Aya

Release 0.3

nick-paul

Sep 07, 2023

CONTENTS

Aya is a terse stack based programming language originally intended for code golf and programming puzzles. The original design was heavily inspired by CJam and GolfScript. Currently, Aya is much more than a golfing language as it supports user-defined types, key-value pair dictionaries, natural variable scoping rules, and many other things which allow for more complex programs and data structures than other stack based languages.

Aya comes with a standard library written entirely in Aya code. The standard library features types such as matrices, sets, dates, colors and more. It also features hundreds of functions for working working on numerical computations, strings, plotting and file I/O. It even features a basic turtle library for creating drawings in the plot window.

Aya also features a minimal GUI that interfaces with Aya's stdin and stdout. The GUI features plotting, tab-completion for special characters, and an interactive way to search QuickSearch help data.

Check out the *Tour of Aya* section for further information.

Note: This project is under active development.

CHAPTER

ONE

CONTENTS

1.1 Tour of Aya

1.1.1 Basic language features

Aya is a stack based language.

```
aya> 1 1 +
2
aya> .# This is a line comment
aya> 1 2 + 10 * 3 / 10 -
0
```

Generally, most symbols that are not a lowercase letter are an operator (including uppercase letters). Extended operators come in the form `.*`, `M*`, `:*`, where `*` is *any* character. Aya has many cool operators. For example:

- Levenshtein distance (`^`)

```
aya> "kitten" "sitting" ^
3
```

- Create range (`R`) and reshape (`L`)

```
aya> 9 R
[ 1 2 3 4 5 6 7 8 9 ]
aya> 9 R [3 3] L
[ [ 1 2 3 ] [ 4 5 6 ] [ 7 8 9 ] ]
```

- List primes up to N (`Mp`)

```
aya> 30 Mp
[ 2 3 5 7 11 13 17 19 23 29 ]
```

- Split string using regex (`|`)

```
aya> "cat,dog, chicken ,pig" "\\W*,\\W*" |
[ "cat" "dog" "chicken" "pig" ]
```

- The *Apply* (`#`) operator is special in that it is parsed as an infix operator which can take another operator (or block) on its right (in this case length (`E`)) and apply to each item in the list

```
aya> 9 R [3 3] L #E
[3 3 3]
aya> 9 R [3 3] L #{E 1 +}
[4 4 4]
```

Many operators are broadcasted automatically. For example: the square root (`.^`), addition (+), multiplication (*), and factorial (`M!`) operators. Aya also supports complex numbers (`:-64i`), fractional numbers (`:1r2` is `1/2`), and extended precision numbers (`:100x`).

```
aya> [4 16 :-64i ] .^
[ 2 4 :0i8 ]
aya> [1 2 3] :1r2 +
[ :3r2 :5r2 :7r2 ]
aya> [1 2 3] [10 20 30] *
[ 10 40 90 ]
aya> [10 100 :100z] M!
[10 100 :100z] M!
[ 3628800 0␣
↪:9332621544394415268169923885626670049071596826438162146859296389521759999322991560894146397615651828
↪]
```

Lowercase letters and underscores are used for variables. The colon (`:`) operator is used for assignment. Like the apply operator (`#`), it is one of the few *"infix"* operators.

```
aya> .# Objects are left on the stack after assignment
aya> "Hello" :first
"Hello"
aya> .# The ; operator pops and disgards the top of the stack
aya> "world!" :second ;
```

As seen above, almost everything else, including all uppercase letters, is an operator. The `:P` operator will print the item on the top of the stack to stdout.

```
aya> first " " + second + :P
"Hello world!"
```

Aya has many types of objects. The `:T` operator is used to get the type. It returns a *Symbol* (`::symbol_name`)

```
aya> aya> 1 :T
::num
aya> [1 2 3] :T
::list
aya> [1 2 3] :T :T
::sym
aya> [ 1 [1 2 3] "hello" 'c {, 1:x } {2+} ::red ] #:T
[ ::num ::list ::str ::char ::dict ::block ::sym ]
```

You can create your own types

```
aya> struct point {x y}
<type 'point'>
aya> 1 2 point! :p
( 1 2 ) point!
```

(continues on next page)

(continued from previous page)

```
aya> p :T
::point
```

Aya supports string interpolation.

```
aya> "$first from Aya! 1 + 1 is $(1 1 +)"
"Hello from Aya. 1 + 1 is 2"
```

Blocks (`{...}`) are first class objects. They can be evaluated with the eval (~) operator.

```
aya> 1 {1 +}
1 {1 +}
aya> 1 {1 +} ~
2
```

When a block is assigned to a variable, it will be automatically evaluated when the variable is de-referenced. This allows the creation of functions.

```
aya> {2*}:double
{2 *}
aya> 4 double
8
```

Blocks may have arguments and local variables. In the example below, `a`, `b`, and `c` are arguments and `x` and `y` are local variables.

```
aya> {a b c : x y,
         a 2 * :x; .# local
         b 3 * :y; .# local

         [a b c x y] .# return a list with vars inside
     }:myfun;
```

The following will call `myfun` and assign 1 to `a`, 2 to `b`, and 3 to `c` within the scope of the function.

```
aya> 1 2 3 myfun
[2 2 3 2 6 8]
aya> .# a b c x y are no longer in scope
aya> a
ERROR: Variable a not found
aya> x
ERROR: Variable a not found
```

Block headers may include type assertions and local variable initializers. By default all local variables are initialized to `0` (see `y` in the example below).

```
aya> {a::num b::str : x(10) y z("hello"),
         [a b x y z]
     }:myfun;
aya> 1 "cats" myfun
[1 "cats" 10 0 "hello"]
aya> "dogs" "cats" myfun
TYPE ERROR: {ARGS}
```

(continues on next page)

(continued from previous page)

```
    Expected: ::num
    Received: "dogs"
```

Aya also supports dictionaries. `{,}` creates an empty dictionary. `.` is used for dictionary access and `.:` is used for assignment.

```
aya> {,} :d
{,
}
aya> 3 d.:x
{,
  3:x;
}
aya> d.x
3
aya> .# Keys can also be assigned in the literal itself
aya> {, 3:x; }
{,
  3:x;
}
```

Aya also supports operator overloading for many operators. Type `\? overloadable` in the Aya interpreter to get a list of all overloadable operators.

```
aya> struct point {x y}
aya> def point::__add__ {other self,
         other.x self.x +
         other.y self.y +
         self!
     }
aya> 3 4 point! 5 6 point! +
( 8 10 ) point!
```

Aya has a growing standard library including:

- 2d Matrix Object
- Dataframes
- JSON, CSV reading/writing
- Image reading/writing
- Sockets
- 2d graphics
- Plotting
- Math & statistics
- And more (see the standard library section)

The Aya core language supports many other cool things such as **closures**, built-in **fraction** and **arbitrary precision** numbers, **macro-like functions** *(the ``struct`` keyword above is defined completely in aya!)*, **exception handling**, built in **plotting** and **GUI dialogs**, **list comprehension**, and **more**!

1.2 Running / Installation

Aya is written in java. Please ensure you have the latest version of java on your system.

Download the latest release from the releases page.

Once downloaded, simply double click `aya.jar` to run the aya GUI.

If your system does not support double clicking a jar to run it, you may optionally run it using the following command:

```
java -jar aya.jar
```

NOTE: `aya.jar` must be in the same directory as the rest of the files included in the download.

1.2.1 Running Examples

There are many example scripts in the `examples/` directory. To run an example, type its name followed by the `example` command:

```
aya> "nth_fib" example
The first 10 fib numbers are [ 1 1 2 3 5 8 13 21 34 55 ]
```

Some examples are in subfolders such as `canvas`, `turtle`, or `plot`. Run them using `subfolder/example_name`:

```
aya> "canvas/mandelbrot" example
```

1.2.2 Command Line Arguments

All arguments are optional.

- The first argument is the directory to point aya at when running.
- If the second argument is `-i`, run the aya repl directly in the terminal.
- All the following arguments are scripts to run

```
$ ls my_scripts/aya_scripts/
hello.aya
$ rlwrap java -jar ~/git/aya/aya.jar my_scripts/aya_scripts/ -i
aya> "hello.aya" G~
Hello world!
```

1.2.3 System Install

This step is only needed if you would like to add aya to your systems path. *Currently only supports OSX/Linux*

Aya supports running scripts directly from the command line. For example:

```
$ cat hello.aya
#!/usr/bin/env aya
"Hello world!" :P

$ chmod +x hello.aya
```

(continues on next page)

