Report on the Programming Language Aiki

William D. Senn – will.senn@gmail.com
August 10, 2026

Dedicated to Go
whose clarity made Aiki possible

Summary

This report gives a defining description of the programming language Aiki. Aiki is a dynamically typed, lexically scoped language designed around a small set of directly composable forms. Binary expressions are evaluated from left to right unless grouping specifies otherwise. Functions are first-class values, lists provide the principal compound data structure, shapes provide named structure over lists, and recoverable errors may be represented as ordinary values. Aiki supports imperative, functional, iterative, recursive, and concurrent forms of computation.

The introduction describes the background, purpose, and status of the language and this report.

The first three chapters present the fundamental ideas of Aiki and describe the lexical conventions, terminology, values, bindings, types, and other concepts needed to understand the language.

Chapters 4 and 5 describe expressions and program structure, including functions, operators, pipelines, conditionals, iteration, matching, binding, assignment, and modules.

Chapter 6 describes the Aiki standard library. It begins with the prelude, whose bindings are available in the initial environment, and then describes the supplied library modules for lists, strings, bytes, hash tables, mathematics, random numbers, regular expressions, files, time, and graphics.

Chapter 7 gives the formal syntax of Aiki and a formal account of its semantics.

An example follows the formal description. The report concludes with references and an alphabetic index of concepts, syntactic forms, and procedures.

The organization of this report is modeled on the compact language-report tradition exemplified by the *Revised⁴ Report on the Algorithmic Language Scheme* (R4RS).

Table of Contents

Summary....1
Introduction....1
Background....2
Acknowledgments....2
Description of the Language....2
1. Overview of Aiki....2
1.1 Semantics....2
1.2 Syntax....3
1.3 Notation and terminology....3
2. Lexical conventions....4
2.1 Identifiers....4
2.2 Whitespace and comments....4
2.3 Other notations....4
3. Basic concepts....5
3.1 Names, bindings, and environments....5
3.2 True and false....5
3.3 Literal and displayed representations....6
3.4 Types....6
3.5 Lists, mutation, and state....6
4. Expressions....6
4.1 Name references....6
4.2 Literal expressions....6
4.3 Function literals....7
4.4 Function calls....7
4.5 List literals....7
4.6 Unary expressions....8
4.7 Binary expressions....8
4.8 Pipelines....8
4.9 Indexing....9
4.10 Field access....9
4.11 Grouping and postfix composition....9
5. Program structure....9
5.1 Programs....9
5.2 Bindings and assignment....9
5.3 Conditional execution....10
5.4 Iteration....10
5.5 Pattern matching....10
5.6 Return....11
5.7 Packages and modules....11
5.8 Concurrent selection....11
6. Standard library....12
6.1 The prelude....12
6.2 Lists....17
6.3 Strings....18
6.4 Bytes....21
6.5 Hash tables....22
6.6 Mathematics....23
6.7 Random numbers....24
6.8 Regular expressions....24
6.9 Files....25
6.10 Time....26
6.11 Canvas graphics....27
6.12 Turtle graphics....28
6.13 Simple turtle graphics....29
6.14 Bit operations....30
6.15 Stores....31
6.16 Testing....31
7. Formal syntax and semantics....31
7.1 Formal syntax....32
7.2 Formal semantics....39
Example....46
References....49
Alphabetical Index....50

Introduction

Aiki is a small general-purpose programming language designed to make computational relationships explicit and readily inspectable. Its design favors a small number of directly composable forms over a larger collection of specialized mechanisms. The language supports ordinary imperative programming together with functional, iterative,

recursive, and concurrent forms of computation, while retaining a compact syntax and value model.

Several choices give Aiki its particular character. Binary expressions are evaluated from left to right unless grouping specifies otherwise. Functions are first-class values and close over their lexical environments. Lists are the principal compound data structure, with shapes providing named structure over lists rather than introducing a separate object system. Pipelines provide an explicit notation for composition, pattern matching provides structural selection and binding, and recoverable failure may be represented directly as data.

Aiki is implemented as a hosted interpreter. The reference implementation is written in Go and separates the language evaluator from host-dependent facilities through a runtime abstraction. This implementation architecture is not part of the definition of the language except where host-visible behavior is explicitly specified by this report. An implementation of Aiki need not be written in Go or reproduce the organization of the reference implementation.

This report describes Aiki 1 and serves as its defining specification. It covers the lexical structure, values, bindings, expressions, program structure, standard environment, syntax, and semantics of the language. Tutorial explanation, implementation detail, and extended discussion of design rationale are kept largely outside the report. This report describes Aiki 1 and serves as its defining specification. The language, standard environment, syntax, and semantics described here are intended to remain stable within this report; an implementation may be written against them. Implementation facilities, experimental modules, and host-specific behavior may continue to change.

Background

Aiki grew from a formal account of apprehension developed in information science (Senn, 2026a). In that account, an informing act occurs when apprehension produces a change in the configuration of a knowing subject, ($K \neq K'$). The criterion shifts attention from whether information is merely present to whether an environment provides conditions under which its structure can be received, inspected, and integrated.

Aiki was designed as a constructive application of that criterion rather than as a language to which the theory was applied afterward. The resulting design treats evaluation rules, syntax, diagnostics, libraries, help, and interaction as part of the programmer’s informational environment. This leads to requirements for visible evaluation, inspectable infrastructure, aligned contextual cues, receivable failure, preserved traces, and simplicity that removes rather than conceals complexity.

Aiki applies that criterion constructively to programming-language design. Its design treats evaluation rules, syntax, diagnostics, libraries, help, and interaction as part of the programmer’s informational environment. This leads to requirements for visible evaluation, inspectable infrastructure, aligned contextual cues, receivable failure, preserved traces, and simplicity that removes rather than conceals complexity.

The formalism supplies the design criterion. The present report is concerned with the resulting language: its syntax, semantics, values, forms, procedures, and program structure.

Acknowledgments

Aiki was implemented in Go, whose clarity, compactness, and tooling materially shaped the development of the reference implementation. Go did not determine the language, but it made possible an unusually direct correspondence between the language design and the system used to realize it.

The organization of this report is closely modeled on the Revised⁴ Report on the Algorithmic Language Scheme (R4RS), edited by William Clinger and Jonathan Rees. Its economy, progression, and treatment of a programming language as something that can be described both informally and formally provided the principal model for the present report.

The present report also follows the broader spirit of R4RS as a document intended for use by a programming-language community rather than solely as a fixed scholarly artifact. R4RS explicitly encouraged implementors to copy, adapt, and use the report as a basis for manuals and other documentation. The Aiki report is offered in the same spirit: implementations, teaching materials, manuals, and related documentation may draw upon and adapt its language description as appropriate.

ChatGPT and Claude were used extensively during the development of Aiki and its implementation. They assisted with implementation, testing, review, documentation, and repeated examination of the correspondence among the grammar, evaluator, runtime, and language description. The design decisions, formalism, language definition, and responsibility for the resulting system remain the author's.

Description of the Language

1. Overview of Aiki

1.1 Semantics

This section gives an overview of Aiki's semantics. A more detailed informal account is given in chapters 3 through 6. Section 7.2 gives a formal semantics of the language.

Aiki is a lexically scoped programming language. Each use of a name is associated with a binding determined by the lexical structure of the program. Functions retain the lexical environment in which they are created.

Aiki is dynamically typed. Types are associated with values rather than with names. A name may therefore be bound to values of different types during the course of a computation.

Functions are values in their own right. They may be created dynamically, bound to names, stored in lists, passed as arguments, returned as results, and called recursively.

Aiki supports proper tail calls. A function call in tail position does not require the accumulation of an additional ordinary call frame, allowing iterative computations to be expressed recursively without unbounded growth of the call stack.

Argument expressions are evaluated before the function body gains control, from left to right, and the resulting values are bound to the parameters.

Aiki binary expressions are evaluated from left to right rather than according to an operator-precedence hierarchy. Parentheses may be used to specify a different grouping. Thus `2 + 3 * 4` has the same grouping as `(2 + 3) * 4`.

Aiki numbers are represented exactly when they can be expressed as rational values. Integer and rational arithmetic therefore use a common numeric representation. Operations whose mathematical results are not rational may produce approximations.

Lists are Aiki's principal compound data structure. A shape adds a name and field interpretation to a list without introducing a distinct aggregate value model. Shaped values remain lists and participate in the ordinary list operations of the language.

Conditional contexts interpret values according to Aiki's truth rules. `false`, the empty list `[]`, and shaped `@error` values are false; all other values are true.

Recoverable failure may be represented as ordinary data, typically by shaped `@error` values that can be returned, inspected, matched, and passed through computations. Evaluation faults are distinct from such values and indicate that the current computation cannot proceed normally.

Aiki provides concurrency through functions for spawning computations and communicating through channels, together with a select statement for choosing among ready receives. Spawned computations execute independently and communicate by explicitly passed values and channel operations.

1.2 Syntax

Aiki uses a conventional surface syntax consisting of names, infix operators, parentheses for grouping and calls, brackets for lists, and braces for blocks. The lexical and syntactic structure of the language is specified by its grammar.

Aiki does not treat programs as ordinary data. There is no quotation syntax, macro system, or general user-level `eval`. Lists are the principal compound data structure, but a list represents data rather than an unevaluated fragment of Aiki source. Shapes add named structure while preserving the underlying list representation.

Aiki has few syntactic forms. Expressions include literals, names, function calls, indexing, field access, unary and binary operations, pipelines, and explicit grouping. Statements provide binding, assignment, conditional execution, iteration, matching, concurrent selection, return, and package declaration. Functions use a single literal syntax whether they are used directly or bound to a name.

Spaces and tabs separate tokens where necessary and are otherwise insignificant. Line endings may separate statements. A comment begins with # and continues to the end of the line. Parentheses, brackets, braces, commas, and other delimiters provide explicit structure.

The formal syntax of Aiki is described in section 7.1.

1.3 Notation and terminology

1.3.1 Status of descriptions

Unless otherwise noted, statements in this report describe the language Aiki 1. Remarks concerning the reference implementation, host environment, experimental modules, or implementation detail are identified separately and are not part of the language description.

1.3.2 Faults and shaped errors

This report uses the term **fault** for a condition that prevents evaluation from continuing normally. A fault is distinct from a shaped error.

Shaped errors are ordinary Aiki data. A value such as `[@error, :kind, "message"]` may be returned, inspected, matched, stored, or passed as an argument like other values.

1.3.3 Entry format

Chapters 4 and 6 contain entries describing syntactic forms and procedures.

An entry for a syntactic form begins with a template showing the form of the construct. Names standing for program fragments are identified in the surrounding text. Repeated or optional components are described explicitly or by the grammar in section 7.1.

An entry for a procedure begins with one or more call templates, for example:

```
append(list, value)
```

Argument names indicate the role of the corresponding values. Type or form restrictions are stated in the entry.

1.3.4 Evaluation examples

The symbol ⇒ used in examples means “evaluates to.” For example,

```
2 + 3 * 4
    ⇒ 20
```

means that the expression `2 + 3 * 4`, evaluated in the initial environment, produces the Aiki number represented by `20`.

Where an example illustrates output rather than an expression value, the output is identified separately.

1.3.5 Naming conventions

Aiki identifiers are case-sensitive. By convention, ordinary names use `snake_case`, while names intended as constants may use `SCREAMING_SNAKE_CASE`.

Shape names begin with `@`. The name following `@`, and shape field names, conventionally use `snake_case`.

Procedure names describe their operations directly. Aiki does not use punctuation suffixes to distinguish predicates, mutation procedures, or conversion procedures.

2. Lexical conventions

This section gives an informal account of the lexical conventions used in writing Aiki programs. For the formal lexical syntax of Aiki, see section 7.1.

Upper- and lower-case letters are distinguished in Aiki. Thus `foo`, `Foo`, and `FOO` are distinct names.

2.1 Identifiers

An Aiki identifier consists of letters, digits, and underscores. The first character must be a letter or underscore. Identifiers are case-sensitive.

Examples of identifiers are:

`x` `count` `add_two` `_internal` `MAX_SIZE`

By convention, ordinary names are written in `snake_case`, while names intended as constants may be written in `SCREAMING_SNAKE_CASE`.

The following words are reserved as syntactic keywords and cannot be used as ordinary identifiers:

let if else while match select default return true false and or not package

Symbols and shape names are lexically distinct from identifiers. A symbol begins with `:`, as in `:ok` or `:error`. A shape name begins with `@`, as in `@point` or `@person`.

See section 7.1.1 for the formal syntax of identifiers.

2.2 Whitespace and comments

Whitespace consists of spaces, tabs, and line endings. Spaces and tabs are otherwise insignificant.

Line endings may separate statements. Outside parentheses and brackets, a line ending terminates a statement when the preceding token is a `name`, `number`, `string`, `rune`, `symbol`, `true`, `false`, `)`, or `]`. In effect, the line ending acts as a semicolon after such a token.

A line ending does not terminate a statement within parentheses or brackets, or when the preceding token cannot complete a statement. Expressions may therefore continue naturally after operators, commas, opening delimiters, and other incomplete forms.

Because braces do not suppress statement termination, the opening brace of an if, while, or match must occur on the same line as the expression that precedes it:

```
    if x {
        println(x)
    }
```

An opening brace on the following line is not equivalent:

```
    if x
    {
        println(x)
    }
```

An explicit semicolon may separate statements on the same line:

```
    let x = 1; let y = 2; println(x + y)
```

Whitespace may occur between tokens but not within a token. Spaces, tabs, and line endings occurring within a string literal are part of the string.

A `#` character begins a comment. The comment continues to the end of the line. Comments do not participate in program evaluation.

For example:

```
# Compute the sum of two values.
let add = (a, b) {
    return a + b
}
```

A comment may also follow program text on the same line:

```
let answer = 42    # the answer
```

2.3 Other notations

The following characters and character sequences have special uses in Aiki.

`+` `-` `*` `/`

Arithmetic operators. - is also used as a unary negation operator.

`< > <= >=`

Comparison operators.

`and` `or` `not`

Boolean operators. and and or are binary operators; not is unary.

`|>`

The pipeline operator. The value on its left is supplied to the computation on its right.

`=`

Used in bindings and assignment.

`.`

Used for field access on shaped lists and modules.

`( )`

Parentheses are used for explicit grouping, function calls, function parameter lists, and function literals.

`[ ]`

Brackets delimit list literals, indexing expressions, shape definitions, and list patterns.

`{ }`

Braces delimit blocks.

`,`

A comma separates elements, arguments, parameters, fields, and related components within delimited forms.

`;`

A semicolon may separate statements appearing on the same line.

`...`

Three successive periods introduce a rest parameter in a function parameter list.

`"`

Double quotation marks delimit string literals. Backslash introduces escape sequences within strings.

`'`

Single quotation marks delimit rune literals. Backslash may introduce an escaped rune.

`:`

A colon introduces a symbol literal, as in `:ready`.

`@`

An at-sign introduces a shape name, as in `@point`.

`#`

A sharp sign begins a comment extending to the end of the line.

The boolean literals are `true` and `false`. Number, string, rune, symbol, and shape notation are described in greater detail with their corresponding value types.

3. Basic concepts

3.1 Names, bindings, and environments

A name may be bound to an Aiki value. A binding associates a name with a value within an environment.

The `let` statement establishes a binding in the current environment:

```
let x = 10
```

Assignment changes an existing binding:

```
x = 20
```

Assignment does not create a new binding. When assignment is evaluated, Aiki searches the current environment and then successively enclosing environments. The nearest existing binding having the specified name is changed. If no such binding exists, evaluation faults.

Name lookup follows the same environment chain. The current environment is searched first, followed by its enclosing environments. The first binding found supplies the value of the name. Referencing a name for which no binding exists faults.

Aiki is lexically scoped. A function retains the environment in which it is created. When the function is called, its parameters and local bindings are established in an environment enclosed by the retained environment. A function may therefore refer to bindings visible at the point where the function was created.

Braces delimit blocks but do not by themselves introduce a new environment. Bindings established within an `if` or `while` block therefore belong to the environment in which the block is evaluated.

A successful `match` arm is evaluated in an enclosed environment containing the bindings established by the matched pattern.

The initial environment supplies the standard bindings described in chapter 6.

3.2 True and false

Any Aiki value may be used in a conditional context.

The boolean value `false`, the empty list, `[]`, and shaped `@error` values are false. Every other value is true. Thus the number `0`, the empty string `""`, symbols, functions, shaped lists, and other nonempty lists are true.

In this report, the words *true* and *false* may refer either to the boolean values `true` and `false` or, when discussing conditional evaluation, to values interpreted according to these rules.

3.3 Literal and displayed representations

Several Aiki values have literal forms that may appear directly in program text. For example:

`42` `1/3` `true` `'a'` `"hello"` `:ready` `[1, 2, 3]`

These denote numbers, booleans, runes, strings, symbols, and lists.

A shaped list includes a shape name:

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

The shape is associated with the list but is not one of its indexed elements.

Functions also have a literal form, described in section 4.3.

Aiki may display values during interactive use, output, or inspection. A displayed representation is not necessarily a literal representation and need not be valid Aiki source text. Aiki does not define a general correspondence in which every displayed value can be read back to produce the same value.

Program text is not ordinary Aiki data. An expression that evaluates to a value is distinct from a representation of that expression as data.

