Gura Language Manual

Yutaka Saito

December 27th, 2015

Contents

Chapter 1

Introduction

We often see a process that applies some operation or transformation on multiple data stored in lists and then put the results into another list. Among them are includes plotting results of a mathematical function fed with sequence of numbers as its parameter and tansforming multiple records extracted from some database into a specific format.
For such a process, many programming language provides sequence control syntax for repeating, with which you can pick up elements from a list subsequently and then create another list that contains result values. Or, if you're a programmer of a functional language, it might be a familiar approach that you prepare a higher-order function with which you apply a certain function on elements in a list.
Either way, you've had to explicitly program "repeat" operation with existing languages. However, when you provide n number of values to a function taking one argument and returning one result, it's obvious that you want n number of answers from it. If a programming language itself has a feature to repeat a function automatically when it's given with a list or an iterator as its arguments, there's no need to explicitly describe repeating syntax any more.
I calls this feature **Implicit Mapping** since it *implicitly* does mapping process.
This may look similar with a feature called "vectorization" that has already been adopted by languages and libraries such as MATLAB and NumPy. A different point is that Implicit Mapping is not limited to mathematic operations with number values, but it can work with various types of value like string, image and even user-defined one. And I've found out Implicit Mapping would be much efficient when it cooperates with more sophisticated iterator operations such as **Member Mapping** that can access members of objects coming from an iterator or a list, and repeat functions capable of generating iterators. These ideas have motivated me to create a brand-new language.
Before the creation of a new language, I made guidelines described below:

- **Inherit Familiar Syntax**

  I don't think it's a good idea to bother creating an original syntax if it has same effects as that of existing languages. I decided to follow other popular languages as much as possible when I need to make syntax and name variables and functions. In fact, as the new language uses a pair of curly brackets to embrace a block, an overwhole appearance of the code may look like one in C or Java.

- **Be Practical**

  Any programming languages are expected to solve problems that actually exist around us. For such purposes, capabilities of reading/writing files and processing text data are still important. However, these days, having such functions is far from enough because various technologies like Internet, graphic image files, database and GUI become so common that most users of computer expect any programs to be capable of handling them. To be practical, the new language should be shipped with these capabilities as standard.

Following these guidelines, I've developed a script language Gura that comes with functions and methods that are aware of Implicit Mapping policy, and published its first version on March 15, 2011.

I found it amazing to develop a new programming language since creating a language doesn't instantly mean that the creater is an expert programmer of it. This may be similar to a situation that you try to come up with an idea of a new game: even if you make its rule, you have to actually play it to know tricks and tactics so that you get a victory on the rule. I also had to create and try a lot of scripts for myself to know how to make programs of Gura. Throughout the process, I've learned that Gura's various features including Implicit Mapping are really practical in actual programming fields.
As one user, I can recommend this script language for you.
Yutaka Saito
March 6th, 2014

Chapter 2

Launch Program

2.1 Program Files

For Windows, there are two types of program files to launch Gura interpreter: `gura.exe` and `guraw.exe` . `guraw.exe` doesn't show command prompt window and you can use it to run a script with graphical user interface.
For Linux, an executable binary `gura` is the interpreter program.

2.2 Interactive Mode

When you run `gura.exe` or `gura` with no script file specified in the argument, it will enter an interactive mode that waits for user inputs.

```
Gura x.x.x [xxxxxxxxxx, xxx xx xxxx] Copyright (C) 2011-2015 ypsitau
>>>
```

When you input a script followed by an enter key after a prompt `>>>` , it will evaluate the script and show its result.

```
>>> 3 + 4
7
>>> println('Hello world')
Hello world
```

To quit the interpreter, enter `Ctrl+C` from keyboard or execute a script `sys.exit()` .
If you want to get a help of a function, put ” ” before the function name and hit the enter key in the prompt. Below is an example to show a help of function `println()` :

```
>>> ~println
println(values*):map:void

Prints out values and a line-break to the standard output.
```

