` `shape` `is_error` `to_str` `to_decimal` `to_number`

General numeric operations:

`abs` `truncate`

Interaction with the live environment:

`help` `doc` `reset` `delete` `quit`

The prelude is deliberately limited. It contains the small set of operations Aiki expects to be immediately useful for computation and interaction rather than attempting to provide every facility as a built-in operation.

4.2 Modules

Facilities beyond the prelude are organized into modules. A module identifies its package:

```
package "mylib"
```

defines ordinary Aiki values:

```
let double = (x) {
    return x * 2
}

let triple = (x) {
    return x * 3
}
```

and makes selected names available outside the module with `export`:

```
export(:double, :triple)
```

The exported names form the module's public surface.

4.3 Importing and Using Modules

A module can be imported as a module value:

```
let list = import("list")
```

Its exported values are then accessed through that value:

```
list.sum([1, 2, 3, 4])
list.reverse([1, 2, 3])
```

Selected names can also be imported directly:

```
import("list", :sum, :reverse)
```

Aiki additionally provides `use` for bringing all exported names into the current environment:

```
use("turtle/simple")

new(800)
forward(50)
left(90)
forward(50)
```

The difference is visible in the source:

```
let list = import("list")
list.sum([1, 2, 3, 4])
```

keeps the module relationship explicit, while:

```
use("list")
sum([1, 2, 3, 4])
```

brings the exported names into the current environment.

Within a module, `export` identifies the names that form its external interface:

```
let square = (x) {
    return x * x
}

export(:square)
```

Defining a value inside a module does not by itself make it public.

`import`, `use`, and `export` are not members of the 34-function prelude. They belong to Aiki's module machinery.

4.4 Supplied Modules

The distribution includes modules for:

bits bytes bytes/ffi bytes/native canvas file hash hash/ffi hash/native list math math/ffi math/native random regex/ffi store string test time turtle turtle/simple

Repository test fixtures may also be discoverable when the source tree is the current working directory, but they are not part of the supplied library surface.

The /ffi and /native distinction concerns implementation rather than a different source-level module mechanism. From an Aiki program, both are modules and are accessed through the same facilities. When X/native exists and no explicit bare package X exists, the bare name X resolves to X/native; /ffi variants are never selected implicitly. The supplied bytes, hash, and math families therefore have native defaults while their /ffi variants remain explicit.

bits operates on non-negative integral numbers without introducing a machine-width integer type. store provides fixed-size mutable indexed storage as an explicit system capability. test supplies assertions and named cases for aiki test.

For example, import("math") selects math/native in the supplied distribution, while import("math/ffi") requests the host-backed implementation explicitly.

4.5 The List Module

The prelude provides the structural list operations:

`first` `rest` `length` `prepend` `append` `empty`

The `list` module adds a richer list-processing vocabulary:

`map` `filter` `reduce` `each` `sum` `max` `min` `reverse` `find` `any` `all` `concat` `range`

For example:

```
let list = import("list")

let double = (x) {
    return x * 2
```

```
}

list.map([1, 2, 3, 4], double)
```

produces:

```
[2, 4, 6, 8]
```

This division is characteristic of Aiki. The prelude supplies the small structural vocabulary needed to work with lists directly; the module provides a richer vocabulary for larger list-processing tasks.

4.6 Discovering Modules

The live environment can be used to discover supplied modules and their operations:

```
help()
help("list")
doc("list")
help("turtle/simple")
doc("turtle/simple")
```

Modules therefore extend Aiki without introducing a separate programming model. Module functions remain functions, module values remain values, and exported facilities operate on the same numbers, strings, lists, functions, and other values used by the rest of the language.

5. Errors and Faults

Aiki distinguishes between two kinds of failure: **faults** and **shaped errors**. A fault means evaluation cannot continue correctly. A shaped error is ordinary recoverable data that a program can inspect, return, match, and propagate.

5.1 Faults

A fault stops the current evaluation. Typical causes include:

- wrong number of arguments
- type mismatch
- undefined name
- division by zero
- index out of bounds
- stack overflow

For example:

```
let add = (a, b) {
    return a + b
}

add(1)
```

produces a diagnostic such as:

```
:1:in '': add: want 2 arguments, got 1
add(1)
```

The call does not produce a value that the program can handle; evaluation faults instead.

5.2 Shaped Errors

Failures that a program may reasonably want to handle can instead be represented as shaped list values. The conventional form is:

```
[@error, :kind, "message"]
```

For example:

```
[@error, :io, "file not found"]
[@error, :parse, "invalid number"]
[@error, :lookup, "not found"]
```

These are ordinary shaped lists. The `@error` portion identifies the shape; it is metadata associated with the list rather than an indexed element.

Thus, for:

```
let result = [@error, :math, "division by zero"]
```

the indexed elements are:

```
result[0]
result[1]
```

which produce:

```
:math
division by zero
```

There is no `result[2]`; attempting to access it faults with an index-out-of-bounds error. This is the same arrangement used by shaped lists generally: in `[@point, 10, 20]`, the shape is `@point`, while the indexed data are `10` and `20`.

5.3 Returning and Testing Errors

A function can construct and return a shaped error directly:

```
let divide = (a, b) {
    if equal(b, 0) {
        return [@error, :math, "division by zero"]
    }

    return a / b
}
```

Calling:

```
divide(10, 0)
```

returns:

```
[@error, :math, "division by zero"]
```

The division itself is never attempted. Because the returned value is ordinary data, the caller remains in control of what happens next.

The prelude provides `is_error()` for recognizing shaped errors:

```
let result = divide(10, 0)

if is_error(result) {
    println(result[0])
    println(result[1])
}
```

which prints:

```
:math
division by zero
```

5.4 Matching Errors

Because errors are shaped lists, they work naturally with `match`:

```
let result = divide(10, 0)

match result {
    [@error, kind, message] {
        println(kind)
        println(message)
    }

    value {
        println(value)
    }
}
```

The first pattern recognizes the `@error` shape and simultaneously binds its two elements.

Modules can use the same representation for failures that callers may reasonably want to handle:

```
[@error, :io, "file not found"]
[@error, :lookup, "not found"]
[@error, :parse, "invalid input"]
```

A shaped error remains an ordinary value. It can be bound, returned, passed to another function, inspected, or matched. Pipelines recognize shaped errors specially: when a stage produces a shaped `@error` value, the pipeline stops and returns that value without invoking subsequent stages.