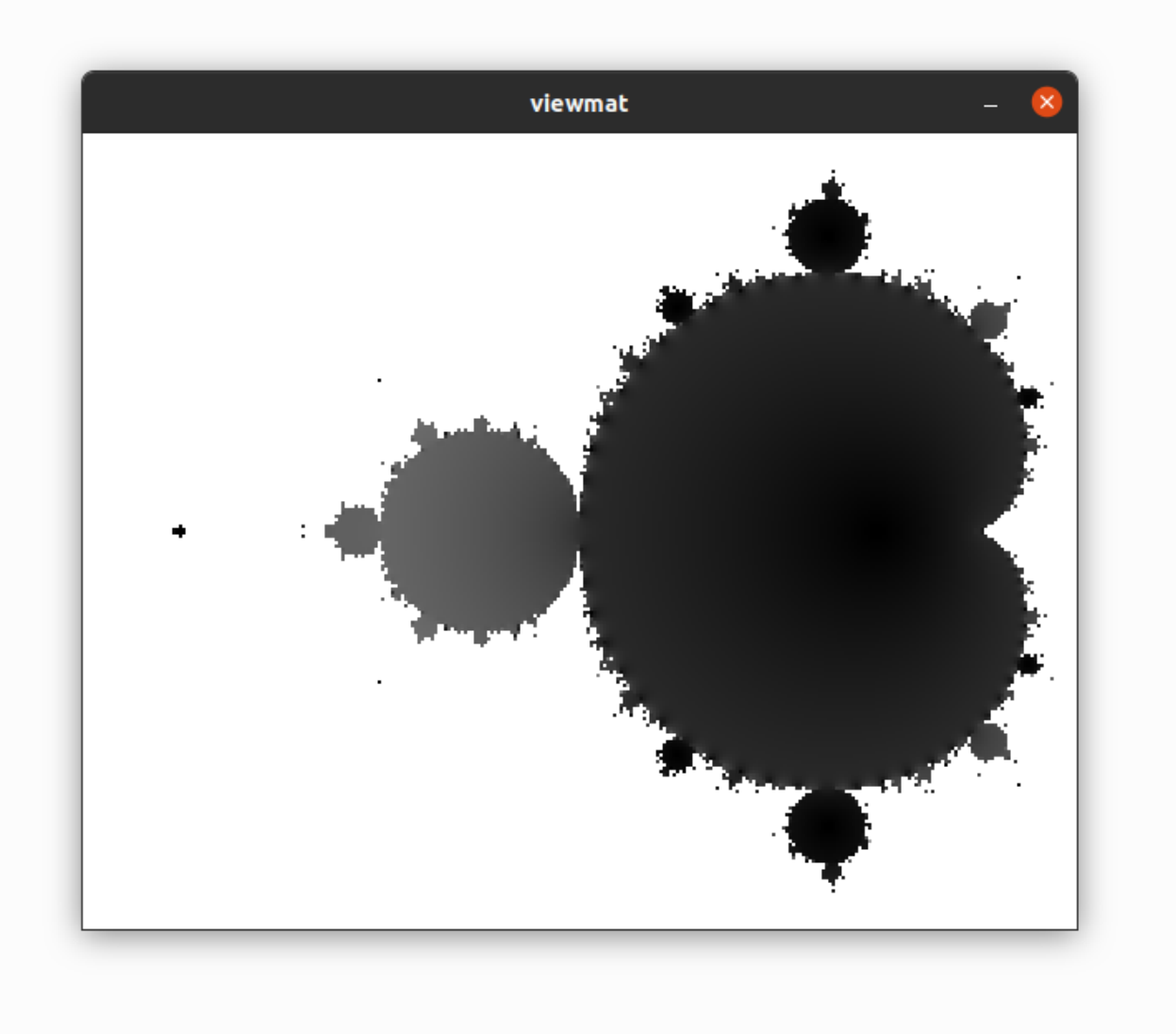

Fig. 1: img/mandelbrot_example.png

(continued from previous page)

```
$ ./hello.aya
Hello world!
```

To enable this, add `aya/runnable/linux` to your path:

For example, you can add this line to your bashrc:

```
PATH="$PATH:/<path_to_aya>/aya/runnable/linux"
```

1.3 Syntax Overview

1.3.1 Execution

Aya is a stack based language. Execution flows from left to right

```
aya> 1 2 +
3
aya> 1 2 + 4 *
12
aya> 1 2 + 4 * 3 /
4
```

1.3.2 Comments

Line Comments

Line comments begin with `.#`

```
aya> aya> .# comment
aya> 1 .# comment
1
aya> .#leading space optional
```

Block Comments

Block comments start with `.{` and end with `.}`

```
.{ This is a
    block comment!  .}
```

```
.{
    Also a block comment
.}
```

```
aya> .{ block .{ comments cannot be .} nested .}
SYNTAX ERROR: .} is not a valid operator
```

1.3.3 Variables

Use `:varname` to assign a variable. Use the plain variable name to access

```
aya> 1 :x
1
aya> x
1
```

Single characters are supported

```
aya> 1 :
1
aya>
1
```

Any string of lowercase letters and underscores can be used as a variable.

```
aya> 1 :this_is_a_valid_variable
1
```

Any string of characters can be used as a variable if the literal is quoted. They cannot be accessed directly. These types of variables are mostly useful for *dictionaries*.

```
aya> 1 :"Quoted Variable!"
1
aya> "Quoted Variable!" :S~
1
```

Numbers and uppercase letters cannot be used for variables

```
aya> 5 :MyVar0
Unexpected empty stack while executing instruction: :M
   in :M .. y V ar 0}
```

Special Variables

Double leading and trailing underscores are used for special variables

See operator overloading and metatables for examples

1.3.4 Numbers

*Main Page:*Numbers

Integers & Decimals

```
aya> 1
1
aya> 1.5
1.5
aya> .5
.5
```

Negative Numbers

`-` is parsed as an operator unless immediately followed by a number

```
aya> 1 2 - 3
-1 3
aya> 1 2 -3
1 2 -3
```

`:` can also be used to specify a negative number

```
aya> 1 2 :3
1 2 -3
```

Big Numbers

Arbitrary precision numbers have the form `:Nz`

```
aya> :12345678901234567890123456789z
:12345678901234567890123456789z
aya> :3.14159265358979323846264338327950288419716939z
:3.14159265358979323846264338327950288419716939z
```

Hexadecimal Literals

Hexadecimal literals have the form `:0xN`

```
aya> :0xfad
4013
```

If the hexadecimal does not fit in a standard integer, it will automatically be promoted to a *big number*.

```
aya> :0xdeadbeef
:3735928559z
```

Binary Literals

Binary literals have the form `:0bN`

```
aya> :0b1011
11
```

If the literal does not fit in a standard integer, it will automatically be promoted to a *big number*.

```
aya> :0b10111010101010010101010010101010010001011
:801704815243z
```

Scientific/"e" Notation

Number literals of the form `:NeM` are evaluated to the literal number `N * 10^M`.

```
aya> :4e3
4000
aya> :2.45e12
2450000000000
aya> :1.1e-3
.0011
```

Fractional Numbers

Fractional literals have the form `:NrM`

```
aya> :1r2
:1r2
aya> :3r
:3r1
aya> :-1r4
:-1r4
```

PI Times

Number literals of the form `:NpM` are evaluated to the literal number `(N * PI)^M`. If no `M` is provided, use the value 1.

```
aya> :1p2
9.8696044
aya> :1p
3.14159265
aya> :3p2
88.82643961
```

Root Constants

Number literals of the form `:NqM` are evaluated to the literal number `N^(1/M)`. The default value of M is 2.

```
aya> :2q
1.41421356
aya> :9q
3
aya> :27q3
3
```

Complex numbers

Complex numbers are built in. `:NiM` creates the complex number `N + Mi`. Most mathematical operations are supported

```
aya> :-1i0
:-1i0
aya> :-1i0 .^
:0i1
aya> :3i4 Ms
:3.85373804i-27.01681326
aya> :3i4 Mi .# imag part
4
aya> :3i4 Md .# real part
3
```

Number Constants

constants follow the format `:Nc`

number	value
`:0c`	pi
`:1c`	e
`:2c`	double max
`:3c`	double min
`:4c`	nan
`:5c`	inf
`:6c`	-inf
`:7c`	int max
`:8c`	int min
`:9c`	char max

1.3.5 Characters

*Main Page:*Characters & Strings

Standard Characters

Characters are written with a single *single quote* to the left of the character:

```
aya> 'a
'a
aya> '   .# space character
'
aya> ''  .# single quote character
''
aya> 'ÿ  .# supports unicode
'ÿ
```

Hex Character Literals

Hex literal characters are written using a `'\x___'` and **require closing quotes**.

```
aya> '\xff'
'ÿ
aya> '\x00a1'
'¡
```

Named Character Literals

Many characters have names. All names consist only of lowercase alphabetical characters. Use `Mk` operator to add new named characters.

```
'\n'         .# => <newline>
'\t'         .# => <tab>
'\alpha'     .# => ''
'\pi'        .# => ''
```

1.3.6 Strings

*Main Page:*Characters & Strings

Standard String Literals

String literals are written with double quotes (").

```
aya> "Hello, world!"
"Hello, world!"
```

Use \\ to escape to double quotes. (string printing in the REPL will still display the escape character)

```
aya> "escape: \" cool"
"escape: \" cool"
aya> "escape: \" cool" println
escape: " cool
```

Strings may span multiple lines.

```
"I am a string containing a newline character
    and a tab."
```

Special Characters in Strings

Strings can contain special characters using \{___}. Brackets can contain named characters or Unicode literals.

```
"sin(\{theta}) = \{alpha}"    .# => "sin() = "
"\{x00BF}Que tal?"            .# => "¿Que tal?"
```

String Interpolation

Use $ for string interpolation

```
aya> 10 :a;
aya> "a is $a"
"a is 10"
```

Use $(...) for expressions

```
aya> "a plus two is $(a 2 +)"
"a plus two is 12"
```

Use \ to keep the $ char

```
aya> 10:dollars;
aya> "I have \$$dollars."
"I have $10"
```

If used with anything else, keep the $

```
aya> "Each apple is worth $0.50"
"Each apple is worth $0.50"
```

Long String Literals

Use triple quotes for long string literals.

```
"""This is
a long string
literal"""
```

No escape characters or string interpolation is processed

```
aya> """This is a long string literal $foo \{theta}"""
"This is a long string literal $foo \{theta}"
```

1.3.7 Symbols

Symbols are primarily used for metaprogramming. Symbols are any valid variable name starting with `::`

```
aya> ::my_symbol
::my_symbol
```

Symbols can be any string if single quotes are used immediately after the `::`

```
aya> ::"My Symbol"
::"My Symbol"
```

1.3.8 Lists

*Main Page:*Lists

List Literals

Lists are written with square brackets (`[]`) and must not contain commas. They may contain any data type:

```
aya> [1 2 3]
[ 1 2 3 ]
aya> []
[ ]
aya> [1 2 "Hello" [3 4]]
[ 1 2 "Hello" [ 3 4 ] ]
```

Lists may also contain expressions:

```
aya> [1 2 + 3 4 +]
[ 3 7 ]
```

List Stack Captures

Use `[N| ... ]` to capture items off the stack into the list

```
aya> 9 [1| 8 7 6]
[ 9 8 7 6 ]
aya> 10 9 [2| 8 7 6]
[ 10 9 8 7 6 ]
aya> 10 9 [2|]
[ 10 9 ]
```

List Comprehensions

*See*list comprehensions

Indexing

Get a value from a list

Use `.[ (index) ]` to get a value from a list

```
aya> [1 2 3 4] :list
[ 1 2 3 4 ]
aya> list.[0]
1
aya> list.[:-1]
4
```

Set a value at an index in a list

Use `(value) (list) .[ (index) ]` to set a the value in a list at an index

```
aya> [1 2 3 4] :list
[ 1 2 3 4 ]
aya> 10 list.:[0]
[ 10 2 3 4 ]
```

1.3.9 Dictionaries

*Main Page:*Dictionaries and User Types

Dictionary Literals

Dictionary literals have the form `{, ... }`. All variables assigned between `{,` and `}` are assigned to the dictionary

```
aya> {, 1:a 2:b }
{,
  2:b;
  1:a;
}
```

`{,}` creates an empty dict

```
aya> {,}
{,}
```

Getting Values

Use dot notation to get values from a dict:

```
aya> {, 1:a 2:b } :d
{,
  2:b;
  1:a;
}
aya> d.a
1
aya> d .b
2
```

Or use strings or symbols with index notation (`.[]`)

```
aya> d.["a"]
1
aya> d.[::a]
1
```

Or use `:I` operator

```
aya> d ::a I
1
aya> d "a" I
1
```

Dot notation can be used with *quoted variables*

```
aya> {, 1:"Hello, world!" } :d
{,
  1:"Hello, world!";
}
aya> d."Hello, world!"
1
```

Setting Values

Use `.:` notation to set values of a dict

```
aya> {,} :d
{,}
aya> 10 d.:a
{,
  10:a;
}
```

Or using strings or symbols with index notation (`.:[]`)

```
aya> 11 d.:["b"]
{,
  11:b;
  10:a;
}
aya> 12 d.:[::c]
{,
  11:b;
  10:a;
  12:c;
}
```

This notation can be used with *quoted variables*

```
aya> {,}:d
{,}
aya> 10 d.:"Hello, world!"
{,
  10:"Hello, world!";
}
```

1.3.10 Blocks

*Main Page:*Blocks & Functions

Basic Blocks

Use `{...}` to define a code block.

```
aya> {2 +}
{2 +}
```

If a code block is assigned to a variable, execute it immediately when the variable is accessed

```
aya> {2 +}:add_two
{2 +}
aya> 4 add_two
6
```

Short Block Notation

Any set of tokens following a tick (```) until an operator or variable will be parsed as a block. Useful for saving a character when golfing

```
aya> `+
{+}
aya> `1 + 1
{1 +} 1
aya> `"hello" 1 'd +
{"hello" 1 'd +}
```

This notation also terminates at variables names

```
aya> `x 1
{x} 1
aya> `1 x 1
{1 x} 1
```

Block Headers

Use a comma in a block to create a block *header*. Block headers define local variables and block arguments

See Variables and Scope and Blocks and Functions for more details.