3.4 Types

Aiki is dynamically typed. Types are associated with values rather than with names.

The basic value types are:

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

These types are distinct. A shaped value has type `list`; its shape does not introduce a separate value type.

Aiki has one numeric type, `number`. Integer, decimal, and rational literals all denote numbers. Numeric values are represented exactly as rational values where possible. Operations whose mathematical results are not rational may produce approximations.

The runtime also provides system values having types including:

bytes file channel module canvas store

These values support facilities provided by the standard environment and host interface and need not have literal representations.

The procedure `type` returns a symbol identifying the type of its argument.

3.5 Lists, mutation, and state

Aiki lists are compound values containing ordered sequences of values. Lists are the principal general-purpose compound data structure of the language.

The language does not expose the storage locations used to represent a list and does not provide general in-place mutation of list elements. Operations such as `prepend`, `append`, and `rest` produce list values. Indexing and field access retrieve values but are not assignment targets.

Shapes do not change this model. A shaped value remains a list with associated shape information used to provide named field access.

Assignment changes bindings rather than list storage. Whether two list values share underlying implementation storage is not defined by the language and has no specified effect on program behavior.

Aiki does not expose mutable list storage, but it does provide an explicit store system value for fixed-size mutable indexed storage. Stores are created and accessed through the store module described in section 6.15. Allocation and representation of ordinary values, and garbage collection, remain matters of implementation.

Some system values represent stateful resources. Stores hold mutable cells, channels participate in communication, file values may refer to host resources, and canvases maintain graphics state. The behavior of such state is described with the facilities that provide those values.

4. Expressions

An Aiki expression is a construct that evaluates to a value. Expressions include literal values, name references, function literals and calls, list literals, unary and binary operations, indexing, field access, pipelines, and grouped expressions.

Expression evaluation is generally left to right. Binary operators do not form a precedence hierarchy; parentheses specify grouping where the default left-to-right grouping is not intended.

4.1 Name references

```
name
```

An expression consisting of a name is a name reference. The value of the expression is the value of the first binding for that name found in the current environment or an enclosing environment.

Referencing an unbound name faults.

For example:

```
let x = 28
x
    ⇒ 28
```

4.2 Literal expressions

Numbers, booleans, runes, strings, and symbols may be written directly as expressions:

```
42
    ⇒ 42
1/3
    ⇒ 1/3

true
    ⇒ true

'a'
    ⇒ 'a'

"hello"
    ⇒ "hello"

:ready
    ⇒ :ready
```

Literal expressions evaluate to the corresponding Aiki values.

List literals and function literals are described separately below.

4.3 Function literals

```
(parameter, ...) { body }
```

A function literal evaluates to a function value.

The function retains the lexical environment in which the literal is evaluated. When the function is called, a new environment enclosed by that retained environment is created. Parameters are bound to the corresponding argument values in the new environment, and the function body is evaluated there.

For example:

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

evaluates to a function.

A function may be bound to a name:

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

Functions may have zero or more parameters:

```
() {
    return 42
}

(a, b) {
    return a + b
}
```

The last parameter may be a rest parameter:

```
(first, ...rest) {
    return rest
}
```

A rest parameter is bound to a list containing argument values remaining after the ordinary parameters have been bound.

Thus:

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

gather(1, 2, 3, 4)
    ⇒ [2, 3, 4]
```

A function body is a block. Evaluation of the function normally produces the value returned by a `return` statement. If evaluation reaches the end of the body without executing `return`, the value of the last statement evaluated is the value of the call.

4.4 Function calls

```
expression(argument, ...)
```

A function call first evaluates the expression denoting the function and then evaluates its argument expressions from left to right. The resulting function is called with the resulting argument values.

For example:

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

add(3, 4)
    ⇒ 7
```

Because functions are values, the expression being called need not be a simple name:

```
let choose = (f) {
    return f
}

choose((x) { return x * 2 })(5)
    ⇒ 10
```

Calls, indexing, and field access may be chained.

Calling a value that is not callable faults.

4.5 List literals

```
[expression, ...]
```

A list literal evaluates its element expressions from left to right and produces a list containing the resulting values.

For example:

```
[1, 2 + 3, 6]
    ⇒ [1, 5, 6]
```

An empty list is written:

`[]`

A shaped list literal begins with a shape name:

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

The shape name is associated with the resulting list as shape information and is not one of the indexed elements.

Thus, if `@point` has fields `x` and `y`:

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

then the indexed elements of `p` are `10` and `20`.

4.6 Unary expressions

```
-expression
not expression
```

The unary `-` operator requires a number and produces its arithmetic negation:

```
-42
    ⇒ -42

-(2 + 3)
    ⇒ -5
```

Applying `-` to a non-number faults.

The `not` operator interprets its operand according to Aiki's truth rules and returns a boolean:

```
not true
    ⇒ false

not false
    ⇒ true

not []
    ⇒ true

not 0
    ⇒ false
```

Unary operators may be combined. They are applied from the operator nearest the operand outward.

4.7 Binary expressions

```
expression operator expression
```

The binary operators are:

`+` `-` `*` `/` `<` `>` `<=` `>=` and or

Binary expressions are evaluated from left to right. Aiki does not assign differing precedence to these operators.

Thus:

```
2 + 3 * 4
    ⇒ 20
```

has the same grouping as:

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

and differs from:

```
2 + (3 * 4)
    ⇒ 14
```

Each operand expression is evaluated in left-to-right order.

The arithmetic operators `+`, `-`, `*`, and `/` operate on numbers. Division by zero faults.

The comparison operators `<`, `>`, `<=`, and `>=` produce boolean values.

The operators `and` and `or` interpret their operands according to Aiki's truth rules and return operand values rather than necessarily returning booleans.

For `and`, if the left value is false, that value is returned; otherwise the right value is returned:

```
0 and "yes"
    ⇒ "yes"

[] and "yes"
    ⇒ []
```

For `or`, if the left value is true, that value is returned; otherwise the right value is returned:

```
0 or 5
    ⇒ 0

[] or 5
    ⇒ 5
```

Both operands of `and` and `or` are evaluated before the operator is applied.

4.8 Pipelines

```
expression |> expression
```

A pipeline passes the value produced by the expression on its left as the first argument to the callable expression on its right.

For example:

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

5 |> double
    ⇒ 10
```

If the right side is a call containing additional arguments, the piped value precedes those arguments:

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

5 |> add(3)
    ⇒ 8
```

Pipelines associate as a left-to-right sequence:

```
value |> f |> g
```

is evaluated by applying `f` to `value` and then applying `g` to the result.

Pipeline stages may therefore be read in the same order in which they are evaluated.

A fault or a shaped error produced while evaluating a pipeline stops evaluation of the remaining stages. A shaped error remains an ordinary value and is returned as the value of the pipeline rather than being passed silently to subsequent stages.

4.9 Indexing

```
expression[index]
```

Indexing evaluates the indexed expression and the index expression and retrieves the element at the specified zero-based position.

Lists and strings may be indexed.

For example:

```
[10, 20, 30][1]
    ⇒ 20

"abc"[1]
    ⇒ 'b'
```

The index must be an integer-valued number. A negative index, an index beyond the available elements, or an attempt to index a value that does not support indexing faults.

String indexing is by rune rather than by byte.

4.10 Field access

```
expression.field
```

Field access retrieves a named field from a shaped list or an exported value from a module.

For a shaped list, the field name is interpreted according to the shape associated with the list:

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

p.x
    ⇒ 10

p.y
    ⇒ 20
```

Field access on an unshaped value, or access to a field not provided by the associated shape, faults.

For modules, field access retrieves an exported binding:

```
module.name
```

Access to a name not exported by the module faults.

4.11 Grouping and postfix composition

```
(expression)
```

Parentheses may be used to group an expression explicitly.

Because Aiki binary operators otherwise group from left to right, parentheses are the means by which another grouping is specified:

```
2 + (3 * 4)
    ⇒ 14
```

Parentheses are also used in function literals and function calls; their role is determined by their syntactic context.

Function calls, indexing, and field access are postfix operations and may be chained:

```
data[i].field
module.make()(x)
xs[0](arg)
```

Each postfix operation is applied from left to right to the value produced by the preceding expression.

5. Program structure

5.1 Programs

An Aiki program consists of a sequence of statements. Statements are evaluated in source order in a common program environment. Evaluation stops if a fault occurs.

Expressions may occur as statements:

```
println("hello")
square(5)
```

Programs may be stored in source files or entered interactively. An interactive session retains its user environment between evaluations.

A source file may also define a module, as described in section 5.7.

5.2 Bindings and assignment

A binding statement has the form:

```
let name = expression
```

The expression is evaluated and its value is bound to `name` in the current environment.

For example:

```
let x = 10

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

```
}
```

If a binding with the same name already exists in the current environment, the new binding replaces it.

Assignment has the form:

```
name = expression
```

The expression is evaluated and its value replaces the nearest existing binding of `name` in the current or an enclosing environment.

Assignment does not create a binding. Assignment to an unbound name faults.

Shape definitions use a second form of `let`:

```
let @shape [field, ...]
```

For example:

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

A shape definition associates the shape name with an ordered sequence of fields.

5.3 Conditional execution

```
if expression {
    statements
}

if expression {
    statements
} else {
    statements
}
```

The condition is evaluated first. If its value is true according to the rules of section 3.2, the first block is evaluated. Otherwise the `else` block, if present, is evaluated.

For example:

```
if x > 0 {
    println("positive")
} else {
    println("not positive")
}
```

An `else` may be followed by another `if`:

```
if x < 0 {
    println("negative")
} else if x > 0 {
    println("positive")
} else {
    println("zero")
}
```

Only the selected branch is evaluated.

The blocks of an `if` statement are evaluated in the surrounding environment and do not by themselves introduce a new environment.

5.4 Iteration

```
while expression {
    statements
}
```

A `while` statement repeatedly evaluates its condition and body. Before each iteration, the condition is evaluated. If it is false, iteration ends. Otherwise the body is evaluated and the condition is tested again.

For example:

```
let n = 3

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

The condition is interpreted according to the truth rules of section 3.2.

The body is evaluated in the surrounding environment. Bindings established in the body therefore remain in that environment.

A `return` executed within a `while` body terminates the surrounding function.

Aiki provides no separate `break` or `continue` statement.

5.5 Pattern matching

```
match expression {
    pattern {
        statements
    }
    ...
}
```

The expression following `match` is evaluated once. Its value is compared with the patterns in source order. The block belonging to the first matching pattern is evaluated. Later patterns are not examined.

Patterns may be literals, names, the wildcard `_`, lists, or shaped lists.

A literal pattern matches an equal literal value:

```
match x {
    0 {
        println("zero")
    }
    1 {
        println("one")
    }
}
```

A name pattern matches any value and binds that value to the name:

```
match x {
    value {
        println(value)
    }
```

```
}
```

The wildcard `_` matches any value without creating a binding.

List patterns match lists having the corresponding structure:

```
match xs {
    [a, b] {
        println(a)
        println(b)
    }
}
```

A shaped-list pattern additionally requires the corresponding shape:

```
match p {
    [@point, x, y] {
        println(x)
        println(y)
    }
}
```

When a pattern matches, its bindings are established in a new environment enclosing the environment of the `match`. Those bindings are available while evaluating the selected arm and do not remain afterward.

If no pattern matches, no arm is evaluated.

5.6 Return

```
return expression
```

A `return` statement evaluates its expression and returns the resulting value from the current function.

For example:

```
let abs_value = (x) {
    if x < 0 {
        return -x
    }
    return x
}
```

A return occurring within an if, while, match, or select propagates outward through the enclosing statements and completes the current function call.

5.7 Packages and modules

A module source file begins with a package declaration:

```
package "name"
```

For example:

```
package "geometry"
```

The package declaration gives the module its package name.

Definitions within a module are private unless explicitly exported.

The module facilities `import`, `use`, and `export` provide access between modules.

```
export(:name, ...)
```

marks named bindings for export.

```
import("module")
```

loads a module and returns a module value. Exported bindings may then be accessed through field notation:

```
let math = import("math")
math.sqrt(9, 8)
```

When no explicit bare package X exists and X/native is supplied, the bare name X resolves to X/native. An explicit X package takes precedence, and an X/ffi package alone never becomes a bare default. Thus import("math") selects math/native in the supplied distribution, while import("math/ffi") remains explicit.

Selected exported names may instead be introduced into the current environment:

```
import("module", :name, ...)
```

The `use` facility introduces the exported names of a module into the current environment:

```
use("turtle/simple")
```

Modules execute in environments separate from those of their importers. Only exported bindings are visible through the module interface.

`import`, `use`, and `export` are ordinary callable facilities rather than statement forms.

5.8 Concurrent selection

```
select {
let value = recv(channel) {
statements
}
recv(other) {
statements
}
default {
statements
}
}
```

A select statement contains one or more receive cases, or a default case, and may contain both. A receive case has the form recv(expression) { ... } or let name = recv(expression) { ... }. The latter binds the received value to name for the selected arm.

Each receive expression is evaluated exactly once when the select is entered and must produce a channel. If one or more receives are ready, one ready case is selected; when several

are ready, Aiki does not specify source-order preference, fairness, or scheduling order.

If no receive is ready and a default arm is present, the default arm is selected immediately. Otherwise evaluation waits until a receive can proceed. Select currently provides receive cases only; sending remains an ordinary send operation.

The selected block is evaluated in a new environment enclosed by the environment of the select. A name bound by a receive case is local to that arm. The value produced by the selected block is the value of the select statement.

6. Standard library

This chapter describes the procedures and facilities supplied by the Aiki standard library.

The initial Aiki environment contains a number of names bound to useful procedures. These bindings form the prelude and are available without importing a module. Additional standard facilities are organized into library modules and are made available through the module operations described in section 5.7.

The procedures described in this chapter operate on the values and system facilities described in the preceding chapters. Unless otherwise stated, applying a procedure to values outside its specified domain faults.

6.1 The prelude

The prelude contains the standard bindings present in the initial Aiki environment. Most are procedures providing fundamental operations needed directly by programs; others provide interaction with the evaluator or runtime.

The prelude includes facilities for function application and program loading, input and output, basic list operations, type and value inspection, conversion, numeric utilities, concurrency, and interactive use.

The following sections describe the bindings supplied by the prelude.

6.1.1 Function application and program loading

```
apply(function, arguments)
```

`arguments` must be a list. `apply` calls `function` using the elements of `arguments` as its argument values.

For example:

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

apply(add, [3, 4])
    ⇒ 7
```

The first argument must be callable and the second must be a list; otherwise evaluation faults. Function application otherwise follows the rules of section 7.2.4.

```
load(path)
```

`path` must be a string naming an Aiki source file. `load` reads the file, performs lexical and syntactic analysis using the Aiki grammar, and evaluates the resulting program in the current environment.

Bindings established by the loaded program are therefore available in that environment after evaluation. The value returned by `load` is the value produced by evaluation of the loaded program.

If the file cannot be located or read, or if lexical or syntactic analysis fails, `load` returns a shaped `@error` value. Supplying an argument of the wrong type faults.

```
stack_limit(n)
```

`n` must be an integer greater than or equal to 1. `stack_limit` sets the maximum number of active non-tail function-call frames.

Proper tail calls reuse the current function frame and therefore do not consume additional depth against this limit. If a non-tail call would exceed the limit, evaluation faults.

`stack_limit` returns `[]`.

6.1.2 Input and output

```
print(value, ...)
```

`print` writes its arguments to standard output without adding a newline or separators between arguments.

String values are written as their contents. Other values are written using their displayed representation.

For example:

```
print("x = ", 42)
```

produces:

```
x = 42
```

`print` accepts zero or more arguments and returns `[]`.

```
println(value, ...)
```

`println` writes its arguments in the same manner as `print` and then writes a newline.

For example:

```
println("x = ", 42)
```

produces:

```
x = 42
```

A call with no arguments writes only a newline.

`println` returns `[]`.

```
read()
```

`read` reads one line from standard input and returns it as a string. The terminating newline, when present, is not included in the returned string.

If end of input is reached before any characters are read, `read` returns:

```
[@end]
```

An input failure other than end of input returns a shaped `@error` value.

```
input()
```

`input` is an alias for `read` and has the same behavior.

6.1.3 Basic sequence operations

```
first(value)
```

`value` must be a list or string.

For a nonempty list, `first` returns its first element:

```
first([1, 2, 3])
    ⇒ 1
```

For a nonempty string, `first` returns its first rune:

```
first("abc")
    ⇒ 'a'
```

Applying `first` to an empty list or empty string faults.

```
rest(value)
```

`value` must be a list or string.

For a list, `rest` returns a list containing all elements except the first:

```
rest([1, 2, 3])
    ⇒ [2, 3]
```

For a string, `rest` returns the string following its first rune:

```
rest("abc")
    ⇒ "bc"
```

The rest of an empty list is `[]`; the rest of an empty string is `""`.

```
length(value)
```

`value` must be a list or string. `length` returns the number of elements in a list or the number of runes in a string.

```
length([1, 2, 3])
    ⇒ 3

length("abc")
    ⇒ 3
```

```
prepend(list, value)
```

Returns a new list whose first element is `value`, followed by the elements of `list`.

```
prepend([2, 3], 1)
    ⇒ [1, 2, 3]
```

If `list` is shaped, the resulting list retains the same shape.

```
append(list, value)
```

Returns a new list containing the elements of `list` followed by `value`.

```
append([1, 2], 3)
    ⇒ [1, 2, 3]
```

If `list` is shaped, the resulting list retains the same shape.

```
empty(value)
```

`value` must be a list or string. `empty` returns `true` when the value contains no elements or runes and `false` otherwise.

```
empty([])
    ⇒ true

empty([1])
    ⇒ false

empty("")
    ⇒ true
```

Applying any of these procedures to a value outside its stated domain faults.