5.5 Faults and Shaped Errors

The distinction can be summarized as:

	Fault	Shaped error
Stops current evaluation	Yes	No
Ordinary program data	No	Yes
User can construct	No	Yes
Can be returned	No	Yes
Can be inspected	No	Yes
Can be matched	No	Yes
Representation	runtime failure	shaped list

A fault means that Aiki cannot correctly continue the requested evaluation. A shaped error means that the computation completed by producing a value that represents failure.

Recoverable errors therefore remain visible, structured, and manipulable using the same lists, shapes, functions, and control structures used everywhere else in Aiki.

6. Concurrency

Aiki provides a complete but deliberately parsimonious concurrency model built on Go's goroutines and channels. Four prelude functions provide spawning and point-to-point communication, and the select statement coordinates alternative receives:

`spawn` `channel` `send` `recv`

spawn() creates an independently running computation. channel() creates a synchronous communication channel. send() and recv() exchange values through that channel by rendezvous. select waits among alternative receives without introducing a task or future abstraction. The surface is small, but the model is not illustrative or simulated: spawned Aiki functions run concurrently in Go goroutines, Aiki channels are backed by unbuffered Go channels, and spawned Aiki functions execute in isolated Aiki environments.

6.1 Spawning a Function

`spawn()` launches an Aiki function concurrently:

```
let worker = () {
    println("worker ran")
}

spawn(worker)
```

The call returns immediately:

```
true
```

while the worker proceeds independently:

```
worker ran
```

Concurrent output is asynchronous. In the REPL, output from a spawned function may appear after the next prompt has already been printed.

Values needed by a spawned computation are supplied explicitly:

```
let worker = (x) {
    println(x)
}

spawn(worker, 42)
```

Several arguments can be supplied, and the ordinary parameter and rest-parameter rules apply.

6.2 Isolation

A spawned function runs in an isolated Aiki environment. It does not inherit ordinary user bindings from the environment in which the function was created.

For example:

```
let x = 42
```

```
let worker = () {
    println(x)
}

spawn(worker)
```

returns:

```
true
```

but the spawned computation faults because `x` was not explicitly supplied:

```
spawn: ... undefined: x
```

The implementation creates a fresh environment enclosed by the prelude and binds only the arguments supplied to `spawn()`. The function's ordinary captured user environment is not reused for spawned execution.

Values therefore cross into concurrent computations explicitly:

```
let worker = (x) {
    println(x)
}

spawn(worker, 42)
```

This isolates concurrent computations from implicit shared user state.

6.3 Channels and Rendezvous

`channel()` creates a channel:

```
let ch = channel()
```

A channel is a system value:

```
ch
type(ch)
```

produce:

```
<channel>
:channel
```

The underlying runtime representation contains an unbuffered Go channel carrying Aiki values. Communication is therefore synchronous: a sender waits until a receiver is available, and a receiver waits until a sender is available.

A typical sender runs concurrently:

```
let ch = channel()

let sender = (ch) {
    println("sender started")
    send(ch, 42)
    println("sender finished")
}

spawn(sender, ch)
```

The sender reaches `send()` and waits:

```
true
sender started
```

The receiving computation can then execute:

```
recv(ch)
```

producing:

```
42
sender finished
```

A successful send and receive are therefore both communication and synchronization:

```
sender                          receiver

send(ch, value)                 recv(ch)
       \                          /
        \                        /
             rendezvous
                 |
               value
```

After the rendezvous, `recv(ch)` returns the transmitted value, `send(ch, value)` returns `true`, and both computations continue independently.

6.4 Communicating Results

`spawn()` returns `true`, not the eventual result of the spawned function. Results are communicated explicitly through channels:

```
let result = channel()

let square = (out, x) {
    send(out, x * x)
}

spawn(square, result, 5)

recv(result)
```

produces:

```
25
```

The same mechanism can signal completion. Aiki does not expose a separate task handle, `join`, or `await`; when one computation needs to know that another has finished, a channel can carry that information.

6.5 Channels Carry Aiki Values

Channels carry ordinary Aiki values: numbers, strings, symbols, lists, shaped values, and other values that can be supplied to `send()`.

```
let @result [value]

send(ch, [@result, 42])
```

The receiver can process the value with ordinary Aiki mechanisms:

```
let message = recv(ch)

match message {
    [@result, value] {
        println(value)
    }
}
```

Concurrency therefore does not introduce a separate message representation. Channels themselves are values as well, so they can be passed as arguments and used to build larger communication arrangements.

6.6 Persistent Concurrent Processes

A spawned function can remain active and repeatedly communicate through channels:

```
let worker = (input, output) {
    while true {
        let x = recv(input)
        send(output, x * 2)
    }
}
```

Once spawned:

```
let input = channel()
let output = channel()

spawn(worker, input, output)
```

the function behaves as a persistent concurrent process. This pattern is sufficient to construct producers, consumers, worker pipelines, request/reply protocols, and other arrangements of communicating processes.

6.7 Selecting among Receives

select waits until one of its receive cases can proceed. A case may bind the received value for the selected arm:

```
select {
let value = recv(left) {
println("left: ", value)
}
recv(right) {
println("right")
}
}
```

Each channel expression is evaluated exactly once on entry. If several receives are ready, Aiki chooses one without promising source-order preference, fairness, or scheduler behavior. If none is ready, select waits.

A default arm makes the selection nonblocking:

```
select {
let value = recv(ch) { println(value) }
default { println("nothing ready") }
}
```

Each selected arm runs in an enclosed environment. A name introduced by let in a receive case is local to that arm. Select currently has receive cases, not selectable send cases.

6.8 Timed Events

The time module provides after(ms), which returns a receive-only channel that produces true once after the requested interval. Unlike sleep(), after() does not block the calling computation.

```
let time = import("time")
select {
let value = recv(ch) { println(value) }
recv(time.after(1000)) { println("timeout") }
}
```

The event channel is receive-only; send() on it faults. Ordinary channel() values remain unbuffered and sendable. The timer's internal one-event storage is an implementation detail, not a general buffered-channel facility.

6.9 Errors in Spawned Functions

A fault in a spawned computation cannot propagate through the original call to `spawn()`, because that call has already returned:

```
let worker = () {
    println(undefined_name)
}

spawn(worker)
```

returns:

```
true
```

while the spawned fault is reported separately:

```
spawn: ... undefined: undefined_name
```

The spawning computation continues independently. Expected failures can instead be communicated explicitly as ordinary shaped error data:

```
send(ch, [@error, :worker, "operation failed"])
```

6.10 A CSP-Style Core

Aiki's concurrency model is close to the communicating-process style associated with CSP and Go. Go supplies scheduling and channel mechanics; Aiki adds isolated spawned environments and a small select form for explicit receive coordination.