(this feature will be provided in v0.6.2 or later)

2.3 Run Script File

You can run a script file by specifying it as an argument for Gura interpreter program.

```
$ gura hello.gura
```

A Gura script file should have a suffix `.gura` or `.guraw`, where `.gura` is for command-line scripts and `.guraw` for ones with GUI. In Windows environment, the suffix `.gura` is associated with the program `gura.exe` and `.guraw` with `guraw.exe`.
As a Gura script is a plain text file, you can use any of your favorite editor to create it. The code below shows the content of `hello.gura` script.

```
println('Hello World')
```

If you want to make a script executable on UNIX-like OS such as Linux, it might be a good idea to add shebang at the top of the script file. Below is a Hello World script with a shebang.

```
#!/usr/bin/env gura
println('Hello World')
```

If you want to use shebang, be careful to save the script file with each line ended with LF code. This is to avoid an error caused by specifications of shell programs, not of Gura.
If a script file contains non-ASCII characters like Japanese and Chinese, you should save it in UTF-8 character code, which is a default code set for the interpreter.
When you need to save the file in other character codes, there are two ways to parse it properly. One is to specify `-d` option in command line as following.

```
$ gura -d shift_jis foo.gura
```

Another one is to describe a magic comment that specifies a character encoding at top of the script but after shebang if exists.

```
#!/usr/bin/env gura
# coding: shift_jis
println('... string that may contain characters in Shift-JIS ...')
```

A magic comment has a format like `coding: XXXXXX` where `XXXXXX` indicates what encoding the parser is to use. It can be detected when it appears at the first or second line of a script and is described as a line comment that begins with `#` or `//`.
The following format is acceptable too.

```
#!/usr/bin/env gura
# -*- coding: shift_jis -*-
```

This is good to make Emacs determine what character encoding it should choose for editing.
Available encoding names are described in Chapter.X.

2.4 Composite File

It often happens that an application consists of multiple script files and other resources such as image files. Consider an application that has following files:

```
foo.gura
utils.gura
message.txt
image.png
```

Assume that `foo.gura` is a main script that imports `utils.gura` and reads files `message.txt` and `image.png`.
It could be bothersome to treat these files separately especially when you try to distribute them. For such a case, Gura has a feature that can run a ZIP archive file containing scripts and any other files. Such a file is called Composite File and can be created by ordinary archiving commands like following:

```
$ zip foo.zip foo.gura utils.gura message.txt image.png
$ mv foo.zip foo.gurc
```

Then you can run it as following:

```
$ gura foo.gurc
```

A Composite File must have a suffix `.gurc` or `.gurcw` where `.gurc` is for command-line scripts and `.gurcw` for ones with GUI. These suffixes are also associated with `gura.exe` and `guraw.exe` respectively in Windows environment. A script file that has the same name with that of the Composite File except for their suffix part is recognized as a main script. The interpreter reads that file at first when given with the Composite File.
You can also use a Gura module to create a Composite File. Below is a script to create a Composite File `foo.gurc` .

```
import(gurcbuild)
gurcbuild.build(['foo.gura', 'utils.gura', 'message.txt', 'image.png'])
```

This script is more useful than using other archiving tools to create a Composite File because the script will embed shebang comment at top of the file and put executable attribute to it so that the created one can run independently under Linux environment.

2.5 Command Line Options

Available command line options are listed below:

Option	**Explanation**
`-h`	Prints a help message.
`-t`	Runs a script file specified and then enters interactive mode.
`-i module[, ...]`	Imports modules in the same way as calling `import` function in a script. You can specify more than one module names for this option by separating them with comma. Or, you can also specify the option in multiple times to import several modules.
`-I dir`	Specifies a directory in which modules are searched. You can specify the option in multiple times to add several directories for module search.
`-c cmd`	Runs `cmd` as a Gura script.
`-T template`	Runs template engine to evaluate the specified template file.
`-C dir`	Changes the current directory before running scripts.
`-d encoding`	Specifies character encoding that the parser uses to read scripts.
`-v`	Prints a version number.