6.1.4 Types, equality, and inspection

```
type(value)
```

`type` returns a symbol naming the type of `value`.

For example:

```
type(42)
    ⇒ :number

type("hello")
    ⇒ :string

type([1, 2])
    ⇒ :list
```

Shaped lists have type `:list`; their shape is obtained separately with `shape`.

```
inspect(value)
```

`inspect` returns a string containing the displayed representation of `value`.

For example:

```
inspect(42)
    ⇒ "42"

inspect(:ready)
    ⇒ ":ready"

inspect([1, 2])
    ⇒ "[1, 2]"
```

The returned string is intended for inspection and is not guaranteed to be Aiki source that can be read back to reproduce the value.

```
equal(a, b)
```

`equal` compares two values for equality and returns a boolean.

Values of different types are not equal. Numbers are equal when they denote the same rational value. Booleans, runes, strings, and symbols are equal when their contained values are equal.

For example:

```
equal(1/2, 2/4)
    ⇒ true

equal("a", "a")
    ⇒ true

equal(:a, :b)
    ⇒ false
```

Other value types, including lists and functions, are not considered equal by `equal`.

```
ord(value)
```

`value` must be a rune or string. For a rune, `ord` returns its Unicode code point as a number.

```
ord('A')
    ⇒ 65
```

For a nonempty string, `ord` returns the code point of its first rune:

```
ord("ABC")
    ⇒ 65
```

For the empty string, `ord` returns `0`. Applying `ord` to another type faults.

```
chr(number)
```

`number` must be an integer representing a Unicode code point. `chr` returns the corresponding rune.

```
chr(65)
    ⇒ 'A'
```

A non-integer argument faults. An integer outside the Unicode code-point range returns a shaped `@error` value.

```
shape(value)
```

If `value` is a shaped list, `shape` returns the symbol naming its shape. Otherwise it returns `:list`.

For example:

```
let @point [x, y]

shape([@point, 10, 20])
    ⇒ :point

shape([1, 2])
    ⇒ :list
```

```
is_error(value)
```

`is_error` returns `true` when `value` is a shaped list whose shape is `@error`, and `false` otherwise.

```
is_error([@error, :data, "bad value"])
    ⇒ true

is_error([1, 2])
    ⇒ false
```

6.1.5 Conversion

```
to_str(value)
```

`to_str` converts value to a string representation.

For strings, the value itself is returned. Runes are converted to the corresponding one-rune string. Numbers are represented in their rational form, booleans as `"true"` or `"false"`, and symbols with their leading colon.

For example:

```
to_str(42)
    ⇒ "42"

to_str(1/3)
    ⇒ "1/3"

to_str('A')
    ⇒ "A"

to_str(true)
    ⇒ "true"
```

```
to_str(:ready)
    ⇒ ":ready"
```

For other values, `to_str` uses their displayed representation.

```
to_decimal(number, places)
```

`number` must be a number and `places` must be a non-negative integer. `to_decimal` returns a string containing a decimal representation of `number` with the specified number of digits following the decimal point.

The value is computed from the exact rational. Digits beyond the requested number of places are discarded by rounding half away from zero, so a value exactly halfway between two representable results rounds away from zero.

For example:

```
    to_decimal(3.14159, 2)
    ⇒ "3.14"

    to_decimal(1/3, 4)
    ⇒ "0.3333"

    to_decimal(1/2, 0)
    ⇒ "1"
```

The result is a string and does not replace the exact numeric value from which it was produced. A negative places faults.

```
to_number(string)
```

`string` must be a string containing a valid Aiki number representation. `to_number` returns the corresponding number.

For example:

```
to_number("42")
    ⇒ 42

to_number("3.14")
    ⇒ 157/50

to_number("1/3")
    ⇒ 1/3
```

If the string does not contain a valid number representation, `to_number` returns a shaped `@error` value. Supplying a non-string argument faults.

6.1.6 Numeric utilities

```
abs(number)
```

`abs` returns the absolute value of `number`.

For example:

```
abs(-5)
    ⇒ 5
```

```
abs(3)
    ⇒ 3

abs(0)
    ⇒ 0
```

The argument must be a number; otherwise evaluation faults.

```
truncate(number)
```

`truncate` returns the integer obtained by discarding the fractional part of `number`. Truncation is toward zero.

For example:

```
truncate(7/2)
    ⇒ 3

truncate(-7/2)
    ⇒ -3

truncate(4)
    ⇒ 4
```

The result is an Aiki number representing an integer. The argument must be a number; otherwise evaluation faults.

6.1.7 Concurrency

```
spawn(function, argument, ...)
```

spawn begins concurrent evaluation of function with the supplied argument values and returns immediately.

The spawned function executes in an isolated environment. Its parameters are bound to the supplied arguments, and the prelude remains available, but bindings from the environment in which the function was created are not visible to the spawned computation. Values needed by the computation must therefore be passed explicitly as arguments.

The isolated environment does contain the shape definitions visible where the function was created. A shape definition is an ordered sequence of field names rather than a value, and cannot be passed as an argument; supplying it allows field access on shaped arguments to behave as it does elsewhere. A shape defined within the spawned function shadows a definition of the same name supplied in this way.

For example:

```
let worker = (ch) {
  send(ch, 42)
}

let ch = channel()

spawn(worker, ch)

recv(ch)
⇒ 42
```

A shaped argument may be read by field name:

```
let @point [x, y]

let reader = (p, out) {
  send(out, p.x)
}

let ch = channel()

spawn(reader, [@point, 10, 20], ch)

recv(ch)
⇒ 10
```

function must be a user function. spawn returns true after starting the computation.

A fault produced within a spawned computation does not propagate to the computation that called spawn. A computation waiting to receive a value that the faulted computation would have sent continues to wait.

```
channel()
```

channel creates and returns a new unbuffered channel.

Channels provide synchronous communication between concurrent computations.

```
send(channel, value)
```

send sends value on channel. Because channels are unbuffered, the operation blocks until another computation receives the value.

send returns true.

The first argument must be a channel; otherwise evaluation faults.

```
recv(channel)
```

recv waits until a value is available on channel and returns that value.

For example:

```
let ch = channel()

let worker = (out) {
  send(out, :done)
}

spawn(worker, ch)

recv(ch)
⇒ :done
```

The argument must be a channel; otherwise evaluation faults.

The select statement described in section 5.8 coordinates one or more receives without adding another prelude procedure. Ordinary channels remain unbuffered and sendable; receive-only event channels may also be supplied by facilities such as time.after.

6.1.8 Interactive facilities

```
quit()
```

`quit` terminates the interactive Aiki session.

```
reset()
```

`reset` restores the interactive environment to its initial state. User bindings and imported-module state associated with the current session are discarded, and open canvases are closed.

`reset` takes no arguments.

```
delete(name)
```

`name` must be a string. `delete` removes the named binding from the current user environment.

It returns `true` if a binding was removed and `false` if no such user binding existed.

For example:

```
let print = 1
delete("print")
    ⇒ true
```

After the user binding is removed, a prelude binding of the same name becomes visible again.

`delete` is available only in an interactive user environment. Supplying an argument of the wrong type or using it where no user environment exists faults.

```
help()
help(name)
```

`help` displays concise help for Aiki syntax, prelude procedures, and standard-library modules.

With no argument, `help` displays an index of available syntax, prelude bindings, and modules. With an argument, it displays concise help for the named facility, including its syntax where available.

Module functions may be named in qualified form, for example:

```
help("list.map")
```

`help` returns `[]`.

```
doc(name)
```

`doc` displays the full documentation available for a syntax form, prelude procedure, module, or qualified module procedure.

For example:

```
doc("spawn")
```

```
doc("list.map")
```

`doc` returns `[]`.

The argument to `doc` must be a string; otherwise evaluation faults.

6.2 Lists

The `list` module provides higher-order operations, traversal and search, aggregation, and list construction utilities.

It is imported in the usual manner:

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

The module exports:

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

6.2.1 Mapping, filtering, and reduction

```
map(list, f)
```

`map` applies `f` to each element of `list`, from first to last, and returns a new list containing the resulting values.

For example:

```
list.map([1, 2, 3], (x) { x * 2 })
    ⇒ [2, 4, 6]
```

```
filter(list, f)
```

`filter` applies `f` to each element of `list`, from first to last, and returns a new list containing those elements for which the result is true according to Aiki's truth rules.

For example:

```
list.filter([1, 2, 3, 4], (x) { x > 2 })
    ⇒ [3, 4]
```

```
reduce(list, initial, f)
```

`reduce` combines the elements of `list` from left to right. The initial accumulator is `initial`. For each element, `f` is called with the current accumulator and that element; the returned value becomes the accumulator for the next step.

For example:

```
list.reduce([1, 2, 3], 0, (acc, x) {
    acc + x
})
    ⇒ 6
```

For an empty list, `reduce` returns `initial` without calling `f`.

```
each(list, f)
```

`each` calls `f` once for each element of `list`, from first to last. It is intended for computations performed for their effects.

For example:

```
list.each([1, 2, 3], (x) {
    println(x)
})
```

produces:

```
1
2
3
```

`each` returns `true`.

The list argument to each of these procedures must support the list operations used by the module, and the function argument must be callable; violations fault during evaluation.

6.2.2 Traversal and search

```
find(list, f)
```

`find` applies `f` to the elements of `list` from first to last and returns the first element for which the result is true according to Aiki's truth rules.

For example:

```
list.find([1, 2, 3, 4], (x) {
    x > 2
})
    ⇒ 3
```

If no element satisfies `f`, `find` returns:

```
[@error, :lookup, "not found"]
```

```
any(list, f)
```

`any` applies `f` to the elements of `list` from first to last. It returns `true` as soon as an application of `f` produces a true value. If no element does so, it returns `false`.

For example:

```
list.any([1, 2, 3], (x) {
    x > 2
})
    ⇒ true
```

For an empty list, `any` returns `false`.

```
all(list, f)
```

`all` applies `f` to the elements of `list` from first to last. It returns `false` as soon as an application of `f` produces a false value. If every application produces a true value, it returns `true`.

For example:

```
list.all([1, 2, 3], (x) {
    x > 0
})
    ⇒ true
```

For an empty list, `all` returns `true`.

The predicate function supplied to these procedures may return any Aiki value; its result is interpreted according to the truth rules of section 3.2.

6.2.3 Aggregation

```
sum(list)
```

`sum` returns the sum of the elements of `list`, evaluated from first to last.

For example:

```
list.sum([1, 2, 3, 4])
    ⇒ 10
```

For an empty list, `sum` returns `0`.

The elements must support numeric addition; otherwise evaluation faults.

```
max(list)
```

`max` returns the greatest element of `list` according to Aiki's `>` operator.

For example:

```
list.max([3, 7, 2, 5])
    ⇒ 7
```

If `list` is empty, `max` returns:

```
[@error, :empty, "empty list"]
```

The elements must support comparison with `>`; otherwise evaluation faults.

```
min(list)
```

`min` returns the least element of `list` according to Aiki's `<` operator.

For example:

```
list.min([3, 7, 2, 5])
    ⇒ 2
```

If `list` is empty, `min` returns:

```
[@error, :empty, "empty list"]
```

The elements must support comparison with `<`; otherwise evaluation faults.

6.2.4 Construction and transformation

```
reverse(list)
```

`reverse` returns a new list containing the elements of `list` in reverse order.

For example:

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

For an empty list, `reverse` returns `[]`.

```
concat(a, b)
```

`concat` returns a list consisting of the elements of `a` followed by the elements of `b`.

For example:

```
list.concat([1, 2], [3, 4])
    ⇒ [1, 2, 3, 4]
```

If `b` is empty, the result has the elements of `a`. If `a` is empty, the result has the elements of `b`.

```
range(start, end)
```

`range` returns a list of successive numbers beginning with `start` and increasing by one while the current value is less than `end`. The upper bound is therefore excluded.

For example:

```
list.range(0, 5)
    ⇒ [0, 1, 2, 3, 4]
list.range(3, 3)
    ⇒ []
```

If `start` is greater than `end`, `range` returns `[]`.

6.3 Strings

The `string` module provides operations for decomposition, searching, comparison, replacement, construction, whitespace handling, classification, and case conversion.

It is imported as:

```
let string = import("string")
```

6.3.1 Substrings and decomposition

```
substring(s, start, end)
```

`substring` returns the portion of `s` beginning at `start` and ending before `end`.

For example:

```
string.substring("hello", 1, 4)
    ⇒ "ell"
```

String positions are measured in runes. If `end` extends beyond the end of the string, only the available runes are returned.

```
split(s, delim)
```

`split` divides `s` at occurrences of `delim` and returns the resulting parts as a list of strings.

For example:

```
string.split("a,b,c", ",")
    ⇒ ["a", "b", "c"]
```

The delimiter is not included in the returned parts.

```
chars(s)
```

`chars` returns a list containing the runes of `s` in order.

For example:

```
string.chars("abc")
    ⇒ ['a', 'b', 'c']
```

```
lines(s)
```

`lines` divides `s` at newline characters and returns the resulting lines as a list of strings.

For example:

```
string.lines("one\ntwo\nthree")
    ⇒ ["one", "two", "three"]
```

```
words(s)
```

`words` removes leading and trailing whitespace, divides the remaining string at spaces, discards empty parts, and returns the resulting strings.

For example:

```
string.words("  one  two three  ")
    ⇒ ["one", "two", "three"]
```

`words` uses spaces as separators after trimming; it does not treat every whitespace character as an internal word separator.

6.3.2 Searching and comparison

```
index_of(s, sub)
```

`index_of` returns the rune index of the first occurrence of `sub` in `s`.

```
string.index_of("hello", "ll")
    ⇒ 2
```

If `sub` does not occur, it returns:

```
[@error, :lookup, "not found"]
```

```
last_index_of(s, sub)
```

`last_index_of` returns the rune index of the last occurrence of `sub` in `s`. If `sub` does not occur, it returns `[@error, :lookup, "not found"]`.

```
contains(s, sub)
```

`contains` returns `true` if `sub` occurs in `s`, and `false` otherwise.

```
starts_with(s, prefix)
ends_with(s, suffix)
```

These procedures return `true` when `s` begins with `prefix` or ends with `suffix`, respectively, and `false` otherwise.

```
occurrences(s, sub)
```

`occurrences` returns the number of occurrences of `sub` in `s`, as determined by splitting `s` at occurrences of `sub`.

```
compare(a, b)
```

`compare` compares two strings lexicographically by rune value. It returns `-1` if `a` precedes `b`, `0` if they are equal, and `1` if `a` follows `b`.

```
string.compare("abc", "abd")
    ⇒ -1

string.compare("abc", "abc")
    ⇒ 0
```

6.3.3 Replacement and construction

```
join(list, delim)
```

`join` concatenates the strings in `list`, placing `delim` between successive elements.

```
string.join(["a", "b", "c"], "-")
    ⇒ "a-b-c"
```

For an empty list, `join` returns `""`.

```
replace(s, old, new)
```

`replace` returns a string in which all occurrences of `old` have been replaced by `new`.

```
string.replace("hello", "l", "L")
    ⇒ "heLLo"
```

```
replace_first(s, old, new)
```

`replace_first` replaces only the first occurrence of `old`. If `old` does not occur, `s` is returned unchanged.

```
repeat(s, n)
```

`repeat` returns a string consisting of `n` successive copies of `s`.

```
string.repeat("ab", 3)
    ⇒ "ababab"
```

```
reverse_string(s)
```

`reverse_string` returns the runes of `s` in reverse order.

```
insert(s, pos, sub)
```

`insert` returns a string with `sub` inserted at rune position `pos`.

```
remove(s, start, end)
```

`remove` returns a string with the runes beginning at `start` and ending before `end` removed.

6.3.4 Whitespace and padding

```
is_whitespace(r)
```

`is_whitespace` returns `true` if `r` is a space, tab, newline, or carriage return, and `false` otherwise.

```
trim_start(s)
trim_end(s)
trim(s)
```

`trim_start` removes leading whitespace, `trim_end` removes trailing whitespace, and `trim` removes both.

```
string.trim("  hello  ")
    ⇒ "hello"
```

```
pad_start(s, length, value)
pad_end(s, length, value)
```

These procedures extend `s` to at least `length` runes by adding the string representation of `value` at the beginning or end. If `s` is already at least that length, it is returned unchanged.

```
center(s, length, value)
```

`center` pads both sides of `s` to reach the requested length. If an unequal amount of padding is required, the extra padding is placed on the right.

```
truncate(s, length)
```

`truncate` returns at most the first `length` runes of `s`. If `s` is no longer than `length`, it is returned unchanged.