There are no separate Aiki constructs for mutexes, condition variables, futures, task handles, joins, or implicitly shared captured state. Concurrent computation is expressed through ordinary functions, explicit values, channels, and select. A store may be passed deliberately when systems code requires shared mutable state; that capability is explicit rather than inherited from a closure or introduced by the concurrency syntax.

7. Graphics

Aiki provides graphics in three layers:

`canvas` `turtle` `turtle/simple`

`canvas` provides direct drawing operations over a graphics surface. `turtle` adds explicit turtle state over a canvas. `turtle/simple` wraps that state in a Logo-like interface with implicit current state.

7.1 Canvas Graphics

Import the low-level `canvas` module and create a graphics surface:

```
let canvas = import("canvas")
let screen = canvas.canvas(400, 300)
```

The returned value has the system type `canvas`:

```
type(screen)
```

produces:

```
:canvas
```

Its dimensions can be inspected directly:

```
canvas.width(screen)
canvas.height(screen)
```

which produce:

```
400
300
```

A canvas maintains drawing state such as foreground color, background color, and pen size:

```
canvas.set_bg(screen, :black)
canvas.set_fg(screen, :white)

canvas.line(screen, 20, 20, 380, 280)
canvas.rect(screen, 50, 50, 100, 75)
canvas.circle(screen, 200, 150, 40)
```

The supplied operations include drawing and filled drawing primitives such as:

`dot` `line` `rect` `fill_rect` `circle` `fill_circle` `arc`

along with operations for changing colors, clearing the surface, querying its dimensions, and destroying the graphics window.

Pen width can be changed:

```
canvas.pen_size(screen, 5)
```

and a low-level canvas can be closed explicitly:

```
canvas.destroy(screen)
```

7.2 Turtle Graphics

The `turtle` module builds turtle movement on top of an existing canvas:

```
let canvas = import("canvas")
let turtle = import("turtle")

let screen = canvas.canvas(400, 400)
let t = turtle.turtle(screen)
```

The turtle is not a separate runtime object type. It is an ordinary shaped Aiki list:

```
shape(t)
```

produces:

```
:turtle
```

Its current logical position and heading can be inspected:

```
turtle.position(t)
turtle.heading(t)
```

which initially produce:

```
[0, 0]
0
```

Heading `0` points north.

7.3 Explicit Turtle State

The `turtle` layer makes state explicit. Moving or turning returns an updated turtle value:

```
let t = turtle.forward(t, 50)
let t = turtle.right(t, 90)
let t = turtle.forward(t, 50)
```

After the first movement:

```
turtle.position(t)
```

produces:

```
[0, 50]
```

and after turning:

```
turtle.heading(t)
```

produces:

```
90
```

The programmer decides where the updated state is bound and how it flows:

```
turtle value
    |
    v
forward(t, 50)
    |
    v
new turtle value
```

The module exports operations including:

`turtle` `forward` `backward` `left` `right` pen_up `pen_down` `set_color` `home` `clear` `position` `heading`

7.4 Turtle Coordinates

The turtle uses logical coordinates centered at the origin. The initial position is `[0, 0]`, and heading `0` points upward.

Turtle movement uses trigonometric calculations internally. Aiki numbers themselves are exact rationals, but trigonometric functions pass through approximate floating-point calculations. As a result, the explicit `turtle` layer can sometimes expose a large rational representing a value extremely close to an expected integer.

A coordinate mathematically expected to be `50`, for example, may inspect as a large rational approximation. It can be presented conventionally with:

```
to_decimal(turtle.position(t)[1], 6)
```

producing:

```
"50.000000"
```

This reflects the boundary between exact rational Aiki numbers and approximate trigonometric computation.

7.5 Turtle Simple

For interactive turtle programming, Aiki provides `turtle/simple`. Rather than requiring the turtle value to be passed through every operation, this module maintains the current turtle state for the session.

```
use("turtle/simple")

new(400)
```

The current position and heading are immediately available:

```
position()
heading()
```

which produce:

```
[@point, 0, 0]
0
```

Movement uses direct Logo-like commands:

```
forward(50)
right(90)
forward(50)
```

after which:

```
position()
```

produces:

```
[@point, 50, 50]
```

The simple layer presents position as a shaped `@point` value.

The difference between the two turtle layers is therefore direct:

```
let t = turtle.forward(t, 50)
let t = turtle.right(t, 90)
```

versus:

```
forward(50)
```

```
right(90)
```

In the first form, turtle state is an ordinary shaped value passed explicitly. In the second, the module manages the active turtle state. The explicit layer is useful when turtle state itself matters as data; the simple layer is convenient for direct interactive drawing and introductory programming.

`turtle/simple` also provides familiar operations such as:

```
home()
pen_up()
pen_down()
```

`home()` returns the turtle to the origin with heading `0` without drawing a line back. Pen control allows movement without drawing and later resumption of drawing.

7.6 Logical and Pixel Coordinates

`turtle/simple` separates the turtle's logical coordinate system from the size of the graphics window. A newly created simple turtle uses a logical space centered at `(0, 0)`, with a default extent of 200 units across each dimension, and maps that space onto the actual canvas.

Thus:

```
new(400)
```

creates a 400-pixel window while turtle movement continues to use logical coordinates such as:

```
[@point, 0, 50]
```

The logical size can also be changed explicitly. This lets turtle programs describe geometry independently of a particular window size.

7.7 Graphics Windows

Canvas graphics are implemented using a separate graphics child process. The evaluator sends drawing commands to that process while the REPL remains active:

```
Aiki evaluator
      |
      v
canvas value
      |
      v
graphics IPC
      |
      v
aiki canvas child
      |
      v
graphics window
```

This allows a graphics window to coexist with the interactive session. Low-level canvases can be closed with `canvas.destroy(screen)`. With `turtle/simple`, close the graphics window normally when finished; another call to `new()` starts a fresh drawing session.

The three layers therefore expose increasing convenience without replacing the lower-level computational model:

```
canvas
    direct drawing
        |
        v
turtle
    explicit shaped state
        |
        v
turtle/simple
    implicit interactive state
```

Graphics surfaces are values, turtle state is shaped list data, operations are functions, and the simple interface is built from those same mechanisms.