2.6 System Directory

The distribution package contains the interpreter executable as well as other various files such as Gura modules and dynamic-loaded libraries. When installed, they are stored in directories that are relative to where the interpreter executable is located.
For Windows, they are stored in the following directories:

```
[install directory] -+- bin
                     +- module
                     +- include
                     +- lib
```

For Linux, they are as below:

```
[install directory] -+- bin
                     +- include
                        +- gura
                     +- lib
                        +- gura
                     +- share
                        +- gura
```

As the interpreter searches these files in directories that are relative to its own location, they are relocatable. This feature makes it easier to install different versions of Gura in a certain system.

2.7 Working Directory

When the interpret is launched, it creates a working directory if it's not exist, which Gura applications can use to store working files.
The directory name comes like below where GURA_VERSION is the Gura's version.
For Windows:

```
%LOCALAPPDATA%\Gura\GURA_VERSION
```

For Linux:

```
$HOME/.gura/GURA_VERSION
```

A variable `sys.localdir` points to the directory.

Chapter 3

Syntax

3.1 Overview

Gura's parser consits of two parts: token parser and syntax parser.
The token parser is responsible of splitting a text into tokens that represent atomic factors in a program. Section "Token" explains about how the tokens should be described in a code and about their traits.
The syntax parser will build up expressions from tokens following Gura's syntax rule. While a program is running, the interpreter reads the expressions and executes them along with Environment status. Section "Expression" explains about what tokens compose each expression and about relationship between expressions using class diagrams.

3.2 Token

3.2.1 Symbol

A symbol is used as a name of variable, function, symbol, type name, attribute and suffix.
A symbol starts with a UTF-8 leading byte or one of following characters:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
_ $ @
```

and is followed by UTF-8 leading or trailing byte or characters shown below:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
_ $ @
0 1 2 3 4 5 6 7 8 9
```

Here are some valid symbols:

```
foo
test_result
$foo
@bar@
test_1_var
```

Special symbols:

```
%

+  *  ?  -
```

3.2.2 Number Literal

A decimal number is the most common number literal.

```
0 1234 999999
```

A floating-point number that sometimes comes with an exponential expression is also acceptable.

```
3.14  10.  .001  1e100  3.14e-10  0e0
```

A sequence of characters that starts with `0b` or `0B` and contains `0` or `1` represents a binary number.

```
0b01010101
```

A sequence of characters that starts with `0` and contains digit characters between `0` and `7` represents an octal number.

```
01234567
```

A sequence of characters that starts with `0x` or `0X` and contains digit characters and alphabet characters between `a` and `f` or between `A` and `F` represents a hexadecimal number.

```
0x7feaa00
0x7FEAA00
```

A suffix symbol can be appended after a number literal to convert it into other types rather than `number` . Two suffix symbols are available as standard.

Suffix Symbol	Function
`j`	Converts into `complex` type. An expression `3j` is equivalent with `complex(0, 3)` .
`r`	Converts into `rational` type. An expression `3r` is equivalent with `rational(3, 0)` .

Importing modules may add other suffix symbols. For instance, importing a module named `gmp` , which calculates numbers in arbitrary precision, would add a suffix `L` that represents numbers that may consist of many digits.
You can also add your own suffix symbols by using Suffix Manager that is responsible for managing suffix symbols and their associated functions.

3.2.3 String Literal

A string literal is a sequence of characters surrounded by a pair of single or double quotations. A string surrounded by single quotations can contain double quotation characters in its body while a string with double quotations can have single quotation characters inside.

```
'Hello "World"'
"Hello 'World'"
```

Although you can choose one of them case by case, single quotation is more preferable in general. Within a string literal, you can use following escape characters.