6.3.5 Rune and string classification

```
is_digit(r)
```

Returns `true` if `r` is one of the ASCII decimal digits `0` through `9`.

```
is_alpha(r)
```

Returns `true` if `r` is one of the ASCII letters `A` through `Z` or `a` through `z`.

```
is_upper(r)
is_lower(r)
```

These procedures test whether `r` is an ASCII upper-case or lower-case letter.

```
is_alnum(r)
```

Returns `true` if `r` satisfies either `is_alpha` or `is_digit`.

```
is_numeric(s)
```

Returns `true` if `s` is nonempty and every rune in it is an ASCII decimal digit.

```
is_alphabetic(s)
```

Returns `true` if `s` is nonempty and every rune in it is an ASCII letter.

The empty string is neither numeric nor alphabetic.

6.3.6 Case conversion

```
upper(s)
lower(s)
```

`upper` and `lower` return strings converted to upper or lower case according to Unicode case-conversion rules.

```
string.upper("hello")
    ⇒ "HELLO"
string.lower("HELLO")
    ⇒ "hello"
```

```
upper_rune(r)
lower_rune(r)
```

These procedures return the Unicode upper-case or lower-case form of rune `r`.

```
capitalize(s)
```

`capitalize` converts the first rune of `s` to upper case and the remaining substring to lower case. The empty string is returned unchanged.

```
string.capitalize("hELLO")
    ⇒ "Hello"
```

```
title(s)
```

`title` divides `s` into words using `words`, capitalizes each word, and joins the resulting words with single spaces.

```
equal_ignore_case(a, b)
```

`equal_ignore_case` returns the result of comparing `lower(a)` with `lower(b)`.

6.4 Bytes

Aiki supplies byte operations through the modules `bytes/ffi` and `bytes/native`. Both modules export the same procedures for conversion, access, length, and slicing, but use different representations.

The byte values in either implementation consist of integers in the range 0 through 255.

6.4.1 Byte values and conversion

```
str_to_bytes(s)
```

`str_to_bytes` encodes string `s` as UTF-8 bytes and returns the resulting byte value.

For example, conceptually:

```
str_to_bytes("ABC")
```

contains the byte values:

`65  66  67`

```
bytes_to_str(bytes)
```

`bytes_to_str` converts a byte value to a string by interpreting its bytes as UTF-8 encoded text.

For example:

```
bytes_to_str(bytes_new([65, 66, 67]))
```

⇒ "ABC"

```
bytes_new(list)
```

`bytes_new` creates a byte value from the elements of `list`. Each element must be an integer between `0` and `255`, inclusive.

For example:

```
bytes_new([65, 66, 67])
```

creates a byte value containing the bytes for `"ABC"`.

Each element must be an integer between 0 and 255, inclusive. Invalid elements are rejected. The supplied implementations differ in whether some invalid inputs produce faults or shaped errors, but valid byte values have the same observable contents.

6.4.2 Byte access and slicing

```
bytes_get(bytes, i)
```

`bytes_get` returns the byte at zero-based index `i` as an Aiki number.

For example:

```
let b = str_to_bytes("Hi")
bytes_get(b, 0)
    ⇒ 72

bytes_get(b, 1)
    ⇒ 105
```

The index must be an integer within the bounds of the byte value; otherwise evaluation faults.

```
bytes_length(bytes)
```

`bytes_length` returns the number of bytes contained in `bytes`.

```
bytes_length(str_to_bytes("Hi"))
    ⇒ 2
```

The result is a number representing a non-negative integer.

```
bytes_slice(bytes, start, end)
```

`bytes_slice` returns a new byte value containing the bytes beginning at `start` and ending before `end`.

For example:

```
let b = bytes_new([65, 66, 67])

bytes_to_str(bytes_slice(b, 0, 2))
    ⇒ "AB"
```

`start` and `end` are zero-based integer indexes. `start` is inclusive and `end` is exclusive.

6.4.3 Supplied implementations

The module `bytes/ffi` represents byte values using the host runtime's opaque `bytes` type. Such values have type:

```
:bytes
```

They are accessed through the byte procedures described above rather than through ordinary list operations.

The module `bytes/native` implements byte values in Aiki as shaped lists having shape `@bytes`. Their elements are ordinary Aiki numbers in the range `0` through `255`.

For example:

```
let b = str_to_bytes("Hi")
shape(b)
    ⇒ :bytes
```

A native byte value remains a list and may therefore also be used with ordinary list operations such as `first`, `rest`, and `length`.

Both implementations export:

`str_to_bytes` bytes_to_str bytes_new bytes_get bytes_length bytes_slice

Programs that use only these exported operations may select either implementation without changing the byte-processing interface.

6.5 Hash tables

Aiki supplies hash-table operations through the modules `hash/ffi` and `hash/native`. Both modules provide the same interface and represent updates functionally: `put` and `del` return new hash tables rather than modifying their arguments.

A hash table associates keys with Aiki values. Key comparison uses the prelude procedure `equal`.

6.5.1 Hash values

```
hash_code(value)
```

`hash_code` returns a numeric hash code for `value`.

Numbers, booleans, runes, strings, symbols, and lists have defined hash computations. The hash of a list incorporates its shape and the hashes of its elements recursively. Other value types produce the hash code `0`.

Hash codes are used by the hash-table implementation to select buckets. Programs should not depend on a particular hash code except when the numeric result itself is required.

6.5.2 Construction and lookup

```
new()
```

`new` returns an empty hash table.

For example:

```
let h = hash.new()
```

```
get(h, key)
```

`get` returns the value associated with `key` in `h`.

For example:

```
let h = hash.put(hash.new(), "name",
"Mochi")
hash.get(h, "name")
    ⇒ "Mochi"
```

If no equal key is present, `get` returns:

```
[@error, :key, "key not found"]
has(h, key)
```

`has` returns `true` if `h` contains `key` and `false` otherwise.

```
hash.has(h, "name")
    ⇒ true

hash.has(h, "age")
    ⇒ false
```

6.5.3 Update and deletion

```
put(h, key, value)
```

`put` returns a new hash table in which `key` is associated with `value`.

If an equal key is already present, its associated value is replaced. Otherwise a new key-value association is added.

For example:

```
let h1 = hash.new()
let h2 = hash.put(h1, :name, "Mochi")
let h3 = hash.put(h2, :name, "Aiki")

hash.get(h3, :name)
    ⇒ "Aiki"
```

`put` does not modify `h`.

```
del(h, key)
```

`del` returns a new hash table from which associations having a key equal to `key` have been removed.

If no such key exists, the returned table has the same associations as `h`.

`del` does not modify `h`.

6.5.4 Keys and values

```
keys(h)
```

`keys` returns a list containing the keys present in `h`.

```
values(h)
```

`values` returns a list containing the associated values present in `h`.

The ordering of the returned lists is determined by the internal bucket organization and should not be relied upon.

For a given hash table, `keys` and `values` traverse associations in the same order, so corresponding positions refer to the same association.

6.5.5 Supplied implementations

The module `hash/ffi` uses the host runtime's `modulo` operation when reducing hash codes to bucket indexes.

The module `hash/native` implements the corresponding modulo operation entirely in Aiki.

Both implementations use 64 buckets and export:

`hash_code` new get put has del keys values

Their hash-table values are ordinary Aiki lists rather than values of a distinct hash-table type. Programs using the exported interface may select either implementation without changing their hash-table operations.

6.6 Mathematics

Aiki supplies mathematical procedures through the modules `math/ffi` and `math/native`. The modules provide arithmetic utilities and selected transcendental functions beyond the operators and prelude procedures described earlier.

Unless otherwise stated, arguments to the procedures in this section must be numbers.

6.6.1 Rounding and remainder

```
floor(x)
```

`floor` returns the greatest integer not greater than `x`.

```
math.floor(7/2)
    ⇒ 3
math.floor(-7/2)
    ⇒ -4
```

```
ceil(x)
```

`ceil` returns the least integer not less than `x`.

```
math.ceil(7/2)
    ⇒ 4
math.ceil(-7/2)
    ⇒ -3
```

```
modulo(a, b)
```

`a` and `b` must be integers. `modulo` returns the remainder corresponding to division of `a` by `b`.

For example:

```
math.modulo(13, 4)
    ⇒ 1
```

The divisor must not be zero.

6.6.2 Trigonometric functions

The `math/ffi` module provides:

```
sin(x)
cos(x)
```

These procedures compute the sine and cosine of `x`, where `x` is interpreted in radians. They use host floating-point mathematics internally and return the resulting approximation as an Aiki number.

The `math/native` module provides:

```
sin(x, terms)
cos(x, terms)
```

These procedures compute sine and cosine using finite Taylor series evaluated with Aiki rational arithmetic. terms specifies the number of `terms` used in the approximation. The term count bounds the work performed rather than specifying a target error. For a fixed argument, additional terms generally improve the approximation, but the gain depends on the magnitude of x and is not linear; no fixed term count guarantees a fixed number of correct digits for all arguments. These procedures perform no range reduction, so larger-magnitude arguments may require more terms and produce larger intermediate rationals.

6.6.3 Square root

The `math/ffi` module provides:

```
sqrt(x)
```

It computes the non-negative square root of `x` using host mathematics and returns the approximation as an Aiki number.

```
let math_ffi = import("math/ffi")
math_ffi.sqrt(9)
    ⇒ 3
```

A negative argument faults.

The `math/native` module provides:

```
sqrt(x, iterations)
```

It computes a square-root approximation using Newton's method with Aiki rational arithmetic. iterations specifies the number of iterations performed.

The result is an exact rational close to the square root rather than the square root itself, which is not in general a rational number. It is therefore shown here as a decimal:

```
to_decimal(math.sqrt(9, 8), 6)
⇒ "3.000000"

to_decimal(math.sqrt(894, 8), 6)
⇒ "29.899833"
```

A negative argument returns `[@error, :math, "sqrt of negative"]`.

The iterations argument bounds the work performed rather than the error tolerated. Newton's method roughly doubles the number of correct digits at each step until the value stops changing, after which further iterations leave the result unchanged while approximately doubling the size of the rational used to represent it. The number of iterations needed depends on the magnitude of the argument, because the method is seeded with `x/2`: the square root of `9` converges by the fifth iteration and the square root of `894` by the eighth.

For `894` the successive approximations are:

```
iteration 5    30.732996
iteration 6    29.911126
iteration 7    29.899835
iteration 8    29.899833
iteration 9    29.899833
```

The eighth and ninth iterations agree to the places shown; the ninth produces a rational approximately twice the size of the eighth.

A caller chooses a count by determining where the value stops changing for the arguments the program supplies, rather than by supplying a large number.

6.6.4 Supplied implementations

The math/ffi module exports:

`floor` `ceil` `modulo` `sin` `cos` `sqrt`

These procedures use host-runtime facilities where appropriate. Trigonometric functions and square root pass through host floating-point computation. A host float64 result is converted at the boundary to the exact rational representation of that returned IEEE-754 binary64 value and thereafter is an ordinary Aiki number.

The `math/native` module exports:

`sin` `cos` `sqrt`

These procedures are implemented in Aiki. Sine and cosine use finite Taylor series, and square root uses Newton's method. Their computations use Aiki rational arithmetic, with approximation controlled explicitly by the number of terms or iterations supplied by the caller.

Both modules return ordinary Aiki numbers; neither introduces a separate floating-point value type or retained provenance flag. The choice of math/ffi or math/native identifies where approximation occurred.

6.7 Random numbers

The `random` module provides pseudo-random integer generation and explicit seeding.

It is imported as:

```
let random = import("random")
```

6.7.1 Seeding

```
seed(n)
```

`seed` initializes the pseudo-random number generator from integer `n`.

Using the same seed produces the same sequence of pseudo-random values.

For example:

```
random.seed(42)
```

`seed` returns `[]`.

The argument must be an integer; otherwise evaluation faults.

6.7.2 Random values

```
random(max)
```

`random` returns a pseudo-random integer `n` satisfying:

```
0 <= n < max
```

For example:

```
random.random(10)
```

returns one of the integers from `0` through `9`.

`max` must be a positive integer. A non-integer or non-positive argument faults.

Successive calls use and advance the current pseudo-random generator state. Calling `seed` establishes a new deterministic sequence.

6.8 Regular expressions

The `regex/ffi` module provides regular-expression matching, searching, replacement, and splitting using RE2-style regular expressions.

It is imported as:

```
let regex = import("regex/ffi")
```

Invalid regular-expression patterns return a shaped `@error` value.

6.8.1 Matching

```
matches(s, pattern)
```

`matches` returns `true` if `pattern` matches anywhere in string `s`, and `false` otherwise.

For example:

```
regex.matches("hello world", "^hello")
    ⇒ true

regex.matches("hello world", "^world")
    ⇒ false
```

The pattern need not match the entire string unless its expression requires this.

6.8.2 Searching

```
find(s, pattern)
```

`find` returns the first non-overlapping match of `pattern` in `s` as a shaped value:

```
[@match, start, end, text]
```

`start` is the position at which the match begins, `end` is the position immediately following the match, and `text` is the matched substring.

The positions are rune indexes into `s`, so a position returned by `find` may be used directly with string indexing and with the string module's procedures.

For example:

```
regex.find("one 123 two", "[0-9]+")
    ⇒ [@match, 4, 7, "123"]
```

If no match is found, `find` returns a shaped `@error` value with category `:regex`.

```
find_all(s, pattern)
```

`find_all` returns a list of all non-overlapping matches, each represented as an `@match` value of the form described above.

For example:

```
regex.find_all("a1 b22 c333", "[0-9]+")
    ⇒ [
        [@match, 1, 2, "1"],
        [@match, 4, 6, "22"],
        [@match, 8, 11, "333"]
      ]
```

If there are no matches, `find_all` returns `[]`.

6.8.3 Replacement

```
replace(s, pattern, replacement)
```

`replace` returns a string in which all non-overlapping matches of `pattern` have been replaced by `replacement`.

Capture groups may be referenced in the replacement using forms such as `$1`, `$2`, and so forth.

For example:

```
regex.replace("abc123", "([a-z]+)([0-9]+)",
"$2-$1")
    ⇒ "123-abc"
```

```
replace_first(s, pattern, replacement)
```

`replace_first` replaces only the first match of `pattern`. If no match occurs, `s` is returned unchanged.

In the current implementation, `replace_first` inserts `replacement` literally; capture references in the replacement are not expanded.

6.8.4 Splitting

```
split(s, pattern)
```

`split` divides `s` at matches of `pattern` and returns the resulting substrings as a list.

For example:

```
regex.split("one, two; three", "[,;] *")
    ⇒ ["one", "two", "three"]
```

The substrings matched by `pattern` are not included in the result.

Invalid patterns return a shaped `@error` value with category `:regex`.

6.9 Files

The `file` module provides handle-based file input and output and basic file-system operations.

It is imported as:

```
let file = import("file")
```

File-system failures are generally returned as shaped `@error` values with category `:io`. Supplying arguments of the wrong type or arity faults.

6.9.1 Opening and closing files

```
open(path, mode)
```

`open` opens the file named by string `path` and returns a file handle.

`mode` is one of the symbols:

`:read` `:write` `:append`

`:read` opens an existing file for reading. `:write` opens a file for writing, creating it if necessary and replacing its previous contents. `:append` opens a file for writing at its end, creating it if necessary.

For example:

```
let f = file.open("data.txt", :read)
```

If the file cannot be opened, or if `mode` is not one of the permitted symbols, `open` returns an `@error` value with category `:io`.

```
close(handle)
```

`close` closes `handle` and returns `true`.

Closing an already closed handle or encountering an operating-system error while closing returns an `@error` value with category `:io`.

6.9.2 Text input and output

```
read_text(handle)
```

`read_text` reads all remaining contents of `handle` and returns them as a string.

At end of file it returns `""`.

```
read_line(handle)
```

`read_line` reads and returns the next line as a string. A terminating newline is not included in the result; a carriage return immediately preceding the newline is also removed.

For example, a file containing:

one

two

may be read as:

```
file.read_line(f)
    ⇒ "one"

file.read_line(f)
    ⇒ "two"
```

If end of file is reached before any further characters are read, `read_line` returns:

```
[@end]
```

If the final line is not terminated by a newline, that line is returned normally; the following call returns `[@end]`.

```
write_text(handle, s)
```

`write_text` writes string `s` to `handle` and returns `true` on success.

It does not add a newline or other characters.

Read and write failures return an `@error` value with category `:io`.

6.9.3 Byte input and output

```
read_bytes(handle)
```

`read_bytes` reads all remaining contents of `handle` and returns an opaque Aiki `:bytes` value containing those bytes.

At end of file it returns an empty bytes value.

```
write_bytes(handle, bytes)
```

`write_bytes` writes the contents of an opaque `:bytes` value to `handle` and returns `true` on success.

The byte values used by these procedures are the host-backed values described for `bytes/ffi` in section 6.4. They are distinct from the shaped-list representation supplied by `bytes/native`.

Read and write failures return an `@error` value with category `:io`.

6.9.4 File-system operations

```
exists(path)
```

`exists` returns `true` if a file-system object exists at string `path`, and `false` otherwise.

Errors encountered while determining existence, including access errors, also produce `false`.

```
delete(path)
```

`delete` removes the file-system object named by string `path` and returns `true` on success.

If the object cannot be removed, `delete` returns an `@error` value with category `:io`.

The `file` module exports:

`open` `read_text` `read_bytes` `read_line`
`write_text` `write_bytes` `close` `exists` `delete`

6.10 Time

The `time` module provides basic time-related operations.

It is imported as:

```
let time = import("time")
```

6.10.1 Sleeping

```
sleep(ms)
```

`sleep` suspends the current computation for approximately `ms` milliseconds and then returns `true`.