If the header is empty, the block is parsed as a dict (see *Dictionary*)

```
aya> {, 1:a }
{,
  1:a;
}
```

Arguments

Add arguments to a block

```
aya> {a b c, a b + c -}:foo
{a b c, a b + c -}
aya> 1 2 3 foo
0
```

Arguments can have type assertions. The block will fail if the type does not match

```
aya> {a::num b::str, "a is $a, b is $b"}:foo
{a::num b::str, "a is $a, b is $b"}
aya> 1 "two" foo
"a is 1, b is two"
aya> "one" 2 foo
        {ARGS}
        Expected:::str
        Received:2
   in a::num b::str, .. "a is $a, b is $b"}
```

(continues on next page)

(continued from previous page)

```
Function call traceback:
  Error in: foo
```

Local Variables

To declare local variables for a block, use a `:` in the header: `{:  ...  ,}`

```
aya> {: local_a local_b, 10:local_a 12:local_b 14:nonlocal_c} ~
10 12 14
aya> local_a
Undefined variable 'local_a'
   in local_a .. }
aya> nonlocal_c
14
```

Use parenthesis after the local variable to set the initial value

```
aya> {: local_a(99) , local_a} ~
99
```

Use `^` after a local variable to "capture" it from the surrounding scope

```
aya> 1:a
1
aya> {: a^, }
{: a(1),}
```

Can mix & match locals and arguments

```
aya> 9 :captured_local
9
aya> { arg typed_arg::str : default_locl initialized_local(10) captured_local^, }
{arg typed_arg::str : default_locl(0)initialized_local(10)captured_local(9),}
```

1.3.11 Operators

*Main Page:*Operators

Standard Operators

All single uppercase letters except `M` are operators

```
aya> 6 R
[ 1 2 3 4 5 6 ]
aya> 4 [5] J
[ 4 5 ]
```

"Dot" Operators

Most characters immediately following a dot (`.`) are an operator

```
aya> 6 .R
[ 0 1 2 3 4 5 ]
aya> 6 .!
1
```

Exceptions

Special Case	Description
`.<grave>`	Deference Without Execution
`.#`	*Line Comment*
`.{`	*Block Comment*
`.'`	*Symbol*

Dereference Without Executing (`.<grave>`)

`.<grave>` Dereference a variable without executing the block

```
aya> {1 2 +}:f
{1 2 +}
aya> f
3
aya> f.`
{1 2 +}
```

If the variable is not a block dereference it normally

```
aya> 1:a
1
aya> a.`
1
```

"Colon" Operators

Most characters immediately following a color (`:`) are an operator

```
aya> [1 2] [2] :|
[ 1 ]
```

Exceptions

Special Case	Description
:"	*Symbol*
:{	*Extension Operator*

"Misc" Operators

`M` plus any character is an operator

```
aya> "Hash" M#
635696504
aya> 0.5 Ms
.47942554
```

Non-Standard "Infix" Stack Operators

List Map (:#)

The `:#` operator takes a block on its *right* and maps it to the list on the stack

```
aya> [1 2 3] :# {1 +}
[ 2 3 4 ]
```

List Map Shorthand (#)

*See*Broadcast Operator

Same as `:#` but automatically creates a block using *short block notation*

```
aya> [1 2 3] # 1 +
[2 3 4]
```

Capture Instructions (:`)

Takes a block `B` and a number `N` from the stack. Captures `N` instructions from the instruction stack.

```
aya> {P} 2 :` 1 +
"[ {1} {+} ]"
```

Extension Operators

Extension operators have the form `:{...}`.

```
aya> 123456789 "dd/MM/yyyy HH:mm:ss" :{date.format}
"02/01/1970 05:17:36"
```

These operators are always wrapped in the standard library. They should almost never be used for normal development

```
aya> import ::date
aya> 123456789 date!
Jan 02, 1970 5:17:36 AM
```

1.3.12 User Types

Struct

Defining A Struct

Create a struct with the following syntax:

```
struct <typename> {<member> <vars> ...}
```

For example:

```
aya> struct point {x y}
aya> point
(struct ::point [ ::x ::y ])
```

Create Instance Of Struct

To create an instance of a struct, use the `!` operator on the type. Member variables should exist on the stack

```
aya> struct point {x y}
aya> 1 2 point!
( 1 2 ) point!
```

Accessing Values of a Struct

Use standard dot notation to acces user type values

```
aya> struct point {x y}
aya> 1 2 point! :p
( 1 2 ) point!
aya> p.x
1
aya> p.y
2
```

Struct Member Functions

Use the `def` keyword to define member functions for structs

```
aya> def point::format {self, "<$(self.x), $(self.y)>"}
aya> 1 2 point! :p
( 1 2 ) point!
aya> p.format
"<1, 2>"
```

1.3.13 Golf Utilities

Golf Constants

Any single-character key stored in `__cdict__` can be accessed using `¢` + that character

```
aya> {, "Hello!":"!" 10:a }:__cdict__
{,
  "Hello!":"!";
}
aya> ¢!
"Hello!"
aya> ¢a
10
```

`golf` standard library defines many useful variables in `__cdict__`

```
aya> import ::golf
aya> ¢Q
[ "QWERTYUIOP" "ASDFGHJKL" "ZXCVBNM" ]
aya> ¢½
[ 1 2 ]
```

1.4 Operators

1.4.1 Type Abbreviations

Type	Abbreviation
Number	N
String	S
Char	C
Block	B
Dict	D
Symbol	J

1.4.2 Operator Table

Name	Args	Ops
!	N|C	N : 1-N (logical not, complementary probability), C : swap case
#	LA..#A	LA..#A : map
$	A	A : deepcopy (duplicate)
%	LB|LS|LC|BN	LB : fold, LS : join, LC : join, BN : repeat
&	NN|SS	NN : bitwise and, SS : list all expressions matching the regex
*	NN	NN : multiply
+	NN|CC|SA|AS	NN : add, CC : add, SA : append string, AS : append string
-	NN|CC	NN : subtract, CC : subtract
/	NN	NN : divide
;	A	A : pop and discard
<	NN|SS|CC	NN : less than, SS : less than, CC : less than
=	AA	AA : equality
>	NN|SS|CC	NN : greater than, SS : greater than, CC : greater than
?	AA	AA : if A1, then A2. If A2 is block, execute it
@	AAA	AAA : rotates the top three elements on the stack [abc->bca]
A	A	A : wrap in list
B	J|L|N|C	J : increment in place, L : uncons from front, N : increment, C : increment
C	L|S|N	L : sort least to greatest, S : sort least to greatest, N : bitwise not
D	ALN	ALN : set index
E	L|N|S	L : length, N : 10^N, S : length
G	S|N	S : read a string from a filename or URL, N : isprime
H	LA|DS|DC|DJ|SA	LA : has; 1 if list contains object, DS : has; 1 if dict contains key, DC : has; 1 if dict contains key, D
I	LB|LL|LN	LB : filter, LL : get index, LN : get index
J	LA|AA|LL|AL	LA : add to list, AA : create list [A A], LL : join lists, AL : add to list
L	LL|AN|NL	LL : reshape, AN : create list by repeating A N times, NL : reshape
N	LA|DS|DJ|SS	LA : return index of first occurance, -1 if not found; keep list on stack, DS : contains key; keep dict
O	LB|DB	LB : Map block to list, DB : Map block to dict
P	A	A : to string
Q	L|N	L : random choice, N : N>0: random number 0-N, N<0: random number N-0, N=0: any int
R	L|N|C	L : len L = 2: range [N1, N1+1, …, N2], len l = 3: range [N1, N2, …, N3], N : range [1, 2 .. N], C
S	SC|LN|SS	SC : split at char, LN : split list at index, SS : split at regex
T	N	N : negate
U	L	L : reverse
V	J|L|N|C	J : decrement in place, L : uncons from back, N : decrement, C : decrement
W	B|L|D	B : while loop (repeat as long as block returns true), L : sum (fold using +), D : export all variable
X	A	A : assign to variable x and pop from stack
Y	A	A : assign to variable y and leave on stack
Z	N|S	N : cast to bignum, S : parse to bignum
\\	AA	AA : swap top two elements on the stack
^	NN|SS	NN : power, SS : levenshtein distance
|	NN	NN : logical or
~	B|L|S|C|D	B : evaluate, L : dump to stack, S : evaluate, C : evaluate, D : set variables if they exist in the local
.!	B|N|S	B : copy block without header, N : signum, S : parse if number
.$	..AN	..AN : copies the Nth item on the stack to the top (not including N)
.%	NN	NN : integer division
.&	SSS|LLB|SNN|LNN|NNN	SSS : replace all occurances of the regex S1 with S2 in S3, LLB : zip with, SNN : convert base of N
.'	L|N|S	L : convert number list to string using UTF-8 encoding, N : cast to char, S : cast to char