Escape Character	**Note**
`\\`	back slash
`\'`	single quotation
`\"`	double quotation
`\a`	bell
`\b`	back space
`\f`	page feed
`\r`	carriage return
`\n`	line feed
`\t`	tab
`\v`	vertical tab
`\0`	null character
`\x`*hh*	any byte of character code *hh* in hexadecimal
`\u`*hhhh*	Unicode character at codepoint *hhhh* in hexadecimal
`\U`*hhhhhhhh*	Unicode character at codepoint *hhhhhhhh* in hexadecimal

If a string is prefixed by `r` , a back slash is treated as a normal character, not one for escaping. This feature is convenient to describe a path name in Windows style and a regular expression that often uses back slash as a metacharacter.

```
r'C:\users\foo\bar.txt'
r'(\w+) (\d+):(\d+):(\d)'
```

You can describe a string containing multiple lines by surrounding it with a triple sequence of single or double quotations.

```
'''
ABCD
EFGH
IJKL
'''

"""
ABCD
EFGH
IJKL
"""
```

These codes are equivalent to an expression `'\nABCD\nEFGH\nIJKL\n'` , which contains a line-feed character at the beginning. If you want to eliminate the first line-feed, you need to begin the string body right after the starting quotations or put a back slash at that position followed by a line feed since a back slash placed at end of a line results in an elimination of the tailing line feed.

```
'''ABCD
EFGH
IJKL
'''

'''\
ABCD
EFGH
IJKL
'''
```

Both of the examples above have the same result `'ABCD\nEFGH\nIJKL\n'` .
You can also specify `r` prefix for the multi-lined string so that it can contain back slash characters without escaping. In this case, you cannot use the second example shown above because a back slash doesn't work to eliminate a line feed. For such a case, a prefix `R` is useful, which eliminates a line feed that appears right after the starting quotation.

```
R'''
ABCD
EFGH
IJKL
'''
```

This is parsed as `'ABCD\nEFGH\nIJKL\n'` .
The prefix `R` also removes indentation characters that appear at each line.

```
if (flag) {
    print(R'''
    ABCD
    EFGH
    IJKL
    ''')
}
```

Assuming that there are four spaces before the expression `print(R'''` , the parser would remove four spaces at top of each line within the multi-lined string. This feature helps you describe multi-lined strings in indented blocks without disarranging the appearance.
A string literal prefixed by `b` would be treated as a sequence of binary data instead of character code.
A string literal can also be appended by a suffix symbol that has been registered in Suffix Manager. There's no built-in suffix for string literals.

3.2.4 Operator

An Operator takes one or two values as its inputs and returns a calculation result. It's categorized in the following types:

- **Prefixed Unary Operator** takes an input value specified after it.

  ```
  +  -  ~  !
  ```

  An example code of a Prefixed Unary Operator comes like "`+x` ".

- **Suffixed Unary Operator** takes an input value specified before it.

  ```
  ?  ..
  ```

  An example code of a Suffixed Unary Operator comes like "`x?` ".

- **Binary Operator** takes two input values specified on both sides of them.

  ```
  +  -  *  /  %  **  ==  !=  >  <  >=  <=  <=>
  in  &  |  ^  <<  >>  ||  &&  ..  =>
  ```

  An example code of a Binary Operator comes like "`x + y` ".

See section Operator for more detail.

3.2.5 Bracket

Multiple expressions can be grouped by surronding them with a pair of brackets. There are three types of brackets as listed below.

- **Square bracket**: `[A, B, C]`

  When it appears right after an expression that has a value as a result of evaluation, it works as an indexer that allows indexing access in the preceding value.

  ```
  x[3]  foo['key']
  ```

  Otherwise, it forms a list of expressions that is set to create a `list` instance after evaluation.

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

- **Parenthesis**: `(A, B, C)`

  When it appears right after an expression that has a value as a result of evaluation, it's used as an argument list to evaluate the preceding value as a callable.

  ```
  f(1, 2, 3)
  ```

  Otherwise, it forms a list of expressions that is set to create an `iterator` instance after evaluation.