For example:

```
time.sleep(1000)
```

suspends evaluation for approximately one second.

The argument must be a number. Fractional values are accepted by the language interface, although the underlying sleep duration is represented at millisecond resolution.

6.10.2 Event channels

```
after(ms)
```

after returns a receive-only event channel that produces true once after approximately ms milliseconds. Unlike sleep, it does not block the calling computation.

For example:

```
let timer = time.after(100)
recv(timer)
⇒ true
```

The argument must be a non-negative number. Fractional milliseconds are accepted. Sending on the returned channel faults.

Ordinary channels created by channel() remain unbuffered and sendable. The timer's internal event storage is an implementation property and does not introduce general buffered channels.

after is intended to compose with select for timeouts and other time-based events.

6.11 Canvas graphics

The `canvas` module provides a small immediate-mode graphics interface based on canvas values.

It is imported as:

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

Coordinates and dimensions are expressed in pixels. The origin is at the upper-left of the canvas, with increasing `x` to the right and increasing `y` downward.

6.11.1 Creating and inspecting canvases

```
canvas(width, height)
```

`canvas` creates a canvas having the specified dimensions and returns a canvas value.

For example:

```
let c = canvas.canvas(800, 600)
```

A new canvas has a black background, a white foreground, and a pen size of `2`.

```
width(c)
height(c)
```

`width` and `height` return the dimensions of canvas `c` as numbers.

```
canvas.width(c)
    ⇒ 800
canvas.height(c)
    ⇒ 600
```

If a canvas cannot be created, `canvas` returns a shaped `@error` value with category `:canvas`.

6.11.2 Drawing primitives

```
dot(c, x, y)
dot(c, x, y, color)
```

`dot` draws a dot at `(x, y)`.

```
line(c, x1, y1, x2, y2)
line(c, x1, y1, x2, y2, color)
```

`line` draws a line from `(x1, y1)` to `(x2, y2)`.

```
rect(c, x, y, width, height)
rect(c, x, y, width, height, color)
```

`rect` draws the outline of a rectangle whose upper-left corner is `(x, y)`.

```
fill_rect(c, x, y, width, height)
fill_rect(c, x, y, width, height, color)
```

`fill_rect` draws a filled rectangle.

```
circle(c, x, y, radius)
circle(c, x, y, radius, color)
```

`circle` draws the outline of a circle centered at `(x, y)`.

```
fill_circle(c, x, y, radius)
fill_circle(c, x, y, radius, color)
```

`fill_circle` draws a filled circle.

```
arc(c, x, y, radius, start, end)
arc(c, x, y, radius, start, end, color)
```

`arc` draws a circular arc centered at `(x, y)` from angle `start` to angle `end`, measured in degrees. An angle of `0` points to the right and an angle of `90` points downward.

If the optional `color` argument is omitted, the current foreground color is used.

The outline primitives use the current pen size. Drawing operations return `true`.

6.11.3 Color and pen state

```
set_bg(c, color)
set_fg(c, color)
```

`set_bg` sets the background color of `c`; `set_fg` sets the foreground color used by subsequent drawing operations.

A color may be specified by one of the symbols:

`:black` `:white` `:red` `:green` `:blue` `:yellow` `:cyan` `:magenta` `:orange` `:purple` `:pink` `:gray` `:grey`

A color may also be specified as a list containing red, green, and blue components, with an optional alpha component:

```
[r, g, b]
[r, g, b, a]
```

```
set_bg_rgb(c, r, g, b)
set_fg_rgb(c, r, g, b)
```

These procedures set the background or foreground color from separate red, green, and blue components.

For example:

```
canvas.set_fg(c, :cyan)
canvas.set_bg_rgb(c, 30, 30, 30)
```

```
pen_size(c, size)
```

`pen_size` sets the thickness used by subsequent outline drawing operations. `size` must be a number whose integer value is at least `1`. Fractional values are truncated to an integer number of pixels.

6.11.4 Clearing and destroying canvases

```
clear(c)
```

`clear` clears the entire canvas using its current background color and returns `true`.

```
destroy(c)
```

`destroy` closes the canvas and releases its associated resources. It returns `true`.

Most operations on a canvas that has been closed return a shaped `@error` value with category `:canvas`.

The `canvas` module exports:

`canvas` `width` `height` `dot` `line` `rect` `fill_rect` `circle` `fill_circle` `arc` `clear` `destroy` `set_bg` `set_fg` `set_bg_rgb` `set_fg_rgb` `pen_size`

6.12 Turtle graphics

The `turtle` module provides Logo-style turtle graphics over a canvas.

It is imported as:

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

Turtle coordinates are logical coordinates centered on the canvas. Position `[0, 0]` is the center, positive `x` extends to the right, and positive `y` extends upward. Heading `0` is north, with headings increasing clockwise.

6.12.1 Turtle values

```
turtle(canvas)
```

`turtle` creates a turtle associated with `canvas`.

A new turtle begins at logical position `[0, 0]`, facing north with heading `0`, with its pen down and its drawing color set to `:white`.

The returned value is a shaped `@turtle` list containing the turtle's current state.

For example:

```
let cv = import("canvas")
let c = cv.canvas(400, 400)
let t = turtle.turtle(c)
```

Operations that change turtle state return a new turtle value. Programs therefore normally rebind the result:

```
t = turtle.forward(t, 100)
```

6.12.2 Movement and heading

```
forward(t, distance)
backward(t, distance)
```

`forward` moves `t` by `distance` units in its current heading. `backward` moves it the same way in the opposite direction.

If the pen is down, movement draws a line on the associated canvas. If the pen is up, only the turtle position changes.

Both procedures return a new turtle at the resulting position.

```
left(t, angle)
right(t, angle)
```

`left` decreases the turtle's heading by `angle` degrees; `right` increases it by `angle` degrees.

For example:

```
t = turtle.right(t, 90)
```

```
turtle.heading(t)
    ⇒ 90
```

Heading `0` is north, `90` is east, `180` is south, and `270` is west.

```
heading(t)
```

`heading` returns the current heading in degrees.

6.12.3 Pen control

```
pen_up(t)
pen_down(t)
```

`pen_up` returns a turtle whose subsequent movement does not draw. `pen_down` returns a turtle whose subsequent movement does draw.

```
set_color(t, color)
```

`set_color` returns a turtle using `color` for subsequent drawing.

Colors are interpreted by the underlying canvas operations and may use the color representations described in section 6.11.

For example:

```
t = turtle.set_color(t, :red)
t = turtle.forward(t, 100)
```

6.12.4 Position and home

```
position(t)
```

`position` returns the current logical position of `t` as:

```
[x, y]
```

The coordinates are relative to the center of the associated canvas.

```
home(t)
```

`home` returns a turtle at position `[0, 0]` with heading `0`.

If the pen is down, returning home draws a line from the turtle's current position to the center. If the pen is up, no line is drawn.

The pen state and drawing color are preserved.

6.12.5 Clearing

```
clear(t)
```

`clear` clears the associated canvas using its current background color and returns `t`.

The turtle's position, heading, pen state, and drawing color are unchanged.

6.13 Simple turtle graphics

The `turtle/simple` module provides Logo-style turtle graphics using an implicit drawing environment and turtle state. Unlike the `turtle` module, its procedures do not require an explicit turtle value to be passed from one operation to the next.

It is imported as:

```
let turtle = import("turtle/simple")
```

The module maintains one current canvas and turtle.

6.13.1 Creating the drawing environment

```
new()
new(dim)
```

`new` initializes the drawing environment and places the turtle at its home position.

With no argument, the canvas is 800 by 800 pixels. If `dim` is supplied, a square canvas of `dim` by `dim` pixels is used.

For example:

```
turtle.new(600)
```

A newly initialized turtle is at logical position `[0, 0]`, has heading `0`, has its pen down, uses color `:white`, and is visible.

`new` establishes a fresh drawing session using the requested canvas dimensions and resets the turtle to its initial state.

6.13.2 Logical coordinates

```
logical(size)
```

`logical` sets the width and height of the logical coordinate space to `size`.

The default logical size is `200`, giving coordinate bounds from `-100` through `100` along each axis. The origin `[0, 0]` is at the center of the canvas, positive `x` extends to the right, and positive `y` extends upward.

For example:

```
turtle.new(800)
turtle.logical(200)
```

maps 200 logical units across an 800-pixel canvas.

Turtle movement is clamped to the logical bounds. A movement that would pass beyond an edge stops at that edge.

Changing the logical size changes the mapping between logical units and pixels but does not change the turtle's logical position.

6.13.3 Movement and heading

```
forward(distance)
backward(distance)
```

`forward` moves the current turtle by `distance` logical units in its current heading. `backward` moves in the opposite direction.

If the pen is down, movement draws a line. If the pen is up, only the position changes.

```
left(angle)
right(angle)
```

`left` decreases the current heading by `angle` degrees; `right` increases it.

Heading `0` points upward, and `90` points to the right.

```
heading()
```

`heading` returns the current heading in degrees.

6.13.4 Pen and background

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

`pen_up` disables drawing during subsequent movement. `pen_down` enables it.

```
pen_size(size)
```

`pen_size` sets the drawing width in logical units. The value is scaled to the current canvas; the resulting pixel width is never less than one pixel.

```
pencolor(color)
```

`pencolor` sets the color used for subsequent turtle drawing.

```
background(color)
```

`background` sets the canvas background color.

Colors use the representations described in section 6.11.

6.13.5 Turtle visibility

show_turtle()

hide_turtle()

`show_turtle` displays the turtle indicator on the canvas. `hide_turtle` removes the indicator from view.

Visibility affects only the displayed turtle indicator; it does not affect movement or drawing.

6.13.6 Position, home, and clearing

```
position()
```

`position` returns the current logical position as a shaped point:

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

The reported coordinates are truncated to integers.

For example:

```
turtle.position()
    ⇒ [@point, 0, 0]
home()
```

`home` moves the turtle to `[0, 0]` and resets its heading to `0`.

Returning home does not draw a line, regardless of the current pen state. The current pen state and drawing color are preserved.

```
clear()
```

`clear` clears the canvas using its current background color. The turtle's position, heading, pen state, color, and visibility are unchanged.

The `turtle/simple` module exports:

`new` `logical` `forward` `backward` `left` `right` `pen_up` `pen_down` `pen_size` `pencolor` `background` `show_turtle` `hide_turtle` `home` `clear` `position` `heading`

6.14 Bit operations

The bits module provides bit operations over non-negative integral Aiki numbers. It does not introduce a fixed-width integer type; widths are explicit where an operation requires them.

It is imported as:

```
let bits = import("bits")
```

6.14.1 Logical operations

```
bit_and(a, b)
```

bit_and returns the bitwise AND of a and b.

```
bit_or(a, b)
```

bit_or returns the bitwise inclusive OR of a and b.

```
bit_xor(a, b)
```

bit_xor returns the bitwise exclusive OR of a and b.

```
bit_not(a, width)
```

bit_not complements the low width bits of a. Bits above width are ignored; the explicit width supplies the word boundary that Aiki numbers otherwise do not have.

6.14.2 Shifts and masks

```
shl(a, count)
```

shl shifts a left by count bits.

```
shr(a, count)
```

shr logically shifts a right by count bits. Negative inputs are not admitted by the module.

```
mask(width)
```

mask returns a number whose low width bits are 1.

```
low(a, width)
```

low returns the low width bits of a.

6.14.3 Bit fields

```
extract(a, start, width)
```

extract returns width bits beginning at zero-based bit position start.

```
replace(a, start, width, value)
```

replace returns a with width bits beginning at start replaced by the low width bits of value.

The module exports bit_and, bit_or, bit_xor, bit_not, shl, shr, mask, low, extract, and replace.

6.15 Stores

The store module provides explicit mutable indexed storage for systems work. A store is a distinct system value rather than a list; ordinary Aiki lists remain immutable.

It is imported as:

```
let store = import("store")
```

6.15.1 Construction

```
new(size)
```

new creates a store containing size cells, each initialized to 0. size must be a non-negative integer.

6.15.2 Access and update

```
get(store, index)
```

get returns the value in the zero-based cell index. An invalid index faults.

```
set(store, index, value)
```

set replaces the value in the zero-based cell index, changes the store in place, and returns []. A store may contain ordinary Aiki values.

```
length(store)
```

length returns the number of cells in store.

The module exports new, get, set, and length.

6.16 Testing

The test module supplies assertions and named test cases for Aiki-native tests. It is used by the reference implementation's aiki test command and is also an ordinary importable module.

It may be brought into the current environment with:

```
use("test")
equal(actual, expected)
```

equal asserts deep equality.

```
not_equal(actual, expected)
```

not_equal asserts that the values are not equal.

```
is_true(value)
```

is_true asserts that value is true according to Aiki truth semantics.

```
is_false(value)
```

is_false asserts that value is false according to Aiki truth semantics.

```
error(value)
```

error asserts that value is a shaped @error value.

```
not_error(value)
```

not_error asserts that value is not a shaped @error value.

```
run(name, function)
```

run executes a named test case. A fault in the test function is recorded as a failure without preventing later named cases from running.

```
faults(function)
```

faults asserts that calling function produces a fault.

The module exports equal, not_equal, is_true, is_false, error, not_error, run, and faults.

7. Formal syntax and semantics

This chapter gives formal descriptions of the syntax and semantics described informally in the preceding chapters.

Section 7.1 gives the formal syntax of Aiki. The grammar is expressed in the extended BNF notation used by the language grammar. Section 7.2 gives a formal account of evaluation corresponding to the semantic operations of the language.

7.1 Formal syntax

The grammar uses the following notation:

```
a b         sequence
a | b       alternative
[ a ]       optional
{ a }       zero or more occurrences
"token"     literal token
NAME        lexical token class
```

Spaces in the grammar are for legibility and do not themselves denote whitespace.

The grammar is stated over lexical tokens. Spaces, tabs, carriage returns, and comments do not participate in the syntactic grammar. Line endings participate only in statement separation as described below. The grammar is expressed in an extended Backus–Naur form (EBNF) notation used by the language grammar.

7.1.1 Lexical structure

The lexical token classes of Aiki are:

```
NAME        = /[a-zA-Z_][a-zA-Z0-9_]*/

NUMBER      = /[0-9]+(\.[0-9]+)?(\/[0-9]+)?/

STRING      = /"([^"\\]|\\.)*"/

RUNE        = /'([^'\\]|\\.)'/

SYMBOL      = /:[a-zA-Z_][a-zA-Z0-9_]*/

SHAPE       = /@[a-zA-Z_][a-zA-Z0-9_]*/
```

The reserved words are:

let if else while match select default return true false and or not package

The symbolic operators are:

`+` `-` `*` `/` `<` `>` `<=` `>=` `|>` `.` `=`

The delimiters are:

`(` `)` `[` `]` `{` `}` `,` `;` `...`

The binary-operator class used by the expression grammar is:

```
BINOP = "+" | "-" | "*" | "/"
      | "<" | ">" | "<=" | ">="
      | "and" | "or"
```

Whitespace between tokens consists of spaces, tabs, carriage returns, and line endings. A comment begins with `#` and extends up to, but does not include, the following line ending.

Spaces, tabs, carriage returns, and comments are discarded before syntactic analysis. Outside parentheses and brackets, a line ending is treated as a statement separator when the preceding token is `NAME`, `NUMBER`, `STRING`, `RUNE`, `SYMBOL`, `)`, `]`, `true`, or `false`. Otherwise the line ending is discarded. Line endings within parentheses or brackets are discarded. Braces do not suppress statement separation.

An explicit semicolon may also separate statements. In the grammar of sections 7.1.2 through 7.1.5, statement sequences therefore admit an optional statement separator after each statement.

Identifiers are case-sensitive. Reserved words are not identifiers. Symbols and shape names form lexical classes distinct from identifiers.

7.1.2 Programs and statements

An Aiki program is a sequence of statements.

```
program = { statement [ ";" ] }
```

A statement is one of the following forms:

```
statement = package_stmt
          | let_stmt
          | assign_stmt
          | if_stmt
          | while_stmt
          | match_stmt
| select_stmt
          | return_stmt
          | expr_stmt
```

A package declaration has the form:

```
package_stmt = "package" STRING
```

A binding statement either binds a name to the value of an expression or defines a shape:

```
let_stmt = "let" NAME "=" expr
         | "let" SHAPE "[" [ field { "," field } ] "]"

field = NAME | SHAPE
```

The second form permits fields to be ordinary field names or embedded shape names.

Assignment has the form:

```
assign_stmt = NAME "=" expr
```

Conditional execution has the form:

```
if_stmt = "if" expr block
          [ "else" ( if_stmt | block ) ]
```

Thus an `else` may contain either a block or another `if` statement.

Iteration has the form:

```
while_stmt = "while" expr block
```

Pattern matching has the form:

```
match_stmt = "match" expr
             "{" { pattern block } "}"
Concurrent selection has the form:
select_stmt = "select" "{"
( select_case { select_case }
[ select_default ] | select_default )
"}"
```

where:

```
select_case = [ "let" NAME "=" ] "recv" "("
expr ")" block
select_default = "default" block
```

A receive case may bind the received value to a name. At least one receive case or a default case is required.

The syntax of patterns is given in section 7.1.5.