Table 1 – cor

.(	NN	NN : left bitwise shift
.)	NN	NN : signed right bitwise shift
.*	B|L	B : decompile, L : compile
.+	NN|BD|BJ|BL<J>|DD	NN : gdc, BD : swap vars in a copy of B for values defined in D, BJ : constant capture variable fro
.-	DS|DJ|LL|NN|LN	DS : remove key from dict, DJ : remove key from dict, LL : remove items at indices L1 from L2, N
./	N	N : ceiling
.;	..A	..A : clear the entire stack
.<	SN|LN|NN|SS|CC	SN : head / pad ' ', LN : head / pad 0, NN : greater of, SS : greater of, CC : greater of
.=	LA|LL|AL	LA : element-wise equivalence, LL : element-wise equivalence, AL : element-wise equivalence
.>	SN|LN|NN|SS|CC	SN : tail / pad ' ', LN : tail / pad 0, NN : lesser of, SS : lesser of, CC : lesser of
.?	AAA	AAA : if A1 then A2, else A3. If A2/A3 are blocks, execute
.@	..AN	..AN : moves the Nth item on the stack (not including N) to the top
.A	..A	..A : wrap entire stack in a list
.B	AL	AL : append item to the back of a list
.C	LB|NN	LB : sort least to greatest by applying B to L, NN : xor
.D	A	A : throw an exception containing A
.E	L	L : length, keep list on stack
.F	L	L : flatten nested list
.G	ASN	ASN : write A as a string to file located at S. N = 0, overwrite. N = 1, append
.I	LNA|DSA|DJA	LNA : getindex with default value, DSA : getindex with default value, DJA : getindex with default v
.K	BB	BB : try B1, if error, execute B2. Neither block has access to the global stack
.M	A	A : get metatable
.N	LB	LB : return the index of the first element of L that satifies E; keep list on stack
.O	AB	AB : apply
.P	A	A : print to stdout
.Q	-	- : return a random decimal from 0 to 1
.R	L|N	L : linspace [from to count], if count not provided, use 100, N : range [0, 1, .., N-1]
.S	LL|LN	LL : rotate [rows cols], LN : rotate]
.T	L	L : transpose a 2d list
.U	S	S : requests a string using a ui dialog, S is the prompt text
.V	AL	AL : append item to back of list
.\\	N	N : floor
.^	N|S	N : square root, S : quote regex
.|	B|N	B : get meta information for a block, N : absolute value
.~	B|J|S|C|D	B : get contents of block, J : deref variable; if not a block, put contents in block, S : parse content
:!	AA	AA : assert equal
:#	L:#B|D:#B	L:#B : map, D:#B : map over key value pairs
:$	..AN	..AN : copies the first N items on the stack (not including N)
:%	NN	NN : mod
:&	A	A : duplicate reference (same as $ but does not make a copy)
:'	S|N|C	S : convert a string to bytes using UTF-8 encoding, N : identity; return N, C : to int
:*	LLB	LLB : outer product of two lists using B
:;	..AA	..AA : clear all but the top of the stack
:<	NN|SS|CC	NN : less then or equal to, SS : less then or equal to, CC : less then or equal to
:=	AJ|AC|AS	AJ : assign A to variable, AC : assign A to variable, AS : assign A to variable
:>	NN|SS|CC	NN : greater than or equal to, SS : greater than or equal to, CC : greater than or equal to
:?	A	A : convert to boolean
:@	AA	AA : isinstance
:A	..AN	..AN : collect N items from stack into list
:B	S	S : interpolate string

Table 1 – cor

:C	J|S	J : convert symbol to string name, S : return S
:D	ASD|AJD	ASD : set dict index, AJD : set dict index
:E	L|D	L : shape, D : number or items in a dict
:G		n/a : Return the variable scope stack as a list of dicts
:I	DS|DJ	DS : get dict item from key, DJ : get dict item from key
:J	LA|AA|LL|AL	LA : add to list (modify list), AA : create list [A A], LL : concatenate lists (modify list 1), AL : add
:K	D	D : return a list of keys as symbols
:M	BD|DD	BD : duplicate block with the given metadata, DD : set D1's meta to D2 leave D1 on stack
:N	LA	LA : find all instances of A in L
:O	AAB	AAB : apply (2-arg)
:P	A	A : println to stdout
:R	-	- : readline from stdin
:S	B|S|C	B : if block has single var or op convert to symbol list, else return empty list, S : convert to symbo
:T	A	A : type of (returns a symbol)
:V	D	D : return a list of values
:Z	N	N : sleep (milliseconds)
:`	BN:`A	BN:`A : wrap next N instructions in a block
:|	LL	LL : remove all elements in L2 from L1
:~	L	L : remove duplicates
M!	N	N : factorial
M#	A	A : hash code of the object
M$	-	- : system time in milliseconds
M?	B|N|S	B : get help data for operator, N : list op descriptions where N=[0:std, 1:dot, 2:colon, 3:misc], S : s
MC	N	N : inverse cosine
MI	NN	NN : create complex number
ML	N	N : base-10 logarithm
MS	N	N : inverse sine
MT	N	N : inverse tangent
Ma	J	J : Aya meta information
Mb	B|J	B : duplicate block, add locals if they do not exist, J : is defined
Mc	N	N : cosine
Md	N|S	N : cast to double, S : parse double, if invalid, return 0.0
Me	N	N : exponential function
Mi	N	N : imag part of complex number
Mk	CS|NN	CS : add special character, NN : unsigned right bitshift
Ml	N	N : natural logarithm
Mm	D	D : true if the dict has a metatable, leave D on stack
Mp	N	N : list primes up to N
Mr	N	N : convert to fractional number
Ms	N	N : sine
Mt	N	N : tangent
Mu	NN	NN : y x Mu => atan2(y,x)

1.5 Numbers

Aya has several representations for numbers: Num (represented by a double), BigNum, Rationals, and Complex (coming soon). Numbers are only promoted when needed. Number literals are always converted to Nums. Aya uses standard mathematical operators.

```
3 4 +   .# => 7
5 6 -   .# => -1
2 0.5 * .# => 1.0
3 2 ^   .# => 9
6 4 /   .# => 1.5
6 2 /   .# => 3
```

- is never a unary operator.

```
8 3 -1  .# is evaluated as (8 3-) 1 => 5 1
-1      .# ERROR: Empty stack at operator '-'
```

To write negative numbers, use a colon (with or without a -)

```
:1.5   .# => -1.5
:-1.5  .# => -1.5
```

1.5.1 Special Number Literals

*See*Syntax Overview: Numbers

Special number literals always begin with a colon. Special number literals can be used to create negative numbers, **bignums**, **rationals**, and **complex numbers** *(coming soon)*.

```
.# A colon paired with a number with no additional formatting is negative
:3    .# -3
:-3   .# -3

.# BigNums end with a z
:123z      .# 123
:-3.1232z  .# -3.1232

.# Rational numbers separated numerator and denominator with a r
:1r2   .# 1/2
:3r    .# 3/1

.# Complex numbers are separated with an i
:1i    .# The imaginary unit
:2i5   .# 2i + 5
```

Special number literals also provide ways for creating numbers using binary and hexadecimal formatting.

```
.# Hexadecimals begin with :0x
.# All letters must be lowercase
:0xff  .# 255
:0x111 .# 273
```

(continues on next page)

(continued from previous page)

```
.# Binary literals begin with :0b
:0b11010  .# 26

.# Large hexidecimal and binary numbers are converted to BigNums
:0xffffffff    .# 268435455 (Num)
:0xffffffff   .# 4294967295 (BigNum)
```

1.5.2 Misc. Number Literals

Like all number literals, these values are evaluated pre-runtime.

Scientific Notation

Number literals of the form `:NeM` are evaluated to the literal number `N * 10^M`.

```
aya> :4e3
4000
aya> :2.45e12
2450000000000
aya> :1.1e-3
.0011
```

PI Times

Number literals of the form `:NpM` are evaluated to the literal number `(N * PI)^M`. If no `M` is provided, use the value 1.

```
aya> :1p2
9.8696044
aya> :1p
3.14159265
aya> :3p2
88.82643961
```

Root Constants

Number literals of the form `:NqM` are evaluated to the literal number `N^(1/M)`. The default value of M is 2.

```
aya> :2q
1.41421356
aya> :9q
3
aya> :27q3
3
```

1.6 Lists

*See*Syntax Overview: Lists

List literals are created using square brackets and do not need commas. Literals are first evaluated as their own stack. The results remaining on the stack become the list items.

```
[1 2 3 4 5]      .# Do not use commas
[1 2 + 7 2 - 3!] .# => [3 5 -3]
```

List literals can grab items from the outer stack using the format `... [num| ...]` where num is an integer literal.

```
aya> 1 2 3 4 5 [3| 6 7 8]
1 2 [3 4 5 6 7 8]

aya> 'h 'e [2|'l 'l 'o]
"hello"

aya> "a" "b" [2|]
["a" "b"]
```

List grabbing only uses integer literals

```
aya> 2 :n
2
aya> 1 2 [n| 3 4]
ERROR: Empty stack at operator '|'
stack:
    1 2
just before:
```

1.6.1 Essential List Operations

List Indexing

Lists are indexed using square bracket syntax following a `.`. For Example:

```
aya> ["the" "cat" "in" "the" "hat"]:list
[ "the" "cat" "in" "the" "hat" ]
aya> list.[0]
"the"
```

Aya supports negative indexing, multiple indexing and filtering with this syntax.

```
aya> list.[:1]
"hat"
aya> list.[1 4]
[ "cat" "hat" ]
aya> list.[{E 3 =}]
[ "the" "cat" "the" "hat" ]
```

Arg Type	Function	Input	Out put
Numb er	Choose the nth item from the list (starting from 0)	`[1 2 3].[1 ]`	``2``
List	Use each item in the second list to index the first	``**"abc".[1 2** 2]``	`" bcc "`
Bloc k	Filter the list. Take all items that satisfy the block	`[1 1 2 2]. [{1=}]`	`[ 1 1 ]`

Lists can also be indexed using the `I` operator:

```
aya> ["the" "cat" "in" "the" "hat"]:list
[ "the" "cat" "in" "the" "hat" ]
aya> list 0 I
"the"
aya> list :1 I
"hat"
```

`.I` operator takes a default value if the index is out of bounds:

```
aya> ["hello" "world"] 0 "nope" .I
"hello"
aya> ["hello" "world"] 9 "nope" .I
"nope"
```

Use the following syntax to set elements of a list

```
item list.:[i]
```

which is equivalent to `list[i] = item` in C-style languages.