8. What Aiki Does Not Have

Aiki is deliberately small. Most of the omissions described here are intentional design choices, not merely features that have not yet been implemented.

The design is governed by a simple constraint: a language feature should make a computational relationship easier to see, inspect, predict, or use. Syntax, evaluation rules, diagnostics, libraries, help, and interaction all affect that relation between the programmer and the behavior of the language.

Aiki therefore favors visible evaluation, inspectable infrastructure, explicit control flow, reduction of redundant forms, and immediate feedback. Hidden rules, overlapping mechanisms, and unnecessary machinery add distance between the written program and the computation it performs.

The result is not minimalism for its own sake. Aiki asks whether a construct exposes a computational relationship that needs to be expressed. Where an existing mechanism already does so clearly and powerfully, another mechanism must justify the additional surface it creates.

8.1 No Operator Precedence

Aiki intentionally has no conventional operator-precedence hierarchy.

Expressions evaluate from left to right unless parentheses state a different grouping:

```
2 + 3 * 4
```

means:

```
(2 + 3) * 4
```

and produces `20`.

To obtain `14`, the grouping is written explicitly:

```
2 + (3 * 4)
```

The issue is not that operator precedence is difficult to learn. It is that precedence introduces a relationship that is not visible in the written expression. In:

```
2 + 3 * 4
```

the source reads addition before multiplication, while a precedence-based language evaluates multiplication first. The programmer must therefore know an external rule in order to reconstruct the actual evaluation order.

Aiki removes that discrepancy. Written order is evaluation order; when a different relationship is intended, parentheses make it visible in the program itself.

This is a direct application of Aiki's design criterion. A rule that the learner must remember but cannot see increases the distance between source and behavior. Left-to-right evaluation reduces that distance by making evaluation order explicit rather than conventional.

8.2 No Static Type Declarations

Aiki is dynamically typed but strongly typed. Values have definite runtime types, and operations enforce the kinds of values they accept. What Aiki omits is a second, declarative type layer describing those values in advance.

Thus:

```
let x = 42
```

binds `x` to a number. The type belongs to the value itself and can be inspected directly:

```
type(x)
:number
```

Static declarations would require the programmer to maintain two related descriptions of the same thing: the value produced by the computation and a declaration stating what that value is expected to be. Such systems can provide important benefits, but they also introduce another representational layer between the learner and the running program.

Aiki chooses not to make static declarations part of its basic model. Type distinctions remain real and enforced, but they are encountered through values, operations, diagnostics, and explicit inspection rather than through a separate annotation language.

Aiki is deliberately non-coercive. Values retain their types unless conversion is requested explicitly. It does not implicitly convert ordinary values merely to satisfy an operation; string-to-number, number-to-string, rune-to-number, and number-to-rune conversions are explicit. Boolean contexts are the exception in form but not in type: they interpret values for truth without converting them to Boolean values.

The result is a dynamically typed language with unusually strict conversion rules, in which type information belongs to values and operations rather than being duplicated in declarations.

8.3 No Numeric Type Hierarchy

Aiki intentionally has one basic numeric type:

`number`

There is no separate family of integer, floating-point, decimal, long, short, or related machine-oriented numeric types.

As described in §2.2, Aiki’s core numeric model is exact arbitrary-precision rational arithmetic, with approximation used where the mathematics requires it. Whole numbers, decimal literals, and rational literals therefore belong to the same numeric model rather than to separate numeric categories.

The learner apprehends **number**, not a hierarchy of machine representations, promotions, and conversion rules.

8.4 No Separate Compound Data Types

Aiki intentionally has one basic compound data structure:

`list`

It does not add separate basic-language tuples, records, structs, maps, dictionaries, sets, or objects.

Named structure is supplied by shapes without replacing the underlying list model described in §§3.7–3.8. This choice follows the Lisp tradition of obtaining substantial structural power from one general compound form while using shape metadata to make that structure locally legible.

Lists plus shapes **are** Aiki's general structuring model.

The store system value does not change this model. A store is an explicit mutable storage capability for systems work, not an additional general-purpose compound data form.

8.5 No Classes or Object System

Aiki intentionally has no classes, inheritance hierarchy, constructors, methods, or object model.

There is no:

`class` `extends` `new` `this`

Data remains data:

```
let @point [x, y]
let p = [@point, 10, 20]
```

and behavior remains in functions:

```
let distance = (p) {
    # computation using p
}
```

The rejection of class-based object orientation has an important design lineage. Go's refusal to organize a modern production language around classes and inheritance demonstrated that class-based OO was a design choice rather than an inevitable endpoint of language design. That was an important influence on Aiki.

Aiki carries the idea further:

```
structure   -> lists
named form  -> shapes
behavior    -> functions
combination -> composition
```

Class hierarchies, inherited behavior, implicit receivers, and dispatch introduce additional conceptual machinery through which the programmer must reconstruct what computation is occurring. Lists, shapes, and functions keep those relationships exposed.

Aiki's rejection of classes is therefore not a rejection of abstraction or structured data. It is a rejection of making objects the organizing principle of the language.

8.6 No Alternate Function Syntax

Aiki intentionally has one function syntax. There is no separate distinction among function declaration, function expression, lambda, arrow function, or another anonymous-function form.

A function is written:

```
(x) {
    return x * x
}
```

and `let` gives it a name when one is useful:

```
let square = (x) {
    return x * x
}
```

The same form can be passed directly:

```
list.map(xs, (x) {
    return x * x
})
```

stored, returned, or used recursively.

Naming does not create a different kind of function. Neither does anonymity. This is the “one way to do each thing” principle in its simplest form: the learner apprehends **function** once, then learns what can be done with a function value.

8.7 No Alternate Conditional Syntax

Aiki intentionally has one conditional construct:

```
if
```

Its optional alternate branch is `else`. There is no separate `elif`, `elseif`, `switch`, or `case` construct.

Conditional execution is written directly:

```
if x < 0 {
    println("negative")
}
```

or:

```
if equal(x, 0) {
    println("zero")
} else {
    println("nonzero")
}
```

More elaborate logic is built from the same form through nesting or guard clauses.

`match` is not another conditional. It has a different role:

```
if      -> conditional execution
match   -> structural matching
```

Aiki therefore has one conditional relationship to learn: if a condition holds, execute a branch; otherwise, optionally execute its alternate.