A return statement has the form:

```
return_stmt = "return" expr
```

An expression may appear as a statement:

```
expr_stmt = expr
```

A block is a sequence of statements enclosed in braces:

```
block = "{" { statement [ ";" ] } "}"
```

As described in section 7.1.1, qualifying line endings may separate statements. An explicit semicolon may be used instead.

The syntax above specifies statement formation only. The evaluation of statements, including the environments in which their blocks are evaluated and the propagation of `return`, is given in section 7.2.

7.1.3 Expressions

An Aiki expression is a pipeline expression:

```
expr = pipe_expr
```

Pipeline expressions consist of an infix expression followed by zero or more pipeline stages:

```
pipe_expr = infix_expr { "|>"
postfix_expr }
```

Binary expressions consist of a unary expression followed by zero or more binary-operator and unary-expression pairs:

```
infix_expr = unary_expr { BINOP
unary_expr }
```

The binary operators are:

```
BINOP = "+" | "-" | "*" | "/"
      | "<" | ">" | "<=" | ">="
      | "and" | "or"
```

Because all binary operators occur in the same production, the grammar defines no precedence hierarchy among them. Their evaluation is described in section 7.2.3.

Unary expressions consist of zero or more unary operators followed by a postfix expression:

```
unary_expr = { "not" | "-" } postfix_expr
```

A postfix expression consists of a primary expression followed by zero or more calls, indexing operations, or field accesses:

```
postfix_expr = primary { call | index |
access }
```

The postfix forms are:

```
call   = "(" [ expr { "," expr } ] ")"

index  = "[" expr "]"

access = "." NAME
```

A primary expression is a literal, a name, a list literal, a function literal, or an explicitly grouped expression:

```
primary = NUMBER | STRING | RUNE | SYMBOL |
NAME
        | "true" | "false"
        | list_literal
        | func_literal
        | "(" expr ")"
```

The forms for list literals and function literals are given in sections 7.1.5 and 7.1.4 respectively.

The grammar permits postfix operations to be chained. Thus calls, indexing, and field access may occur successively after the same primary expression. Parenthesized expressions re-enter the complete expression grammar and therefore provide explicit grouping.

The complete grammar is given in section 7.1.6.

7.1.4 Functions and parameters

A function literal consists of a parameter specification followed by a block:

```
func_literal = "(" params ")" block
```

The parameter specification is optional and may contain ordinary parameters, a rest parameter, or both:

```
params = [ param_list [ "," rest_param ]
         | rest_param ]
```

An ordinary parameter list consists of one or more names separated by commas:

```
param_list = NAME { "," NAME }
```

A rest parameter consists of three successive periods followed by a name:

```
rest_param = "..." NAME
```

Thus the following forms are syntactically valid:

```
() { ... }

(x) { ... }

(x, y) { ... }

(...args) { ... }

(x, y, ...rest) { ... }
```

A rest parameter, when present, must be the final parameter.

Function literals are primary expressions and may therefore occur wherever a primary expression is permitted. Their evaluation and the binding of ordinary and rest parameters are described in section 7.2.4.

The syntax of blocks is given in section 7.1.2. The complete grammar is given in section 7.1.6.

7.1.5 Lists, shapes, and patterns

A list literal is either an ordinary list or a shaped list:

```
list_literal = "[" [ expr { "," expr } ]
"]"
             | "[" SHAPE { "," expr } "]"
```

An ordinary list contains zero or more expressions separated by commas:

```
[]

[1, 2, 3]
```

A shaped list begins with a shape name followed by zero or more expressions:

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

The shape name is syntactically distinct from the expressions that follow it.

Shape definitions are given by the `let` statement described in section 7.1.2. A field in a shape definition may be either a name or an embedded shape name:

```
field = NAME | SHAPE
```

Patterns have the following syntax:

```
pattern = "_"
        | NAME
        | literal
        | "[" [ pattern { "," pattern } ]
"]"
        | "[" SHAPE { "," pattern } "]"
```

The wildcard `_` and ordinary names form atomic patterns. Literal patterns use the following literal class:

```
literal = NUMBER | STRING | RUNE | SYMBOL
        | "true" | "false"
```

List patterns contain zero or more nested patterns:

```
[]

[a, b]

[1, _, value]
```

A shaped-list pattern begins with a shape name followed by zero or more nested patterns:

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

Patterns may therefore be nested recursively.

The formation of patterns is specified here; their matching behavior and the bindings established by successful matches are described in section 7.2.7.

The complete grammar is given in section 7.1.6.

7.1.6 Complete grammar

The following is the complete Aiki grammar used by the language engine. It is not a separately maintained descriptive grammar. The implementation reads this EBNFX grammar at runtime and uses it to recognize source text and construct the parse structure presented to the evaluator. The grammar therefore directly determines the syntactic forms on which evaluation operates.

The grammar is presented here in the implementation's extended BNF form, including the lexical definitions and grammar metadata used for diagnostics, templates, and help.

The `@tokens` section defines lexical token classes. Tokens marked `@skip` are recognized by the lexer but discarded before syntactic analysis. The productions that follow define programs, statements, expressions, functions, lists, shapes, and patterns. The treatment of line endings as statement separators is described in section 7.1.1.

```
@tokens {
    WHITESPACE  /[ \t\r]+/  @skip
    NEWLINE     /\n+/
    COMMENT     /#[^\n]*/  @skip

    NUMBER      /[0-9]+(\.[0-9]+)?(\/[0-9]+)?/
        @error "expected number"
        @template "42, 3.14, 1/3"
        @help "integer, decimal, or rational literal"

    STRING      /"([^"\\]|\\.)*"/
        @error "expected string"
        @template "\"hello\""
        @help "double-quoted text"

    RUNE        /'([^'\\]|\\.)'/
        @error "expected rune"
        @template "'a', '\\n'"
        @help "single character in single quotes"

    SYMBOL      /:[a-zA-Z_][a-zA-Z0-9_]*/
        @error "expected symbol"
        @template ":name, :ok, :error"
        @help "colon-prefixed atomic identifier"

    SHAPE       /@[a-zA-Z_][a-zA-Z0-9_]*/
        @error "expected shape"
        @template "@point, @error"
        @help "at-prefixed shape name"

KEYWORD let if else while match select default return true false and or not package
    OPERATOR    + - * / < > <= >= |> . =
    DELIMITER   ( ) [ ] { } , ; ...

    NAME        /[a-zA-Z_][a-zA-Z0-9_]*/
        @error "expected name"
        @template "foo, my_var"
        @help "identifier in snake_case"
}

program = { statement [ ";" ] }
    @error "expected statement"
    @template "statement ..."
    @help "sequence of statements"

statement = package_stmt | let_stmt | assign_stmt | if_stmt | while_stmt
| match_stmt | select_stmt | return_stmt | expr_stmt
    @error "expected statement"
@template "package | let | assign | if | while | match | select | return | expr"
    @help "one of the statement forms"

package_stmt = "package" STRING
    @error "expected 'package \"name\"'"
    @template "package \"name\""
    @help "declares the package name for this module"

let_stmt = "let" NAME "=" expr
         | "let" SHAPE "[" [ field { "," field } ] "]"
```

```
    @error "expected 'let name = expr' or 'let @shape [fields]'"
    @template "let name = expr | let @shape [fields]"
    @help "binds a value or defines a shape"

field = NAME | SHAPE
    @error "expected field name or embedded shape"
    @template "name | @shape"
    @help "field in a shape definition"

assign_stmt = NAME "=" expr
    @error "expected 'name = expr'"
    @template "name = expr"
    @help "assigns to existing binding"

if_stmt = "if" expr block [ "else" ( if_stmt | block ) ]
    @error "expected 'if expr { ... }'"
    @template "if cond { ... } [else { ... }]"
    @help "conditional execution"

while_stmt = "while" expr block
    @error "expected 'while expr { ... }'"
    @template "while cond { ... }"
    @help "loop while condition is true"

match_stmt = "match" expr "{" { pattern block } "}"
    @error "expected 'match expr { pattern { ... } ... }'"
    @template "match expr { pattern { ... } }"
    @help "pattern matching on value"

select_stmt = "select" "{" ( select_case { select_case } [ select_default ] |
select_default ) "}"
@error "expected 'select { recv(ch) { ... } ... }'"
@template "select { [let name =] recv(ch) { ... } [default { ... }] }"
@help "waits until one receive case is ready; default runs only when none is ready"

select_case = [ "let" NAME "=" ] "recv" "(" expr ")" block
@error "expected '[let name =] recv(channel) { ... }'"
@template "let value = recv(ch) { ... } | recv(ch) { ... }"
@help "receive case in a select statement"

select_default = "default" block
@error "expected 'default { ... }'"
@template "default { ... }"
@help "nonblocking select case used only when no receive case is ready"

pattern = "_" | NAME | literal
        | "[" [ pattern { "," pattern } ] "]"
        | "[" SHAPE { "," pattern } "]"
    @error "expected pattern"
    @template "_, name, literal, [patterns], [@shape, patterns]"
    @help "pattern for matching"

return_stmt = "return" expr
    @error "expected 'return expr'"
    @template "return expr"
    @help "returns value from function"

expr_stmt = expr
    @error "expected expression"
    @template "expr"
    @help "expression as statement"

expr = pipe_expr
    @error "expected expression"
    @template "expr"
    @help "any expression"

pipe_expr = infix_expr { "|>" postfix_expr }
    @error "expected expression"
    @template "expr |> func |> func"
    @help "left-to-right pipeline"

infix_expr = unary_expr { BINOP unary_expr }
    @error "expected expression"
    @template "expr op expr op expr"
```

```
    @help "left-to-right binary operations"

unary_expr = { "not" | "-" } postfix_expr
    @error "expected expression"
    @template "not expr, -expr"
    @help "prefix unary operators"

postfix_expr = primary { call | index | access }
    @error "expected expression"
    @template "expr(), expr[i], expr.field"
    @help "call, index, or field access"

call = "(" [ expr { "," expr } ] ")"
    @error "expected '(args)'"
    @template "(arg, arg, ...)"
    @help "function call"

index = "[" expr "]"
    @error "expected '[index]'"
    @template "[i]"
    @help "list or string index"

access = "." NAME
    @error "expected '.field'"
    @template ".field"
    @help "shaped list field access"

primary = NUMBER | STRING | RUNE | SYMBOL | NAME
        | "true" | "false"
        | list_literal
        | func_literal
        | "(" expr ")"
    @error "expected value"
    @template "42, \"str\", 'r', :sym, name, true, [], (), (expr)"
    @help "literal, name, or grouped expression"

list_literal = "[" [ expr { "," expr } ] "]"
             | "[" SHAPE { "," expr } "]"
    @error "expected list '[expr, ...]' or '[@shape, expr, ...]'"
    @template "[a, b], [@point, x, y]"
    @help "raw or shaped list"

func_literal = "(" params ")" block
    @error "expected '(params) { body }'"
    @template "(a, b) { return a + b }"
    @help "anonymous function"

params = [ param_list [ "," rest_param ] | rest_param ]
    @error "expected parameters"
    @template "a, b, ...rest"
    @help "function parameters"

param_list = NAME { "," NAME }
    @error "expected parameter names"
    @template "a, b, c"
    @help "list of parameter names"

rest_param = "..." NAME
    @error "expected '...name'"
    @template "...args"
    @help "rest parameter"

block = "{" { statement [ ";" ] } "}"
    @error "expected '{ ... }'"
    @template "{ statement ... }"
    @help "block of statements"

literal = NUMBER | STRING | RUNE | SYMBOL | "true" | "false"
    @error "expected literal"
    @template "42, \"str\", 'r', :sym, true, false"
    @help "literal value"

BINOP = "+" | "-" | "*" | "/"
      | "<" | ">" | "<=" | ">="
      | "and" | "or"
```

```
    @error "expected operator"
    @template "+ - * / < > <= >= and or"
    @help "binary operator"
```

7.2 Formal semantics

This section gives a formal account of Aiki evaluation.

The semantics follow the structure of the language evaluator. Parsed forms are evaluated in an environment according to their syntactic kind, producing an Aiki value or transferring control through return or fault propagation. The evaluator provides a corresponding semantic operation for every syntactic production that participates in evaluation.

The presentation below abstracts from the organization of the reference implementation while preserving its language-visible behavior. It describes values, environments, expression and statement evaluation, function application, pattern matching, control propagation, and top-level execution.

Unless otherwise stated, subexpressions are evaluated from left to right. Evaluation that produces a fault does not continue through the enclosing computation. Shaped error values are ordinary Aiki values and are not faults.

The notation and semantic domains used in the following sections are introduced in section 7.2.1.

7.2.1 Semantic domains and notation

The following domains and notation are used in the semantic descriptions that follow.

Let

```
v ∈ V       Aiki values
n ∈ N       numbers
b ∈ B       booleans
r ∈ R       runes
s ∈ S       strings
y ∈ Y       symbols
l ∈ L       lists
f ∈ F       functions
m ∈ M       modules
c ∈ C       channels
ρ ∈ Env     environments
d ∈ Shape   shape definitions
```

Other standard-library values, including bytes, files, stores, and canvases, are members of V but require no special treatment in the core evaluation rules.

A list value consists of an ordered sequence of values together with an optional shape name:

```
l = ⟨q, v₁, ..., vₙ⟩
```

where `q` is either a shape name or absent. An unshaped list has no shape name. A shaped list and an unshaped list belong to the same value domain `L`.

A function value consists of its ordinary parameters, optional rest parameter, body, and retained lexical environment:

```
f = ⟨p₁, ..., pₙ, rest?, body, ρ⟩
```

A shape definition associates a shape name with an ordered sequence of fields.

An environment contains bindings from names to values and shape names to shape definitions, together with an optional enclosing environment. Environments form a lexical chain.

Evaluation is written

```
⟨e, ρ⟩ ⇓ v
```

to mean that expression `e`, evaluated in environment `ρ`, produces value `v`.

Statement evaluation is written

```
⟨s, ρ⟩ ⇓ o
```

where an outcome `o` is either an ordinary value, a returned value, or a fault:

```
o ∈ V ∪ return(V) ∪ Fault
```

`return(v)` represents the internal propagation of a value from a return statement to the function call that encloses it. A fault represents an evaluation failure that prevents the enclosing computation from continuing normally. Neither `return(v)` nor a fault is an ordinary Aiki value.

Environment extension is written

```
ρ[x ↦ v]
```

for an environment in which `x` is bound to `v` in a newly enclosed environment.

Updating an existing binding is written

```
ρ[x := v]
```

and denotes replacement of the nearest binding for `x` found by searching `ρ` and its enclosing environments.

Lookup is written

```
ρ(x)
```

and denotes the value of the nearest binding for `x` in the same search order.

The semantic rules in the following sections describe language-visible behavior. Implementation structures used to realize these domains are not part of the formal semantics except where their behavior is explicitly reflected by a rule.

7.2.2 Environments, bindings, and lookup

An environment contains bindings from names to values and shape names to shape definitions and may have an enclosing environment. Environments therefore form a lexical chain.

For a name `x`, lookup searches the current environment first and then successive enclosing environments:

```
lookup(ρ, x)
```

The first binding for `x` found in this search supplies its value. If no binding is found, lookup faults.

A `let` binding establishes or replaces a binding in the current environment:

```
bind(ρ, x, v)
```

does not search enclosing environments. If `x` is already bound in `ρ`, its value is replaced; otherwise a new binding is established in `ρ`.

Assignment searches the lexical chain for an existing binding:

```
assign(ρ, x, v)
```

If `x` is bound in the current environment, that binding is changed. Otherwise enclosing environments are searched in order, and the nearest existing binding is changed. If no binding exists, assignment faults.

Bindings supplied by the prelude are protected from direct assignment. If lookup of `x` reaches a prelude binding without encountering a user binding for `x`, assignment to `x` faults. A program may instead establish a binding of the same name with `let`; subsequent assignment then changes that user binding.

Thus a prelude name may be shadowed but not overwritten.

Function creation retains the environment in which the function is evaluated. Function application creates a new environment enclosed by that retained environment. Parameter bindings and bindings established by `let` in the function body belong to the new environment.

Blocks do not by themselves create environments. The bodies of `if` and `while` are evaluated in their surrounding environment. A successful `match` arm is evaluated in a newly enclosed environment containing the bindings established by its pattern.

Shape definitions follow the same lexical organization. Shape lookup searches the current environment and then successive enclosing environments for the nearest definition of the requested shape name.

These rules determine lexical visibility independently of the lifetime or representation of the implementation objects used to realize environments.

7.2.3 Expression evaluation

Expression evaluation is deterministic and proceeds from left to right except where explicit grouping establishes a nested expression.

Literal expressions evaluate to their corresponding values:

```
⟨literal, ρ⟩ ⇓ v
```

A name expression evaluates by lookup in the current environment as described in section 7.2.2:

```
⟨x, ρ⟩ ⇓ lookup(ρ, x)
```

If lookup fails, evaluation faults.

A parenthesized expression has the value of the expression it contains.

Unary expressions are evaluated by first evaluating their operand. Prefix operators are then applied from right to left. For `not`, the operand is interpreted according to Aiki truth semantics. For unary `-`, the operand must be a number; otherwise evaluation faults.