Essential List Operators

*See*Operators

Extend (K)

```
aya> [1 2 3] :list
[ 1 2 3 ]
aya> list [4 5 6] K
[ 1 2 3 4 5 6 ]
aya> list
[ 1 2 3 4 5 6 ]
```

Join (J)

Similar to ``K`` but never modifies either list

```
aya> [1 2 3] :list;
aya> list [4 5 6] J
[ 1 2 3 4 5 6 ]
aya> list
[ 1 2 3 ]
```

Reshape (L)

```
aya> 9R [3 3] L
[ [ 1 2 3 ] [ 4 5 6 ] [ 7 8 9 ] ]
aya> [1 2] [2 2 2] L
[ [ [ 1 2 ] [ 1 2 ] ] [ [ 1 2 ] [ 1 2 ] ] ]
aya> 100R [2 3] L
[ [ 1 2 3 ] [ 4 5 6 ] ]
```

Flatten (.F)

```
aya> [[1 2] [3] 4 [[5] 6]] .F
[ 1 2 3 4 5 6 ]
```

Pop from front / back

```
aya> [1 2 3] B
[ 1 2 ] 3
aya> [1 2 3] V
[ 2 3 ] 1
```

Append to front / back

```
aya> 1 [2 3] .B
[ 2 3 1 ]
aya> 1 [2 3] .V
[ 1 2 3 ]
```

Generators

Range (R)

One item: create a range from 1 (or `'a'`) to that number.

```
10 R    .# => [1 2 3 4 5 6 7 8 9 10]
'B R    .# => "...56789:;<=>?@AB" (from char code `1` to the input char)
```

Two items: create a range from the first to the second.

```
[5 10] R     .# => [5 6 7 8 9 10]
['z 'w] R    .# => "zyxw"
"zw" R       .# => "zyxw"
```

Three items: create a range from the first to the third using the second as a step.

```
[0 0.5 2] R      .# => [0 0.5 1.0 1.5 2.0]
[2 2.5 4] R      .# => [2 2.5 3 3.5 4]
```

1.6.2 List comprehension

When commas are used inside of a list literal, the list is created using list comprehension. List comprehension follows the format `[range, map, filter1, filter2, ..., filterK]`. The range section is evaluated like the R operator. When the list is evaluated, the sections are evaluated from left to right; first create the range, then map the block to the values, then apply the filters. All filters must be satisfied for an item to be added to the list.

If the map section is left empty, the list is evaluated as a basic range.

```
aya> [10 ,]
[1 2 3 4 5 6 7 8 9 10]

aya> ['\U00A3' '\U00B0' ,]
"£¤¥¦§¨©ª«¬®¯°"

aya> [0 3 15 , T]
[0 -3 -6 -9 -12 -15]
```

Here are some examples using map and filter.

```
aya> [10, 2*]
[2 4 6 8 10 12 14 16 18 20]

aya> [10, 2*, 5<]
[2 4]

aya> [10, 2*, 5<, 4=!]
[2]

.# Can grab from stack
aya> 3 [1| 6 18, 2*]
[ 6 12 18 24 30 36 ]
```

If a list literal is used as the first section of a list comprehension, the list comprehension is simply applied to the inner list.

```
aya> [ [1 2 3 4 5], 2*, 7<]
[2 4 6]
```

If there are two or more lists used as the first argument of a list comprehension, and each list is the same length, all respective elements of each list will be added to the stack when applying the map and filter sections.

```
aya> [ [1 2 3][4 5 6], +]
[5 7 9]

aya> [ "hello" "world", J]
[ "hw" "eo" "lr" "ll" "od" ]
```

1.6.3 The Broadcast Operator

is a very powerful *infix* operator. It's primary function is map. It takes the arguments from its right side and maps them to the list on the left side.

```
[1 2 3] # {1 +} .# => [2 3 4]
```

If a block is not given on the right side, # will collect items until an operator or variable is encountered.

```
.# Same as the previous example
[1 2 3] # 1 + .# => [2 3 4]
```

will also collect items on its left side until a list is hit. It will add these items to the front of the block being mapped to.

```
.# Also the same as the previous line
[1 2 3] 1 # + .# => [2 3 4]
```

This operator can be used to construct *"for loops"* on variables

```
"hello" :str;
str # {c,
    c toupper
}
=> "HELLO"
```

The :# operator works the same way except it *always* takes a list on the left and a block on the right:

```
list :# {block}
```

```
aya> [1 2 3] :# {3+}
[ 4 5 6 ]

aya> [1 2 3] 3 :# +
ERROR: Empty stack at operator ':#'

aya> [1 2 3] 3 # +
[ 4 5 6 ]
```

(continues on next page)

(continued from previous page)

```
aya> [1 2 3] 3 :# {+}
TYPE ERROR: Type error at (:#):
    Expected ((L:#B|D:#B))
    Recieved ({+} 3 )
stack:
    [ 1 2 3 ]
just before:
```

1.7 Characters

*See*Syntax Overview: Characters

Character literals are created using single quotes. Most characters do not need closing quotes.

```
'a      .# => 'a
'p'q    .# => 'p 'q
```

1.7.1 Special Characters

Using a \ after a single quote denotes a special character. Special characters always need a closing single quote.

Hex character literals

Hex literal characters are written using a `'\x___'` and need closing quotes.

```
aya> '\x00FF'
'ÿ'
aya> '\x00A1'
'¡'
```

Named Characters

Many characters have names. All names consist only of lowercase alphabetical characters. Named characters can be used like so within Aya:

```
'\alpha'      .# => "
'\pi'         .# => "
'\because'    .# => "
'\n'          .# => <newline>
'\t'          .# => <tab>
```

To add or override a named character from within Aya, use the `Mk` operator.

```
aya> '\integral'
SYNTAX ERROR: '\integral' is not a valid special character

aya> '\U222b' "integral" Mk
```

(continues on next page)

(continued from previous page)

```
aya> '\integral'
''
```

1.8 Strings

*See*Syntax Overview: Strings

Strings are created using the double quote character ".

```
"I am a string"
"I am a string containing a newline character\n\t and a tab."
```

Strings may span multiple lines.

```
"I am a string containing a newline character
    and a tab."
```

Strings can contain special characters using \{___}. Brackets can contain named characters or Unicode literals.

```
"Jack \{heart}s Jill"          .# => "Jack s Jill"
"sin(\{theta}) = \{alpha}"     .# => "sin() = "
"\{x00BF}Que tal?"             .# => "¿Que tal?"
```

Strings are essentially a list of characters, so any list operator that can be used on lists can be can be used on strings.

```
"Hello " "world!" K  .# => "Hello world!"
['s't'r'i'n'g]       .# => "string"
"abcde".[2]          .# => 'c'
```

1.8.1 String Interpolation

Use the $ character within a string to evaluate the variable or statement following it. If used with a variable name, evaluate the variable name.

```
aya> 5:num;
aya> "I have $num apples"
"I have 5 apples"
```

If used with a group (), evaluate the group.

```
aya> "I have $(1 num +) bananas"
"I have 6 bananas"
```

If there are more than one item left on the stack, aya dumps the stack inside square brackets.

```
aya> 123:playera;
aya> 116:playerb;
aya> "The final scores are $(playera playerb)!"
"The final scores are [ 123 116 ]!"
```

If used after a \, keep the $ char.

```
aya> 10:dollars;
aya> "I have \$$dollars."
"I have $10"
```

If used with anything else, keep the $.

```
aya> "Each apple is worth $0.50"
"Each apple is worth $0.50"
```

Here are some additional examples:

```
aya> 5:num;
aya> 0.75:price;
aya> "I sold $num apples for \$$price each and I made \$$(num price*)"
"I sold 5 apples for $0.75 each and I made $3.75"

aya> "Inner $(\"strings\")"
"Inner strings"

aya> "Inner $(\"$a\") interpolation requires backslashes"
"Inner 1 interpolation requires backslashes"

aya> "Inner-$(\"$(\\\"inner\\\")\") interpolation can be messy"
"Inner-inner interpolation can be messy"
```

1.8.2 Long String Literals

Long strings are entered using triple quotes. No characters are escaped within long strings. In the following code…

```
"""<div id="my_div">
    <h1>\n: the newline character</h1>
    <p>\{alpha}<p>
    <p>$interpolate</p>
</div>"""
```

…no escape characters are parsed in the output:

```
"<div id="my_div">
    <h1>\n: the newline character</h1>
    <p>\{alpha}<p>
    <p>$interpolate</p>
</div>"
```

1.9 Blocks

Blocks contain expressions. They can be used to define functions, map instructions to lists, etc. They are denoted using curly braces `{}`.

```
{20 50 +}
```

They can be evaluated by using the ~ operator.

```
{20 50 +}~ .# => 70
```

When blocks are evaluated, their contents are dumped to the stack and the stack continues as normal. This is what happens when we call functions as well.

```
100 10+ {1 + 2 *}~
    110 {1 + 2 *}~
       110 1 + 2 *
           111 2 *
               222
```

1.9.1 Block Header

A comma (`,`) is used to specify that the block has a header. Anything before the comma is considered the header and everything after is considered the instructions. A block header is used to introduce local variables to the block in the form of arguments or local declarations. Arguments and declarations are separated by a colon (`:`). Arguments must go on the left hand side of the colon and local declarations on the right.

```
{<arg1> <arg2> ... <argN> : <local dec 1> ... <local dec M>, <block body>}
```

If no colon is included in the header, all variable names will be used as arguments.

```
{<arg1> <arg2> ... <argN>, <blovk body>}
```

If a colon is the first token in a block header, all variable names are considered local declarations.

```
{: <local dec 1> <local dec 2> ... <local dec M>, <block body>}
```

Finally, if nothing is included in the block header, the block will be parsed as a dictionary.

```
{, <dict body>}
```

1.9.2 Arguments

Arguments work like parameters in programming languages with anonymous/lambda functions. Before the block is evaluated, its arguments are popped from the stack and assigned as local variables for the block.

```
aya> 4 {a, a2*}~
8
```

Arguments are popped in the order they are written.

```
aya> 8 4 {a b, [a b] R}~
[8 7 6 5 4]
```

Arguments are local variables.

```
aya> 2:n 3{n, n2^}~ n
2 9.0 2
```

1.9.3 Argument Type Assertions

Arguments may have type assertions. Write a variable name followed by a symbol corresponding to the type.

```
1 2 {a::num b::num, a b +}~     .# => 3
"1" 2 {a::num b::num, a b +}~   .# TYPE ERROR: Type error at ({ARGS}):
                                Expected (::num)
                                Recieved ("1" )
```

If a user defined type defines a `type` variable as a symbol. The symbol will be used for type assertions.

```
{,
  ::vec :type;

  ... define vec variables and functions ...
}:vec;

{v::vec,
  v :P
}:printvec;
```

The type checker will use the `.type` variable:

```
aya> 1 2 vec! :v
<1,2>
aya> v printvec
<1,2>
aya> 3 printvec
TYPE ERROR: Type error at ({ARGS}):
    Expected (::vec)
    Recieved (3 )
```

1.9.4 Local Declarations

Local declarations introduce a local scope for that variable. Scope is discussed in greater detail in the Variable Scope section of this document. Local declarations can not have type declarations.

```
aya> "A":a
"A"
aya> a println {:a, "B":a; a println}~ a println
A
B
A
```