8.8 No Additional Loop Syntax

Aiki intentionally has one loop construct:

```
while
```

There is no separate:

`for` `until` `repeat` `do` `foreach`

Explicit iteration uses `while`:

```
let n = 0

while n < 10 {
    println(n)
    n = n + 1
}
```

Repetition can also be expressed recursively or through higher-order operations:

```
list.map(xs, fn)
list.filter(xs, pred)
list.reduce(xs, initial, fn)
```

These forms expose different computational relationships:

```
while       -> repeated state change
recursion   -> computation expressed in terms of itself
map/filter/
reduce      -> higher-order transformation
```

Additional loop syntax would add more forms for repetition without adding another fundamental capability.

8.9 No Exception Syntax

Aiki intentionally has no exception mechanism.

There is no:

`throw` `try` `catch` `finally`

Aiki distinguishes instead between faults and recoverable error values. A fault stops evaluation: wrong arity, type mismatch, undefined name, division by zero, index out of bounds, and similar failures.

A recoverable error is ordinary shaped data:

```
[@error, :io, "file not found"]
```

It can be returned, tested with `is_error()`, or decomposed with `match`.

Exceptions introduce a second, nonlocal path through a computation. Aiki instead keeps recoverable failure inside the ordinary value and control model:

```
fault       -> evaluation stops
error value -> ordinary value flow
```

No separate exception-control system is required.

8.10 Other Deliberate Omissions

The same design discipline accounts for a number of smaller omissions. Aiki has no separate method syntax, no special recursion syntax, no `break` or `continue`, no arrow lambdas, no macros, and no general user-level `eval`.

Its concurrency surface likewise does not multiply abstractions into futures, promises, task objects, async/await, mutexes, or atomics. Concurrent coordination is expressed through spawn, channels, send, recv, and select. The store module provides explicit mutable storage for systems work, but Aiki does not add ambient shared state or a separate locking syntax.

Some details of the alpha surface may still change. The principle is more stable than any particular omission: a new mechanism should expose something important that the existing language cannot already express clearly.

8.11 A Deliberately Small Surface

Aiki does not try to provide a distinct construct for every familiar programming idiom.

Its central reductions are straightforward:

```
precedence          -> left-to-right evaluation
```

```
numeric hierarchy   -> number
compound structures -> lists + shapes
objects             -> data + functions + composition
function variants   -> one function form
conditionals        -> if
loop variants       -> while
exceptions          -> faults + error values
```

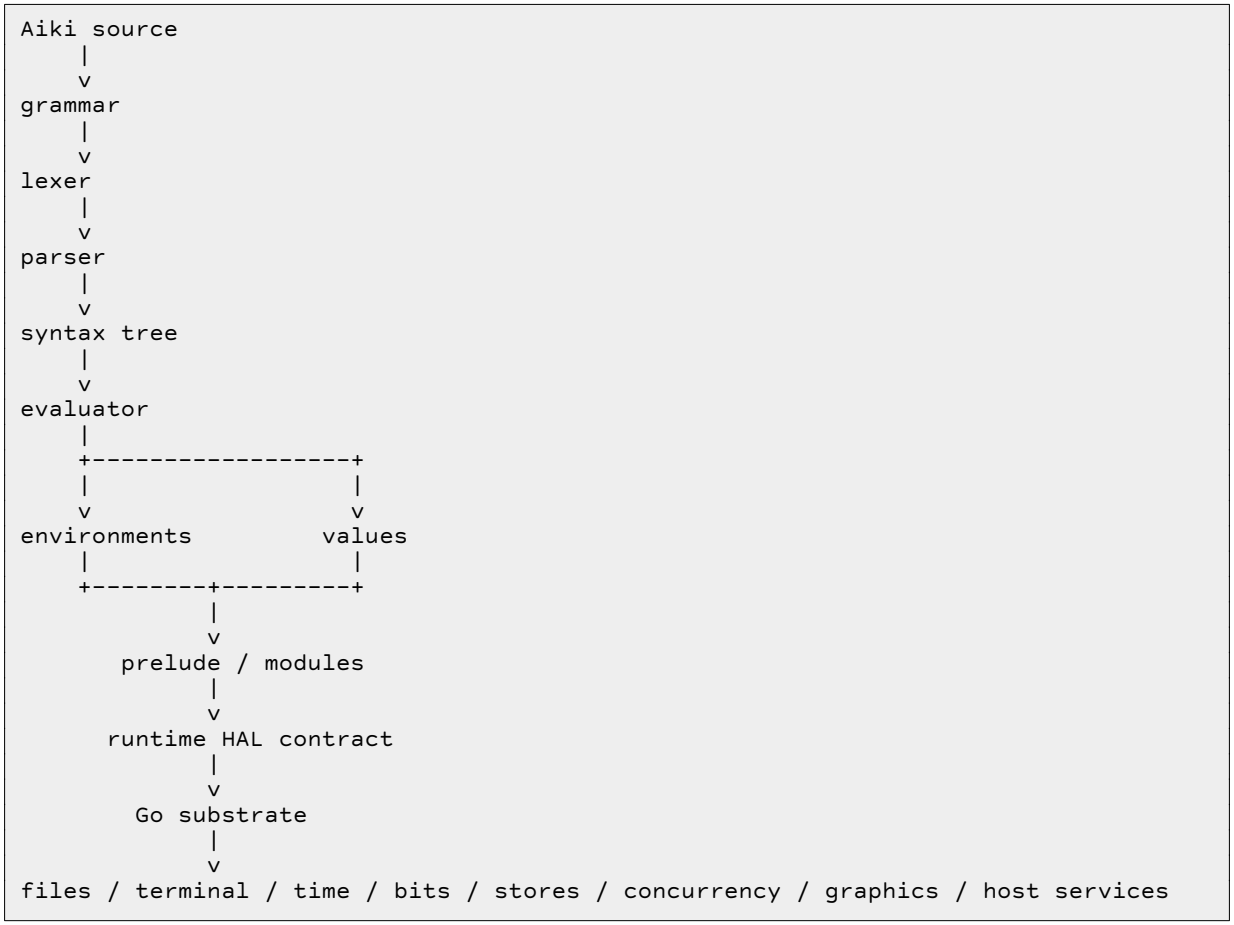
Each reduction leaves fewer overlapping mechanisms between the written program and the computation it expresses.

That is the practical consequence of treating Aiki as a learning field. The aim is not simply fewer features. The aim is to make the computational relationships that remain easier to see, inspect, predict, and apprehend.