Binary expressions have the form

```
e₀ op₁ e₁ op₂ e₂ ... opₙ eₙ
```

and are evaluated as a left fold. First e_0 is evaluated. Each following expression is then evaluated from left to right and the corresponding operator is applied to the accumulated result:

```
v₀ = eval(e₀, ρ)
v₁ = op₁(v₀, eval(e₁, ρ))
v₂ = op₂(v₁, eval(e₂, ρ))
...
vₙ = opₙ(vₙ₋₁, eval(eₙ, ρ))
```

Thus:

```
2 + 3 * 4
```

is evaluated as:

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

There is no precedence distinction among binary operators.

The operators `and` and `or` follow the same left-to-right evaluation rule as the other binary operators. Both operands are evaluated. They return operand values rather than necessarily returning booleans. For values `a` and `b`:

```
a and b
```

returns `a` when `a` is false and otherwise returns `b`; and

```
a or b
```

returns `a` when `a` is true and otherwise returns `b`.

The operators `+`, `-`, `*`, and `/` operate on numbers, except that `+` also concatenates two strings. Division by zero faults. The comparison operators `<`, `>`, `<=`, and `>=` compare numbers and produce boolean values. Application of an operator to unsupported operand types faults.

A pipeline

```
e |> p₁ |> p₂ ... |> pₙ
```

first evaluates `e`. Its value becomes the first argument supplied to `p₁`. The result of each stage becomes the first argument supplied to the next stage. Any explicit arguments written at a pipeline stage follow the piped value.

Pipeline stages are evaluated from left to right. A fault or shaped error produced by the initial expression or by any stage stops evaluation of the remaining stages. A shaped error remains an ordinary Aiki value and becomes the value of the pipeline.

Postfix operations are also evaluated from left to right. The primary expression is evaluated first, after which calls, indexing operations, and field accesses are applied successively to the current result. Function application is specified in section 7.2.4. List and string indexing and field access are specified in section 7.2.6.

If evaluation of a subexpression produces a fault or propagating control outcome, the enclosing expression does not evaluate subsequent components and propagates that outcome.

7.2.4 Function creation and application

Evaluating a function literal creates a function value containing its parameter specification, body, and the environment in which the literal is evaluated:

```
⟨(p₁, ..., pₙ) { body }, ρ⟩
⇓ ⟨p₁, ..., pₙ, body, ρ⟩
```

The retained environment gives Aiki functions lexical scope. Bindings visible where the function is created remain available when the function is later applied.

To apply a function, the function expression is evaluated first. Its argument expressions are then evaluated from left to right. If evaluation of the function expression or any argument faults, application does not proceed.

For a user function

```
f = ⟨p₁, ..., pₙ, rest?, body, ρf⟩
```

applied to argument values

```
a₁, ..., aₘ
```

a new call environment `ρc` is created whose enclosing environment is `ρf`.

The ordinary parameters are bound in `ρc`:

```
p₁ ↦ a₁
...
pₙ ↦ aₙ
```

A call must supply at least as many arguments as there are ordinary parameters. If fewer than `n` arguments are supplied, application faults.

If the function has a rest parameter, all arguments following the ordinary parameters are collected into a list and bound to that parameter:

```
rest ↦ [aₙ₊₁, ..., aₘ]
```

If there are no remaining arguments, the rest parameter is bound to `[]`.

If the function has no rest parameter, arguments beyond those required by its ordinary parameters are evaluated but are otherwise ignored.

The function body is evaluated in `ρc`. An explicit

```
return e
```

evaluates `e` and transfers its value from the function. Otherwise the value produced by the function body becomes the value of the application. An empty body produces `[]`.

A function value may be applied wherever a value in function position is permitted. Attempting to apply a value that is not callable faults.

Calls occurring in tail position are evaluated as proper tail calls: they do not require an additional active function frame. This optimization does not alter the value produced by the call or the lexical environment used for evaluation.

Isolated application. A function applied by spawn is evaluated in an isolated environment. In this case `ρc` encloses the prelude environment rather than ρf, so no binding of `ρf` is visible to the application. The parameter and rest-parameter rules above apply unchanged.

`ρc` additionally contains the shape definitions visible in `ρf`, with nearer definitions taking precedence. Shape definitions are ordered sequences of field names rather than values, and are supplied so that field access on shaped arguments follows the rules of section 7.2.6. A shape defined within the body is established in `ρc` and shadows a definition of the same name supplied in this way.

A fault produced by an isolated application does not propagate to the application of spawn, which has already produced its value.

7.2.5 Statement evaluation

Statements are evaluated in source order. Unless a statement explicitly establishes a new environment, it is evaluated in the environment in which it occurs.

An expression statement evaluates its expression and produces the resulting value:

$$\frac{\langle e, \rho\rangle \Downarrow v}{\langle e;, \rho\rangle \Downarrow v}$$

A binding statement

```
let x = e
```

first evaluates `e` in the current environment. If evaluation succeeds, `x` is bound to the resulting value in that environment:

$$\frac{\langle e, \rho\rangle \Downarrow v}{\langle \texttt{let x = e}, \rho\rangle \Downarrow []}$$

The binding established by `let` does not search or alter enclosing environments. A binding statement itself produces `[]`.

A shape definition

```
let @q [f₁, ..., fₙ]
```

associates the shape name `q` with its ordered fields in the current environment and produces `[]`.

Assignment

```
x = e
```

requires an existing assignable binding for `x`. The expression `e` is evaluated first; its value then replaces the nearest eligible binding found through the lexical environment chain as described in section 7.2.2. Assignment produces `[]`.

A conditional

```
if e {
    s₁
} else {
    s₂
}
```

first evaluates `e`. If its value is true according to Aiki truth semantics, the first block is evaluated. Otherwise the `else` branch, when present, is evaluated. The selected block is evaluated in the surrounding environment and its result is the result of the conditional. If no branch is selected, the result is `[]`.

An `else if` is evaluated as a nested conditional.

A `while` statement repeatedly evaluates its condition in the surrounding environment. Before each iteration, the condition is interpreted according to Aiki truth semantics. If it is true, the body is evaluated in the same environment and the condition is evaluated again.

The value of a `while` statement is the value produced by the final execution of its body. If the body is never executed, the value is `[]`. A return or fault produced while evaluating the condition or body terminates the loop and is propagated.

A `match` statement first evaluates its subject expression once. Its patterns are then considered in source order. The first successful pattern selects its associated block. Pattern matching is described in section 7.2.7.

A successful match creates a new environment enclosed by the surrounding environment. Bindings established by the pattern are placed in this new environment, and the selected block is evaluated there. Its result is the result of the `match` statement. If no pattern matches, the result is `[]`.

A select statement first evaluates the channel expression of each receive case exactly once. Each resulting value must be a channel. A fault produced while preparing a case is propagated and no arm is selected.

If one or more receive cases can proceed, one ready case is selected. The language does not specify which ready case is chosen when several can proceed. If none can proceed and a default case is present, the default case is selected; otherwise evaluation waits until a receive case can proceed.

The selected arm is evaluated in a new environment enclosed by the surrounding environment. For a case of the form let x = recv(e) { ... }, x is bound to the received value in that environment. The result of the selected block is the result of the select statement.

A return statement

```
return e
```

evaluates `e` and produces the control outcome `return(v)`, where `v` is the resulting value. This outcome propagates through enclosing blocks and control statements until it reaches the current function application. Return propagation is described further in section 7.2.8.

A block evaluates its statements in source order in the environment supplied to the block:

```
{s₁; s₂; ...; sₙ}
```

Its value is the value of `sₙ`. An empty block produces `[]`. A return or fault stops evaluation of the remaining statements and is propagated.

A shaped error is an ordinary value and does not by itself stop ordinary statement sequencing. Pipeline evaluation gives shaped errors the additional propagation behavior described in section 7.2.3.

A package declaration records the package name associated with the current module and produces `[]`.

7.2.6 Lists, shapes, indexing, and field access

A list literal evaluates its element expressions from left to right and produces a list containing the resulting values.

For an ordinary list

```
[e1, ..., en]
```

evaluation produces

```
[v1, ..., vn]
```

where each `vi` is the value of `ei`.

A shaped list

```
[@q, e1, ..., en]
```

is evaluated in the same manner but carries the shape name `q` in addition to its elements. Shaped and unshaped lists are members of the same list value domain.

The presence of a shape name does not by itself validate the number or interpretation of the elements. A shape definition supplies the field interpretation used by field access.

A shape definition

```
let @q [f1, ..., fn]
```

associates `q` with an ordered sequence of fields. Shape definitions are looked up through the lexical environment chain as described in section 7.2.2.

Indexing has the form

```
e0[e1]
```

The indexed expression `e0` is evaluated first and the index expression `e1` second.

Lists and strings may be indexed. The index must evaluate to an integer-valued number greater than or equal to zero and less than the length of the indexed value. Otherwise evaluation faults.

For a list,

```
[v0, v1, ..., vn][i]
```

produces the element at zero-based position `i`.

For a string, indexing is by rune rather than by byte. The result of string indexing is a rune value.

Field access has the form

```
e.name
```

The expression `e` is evaluated first.

If its value is a shaped list with shape `q`, the evaluator looks up the definition of `q` and finds the position of `name` in its ordered fields. If `name` is field `fi`, the corresponding list element is returned.

Thus, for

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

the expression

```
p.x
```

produces `10`.

Field access faults if the value is not a shaped list, if its shape is unknown, if the requested field does not occur in the shape definition, or if the shaped list does not contain an element at the field's position.

The same access syntax is used for modules. If `e` evaluates to a module, then

```
e.name
```

produces the value exported by that module under `name`. Access to a name not exported by the module faults.

Indexing and field access are postfix operations and may be chained with calls and with one another as described in section 7.2.3.

7.2.7 Pattern matching

A `match` statement evaluates its subject expression once. The resulting value is compared with each pattern in source order. The first successful pattern selects its associated block.

Pattern matching is recursive and may establish bindings. A match attempt either succeeds with a set of bindings or fails.

The wildcard pattern

```
_
```

matches every value and establishes no binding.

A name pattern

```
x
```

matches every value and binds `x` to that value.

Literal patterns compare their literal value with the matched value. Number, string, rune, symbol, and boolean patterns match when the values are equal and of the same type.

An ordinary list pattern

```
[p1, ..., pn]
```

matches only a list containing exactly `n` elements. Each pattern `pi` is matched recursively against the corresponding element. The match succeeds only if every element pattern succeeds.

Thus:

```
[a, b]
```

matches a two-element list but does not match a one-element or three-element list.

A shaped-list pattern

```
[@q, p₁, ..., pₙ]
```

matches only a shaped list whose shape name is `q` and which contains exactly `n` elements. The element patterns are then matched recursively in order.

For example:

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

matches

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

and establishes

```
x ↦ 10
y ↦ 20
```

but does not match an unshaped list or a list carrying a different shape name.

Nested list and shaped-list patterns are matched recursively:

```
[@wrap, [a, b]]
```

matches a shaped list whose sole element is a two-element list.

If the same name occurs more than once in a pattern, each occurrence is a binding occurrence. Matching proceeds from left to right, and a later occurrence replaces the binding established by an earlier occurrence. Repetition of a name therefore does not impose an equality constraint.

Bindings accumulated while matching a successful pattern are installed in a new environment enclosed by the environment of the `match` statement. The selected block is evaluated in that environment. Pattern bindings therefore do not escape the selected arm.

If a pattern fails, its tentative bindings are discarded and the next pattern is tried. If no pattern matches, the `match` statement produces `[]`.

Pattern order is significant. Because name and wildcard patterns match any value, placing either before a more specific pattern may make later arms unreachable.

7.2.8 Return, faults, and control propagation

Aiki distinguishes ordinary values from two forms of control propagation: return and fault.

A return statement

```
return e
```

first evaluates `e`. If evaluation succeeds with value `v`, the statement produces the internal outcome

```
return(v)
```

rather than an ordinary value.

A return outcome immediately stops evaluation of the enclosing block. It propagates through enclosing control statements, including if, while, match, and select, without evaluating any remaining statements.

At a function-call boundary, the return outcome is consumed and its contained value becomes the value of the function application:

```
return(v)  ⇒  v
```

Thus `return` controls evaluation within a function but is not itself an Aiki value visible to the program.

A fault represents an unrecoverable failure of evaluation. Faults may result from such conditions as an undefined name, invalid operand types, division by zero, invalid indexing or field access, an invalid function application, or violation of another required evaluation condition.

If evaluation of a subexpression or statement produces a fault, evaluation of the enclosing construct stops immediately and the same fault propagates outward. No subsequent operand, statement, pipeline stage, loop iteration, or match arm is evaluated.

Fault propagation continues across function calls. A function does not convert a fault into an ordinary return value.

A shaped error

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

is distinct from a fault. It is an ordinary Aiki value representing a recoverable failure. Ordinary expression and statement evaluation therefore permits a shaped error to be stored, returned, passed to a function, inspected, or matched like other values.

Pipelines give shaped errors additional propagation behavior. If the initial value or any stage of a pipeline produces a shaped error, the remaining stages are not evaluated and the shaped error becomes the value of the pipeline. This prevents a recoverable error from being passed silently through subsequent pipeline stages.

Return and fault propagation take precedence over ordinary value production. A construct that receives either control outcome does not treat it as its own result; it propagates the outcome until the corresponding boundary is reached.

7.2.9 Programs and top-level evaluation

An Aiki program is evaluated as a sequence of statements in a top-level environment.

Let

```
P = s₁; s₂; ...; sₙ
```

be a program evaluated in environment ρ_0. The statements are evaluated from left to right in ρ_0.

The value of the program is the value produced by the final statement evaluated successfully. An empty program produces `[]`.

If evaluation of a statement produces a fault, evaluation of the remaining statements stops and the fault propagates to the top-level boundary.

If a return outcome reaches the top-level program boundary, the boundary consumes the return outcome and its contained value becomes the result of the program:

```
return(v)  ⇒  v
```

Thus a top-level `return` terminates evaluation of the remaining program and yields its value.

Bindings established by top-level `let` statements are placed in the top-level environment and remain available to subsequent statements evaluated in that environment. Assignments and shape definitions at top level follow the environment rules of section 7.2.2.

A package declaration associates a package name with the top-level environment in which the program is evaluated. It does not otherwise alter the order or rules of evaluation.

Before evaluation, source text is tokenized and parsed according to the grammar of section 7.1. The resulting parse tree is then evaluated according to the rules of this section. The evaluator provides semantic handling for every grammar production and every non-skipped token that can occur in the *parse* tree.

The complete process may therefore be summarized as:

```
source
    ↓
lexical analysis
    ↓
parsing
    ↓
parse tree
    ↓
evaluation in ρ₀
    ↓
value or fault
```

This relation between grammar and evaluator is the basis of Aiki execution: the grammar determines the structure presented for evaluation, and the evaluator determines the meaning of that structure.

Example

The following program computes two ideal ballistic trajectories and displays them. For an initial speed `v`, launch angle `θ`, and gravitational acceleration `g`, the position at time `t` is

```
x(t) = v cos(θ)t
y(t) = v sin(θ)t - gt²/2
```

The two trajectories are computed concurrently for complementary launch angles of `30` and `60` degrees, which produce equal ranges and unequal peak heights.

A sample records a trajectory, a time, and a position; a result records a trajectory's range, peak height, and flight time. The program imports the canvas, mathematics, list, and time modules and binds the constants of the computation.

```
let @sample [flight, time, x, y]
let @result [flight, range, peak, duration]

let cv = import("canvas")
let math = import("math/native")
let list = import("list")
let time = import("time")

let PI = 355/113
let SPEED = 50
let GRAVITY = 981/100
let DT = 1/10
let TERMS = 6
let ITERATIONS = 8
let CROSS = 50
let SCALE = 3
```

Aiki numbers are exact rationals, so `355/113`, `981/100`, and `1/10` denote exact values rather than approximations of a floating-point representation. Sine, cosine, and square root have no exact rational form in general, and the two supplied mathematics modules differ in how they approximate them: `math/ffi` computes them through host floating-point arithmetic, while `math/native` computes them with Aiki rational arithmetic, using a finite Taylor series or Newton's method and taking the number of terms or iterations as an argument. This program uses `math/native`, so approximation occurs at three places: the rational value `PI`, the series length `TERMS`, and the iteration count `ITERATIONS`.

`simulate` computes one trajectory, sending a sample at each interval of `dt` and a result at the end of the flight. Spawned computations are evaluated in isolated environments, so every value `simulate` requires, including the mathematics module and the channel, is passed to it as an argument.

```
let simulate = (out, math_module, flight, degrees, speed, gravity, dt, pi, terms) {
  let angle = degrees * pi / 180
  let vx = speed * math_module.cos(angle, terms)
  let vy = speed * math_module.sin(angle, terms)
  let duration = 2 * vy / gravity
  let range = vx * duration
  let peak = vy * vy / (2 * gravity)
  let t = 0
  while t < duration {
    let x = vx * t
    let y = (vy * t) - ((gravity * t * t) / 2)
    send(out, [@sample, flight, t, x, y])
    t = t + dt
  }
  send(out, [@sample, flight, duration, range, 0])
  send(out, [@result, flight, range, peak, duration])
  return true
}
```

The parentheses in the expression for `y` specify a grouping other than the left-to-right one.