All local declarations default to the value 0.

```
{: a, "a is $a" :P } ~
```

Change the default value for a local variable using an initializer.

```
aya> {: a(10) b c("hello") d([1 2]), [a b c d] } ~
[ 10 0 "hello" [ 1 2 ] ]
```

Variables are initialized before run time and therefore can not be variables.

```
aya> 99 :l
99

aya> {: a(l), a} ~
SYNTAX ERROR: Block header: Local Variables Initializer: Must contain only const values

aya> .# define a as a function which evaluates to l
aya> {: a({l}), a} ~
99

aya> .# define a as a list which evaluates to l
aya> {: a([l]), a} ~
[ 99 ]
```

1.9.5 Keyword Arguments

Aya provides a way to use keyword arguments using dictionaries and local declarations. Consider the following function:

```
{kwargs::dict : filename("") header dtype(::num),
  kwargs .W

  "filename=\"$filename\", header=$header, dtype=$dtype" :P
}:fn;
```

The function `fn` contains 1 argument `kwargs` (the name can be anything) and three local declarations. The operator `.W` will export variables from the `kwargs` dict only if they are defined in the local scope. This means that any variables defined in `kwargs` will overwrite the initialized local variables. Every variable not given by `kwargs` dict will remain in its default state.

```
aya> {, "sales.csv":filename 1:header} fn
filename="sales.csv", header=1, dtype=::num

aya> .# The variable `useless` does not exist in the local scope of `fn`
aya> .#    and will therefore be ignored
aya> {, "colors.csv":filename "blah":useless} fn
filename="colors.csv", header=0, dtype=::num

aya> {, "names.csv":filename ::str:dtype} fn
filename="names.csv", header=0, dtype=::str
```

(continues on next page)

(continued from previous page)

```
aya> {, } fn
filename="", header=0, dtype=::num
```

1.10 Functions

We now have the basic building blocks for defining functions: variable assignment and blocks. A function is simply a variable that is bound to a block. When the variable is called, the interpreted dumps the contents of the block onto the instruction stack and then continues evaluating. Functions can take advantage of anything that a normal block can including arguments and argument types.

Here are a few examples of function definitions: `swapcase` takes a character and swaps its case.

```
aya> {c::char, c!}:swapcase;
aya> 'q swapcase
'Q'
```

Below is the definition of the standard library function roll, This function will move the last element of a list to the front.

```
aya> {B\.V}:roll;
aya> [1 2 3 4] roll;
[4 1 2 3]
```

When used with block arguments, functions can be written in very readable ways. The following function swapitems takes a list and two indices and swaps the respective elements. It uses block arguments and type assertions.

```
{listL i::num j::num : tmp,
  list i I : tmp;
  list j I  list i D
  tmp list j D
  list
}:swapitems;

aya> [1 2 3 4 5] 0 3 swapitems
[ 4 2 3 1 5 ]
```

To see more examples check out the standard library located at /base/std.aya

1.11 Dictionaries

A dictionary is a set of key-value pairs. The keys must always be valid variable names. A dictionary literal is created using a block with an empty header. The block is evaluated and all variables are in assigned in the scope of the dictionary.

```
{,
  <dictionary body>
}
```

Below is a simple dictionary example.

```
.# Define a simple dictionary
{,
  1:one;
  2:two;
  3:three;
}:numbers;
```

Empty dictionaries are created if the block and the header are empty.

```
aya> {,}
{,
}
```

1.11.1 Accessing Values

Access variables are used to access variables in dictionaries and user types. To create an access variables, use a dot before the variable name.

```
aya> numbers.two
2

.# whitespace is optional
aya> numbers .one
1
```

1.11.2 Assigning / Creating Values

Dictionary values can be assigned using the `.:` operator.

```
aya> {, 1:a 2:b} :d
{,
  1:a;
  2:b;
}
aya> 4 d.:a
{,
  4:a;
  2:b;
}
aya> 9 d.:c
{,
  4:a;
  2:b;
  9:c;
}
```

They may also be assigned using the following syntax:

```
item dict.:[key]
```

where `key` is a string or a symbol.

For example:

```
aya> {, 0:x } :dict;
aya> 1 dict.:[::y]
aya> dict
{,
  0:x;
  1:y;
}

aya> 1 dict.:["y"]
aya> dict
{,
  0:x;
  1:y;
}
```

Loop over k/v pairs in a dict using the `:#` operator

```
aya> dict :# {k v, v 1 + dict.:[k]}
aya> dict
{,
  1:x;
  2:y;
}
```

1.11.3 Metatables

In Aya, metatables can be used to define custom types with separate functionality and moderate operator overloading. User types are represented internally as an array of objects paired with a dictionary. Any dictionary can contain a read-only set of variables as a metatable. Metatables typically contain functions that act on the dictionaries values. For example, if we define the metatable

```
{, {self, self.x self.y +}:sum;  {}:donothing; } :meta;
```

and the dictionary

```
{, 1:x 2:y {}:none } :dict;
```

we can set the metatable using the MO operator like so

```
aya> meta meta.:__meta__
{,
  1:x;
  2:y;
  {}:none;
}
```

We can see that the dict still has the values `x` and `y` but it also now has a hidden entry for the key sum in its metatable. When we call the metatable variable, the dictionary will be left on the stack and the metatable value will be evaluated.

```
aya> dict.sum
3
```

(continues on next page)

(continued from previous page)

```
aya> dict.donothing
{,
  1:x;
  2:y;
  {}:none;
}
```

1.12 Variables

*See*Syntax Overview: Variables

Variables may only contain lower case letters and underscores. They are assigned using the colon (:) operator. The value is left on the stack after the assignment has occurred.

```
aya> 1 :a
1
aya> 3:b a +
4
```

1.12.1 Variable Scope

A new scope is introduced if a block contains any variable declaration in its header. When a variable assignment occurs, the interpreter will walk outward until a reference to that variable appears. If it does not appear in any of the scopes before the global scope, a new reference will be created there. In order to ensure a variable is using local scope, the variable name must be included in the block header. If a block does not contain a header, a new scope will not be introduced. These concepts are best demonstrated by example.

Let us introduce the variables a and b:

```
"A":a; "B":b;
```

When blocks have arguments, a scope is introduced for that variable. Here, the number zero is assigned to b within the scope of the block. When the block ends, the scope is destroyed and we reference the now global variable b.

```
aya> 0 {b, b.P}~ b.P
0B
```

Local variables also create local scopes for that variable. Here, we create a local scope for the variable b. a is not included in the new scope.

```
aya> .# Local variable b declared in header
  {:b,
    0:a;
    1:b;
    "a = $a," .P
    "b = $b\n" .P
  }~
  "a = $a," .P
  "a = $b\n" .P
```

(continues on next page)

(continued from previous page)

```
a = 0,b = 1
a = 0,a = B
```

1.13 User-Defined Types

1.13.1 Classes

Classes are defined using the `class` keyword

```
aya> class person
aya> person
<type 'person'>
```

Constructor

The constructor (`__init__`) takes any number of optional arguments followed by a `self` argument. `self` must always be the last argument in the list:

```
def person::__init__ {name age self,
    name self.:name;
    age  self.:age;
}
```

Create an object with the `!` operator:

```
aya> "Jane" 25 person! :jane;
(person 0x259984df)
aya> jane.name
"Jane"
aya> jane.age
25
```

Functions

Like the constructor, a member function takes `self` as an argument:

```
def person::greet {self,
    "Hi it's $(self.name)"
}
```

It is called like any other class variable:

```
aya> jane.greet
"Hi it's Jane"
```

Print/String Overloading

`__repr__` is a special function that is called when the object is printed. Overload it to change how an object is printed to the console

```
def person::__repr__ {self,
    "person: $(self.name)"
}
```

For example

```
aya> jane
person: Jane
```

`__str__` is a special function that is called when the object is converted to a string

```
def person::__str__ {self,
    self.name
}
```

For example

```
aya> jane P
"Jane"
aya> "I saw $jane the other day"
"I saw Jane the other day"
```

Operator Overloading

Many operators can be overloaded for user types. Type `\? overloadable` in the repl for a full list. Many of the standard libraries use this feature to seamlessly integrate with the base library. For example, the matrix library uses it for all math operators:

```
aya> import ::matrix
aya> [[1 2][3 4]] matrix! :m
[[ 1 2 ]
 [ 3 4 ]]
aya> m 10 + 2 /
[[ 5.5 6 ]
 [ 6.5 7 ]]
```

It is especially useful when writing libraries for code golf. The `asciiart` library uses it to create specialized operators on it's custom string type. Here is a 13 character function for creating a size `N` serpinski triangle:

```
aya> 4 "##`#"_\L{I}/
asciiart:
################
# # # # # # # #
##  ##  ##  ##
#   #   #   #
####    ####
# #     # #
##      ##
```

(continues on next page)

(continued from previous page)

```
#       #
########
# # # #
##  ##
#   #
####
# #
##
#
```

Let's overload the increment operator (`B`) to increment a person's age.

Here we modify the object directly

```
def person::__inc__ {self,
    self.age B self.:age;
}
```

Gives us

```
aya> jane.age
25
aya> jane B
aya> jane.age
```

If we don't want to modify the object but return a modified copy we could have chose to use the `$` syntax to pass a copy of the object instead:

```
def person::__inc__ {self$,
    self.age B self.:age;
    self .# Leave the copy on the stack
}
```

Usage

```
aya> jane.age
25
aya> jane B :jane_older;
aya> jane.age
25
aya> jane_older.age
26
```

Class Variables & Functions

To define a shared class variable, assign it to the class directly:

```
def person::counter 0
```

or

```
0 person.:counter;
```

We can then redefine our construtor to keep track of how many times we've called the constructor.

Note that we can access `counter` directly from `self` but we need to use `__meta__` to update it to ensure we are updating the shared variable.

```
def person::__init__ {name age self,
    name self.:name;
    age  self.:age;
    self.counter 1+ self.__meta__.:counter;
}
```

Class functions take the class as an argument:

```
def person::create_anon {cls,
    "Anon" 20 cls!
}
```

They are called with the class (rather than with an instance)

```
aya> person.create_anon :anon
(person 0x7a1fe926)
aya> anon.name
"Anon"
```

Inheritance

Aya classes support single inheritance. We can use the extend operator to create a class that is derived from another class. Here we create an `employee` class which extends the `person` class. It will simply add a `job` field.