9. Architecture

Aiki is implemented in Go, but the language is deliberately separated from the Go implementation beneath it.

At a high level, execution follows this path:



The important boundary is between **Aiki semantics** and **host implementation**. The evaluator implements the meaning of the language. Host-dependent operations are delegated through a runtime contract rather than being spread throughout the evaluator.

9.1 Source Organization

The central implementation layers live under `$AIKIHOME/engine/`:

```
$AIKIHOME/
├── cmd/
│   ├── aiki/
│   └── subcommands/
├── engine/
│   ├── runner/
│   ├── runtime/
│   │   ├── hal/
│   │   │   └── substrate/
│   │   ├── help/
│   │   └── prelude/
│   ├── semantics/
│   │   ├── evaluator/
│   │   └── value/
│   └── syntax/
│       └── grammar/
└── lib/
```

The divisions correspond to different responsibilities:

- syntax what Aiki programs can say
- semantics what those programs mean
- runtime how Aiki reaches host facilities
- runner how an execution environment is assembled
- cmd command-line and interactive entry points
- lib supplied Aiki modules

9.2 Grammar, Syntax, and Evaluation

Aiki's syntax begins with its grammar. The lexer and parser operate from the loaded grammar definition:

```
source
  |
  v
lexer
  |
  v
tokens
  |
  v
parser
  |
  v
AST
```

The same grammar is used for ordinary execution, the REPL, modules, the formatter, the linter, and engine-level tests.

The parser produces syntax nodes representing constructs such as:

program statement let_stmt assign_stmt if_stmt while_stmt match_stmt select_stmt select_case select_default return_stmt expr func_literal list_literal call index access

The evaluator operates on this parsed structure rather than reinterpreting source text.

Evaluation is handler-driven. Each significant grammar production has a corresponding evaluator handler, and the evaluator validates that relationship when it is initialized. A grammar production without semantic support therefore fails structurally rather than remaining silently parseable:

```
grammar construct
        <=>
```

```
semantic handler
```

This is one of Aiki's explicit drift-prevention boundaries.

9.3 Values and Environments

Evaluation produces Aiki values. The value layer contains the representations for numbers, booleans, runes, strings, symbols, lists, functions, and system values including channels, modules, canvases, files, bytes, and stores.

A shaped list, for example, stores its payload and its shape separately:

```
type List struct {
    Elements []Value
    Shape    string
}
```

This is why:

```
[@point, 10, 20]
```

has two indexed elements rather than three. `@point` is shape metadata rather than element zero.

Likewise, Aiki numbers are represented as arbitrary-precision rational values rather than ordinary Go machine integers or floating-point values. Important language semantics therefore live in the value layer rather than being incidental consequences of the host language.

Names are resolved through lexical environments. An environment contains bindings and may point to an enclosing environment:

```
local environment
       |
       v
outer environment
       |
       v
...
```

A lookup begins locally and proceeds outward. New enclosed environments support function calls, successful match arms, selected receive arms, the prelude, and modules using the same mechanism.

An Aiki function value contains its parameters, optional rest parameter, body, and defining environment. Ordinary calls therefore have closure behavior naturally. `spawn()` deliberately changes that rule by executing a function in a fresh environment enclosed by the prelude rather than reusing its captured user environment.

9.4 The Prelude and HAL Boundary

At startup, Aiki creates a prelude environment and evaluates the prelude as Aiki source. The user environment is then enclosed by it:

```
user environment
      |
      v
prelude environment
```

This is why user programs can see names such as `println`, `first`, `type`, `spawn`, and `channel` through ordinary environment lookup.

The prelude also forms a boundary between ordinary Aiki names and lower-level host operations. The Go substrate registers `_`-prefixed HAL primitives such as:

_print _read _first _rest _length _type _equal _import _export _load _spawn _channel _send _recv _after _store_new _store_get _store_set _store_length _bits_and _bits_or _bits_xor _bits_not _bits_shl _bits_shr

The user normally encounters Aiki-level names instead:

`println` `first` `type` `spawn` `channel` `send` `recv`

Conceptually:

```
Aiki program
    |
Aiki prelude
    |
HAL
    |
Go substrate
    |
host
```

Ordinary user code cannot simply reach through the prelude into the general HAL primitive registry. The boundary is part of the language architecture.

9.5 The HAL and Go Substrate

The `runtime/hal` layer defines the contract through which the evaluator reaches host facilities. Some primitives need only arguments; others, such as `import`, `export`, `apply`, `load`, and `spawn`, also require evaluation context because they interact with environments, grammar, or evaluation itself.

The current implementation of that contract is the Go substrate. It supplies facilities including:

- terminal I/O
- list support
- type operations
- numeric support

  bit operations

  explicit mutable stores
- bytes
- time
- loading
- modules
- concurrency
- graphics
- REPL interaction

For example:

```
Aiki channel    -> Go channel
Aiki store -> synchronized Go-backed store
time.after -> receive-only event channel
Aiki spawn      -> Go goroutine
Aiki canvas     -> graphics child process
Aiki file I/O   -> host filesystem
```

Go supplies the underlying mechanisms, but Aiki determines which mechanisms become visible and under what rules. Go channels become Aiki channels; Go goroutines become isolated Aiki spawned computations. The host provides capability without defining the language semantics.

9.6 Modules

Modules use the same lexer, parser, evaluator, values, and environments as ordinary programs. At startup, a module registry scans configured roots, including:

`.` `lib/` `vendor/` `~/.aiki/lib/`

A module is evaluated in an environment enclosed by the prelude, not by the importing user's environment:

```
module environment
       |
       v
prelude environment
```

A module can therefore see the prelude but does not automatically see arbitrary bindings belonging to its importer.

After evaluation, names identified by `export()` are collected into a module value. `import("name")` can return that module value, while selective import copies named exports into the caller's environment. Registry-loaded modules are cached.

After scanning, the registry also establishes native defaults: when X/native exists and no explicit X package exists, X becomes a public alias for X/native. Explicit bare packages win, /ffi-only variants never become defaults, and an alias retains its bare public module identity.

An important consequence is that much of Aiki's larger functionality is written in Aiki itself. Supplied modules under `$AIKIHOME/lib/` are parsed and evaluated by the same engine used for user programs. Only when a facility reaches a host boundary does it descend through the HAL.