A channel carries the values produced by the two computations, and a canvas is prepared for their display.

```
let out = channel()
let screen = cv.canvas(800, 500)
cv.set_bg(screen, :white)
```

```
cv.clear(screen)
cv.line(screen, 50, 450, 750, 450, :black)
cv.line(screen, 50, 450, 50, 50, :black)

spawn(simulate, out, math, :low, 30, SPEED, GRAVITY, DT, PI, TERMS)
spawn(simulate, out, math, :high, 60, SPEED, GRAVITY, DT, PI, TERMS)
```

The two computations proceed concurrently over one channel, so the order in which their messages arrive is not determined. Matching on shape and tag distinguishes the four kinds of message: samples of the 30-degree trajectory are drawn as circles and those of the 60-degree trajectory as squares, and each sample is retained in a list.

```
let low_result = []
let high_result = []
let low_samples = []
let high_samples = []
let remaining = 2

while remaining > 0 {
  let message = recv(out)
  match message {
    [@sample, :low, _, x, y] {
      low_samples = append(low_samples, message)
      cv.fill_circle(screen, 50 + (x * SCALE), 450 - (y * SCALE), 2, :black)
    }
    [@sample, :high, _, x, y] {
      high_samples = append(high_samples, message)
      cv.fill_rect(screen, 48 + (x * SCALE), 448 - (y * SCALE), 4, 4, :black)
    }
    [@result, :low, _, _, _] {
      low_result = message
      remaining = remaining - 1
    }
    [@result, :high, _, _, _] {
      high_result = message
      remaining = remaining - 1
    }
  }
}
```

`crossing_time` takes a result and a height and produces the time at which that trajectory first reaches the height, obtained by solving `y(t) = h` for the ascending crossing:

```
t = (v_y - √(v_y² - 2gh)) / g
```

The solution is real only when the height does not exceed the trajectory's peak; otherwise `crossing_time` produces a shaped error. `describe` takes either and produces a printable form.

```
let crossing_time = (result, height) {
  if height > result.peak {
    return [@error, :flight, "height not reached"]
  }
  let vy = GRAVITY * result.duration / 2
  let discriminant = (vy * vy) - (2 * GRAVITY * height)
  let root = math.sqrt(discriminant, ITERATIONS)
  return (vy - root) / GRAVITY
}

let low_crossing = crossing_time(low_result, CROSS)
let high_crossing = crossing_time(high_result, CROSS)

let describe = (t) {
  match t {
    [@error, _, message] {
      return message
    }
    _ {
      return to_decimal(t, 2) + " s"
    }
  }
}
```

`above` tests one sample against the height, and the pipelines count the samples of each trajectory that lie above it.

```
let above = (s) {
  return s.y > CROSS
}

let low_above = low_samples |> list.filter(above) |> length
let high_above = high_samples |> list.filter(above) |> length

println("30 deg range:    ", to_decimal(low_result.range, 2), " m")
println("30 deg peak:     ", to_decimal(low_result.peak, 2), " m")
println("30 deg flight:   ", to_decimal(low_result.duration, 2), " s")
println("60 deg range:    ", to_decimal(high_result.range, 2), " m")
println("60 deg peak:     ", to_decimal(high_result.peak, 2), " m")
println("60 deg flight:   ", to_decimal(high_result.duration, 2), " s")
println("30 deg reaches 50 m at: ", describe(low_crossing))
println("60 deg reaches 50 m at: ", describe(high_crossing))
println("30 deg samples above 50 m: ", to_str(low_above))
println("60 deg samples above 50 m: ", to_str(high_above))
println(
  "range difference: ",
  to_decimal(abs(low_result.range - high_result.range), 6),
  " m"
)

time.sleep(3000)
```

A run produces

```
30 deg range:    220.70 m
30 deg peak:     31.86 m
30 deg flight:   5.10 s
60 deg range:    220.70 m
60 deg peak:     95.57 m
60 deg flight:   8.83 s
30 deg reaches 50 m at: height not reached
60 deg reaches 50 m at: 1.37 s
30 deg samples above 50 m: 0
60 deg samples above 50 m: 61
range difference: 0.000036 m
```

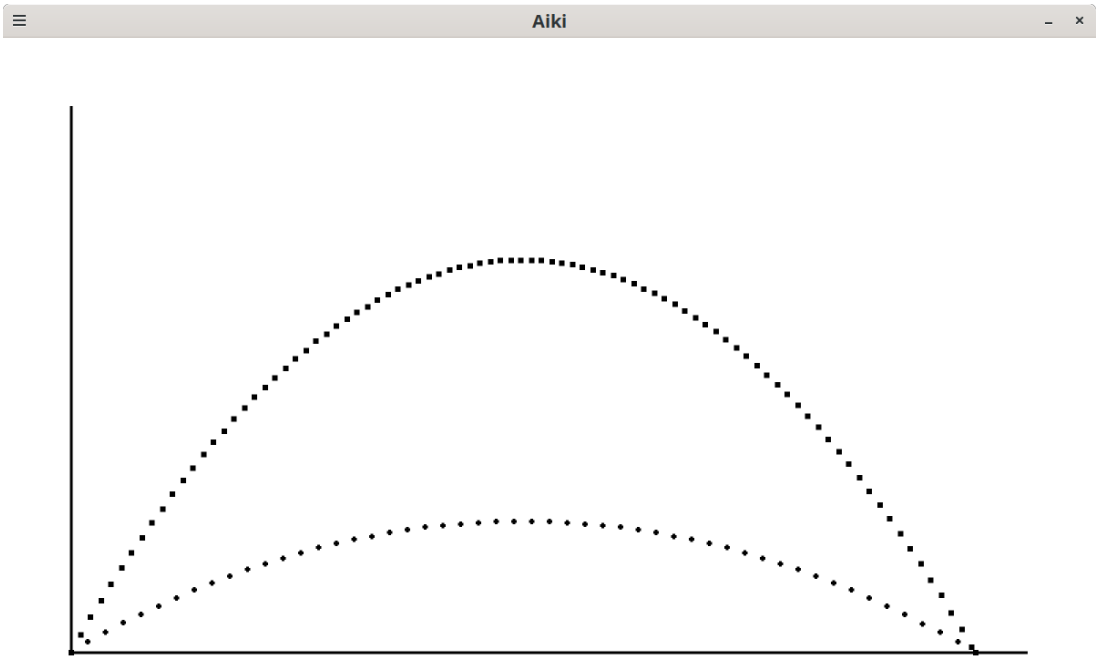

Figure 1. Concurrent ballistic trajectories computed in Aiki. Circular markers show the 30-degree trajectory and square markers show the 60-degree trajectory.

The range difference of `0.000036` meters follows from the approximations used in computing the trajectories: the rational value used for `π` and the finite trigonometric series. After those approximations have supplied the trigonometric values, the trajectory calculations proceed using Aiki rational arithmetic. The 60-degree trajectory reaches `50` meters and the 30-degree trajectory does not; accordingly, one call to `` `crossing_time` `` yields a time and the other yields a shaped error.

The example combines rational arithmetic with explicit approximation parameters, functions, modules, shapes and field access, pattern matching, recoverable shaped errors, pipelines, isolated concurrency, synchronous channels, lists, and canvas graphics in a single computation.

References

Clinger, W., & Rees, J. (Eds.). (1991). *Revised⁴ report on the algorithmic language Scheme.*

Senn, W. D. (2026a). The informing act: apprehension in AI-mediated systems. *Journal of Documentation*. https://doi.org/10.1108/JD-03-2026-0125.

Alphabetical Index

The principal defining entry for each concept, syntactic form, keyword, or procedure is listed first, followed by additional sections containing significant discussion.

A

abs 6.1.6
acos 6.6.2
time.after 6.10.2
list.all 6.2.2
and 4.7
list.any 6.2.2
append 6.1.3
argument evaluation 4.4, 7.2.4
arithmetic operators 4.7
canvas.arc 6.11.2
asin 6.6.2
assignment 3.1, 5.2, 7.2.2
atan 6.6.2

B

turtle.backward 6.12.2
turtle/simple.backward 6.13.3
turtle/simple.background 6.13.4
binary expressions 4.7, 7.2.3
bits 6.14
bits.bit_and 6.14.1
bits.bit_not 6.14.1
bits.bit_or 6.14.1
bits.bit_xor 6.14.1
bindings 3.1, 5.2, 7.2.2
blocks 7.1.2, 7.2.5
boolean 3.2, 3.4
braces, placement of 2.2
bytes 6.4
bytes/ffi 6.4.3
bytes/native 6.4.3
bytes_get 6.4.2
bytes_length 6.4.2
bytes_new 6.4.1
bytes_slice 6.4.2
bytes_to_str 6.4.1

C

canvas 6.11
canvas.canvas 6.11.1
string.capitalize 6.3.6
ceil 6.6.1
string.center 6.3.4
channel 6.1.7
channels 6.1.7
string.chars 6.3.1
chr 6.1.4
canvas.circle 6.11.2
canvas.clear 6.11.4
turtle.clear 6.12.5
turtle/simple.clear 6.13.6
file.close 6.9.1
closures 4.3, 7.2.4
comments 2.2
string.compare 6.3.2
comparison operators 4.7
list.concat 6.2.4
concurrency 6.1.7
string.contains 6.3.2
cos 6.6.2

D

hash.del 6.5.3
delete 6.1.8
default 5.8, 7.1.2, 7.2.5
file.delete 6.9.4
displayed representation 3.3
canvas.destroy 6.11.4
doc 6.1.8
canvas.dot 6.11.2
dynamic typing 1.1, 3.4

E

list.each 6.2.1
else 5.3
empty 6.1.3
end of input, @end 6.1.2, 6.9.2
string.ends_with 6.3.2
environments 3.1, 7.2.2
equal 6.1.4
test.equal 6.16
test.error 6.16
string.equal_ignore_case 6.3.6
evaluation order 1.1, 4.7, 7.2.3
file.exists 6.9.4
bits.extract 6.14.3
export 5.7
expressions 4, 7.1.3, 7.2.3
exact rational arithmetic 1.1, 3.4, 6.6.4

F
false 3.2, 4.2
test.faults 6.16
faults 1.3.2, 7.2.8
field access 4.10, 7.2.6
file 6.9
file handles 6.9.1
canvas.fill_circle 6.11.2
canvas.fill_rect 6.11.2
list.filter 6.2.1
list.find 6.2.2
regex.find 6.8.2
regex.find_all 6.8.2
first 6.1.3
first-class functions 1.1, 4.3
floor 6.6.1
turtle.forward 6.12.2
turtle/simple.forward 6.13.3
function application 4.4, 7.2.4
function literals 4.3, 7.1.4
functions 1.1, 3.4, 4.3
G
store.get 6.15.2
grammar 7.1, 7.1.6
grouping 4.11, 7.1.3
H
handles 6.9
hash/ffi 6.5.5
hash/native 6.5.5
hash.hash_code 6.5.1
hash tables 6.5
hash.has 6.5.2
turtle.heading 6.12.2
turtle/simple.heading 6.13.3
canvas.height 6.11.1
help 6.1.8
turtle/simple.hide_turtle 6.13.5
turtle.home 6.12.4
turtle/simple.home 6.13.6
I
identifiers 2.1, 7.1.1
if 5.3
import 5.7
indexing 4.9, 7.2.6
initial environment 3.1, 6.1
input 6.1.2
inspect 6.1.4
string.insert 6.3.3
interactive environment 5.1, 6.1.8
string.is_alnum 6.3.5
string.is_alpha 6.3.5
string.is_alphabetic 6.3.5
string.is_digit 6.3.5
is_error 6.1.4
test.is_false 6.16
test.is_true 6.16
string.is_lower 6.3.5
string.is_numeric 6.3.5
string.is_upper 6.3.5
string.is_whitespace 6.3.4
iteration 5.4
J
string.join 6.3.3
K
hash.keys 6.5.4
keywords 2.1, 7.1.1
L
string.last_index_of 6.3.2
turtle.left 6.12.2
turtle/simple.left 6.13.3
left-to-right evaluation 1.1, 4.7, 7.2.3
length 6.1.3
store.length 6.15.2
let 5.2
lexical scope 1.1, 3.1, 7.2.2
canvas.line 6.11.2
line continuation 2.2, 7.1.1
line endings 2.2, 7.1.1
string.lines 6.3.1
lists 3.5, 4.5, 7.2.6
list literals 4.5, 7.1.5
list module 6.2
literal expressions 4.2
literal representation 3.3
load 6.1.1
turtle/simple.logical 6.13.2
bits.low 6.14.2
string.lower 6.3.6
string.lower_rune 6.3.6
M
list.map 6.2.1
match 5.5

pattern matching 5.5, 7.2.7
math/ffi 6.6.4
math/native 6.6.4
bits.mask 6.14.2
mathematics 6.6
regex.matches 6.8.1
list.max 6.2.3
list.min 6.2.3
modulo 6.6.1
modules 5.7
mutation 3.5
N
name lookup 3.1, 7.2.2
name references 4.1
hash.new 6.5.2
store.new 6.15.1
turtle/simple.new 6.13.1
not 4.6
test.not_equal 6.16
test.not_error 6.16
number 3.4
numeric approximation 3.4, 6.6
O
string.occurrences 6.3.2
file.open 6.9.1
operator precedence 1.1, 4.7, 7.2.3
ord 6.1.4
or 4.7
P
package 5.7, 7.1.2
packages 5.7
string.pad_end 6.3.4
string.pad_start 6.3.4
parameters 4.3, 7.1.4
pattern bindings 5.5, 7.2.7
patterns 5.5, 7.1.5, 7.2.7
turtle.pen_down 6.12.3
turtle/simple.pen_down 6.13.4
canvas.pen_size 6.11.3
turtle/simple.pen_size 6.13.4
turtle.pen_up 6.12.3
turtle/simple.pen_up 6.13.4
turtle/simple.pencolor 6.13.4
pipeline, |> 4.8, 7.2.3
postfix composition 4.11, 7.1.3
turtle.position 6.12.4
turtle/simple.position 6.13.6
prelude 6.1
prepend 6.1.3
print 6.1.2
println 6.1.2
programs 5.1, 7.2.9
proper tail calls 1.1, 7.2.4
hash.put 6.5.3
Q
quit 6.1.8
R
random module 6.7
random.random 6.7.2
list.range 6.2.4
rational arithmetic 1.1, 3.4, 6.6.4
read 6.1.2
file.read_bytes 6.9.3
file.read_line 6.9.2
file.read_text 6.9.2
recv 6.1.7
recoverable errors 1.3.2, 7.2.8
canvas.rect 6.11.2
list.reduce 6.2.1
regex/ffi 6.8
regular expressions 6.8
string.remove 6.3.3
string.repeat 6.3.3
string.replace 6.3.3
regex.replace 6.8.3
string.replace_first 6.3.3
regex.replace_first 6.8.3
bits.replace 6.14.3
reset 6.1.8
rest 6.1.3
rest parameters, ... 4.3, 7.1.4
return 5.6, 7.2.8
list.reverse 6.2.4
string.reverse_string 6.3.3
turtle.right 6.12.2
turtle/simple.right 6.13.3
round 6.6.1
test.run 6.16
rune 3.4, 4.2
rune indexing 4.9
S
random.seed 6.7.1

select 5.8, 7.1.2, 7.2.5
semicolon 2.3, 5.1, 7.1.1
send 6.1.7
canvas.set_bg 6.11.3
canvas.set_bg_rgb 6.11.3
turtle.set_color 6.12.3
canvas.set_fg 6.11.3
canvas.set_fg_rgb 6.11.3
store.set 6.15.2
shape 3.5, 5.2, 7.2.6
shape 6.1.4
shape definitions 5.2, 7.2.6
shaped errors, @error 1.3.2, 4.8, 7.2.8
shaped lists 3.3, 4.5, 7.2.6
bits.shl 6.14.2
bits.shr 6.14.2
turtle/simple.show_turtle 6.13.5
sin 6.6.2
time.sleep 6.10.1
spawn 6.1.7
spawn isolation 6.1.7
string.split 6.3.1
regex.split 6.8.4
sqrt 6.6.3
stack_limit 6.1.1
standard library 6
statement separation 2.2, 7.1.1
string.starts_with 6.3.2
statements 5.1, 7.1.2, 7.2.5
store 3.5, 6.15
str_to_bytes 6.4.1
string 3.4, 6.3
string module 6.3
string.substring 6.3.1
list.sum 6.2.3
symbol 3.4, 4.2
synchronous channels 6.1.7
syntax 1.2, 7.1
system values 3.4
T
tail calls 1.1, 7.2.4
tan 6.6.2
test 6.16
time 6.10
string.title 6.3.6
to_decimal 6.1.5
to_number 6.1.5
to_str 6.1.5
top-level evaluation 7.2.9
string.trim 6.3.4
string.trim_end 6.3.4
string.trim_start 6.3.4
true 3.2, 4.2
truncate 6.1.6
string.truncate 6.3.4
turtle 6.12
turtle.turtle 6.12.1
turtle/simple 6.13
type 3.4, 6.1.4
types 3.4
U
unary expressions 4.6, 7.1.3
unbuffered channels 6.1.7
Unicode code points 6.1.4
string.upper 6.3.6
string.upper_rune 6.3.6
use 5.7
UTF-8 bytes 6.4.1
UTF-8 match offsets 6.8.2
V
values 3.3, 3.4, 7.2.1
hash.values 6.5.4
W
while 5.4
whitespace 2.2, 7.1.1
wildcard pattern, _ 5.5, 7.2.7
canvas.width 6.11.1
string.words 6.3.1
file.write_bytes 6.9.3
file.write_text 6.9.2