Note that `extend` is not a keyword like `class` but an operator that takes the class as a symbol argument

```
::employee person extend;
```

or more generally

```
::derived base extend;
```

Our constructor calls the person constructor with `name` and `age` and then adds a `job` field.

```
def employee::__init__ {name age job self,
    .# call super constructor
    name age self super.__init__

    .# derived-specific code
    job self.:job;
}
```

In the example below, not that employee still calls `__repr__` we defined for the `person` class.

```
aya> "Bob" 30 "salesman" employee!
person: Bob
```

We can overload the `greet` function to include the job:

```
def employee::greet {self : greeting,
    .# call super greet
    .# must pass `self` to super
    self super.greet :greeting;

    .# append derived-specific greeting to output
    greeting ", I'm a $(self.job)" +
}
```

Calling it:

```
aya> bob.greet
"Hi it's Bob, I'm a Salesman"
```

1.13.2 Structs

In Aya, structs are classes. The `struct` keyword simply creates a class with a few convience functions already defined.

The syntax is

```
struct <name> {<member1>, <member2>, ...}
```

For example, lets create a `point` struct for representing a 2d point:

```
struct point {x y}
```

The constructor is created automatically for us. It takes each member as an argument in the same order they are defined

```
aya> 3 4 point! :p;
aya> p.x
3
aya> p.y
4
```

`__repr__` and `__str__` functions are also automatically created:

```
aya> p
( 3 4 ) point!
aya> p P
"( 3 4 ) point!"
```

1.13.3 Internals

Keywords such as `class`, `struct`, and `def` are not actually keywords at all. They are regular aya functions defined completely in aya code (see base/**aya**.aya).

Classes, structs, and object instances are simply dictionaries with special **meta** dictionaries. If you are interested in seeing how these are implemented entirely in aya, read on.

Below is an example of a 2d vector "class" definition written *from scratch* without using any convience functions. Member functions and overloads work the same as they do for normal classes. The only major difference is object creation (`__new__` vs `__init__`) and the special variables `__pushself__` and `__type__` at the top of the metatable.

```
{,

  1:__pushself__;
  ::vec:__type__;

  .# Constructor
  {x y cls,
    {,
      x:x;
      y:y;
      cls:__meta__;
    }
  }:__new__;

  .# Member functions

  .# Print overload
  {self,
    "<$(self.x),$(self.y)>"
  }:__repr__;

  .# Compute vector length
  {self,
    self.x 2^ self.y 2^ + .^
  }:len;

  .# Operator overload
  {other self,
    other.x self.x +
    other.y self.y +
    vec!
  }:__add__;

}:vec;
```

Special Metatable Variables

```
1:__pushself__;
::vec:__type__;
```

`__pushself__` tells aya to push a reference of the object to the stack when calling functions on it. It effectively enables the use of `self`

The symbol assigned to `__type__` is used for type checking and overloading the `:T` (get type) and `:@` (is instance) operators.

Constructor

```
{x y cls,
    {,
        x:x;
        y:y;
        cls:__meta__;
    }
}:__new__;
```

Object construction with the `!` operator is just a standard operator overload that calls `__new__`.

Note: For classes, `__new__` creates an instance of the object (i.e. `self`) and then calls `__init__` wich takes self as an argument.

1.14 Metaprogramming

1.14.1 Blocks

Aya provides a basic data structure for representing code called a *block*. A block is a list of instructions. Internally, every Aya program is a block.

```
aya> {1 1 +}
{1 1 +}
```

Evaluate it with the ~ operator

```
aya> {1 1 +} ~
2
```

By default, blocks assigned to variables are automatically evaluated when de-referenced. Use `.`` to get the block without evaluating it.

```
aya> {1 1 +} :a
{1 1 +}
aya> a
2
aya> a.`
{1 1 +}
```

Split a block into parts using the `.*` operator.

```
aya> {3 4 *} .*
[ {3} {4} {*} ]
```

The same operator is used to join a list into a block:

```
aya> [ {3} {4} {*} ] .*
{3 4 *}
```

`.*` automatically converts data into instructions

```
aya> [ 3 4 {*} ] .*
{3 4 *}
```

For example, `make_adder` is a function that takes a number `N` and creates a block of code that adds `N` to its input

```
aya> { {+} J .* }:make_adder
{{+} J .*}
aya> 5 make_adder :add_five
{5 +}
aya> 4 add_five
9
```

1.14.2 Macros

In Aya, programs are evaluated from left to right

```
aya> 1 2 +
3
aya> 1 2 + 4 *
12
```

Above, the + and * operators read data from their left. When evaluating +, everything to the left is considered **data** and everything to the right is considered **instructions**.

```
     1 2 + 4 *
<-- data | instructions -->
```

All standard operators and functions operate only on *data*; that is, things to their left.

A macro is a function that operates on *instructions*; or things to its right. Macros may also operate on *data* and *instructions*.

For example, `struct` is a macro that reads two instructions: the type name and the list of member variables.

```
aya> struct point {x y}
<type 'point'>
```

`if` is a macro that reads 3 instructions to achieve behavior similar to `if` keywords from imperitive languages

```
aya> if (1) {"true!"} {"false!"}
"true!"
```

The `:`` operator is used to create macros. It takes 2 *data* arguments. A block `B` and an integer `N`. When evaluated, it will wrap each of the next `N` *instructions* in a block (converting them to *data*) then wrap the whole thing in a list. Then it will run `B` after the newly created block.

```
aya> { "data block" } 1 :` instruction
[ {instruction} ] "data block"
aya> {1} 2 :` 3 +
[ {3} {+} ] 1
```

Macro Example

Lets define a macro `apply` that applies the instruction after it to each element of a list.

```
aya> ["three" "two" "one"] apply .upper
[ "THREE" "TWO" "ONE" ]
```

First we use `:`` to capture the instruction we want to apply then use the ~ operator to unwrap the instruction list

```
aya> ["three" "two" "one"] { } 1 :` .upper
[ "three" "two" "one" ] [ {.upper} ]
aya> ["three" "two" "one"] { ~ } 1 :` .upper
[ "three" "two" "one" ] {.upper}
```

We use the map operator `0` to apply the block to each element of the list

```
aya> ["three" "two" "one"] { ~ 0 } 1 :` .upper
[ "THREE" "TWO" "ONE" ]
```

Now we can replace `.upper` with the reverse operator `U` to reverse the strings in the list instead

```
aya> ["three" "two" "one"] { ~ 0 } 1 :` U
[ "eerht" "owt" "eno" ]
```

Finally, we can remove our example data and define our macro.

```
aya> { { ~ 0 } 1 :` } :apply
{{~ 0} 1 :`}
```

Usage:

```
aya> ["three" "two" "one"] apply .upper
[ "THREE" "TWO" "ONE" ]
aya> ["three" "two" "one"] apply U
[ "eerht" "owt" "eno" ]
aya> ["three" "two" "one"] apply .[0]
"tto"
```

Apply multiple instructions by wrapping them in `()`

```
aya> ["three" "two" "one"] apply ("!" +)
[ "three!" "two!" "one!" ]
```

1.15 Standard library

This section is still a work in progress

The Aya standard library consists of type definitions, mathematical functions, string and list operations, plotting tools and even a small turtle graphics library. It also defines functions and objects for working with colors, dates, files, GUI elements, and basic data structures such as queues, stacks, and sets. The standard library also contains a file which defines extended ASCII operators for use when code golfing.

1.15.1 asciiart

Provides an asciiart datatype and several operator overloads for drawing complex ascii art pictures with only a few characters.

Run length encoding:

```
aya> "  #` # #`5#"_
asciiart:
  #
 # #
#####
```

Operator overloads

```
aya> "  #` # #`5#"_ T
asciiart:
  #
 ##
# #
 ##
  #

aya> "  #` # #`5#"_ $I
asciiart:
            #
           # #
          #####
       #         #
      # #       # #
     #####     #####
  #    #    #    #    #
 # #  # #  # #  # #  # #
#########################
```

1.15.2 bitset

Provides the `bitset` type

```
aya> 8 bitset! :b
[ 0 0 0 0 0 0 0 0 ]bitset!
aya> 3 b.set
aya> 5 b.set
aya> b
[ 0 0 0 1 0 1 0 0 ]bitset!
aya> b.count
2
```

1.15.3 canvas

Graphics library for creating images and animations. See `examples/canvas` for more examples.

Fig. 2: Vaporwave City

Fig. 3: 3D Cube

1.15.4 color

The `color` library defines basic color constructors and conversions.

```
aya> 14 57 100 color!
(14 57 100) color!

aya> color.colors.violet :violet
(238 130 238) color!

aya> violet.hsv
[ 300 .45378151 .93333333 ]

aya> violet.hex
"ee82ee"

aya> 5 color.colors.red color.colors.blue.grad
[
  (255 0 0) color!
  (191 0 63) color!
  (127 0 127) color!
  (63 0 191) color!
  (0 0 255) color!
]
```

1.15.5 csv

Provides functions for reading and writing CSV files

```
aya> "examples/data/simple.csv" csv.read
{,
  [
    [ 1 2 3 ]
    [ 4 5 6 ]
    [ 7 8 9 ]
  ]:data;
  nil:rownames;
  [ "A" "B" "C" ]:colnames;
}
```

1.15.6 dataframe

The `dataframe` type is an interface for working with tables. CSV files can be directly imported and modified or the data can be generated by the program itself.

```
aya> {, [[1 2 3][4 5 6]]:data ["x" "y" "z"]:colnames} dataframe!
    x y z
0 | 1 2 3
1 | 4 5 6

aya> {, [[1 2 3][4 5 6]]:data ["x" "y" "z"]:colnames} dataframe! :df
    x y z
0 | 1 2 3
1 | 4 5 6

aya> df.["x"]
[ 1 4 ]

aya> "examples/data/simple.csv" dataframe.read_csv
    A B C
0 | 1 2 3
1 | 4 5 6
2 | 7 8 9
```

1.15.7 date

The date script provides a basic interface for the date parsing operators `Mh` and `MH`. It also provides basic date unit addition and subtraction.

```
aya> date.now
May 01, 2017 12:53:25 PM

aya> date.now.year
2017

aya> date.now 2 dates.month +
```

(continues on next page)

(continued from previous page)

```
Jul 01, 2017 8:53:42 AM

aya> date.now 2 dates.month + .mmddyy
"07/01/17"
```

1.15.8 enum

The enum library defines the enum keyword which uses dictionaries and metatables to create enums.

```
aya> enum shape {circle triangle square}

aya> shape
shape

aya> shape :T
::enum

aya> shape.circle
shape.circle

aya> shape.circle :T
::shape

aya> shape.circle shape.circle =
1
```