9.7 Runner and REPL

The runner assembles the pieces required for execution: grammar, module registry, runtime, prelude environment, user environment, lexer, parser, and evaluator. The command-line executable is therefore an entry point into the engine rather than a separate implementation of the language.

The REPL uses the same engine and retains four principal pieces of state:

- grammar
- runtime
- evaluator
- environment

Each entry is lexed, parsed, and evaluated in the persistent session environment. `reset()` rebuilds the module registry, reloads the prelude, and replaces the user environment with a fresh one. The REPL is another client of the engine, not a reduced interpreter beside it.

9.8 Graphics Processes

Graphics introduce one additional host boundary. A canvas window runs as a child process of the `aiki` executable. The parent sends line-oriented JSON drawing commands to the child, which owns the graphics event loop while the parent REPL remains usable. Closing the window ends that graphics session without requiring the evaluator itself to become a graphics event loop.

9.9 Tooling and Structural Invariants

Aiki's tools reuse the language engine rather than maintaining independent definitions of the language.

`enginesmoke` can exercise:

- lexer
- parser
- evaluator
- ebnf parser

and compare their output with blessed gold files. enginesmoke also records which EBNF productions are traversed by the structural specimens and fails coverage when any declared production is unexercised.

The formatter parses the original source, formats it, parses the result again, and compares the two syntax trees:

```
parse(original)
        |
      format
        |
parse(formatted)
        |
compare ASTs
        |
same structure? -> write
different?      -> refuse
```

The linter likewise operates structurally over parsed Aiki source.

The runner also checks the public prelude against the help registry. Public prelude functions and help entries must agree.

Several important relationships are therefore made executable rather than left as conventions:

```
grammar        <=> evaluator handlers
prelude        <=> help
formatted AST  <=> original AST
modules        <=> explicit exports
behavior       <=> gold files
grammar productions <=> structural specimens
```

The engine additionally exposes observer hooks for lexing, parsing, evaluation, and effects, allowing diagnostic tooling to inspect the same execution pipeline without creating a second semantics.

Golds are deliberately treated as reference snapshots rather than as proof of correctness. Independent checks establish that an intentional change is sound; bless records the resulting behavior and structure; validate later compares the tree against that blessed state without rewriting it.

9.10 Implementation Dependencies

Aiki is implemented largely within its own code base, but it relies on two external Go libraries for host-facing facilities.

`chzyer/readline` provides the terminal line-editing support used by the interactive console. `Ebiten` provides the graphics and windowing substrate used by Aiki's canvas process. Aiki does not implement a terminal line editor or graphics engine of its own.

These libraries supply host capabilities rather than Aiki language semantics. The REPL, graphics model, canvas protocol, turtle layers, and their behavior are defined by Aiki; the underlying terminal and graphics machinery is provided by those libraries.

Other packages listed in the Go module definition are indirect dependencies of the implementation.

10. Current Status

Aiki 1 is specified by the Report on the Programming Language Aiki. The reference implementation is currently a **working alpha and research artifact**. It supports interactive use, nontrivial programs, modules, graphics, concurrency, tooling, and the examples described in this document. It is not yet presented as a stable production implementation. The language and standard environment described by the Report are intended to remain stable; alpha refers to the reference implementation and to compatibility guarantees not yet established.

The present code base is a working alpha implementation rather than a prototype specification.

10.1 What Works

The core language is operational:

- bindings and mutation
- exact rational numbers
- strings, runes, symbols, booleans
- lists and shapes
- functions and closures
- if
- while
- match

  select
- recursion and proper tail calls
- pipelines
- modules
- recoverable error values

The evaluator implements proper tail-call handling rather than relying on the Go call stack for tail-recursive execution. Ordinary non-tail recursion remains bounded by Aiki's stack limit.

The module system, prelude, persistent REPL, canvas graphics, both turtle layers, and the concurrency model are operational. The concurrency surface includes spawn, channel, send, recv, select, and time.after event channels. Modules execute in environments isolated from their importers; spawned computations execute in isolated user environments; graphics run separately from the evaluator so the REPL remains active.

The supplied library surface currently includes:

bits bytes canvas file hash list math random regex/ffi store string test time turtle turtle/simple

Several facilities have both native-Aiki and host-backed implementations:

bytes/native bytes/ffi hash/native hash/ffi math/native math/ffi

10.2 Tooling and Testing

Aiki includes:

aiki fmt aiki lint aiki test aiki smoke aiki enginesmoke

The formatter checks that formatting preserves the syntax tree. The linter works structurally over parsed source. aiki test runs Aiki-native tests. smoke compares program behavior with expected transcripts, while enginesmoke preserves lexer, parser, evaluator, and grammar representations as gold files and verifies complete EBNF production coverage.

Testing is unusually substantial for an alpha language. The current source tree contains:

```
128 Go source files
99 Aiki source files
43 Go test files
34      behavior programs
10 structural engine programs
70 gold files
17      sample programs
```

Tests are organized by purpose:

```
structure     intermediate engine representations
behavior      observable output
boundary      separation between scopes and layers
invariant     architectural conditions
canary        detection of disappearing features
contract      interface behavior
property      invariant testing over generated inputs
regression    previously observed failures
fuzz          malformed input and panic resistance
```

The build system defines increasingly broad validation targets:

```
make test
make check
make bless
make validate
make fuzz
make rigorous
```

with:

```
check =
    build
    + fmt
    + lint
    + Go tests
+ Aiki-native tests
+ complete EBNF production coverage
```

Blessing and validation are separate:

```
bless = check + rewrite behavior and engine golds
validate = check + compare behavior and engine golds
```

check never writes golds. bless first requires check to pass and then records the current reference snapshots. validate is read-only with respect to gold files. For ordinary development, make validate is the completion gate; make bless is used only after an intentional behavior or structural change has been independently established as correct.

and:

```
rigorous =
    validate
    + lexer fuzzing
    + parser fuzzing
```

This does not prove correctness. It does mean the alpha is guarded by substantially more than a collection of demonstration programs.

10.3 Research Artifact

Aiki is also a research artifact. The present release includes the interpreter, grammar, prelude, libraries, examples, behavior programs, and expected outputs needed to inspect the system as implemented at this stage.

The artifact therefore has two simultaneous roles:

- programming language
- research object

The first requires that programs actually run. The second requires that claims about the language be inspectable against a particular implementation.

Retaining versioned releases makes those roles compatible: later Aiki can evolve without changing the artifact described by this document.