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

- **Curly bracket**: {`A, B, C`}

  It forms a list of expressions called Block. In general, a Block is used as a body for function assignment or provides a procedual part in calling a function.

  ```
  f() = { println('hello') }
  ```

- **Vertical Bar**: `|A, B, C|`

  This only appears right after opening bracket of Block and is called Block Parameter.

  ```
  repeat (3) {|i| println(i)}
  ```

  If an element contains an operator "`|` " in it, it must be embraced by parentheses to avoid the parser from mistaking the operator as Block Parameter's terminater.

  ```
  |(a | b), c, d|
  ```

Expressions within brackets can be separated by a comma character or a line feed. The following two codes have the same result.

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

[1
2
3
4
]
```

3.2.6 Back Quote

A symbol preceded by a back quote creates an instance of `symbol` data type.

```
`foo  `bar
```

Each values of `symbol` data type has a unique number that is assigned at parsing phase, which enables quick identification between them.
Any other expressions that have a back quote appended ahead create an instance of `expr` data type.

```
`(a + b)  `func()
```

As an `expr` instance can hold any code without any evaluation, it can be used to pass a procedure itself to a function as one of the arguments.

3.2.7 Comment

There are two types of comments: line comment and block comment.
A line comment begins with a marker `#` or `//` and lasts until end of the line.

```
# this is a comment

// and this is too

x = 10 // comment after code
```

A block comment begins with a marker `/*` and ends with `*/`. It can contain multiple lines and even other block comments nested as long as pairs of the comment markers are matched. Following are valid examples of block comment.

```
/* block comment */

/*
block comment
*/

/* /* /* nested comment */ */ */
```

3.3 Expression

3.3.1 Class Diagram of Expression

The following figure shows a hierarchy of expressions.

```
Expr <-+- Value
       +- Identifier
       +- Suffixed
       +- Unary <-----+- UnaryOp
       |              `- Quote
       +- Binary <----+- BinaryOp
       |              +- Assign
       |              `- Member
       +- Collector <-+- Lister
       |              +- Iterer
       |              +- Block
       |              `- Root
```

```
      `- Compound <--+- Indexer
                     `- Caller
```

All the expressions are derived from `Expr` that is an abstract expression.
Other abstract expressions, Unary, Binary, Collector and Compound, don't appear in the actual code either, but just provide common functions for their derivations.

3.3.2 Value

A `Value` expression holds a value of `number`, `string`, `binary` type.
Class diagram is:

```
+----------------------------------+
|              Value               |
|----------------------------------|
|- value: number, string or binary |
+----------------------------------+
```

Those types of value are described with string literal, number literal and b-prefixed string literal in a script respectively.
Consider the following expressions:

- `3.141`

  It has a value of `number` type.

- `'hello'`

  It has a value of `string` type.

- `b'\x00\x01\x02\0x03'`

  It has a value of `binary` type.

3.3.3 Identifier

An `Identifier` expression consists of a symbol and zero or more attributes trailing after it. An `Identifer` expression can also contain attributes, where an attribute is a symbol preceded by a colon character. One or more attributes can be described after a symbol of the `Identifier`.
Class diagram is:

```
+----------------------------+
|         Identifier         |
|----------------------------|
|- symbol: symbol            |
|- attrs: set of symbol      |
|- attrFront: list of symbol |
|- attrsOpt: set of symbol   |
+----------------------------+
```

Consider the following expressions:

- `foo`

  It has a symbol `foo`. Other elements are all blank.

- `foo:attr1:attr2`

  It has a symbol `foo` and has symbols `attr1` and `attr2` as its attrs element.

3.3.4 Suffixed

A `Suffixed` expression has a suffix symbol and a preceding literal of string or number.
Class diagram is:

```
+---------------------+
|      Suffixed       |
|---------------------|
|- body: string       |
|- suffix: symbol     |
+---------------------+
```

Even with a number literal, the body element is stored as a string.
Consider the following expressions:

- `123.45foo`

  It has a string `'123.45'` as its body and a symbol `foo` as its suffix.

- `'hello world'bar`

  It has a string `'hello world'` as its body and a symbol `bar` as its suffix.

3.3.5 UnaryOp

A `UnaryOp` expression consists of a unary operator and a child expression on which the operator is applied.
Class diagram is:

```
+---------------------+         +----------------+
|       UnaryOp       |   child |      Expr      |
|---------------------*---------+----------------|
|- operator: operator |         |                |
+---------------------+         +----------------+
```

Consider the following expression:

- `-foo`

  It has an operator "`-`" and owns an Identifer expression as its child.

3.3.6 Quote

A `Quote` expression consists of a back quotation and a child expression that is to be quoted by it.
Class diagram is:

```
+---------------------+         +----------------+
|        Quote        |   child |      Expr      |
|---------------------*---------+----------------|
|                     |         |                |
+---------------------+         +----------------+
```

Consider the following expression:

- `` `12345 ``

  It owns an Value expression with a number value as its child.

3.3.7 BinaryOp

A `BinaryOp` expression consists of a binary operator and two child expressions on which the operator is applied.
Class diagram is:

```
                                   +-----------------+
                              left |      Expr       |
                          +--------+-----------------|
+---------------------+   |        |                 |
|       BinaryOp      *---+        +-----------------+
|---------------------|
|- operator: operator *---+        +-----------------+
+---------------------+   |  right |      Expr       |
                          +--------+-----------------|
                                   |                 |
                                   +-----------------+
```

Consider the following expression:

- `x + y`

  It has an operator "`+`" and owns an Identifer expression `x` as its left and also an Identifier expression `y` as its right.

3.3.8 Assign

An `Assign` expression consists of an equal symbol, an expression on the left side that is a target of the assignment and an expression on the right side that is an assignment source. An expresion that can be specified on the left is one of `Identifer`, `Lister`, `Indexer`, `Caller` and `Member`.
Class diagram is:

```
                                   +-----------------+
                              left |      Expr       |
                          +--------+-----------------|
+---------------------+   |        |                 |
|       Assign        *---+        +-----------------+
|---------------------|
|- operator: operator *---+        +-----------------+
+---------------------+   |  right |      Expr       |
                          +--------+-----------------|
                                   |                 |
                                   +-----------------+
```

The `Assign` expression also has an operator that is to be applied before assignment. For a normal assignment, that is set to invalid operator.
Consider the following expressions:

- `x = y`

  It owns an Identifer expression `x` as its left and also an Identifier expression `y` as its right. The operator is set to invalid.

- `x += y`

  It owns an Identifer expression `x` as its left and also an Identifier expression `y` as its right. It also has an operator "`+`".

3.3.9 Member

A `Member` expression is responsible for accessing variables in a property owner like instance, class and module. Below are available Member accessors.
Class diagram is:

```
                                   +----------------+
                             left  |      Expr      |
                          +--------+----------------|
+---------------------+   |        |                |
|       Member        *---+        +----------------+
|---------------------|
|- mode: mode         *---+        +----------------+
+---------------------+   |  right |      Expr      |
                          +--------+----------------|
                                   |                |
                                   +----------------+
```

Consider the following expression:

- `x.y`

  It has a `normal` mode and owns an Identifer expression `x` as its left and also an Identifier expression `y` as its right.

A Member expression may take one of the following modes.

Expression	Mode
`x.y`	`normal`
`x::y`	`map-to-list`
`x:*y`	`map-to-iterator`
`x:&y`	`map-along`

Mode `normal` takes a reference to a property owner as its left's result value.
Others are for what is called Member Mapping and take a list or an iterator as its left's result value, each of which expressions is a reference to a property owner.

3.3.10 Lister

A `Lister` expression is a series of element expressions embraced by a pair of square brackets.
Class diagram is:

```
+---------------------+          +----------------+
|        Lister       | elements |      Expr      |
|---------------------*----------+----------------|
|                     |        * |                |
+---------------------+          +----------------+
```

Consider the following expression:

- `[x, y, z]`

  It contains three Identifier expressions `x` , `y` and `z` as its elements.

3.3.11 Iterer

An `Iterer` expression is a series of element expressions embraced by a pair of parentheses. Class diagram is:

```
+---------------------+           +----------------+
|        Iterer       |  elements |      Expr      |
|---------------------*-----------+----------------|
|                     |         * |                |
+---------------------+           +----------------+
```

Consider the following expression:

- `(x, y, z)`

  It contains three Identifier expressions `x` , `y` and `z` as its elements.

3.3.12 Block

A `Block` expression is a series of element expressions embraced by a pair of curly brackets. Class diagram is:

```
+---------------------+            +----------------+
|        Block        |   elements |      Expr      |
|---------------------*------------+----------------|
|                     |          * |                |
+---------------*-----+            +----------------+
                |
                |                  +----------------+
                | block-parameters |      Expr      |
                +------------------+----------------|
                                 * |                |
                                   +----------------+
```

The `Block` expression also has a list of block-parameters that appear in a code embraced by a pair of vertical bars right after block's opening curly bracket.
Consider the following expression:

- `{x, y, z}`

  It contains three Identifier expressions `x` , `y` and `z` as its elements.

- `{|a, b, c| x, y, z}`

  It contains three Identifier expressions `x` , `y` and `z` as its elements. It also owns Identifier expressions `a` , `b` and `c` as its block-parameters.

3.3.13 Root

A `Root` expression represents a series of element expressions that appear in the top sequence. Class diagram is:

```
+---------------------+           +----------------+
|        Root         |  elements |      Expr      |
|---------------------*-----------+----------------|
|                     |         * |                |
+---------------------+           +----------------+
```

Consider the following expression:

- `x, y, z`

  It contains three Identifier expressions `x` , `y` and `z` as its elements.

3.3.14 Indexer

An `Indexer` expression consists of a car element and a series of expressions that represent indices.
Class diagram is:

```
                                                  +----------------+
                                              car |      Expr      |
                                      +-----------+----------------|
+---------------------+               |           |                |
|       Indexer       *----+          +----------------+
|---------------------|
|                     *----+          +----------------+
+---------------------+    |  indices |      Expr      |
                           +-----------+----------------|
                                    * |                |
                                      +----------------+
```

Consider the following expression:

- `a[x, y, z]`

  It owns an Identifier expression `a` as its car element and three Identifier expressions `x` , `y` and `z` as its indices.

3.3.15 Caller

A `Caller` expression consists of a car element and a series of expressions that represent arguments. It may optionally own a Block expression if a block is specified and may own a Caller expression as its trailer if that is described in a leader-trailer syntax.
As with an `Identifier` expression, a `Caller` expression can also have attributes. They can be described just after a closing parenthesis of an argument list.
Class diagram is:

```
                                                 +----------------+
+-----------------------------+              car |      Expr      |
|           Caller            |    +-----------+----------------|
|-----------------------------|    |           |                |
|- symbol: symbol             *----+           +----------------+
|- attrs: set of symbol       |
|- attrFront: list of symbol *----+           +----------------+
|- attrsOpt: set of symbol    |    | arguments |      Expr      |
+--------*--------------*----+    +-----------+----------------|
         |              |                   * |                |
         |              +----+                +----------------+
   block | 0..1      trailer | 0..1
+--------+-------+  +--------+-------+
|      Block     |  |     Caller     |
+----------------|  +----------------|
|                |  |                |
+----------------+  +----------------+
```

Consider the following expressions:

- `a(x, y, z)`

  It owns an Identifier expression `a` as its car element and three Identifier expressions `x` , `y` and `z` as its arguments. Its block and trailer elements are both invalid.

- `a()`

  It owns an Identifier expression `a` as its car element. Its arguments is blank.

- `a(x, y, z) {xx, yy, zz}`

  It owns an Identifier expression `a` as its car element and three Identifier expressions `x` , `y` and `z` as its arguments. It also owns a Block expression as its block element.

If two or more callers are described in the same line, they have a leader-trailer relationship each other, in which the preceding caller is dubbed a leader and following one a trailer. A caller that acts as a leader is the owner of its trailing caller.
Consider the following expressions:

- `a() b()`

  The Caller expression `a()` owns a Caller expression of `b()` as its trailer.

- `a() b() c()`

  The Caller expression `a()` owns a Caller expression of `b()` as its trailer, and the Caller expression `b()` owns the Caller expression `c()` as well.

The parser determines whether a following Caller is a trailer by checking if top of the following Caller is described in the same line as a closing parenthesis of the preceding one. It means that the example below is a valid leader-trailer form.