1.15.9 golf

golf defines many short variables that are useful when golfing. It also uses the Mk operator to add additional single character operators. In the following code, all variables i, ¶, ¦, ¥ and r are defined in the golf script.

```
aya> .# Generate and print an addition table
aya> 6r_i¶¦¥
   0   1   2   3   4   5
   1   2   3   4   5   6
   2   3   4   5   6   7
   3   4   5   6   7   8
   4   5   6   7   8   9
   5   6   7   8   9  10
```

Sets default values for many variables

```
aya> [ a b c d k l p w z i í]
[ 2 3 10 1000 [ ] 3.14159265 -1 0 {+} {-} ]
```

1.15.10 image

Library for reading and writing images.

```
aya> "images/logo.png" image.read :img
(image 300x300)
aya> img.width
300
aya> img.pixels 5 .<
[
  [ 255 255 255 ]
  [ 255 255 255 ]
  [ 255 255 255 ]
  [ 255 255 255 ]
  [ 255 255 255 ]
]
```

1.15.11 io

Defines the `file` and `path` types

1.15.12 json

Library for reading and writing json

1.15.13 math

The `math` library provides many math functions

1.15.14 matrix

The `matrix` library provides a basic interface and operator overloads for working with matrices.

```
aya> 3 3 10 matrix.randint :mat
|  7  8  2 |
|  8  7  3 |
|  8  4  4 |

aya> mat [[0 1] 0] I
|  7 |
|  8 |

aya> mat [[0 1] 0] I .t
|  7  8 |

aya> mat 2 ^ 100 -
|   29   20  -54 |
|   36   25  -51 |
|   20    8  -56 |
```

1.15.15 missing

Provides the `missing` type for working with missing data

1.15.16 mp

Metaprogramming library

1.15.17 plot

Plotting interface. See `examples/plot`

1.15.18 queue

Queue data structure.

1.15.19 random

Functions for woring with random numbers.

1.15.20 set

The `set` script defines a `set` type and many operator overloads. It defines `s` as a prefix operator for the set constructor allowing the syntax `s[ ... ]` to create sets.

```
aya> s[1 2 3 2 2 1]  .# == ([1 2 3 2 2 1] set!)
s[ 1 2 3 ]

aya> s[1 2 3] s[2 3 4] |
s[ 1 2 3 4 ]

aya> s[1 2 3] s[2 3 4] &
s[ 2 3 ]

aya> s[1 2 3] s[2 5] /
s[ 1 3 ]
```

1.15.21 shell

A shell-like interface for the aya REPL.

1.15.22 socket

Socket and socket server interface.

1.15.23 stack

Stack data structure.

1.15.24 stats

Provides several statistics functions.

1.15.25 sys

Provides functions for working with the system such as getting or changing the working directory.

1.15.26 terminal

Functions for formatting text in the terminal (bold, color, etc)

1.15.27 turtle

Turtle library. See `examples/turtle`

1.15.28 viewmat

Provides the `viewmat` function which is used to generate a heatmap visualization of a 2d array. See `examples/canvas/julia`

1.16 Canvas Input

You can poll for mouse and keyboard input using these standard library instructions.

The `graphics.click_events`, `graphics.move_events` and `graphics.typed_chars` instructions provide you with a list of events that occurred since the last time the instruction was called.

The `graphics.pressed_buttons` and `graphics.pressed_keys` instructions yield the *currently* pressed / held buttons and keys.

1.16.1 Mouse Events

Click Events

Lists the mouse clicks since the last time this instruction was executed.

`graphics.click_events` pushes a list of dictionaries with the following keys onto the stack:

- `x` (num) The x-coordinate the click occurred at.
- `y` (num) The y-coordinate the click occurred at.
- `button` (num) The button number the was clicked. (See *Mouse Buttons*)
- `clicks` (num) The amount of successive clicks. (Useful for detecting double-clicks)

```
aya> my_canvas.id :{graphics.click_events}
[ {
    128 :x;
    256 :y;
    1 :button;
    3 :clicks;
} ]
```

Pressed Buttons

Lists the currently held mouse buttons.

`graphics.pressed_buttons` pushes a list of currently held button-numbers onto the stack. (See *Mouse Buttons*)

```
aya> my_canvas.id :{graphics.pressed_buttons}
[ 1 3 ]
```

Move Events

Lists the mouse movements since the last time this instruction was executed.

`graphics.move_events` pushes a list of dictionaries with the following keys onto the stack:

- `x` (num) The x-coordinate the cursor moved to.
- `y` (num) The y-coordinate the cursor moved to.

```
aya> my_canvas.id :{graphics.move_events}
[ {
    128 :x;
    256 :y;
} {
    130 :x;
    260 :y;
} {
    132 :x;
    264 :y;
} ]
```

1.16.2 Keyboard Events

Pressed Keys

Lists the currently held keyboard keys.

`graphics.pressed_keys` pushes a list of dictionaries with the following keys onto the stack:

- `key_name` (str) The name of the pressed key. (See *Keyboard Keys*)
- `keycode` (num) An integer representation of the key.
- `location_name` (str) The name of the location of the key. (See *Keyboard Locations*)
- `location` (num) An integer representation of the location.

```
aya> my_canvas.id :{graphics.pressed_keys}
[ {
    "A" :key_name;
    65 :keycode;
    "STANDARD" :location_name;
    1 :location;
} {
    "CONTROL" :key_name;
    17 :keycode;
    "LEFT" :location_name;
    2 :location;
} ]
```

Typed Characters

Lists the Unicode characters that were typed since the last time this instruction was executed.

`graphics.typed_chars` pushes a string of typed characters onto the stack.

```
aya> my_canvas.id :{graphics.typed_chars}
"Hello, World!"
```

1.16.3 Overview of possible values

Mouse Buttons

Number	Button
1	left
2	middle
3	right
4	back
5	forward

If your mouse has more than 5 buttons, you may see larger numbers as well.

Keyboard Keys

Keycode	Key Name
0	UNDEFINED
3	CANCEL
8	BACK_SPACE
9	TAB
10	ENTER
12	CLEAR
16	SHIFT
17	CONTROL
18	ALT
19	PAUSE
20	CAPS_LOCK
21	KANA
24	FINAL
25	KANJI
27	ESCAPE
28	CONVERT
29	NONCONVERT
30	ACCEPT
31	MODECHANGE
32	SPACE
33	PAGE_UP
34	PAGE_DOWN
35	END
36	HOME
37	LEFT
38	UP
39	RIGHT
40	DOWN
44	COMMA
45	MINUS
46	PERIOD
47	SLASH
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
59	SEMICOLON
61	EQUALS
65	A
66	B
67	C

continues on next page

Table 2 – continued from previous page

Keycode	Key Name
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	OPEN_BRACKET
92	BACK_SLASH
93	CLOSE_BRACKET
96	NUMPAD0
97	NUMPAD1
98	NUMPAD2
99	NUMPAD3
100	NUMPAD4
101	NUMPAD5
102	NUMPAD6
103	NUMPAD7
104	NUMPAD8
105	NUMPAD9
106	MULTIPLY
107	ADD
108	SEPARATOR
109	SUBTRACT
110	DECIMAL
111	DIVIDE
112	F1
113	F2
114	F3
115	F4
116	F5
117	F6
118	F7

continues on next page

Table 2 – continued from previous page

Keycode	Key Name
119	F8
120	F9
121	F10
122	F11
123	F12
127	DELETE
128	DEAD_GRAVE
129	DEAD_ACUTE
130	DEAD_CIRCUMFLEX
131	DEAD_TILDE
132	DEAD_MACRON
133	DEAD_BREVE
134	DEAD_ABOVEDOT
135	DEAD_DIAERESIS
136	DEAD_ABOVERING
137	DEAD_DOUBLEACUTE
138	DEAD_CARON
139	DEAD_CEDILLA
140	DEAD_OGONEK
141	DEAD_IOTA
142	DEAD_VOICED_SOUND
143	DEAD_SEMIVOICED_SOUND
144	NUM_LOCK
145	SCROLL_LOCK
150	AMPERSAND
151	ASTERISK
152	QUOTEDBL
153	LESS
154	PRINTSCREEN
155	INSERT
156	HELP
157	META
160	GREATER
161	BRACELEFT
162	BRACERIGHT
192	BACK_QUOTE
222	QUOTE
224	KP_UP
225	KP_DOWN
226	KP_LEFT
227	KP_RIGHT
240	ALPHANUMERIC
241	KATAKANA
242	HIRAGANA
243	FULL_WIDTH
244	HALF_WIDTH
245	ROMAN_CHARACTERS
256	ALL_CANDIDATES
257	PREVIOUS_CANDIDATE

continues on next page

Table 2 – continued from previous page

Keycode	Key Name
258	CODE_INPUT
259	JAPANESE_KATAKANA
260	JAPANESE_HIRAGANA
261	JAPANESE_ROMAN
262	KANA_LOCK
263	INPUT_METHOD_ON_OFF
512	AT
513	COLON
514	CIRCUMFLEX
515	DOLLAR
516	EURO_SIGN
517	EXCLAMATION_MARK
518	INVERTED_EXCLAMATION_MARK
519	LEFT_PARENTHESIS
520	NUMBER_SIGN
521	PLUS
522	RIGHT_PARENTHESIS
523	UNDERSCORE
524	WINDOWS
525	CONTEXT_MENU
61440	F13
61441	F14
61442	F15
61443	F16
61444	F17
61445	F18
61446	F19
61447	F20
61448	F21
61449	F22
61450	F23
61451	F24
65312	COMPOSE
65368	BEGIN
65406	ALT_GRAPH
65480	STOP
65481	AGAIN
65482	PROPS
65483	UNDO
65485	COPY
65487	PASTE
65488	FIND
65489	CUT

For more information, check the KeyEvent javadoc

Keyboard Locations

Location Code	Location Name
0	UNKNOWN
1	STANDARD
2	LEFT
3	RIGHT
4	NUMPAD

For more information, check the KeyEvent javadoc

1.17 Debugging

Aya has built-in support for setting breakpoints using the `bp` command. For example:

```
{a b : c,
    a b + :c;
    [a b]
    bp
    c J
}:fn;
```

Calling this function with pause execution at the location of `bp` and open a shell for inspection.

```
aya> 1 2 fn
Execution paused, enter '.' to continue
Stack: [ 1 2 ]
Next instructions:   c J

aya (debug)> a
1

aya (debug)> c
3

aya (debug)> .
[ 1 2 3 ]
```

Setting `__aya__.ignore_breakpoints` to `1` will disable breakpoints in the session and setting it to `0` will enable them. It is set to `0` by default.

```
aya> 1 __aya__.:ignore_breakpoints;

aya> 1 2 fn
[ 1 2 3 ]

aya> 0 __aya__.:ignore_breakpoints;

aya> 1 2 fn
Execution paused, enter '.' to continue
...
```

Aya has its documentation hosted on Read the Docs.