10.4 What Alpha Means Here

The language model is coherent, operational, and specified. Alpha refers to the reference implementation rather than to the language: compatibility guarantees have not yet been established, and arbitrary existing Aiki programs should not yet be assumed to run unchanged across future releases.

The command's compiled version is supplied at build time from the repository tag or commit; without that metadata the source defaults to:

```
dev
```

10.5 What Is Still Open

The remaining work is increasingly about **stabilization, breadth, and experience** rather than establishing whether the language model can work.

Questions appropriate to the alpha stage include:

- Which implementation and library guarantees should be established for beta?
- Which library interfaces deserve compatibility guarantees?
- Where does the standard library stop?
- Which additional capabilities justify their surface cost?
- How should releases and library versions be managed?
- What behavior needs more adversarial testing?
- What does sustained classroom use reveal about the design?

The grammar parses programs. The evaluator runs them. Functions, lists, shapes, matching, selection, modules, tail recursion, files, explicit stores, bit operations, timed events, graphics, and concurrency work. The REPL is live. The prelude and libraries are inspectable. Formatter, linter, Aiki-native tests, smoke tests, structural coverage, structural gold tests, property tests, and fuzzing infrastructure exist.

The principal task now is not to discover what Aiki is. It is to use it hard enough to discover where Aiki is wrong.

11. A Note on AI

Aiki was created with extensive assistance from generative AI. Every line of implementation code in the current code base was generated through interaction with AI systems. AI also participated in architecture, tooling, testing, and the mechanical expansion of the system. That sounds cleaner than it was. At times, it was an unmitigated disaster.

The central problem was drift. An AI system could make a locally reasonable choice, propagate it through several parts of the code base, adapt surrounding code to it, and leave behind a system that was internally coherent and wrong. The numeric model was the clearest case. Aiki was intended to use exact rational arithmetic, but floating-point computation entered the ordinary numeric path and the results were later converted back into rationals. The problem was not the existence of an explicit foreign floating-point boundary; current math/ffi deliberately converts each returned float64 to the exact rational representation of that binary64 result. The failure was that approximation had entered the language's normal arithmetic path without being an explicit boundary at all.

Worse, the tests passed. Implementation and tests had drifted together. They agreed with one another while no longer agreeing with the language design. That was the critical lesson: a test suite can preserve behavior perfectly and still preserve the wrong behavior. When tests inherit the same mistaken assumption as the code they exercise, consistency becomes evidence of internal agreement, not correctness.

The error also propagated. The numeric choice spread through arithmetic, conversions, libraries, examples, and assumptions elsewhere in the engine. Correcting it meant unwinding a decision that had already become structural. After that, it was no longer enough to specify a design, generate an implementation, run the tests, and inspect the result. I needed mechanisms capable of detecting when implementation was beginning to redefine the language.

The same failure appeared architecturally. Host-language calls had spread through the implementation until the boundary between Aiki and Go was no longer reliably structural. The remedy was not another convention but an enforced boundary: an explicit HAL contract, scope-gated registration, and tests that verify the separation. A boundary that exists only in the programmer’s intentions is not an architectural boundary.

Much of Aiki’s later architecture grew from a deliberate programme of drift detection and prevention. Known-good behavior and intermediate engine results were captured so later changes could be checked against established expectations. The same impulse produced structural checks tying grammar productions to evaluator handlers, public prelude functions to help entries, module visibility to explicit exports, and formatted source to the original syntax tree.

Where possible, these relationships are enforced mechanically rather than left as conventions:

```
grammar productions <-> evaluator handlers
public prelude       <-> help entries
module visibility    <-> explicit exports
formatted source     <-> original syntax tree
behavior             <-> gold files
engine structure     <-> structural gold files
grammar productions <-> structural specimen coverage
architectural rules  <-> invariant and boundary tests
```

The grammar cannot quietly grow syntax that the evaluator does not implement, and a new grammar production cannot leave enginesmoke coverage complete unless some structural specimen actually exercises it. The help surface cannot drift away from the prelude. The formatter cannot change the syntax tree and still succeed. Modules cannot expose names that were not explicitly exported. Behavioral, structural, boundary, canary, regression, property, differential, fuzz, and gold-file tests all serve the same purpose: to make important assumptions executable enough that the system can detect when they stop being true. Golds preserve a state already judged correct; they do not establish correctness by themselves.

This was not an elegant plan conceived at the beginning. It was scar tissue. AI made implementation extraordinarily fast, but it also made bad assumptions propagate extraordinarily fast. A human programmer can make the same mistake, but AI changes the scale: one locally plausible choice can spread through implementation, tests, examples, and surrounding assumptions before the underlying error is recognized.

That forced a different development cycle:

```
design
  ->
encode constraints
  ->
generate
  ->
execute
  ->
compare
  ->
detect drift
  ->
correct
  ->
strengthen constraints
```

The human role was therefore primary, not merely supervisory. AI could propose architecture, elaborate it, implement it, test it, and sometimes improve it. But the authority to decide what Aiki was remained outside the generated system. That meant deciding which abstractions belonged, which apparent improvements were violations, where boundaries had to remain fixed, and when an internally coherent implementation had ceased to be Aiki.

Without that independent design authority, the float-to-rational drift would not have been recognizable as a failure. It would simply have become the language.

Much of the consequential work was refusal. AI is good at answering, “How can this be implemented?” It is less reliable at preserving, across thousands of interacting decisions, the prior answer to, “What are we building?” The programme of invariants, gold files, boundary tests, differential tests, canaries, and constrained changes arose because that second question had to remain under human control.

This project could not have been produced at this scale without AI. But AI alone would not have produced Aiki. It could just as readily have produced a larger, smoother, internally consistent language bearing the same name while erasing the decisions that made Aiki Aiki.

The lesson is not that AI replaces software design, but nearly the opposite. AI supplies extraordinary implementation leverage, and that leverage magnifies the cost of architectural error. The faster code can be generated and propagated, the more important it becomes to maintain an external account of what the system is supposed to be and mechanisms that detect when implementation departs from it.

Aiki exists because AI supplied implementation leverage and human judgment supplied continuity of intent. Only one of them knew when the system had stopped being Aiki.

12. Closing

Aiki is intentionally small, but it is not finished. Its language surface, libraries, tooling, and implementation remain open to revision as use exposes mistakes, unnecessary machinery, and missing capability.

The aim is not to preserve Aiki exactly as it is. It is to preserve the constraints that make Aiki Aiki while continuing to test whether the language earns them.

The next step is simple: use it.