```
a(
) b(
)
```

If the preceding Caller has a block, the closing curly bracket must be in the same line as top of the following one like below.

```
a() {
} b(
)
```

Chapter 4

Data Type

4.1 Overview

A value has a corresponding Data Type that defines its behavior and properties.
Each Data Type is bound with a type name, which usually appears in argument list of function call.
Name spaces for Data Type are completely isolated from those for variable and function names.
As each Data Type has a one-to-one relationship with a corresponding Class, those terms have almost the same meaning within documents in many cases.
Data types are categorized into two types: **Primitive Data Type** and **Object Data Type**.
A value of Primitive Data Type holds its content in as small memory as possible. It doesn't include any Environment in it and doesn't have any methods with side effects. Among them are `nil` , `boolean` , `complex` , `number` , `rational` , `string` and `symbol` types.
A value of Object Data Type owns Object data that is a sort of Environment, which allows operations with side effects. Most Data Types except for what are picked up as Primitive Data Types above belong to this.

4.2 Primitive Data Types

Below is a list of Primitve Data Types, which also shows one of the typical ways to instantiate values of each type.

- `nil`

  A value of `nil` type is used to indicate an invalid result or status. It is often used as a returned value of a function when it fails its expected work. A variable `nil` has a value of nil type.

  ```
  nil
  ```

  Since `nil` is the only instance of `nil` type, the term `nil` can both mean the name of the value and its type.

- `boolean`

  Values of `boolean` type are used to determine whether something is in a true or a false state. Variables named `true` and `false` are assigned with a true value and a false value respectively.

  ```
  true  false
  ```

  In a function like `if` having arguments to check true/false condition and in a logical calculation, `false` and `nil` only are determined as a false state while other values are treated as a true state. Note that a zero value of `number` type is recognized as a true, not a false.

- `complex`

  A number literal suffixed by `j` instantiates a value of `complex` type that represents a complex number.

  ```
  3.14j  1000j  1e3j
  ```

  See chapter Mathematic Functions for more detail.

- `number`

  A number literal without any suffix instantiates a value of `number` type.

  ```
  3.14  1000  1e3  0xaabb
  ```

- `rational`

  A number literal suffixed by `r` instantiates a value of `rational` type that represents a rational number.

  ```
  3r  123r
  ```

  See chapter Mathematic Functions for more detail.

- `string`

  A string literal without any suffix instantiates a value of `string` type.

  ```
  'hello world'

  R'''
  message text
  '''
  ```

- `symbol`

  An identifier preceded by a back quote instantiates a value of `symbol` type.

  ```
  `foo  `bar
  ```

4.3 Object Data Types Frequently Used

4.3.1 List

If one or more elements are surrounced by a pair of square brackets, it would instantiate a value of `list` type. Any type of value can be an element of lists.

```
[3, 1, 4, 1, 5, 9]
['hello', 'world', 3, 4, 5]
```

4.3.2 Iterator

If one or more elements are surrounced by a pair of parentheses, it would instantiate a value of `iterator` type. Any type of value can be an element of iterators.

```
(3, 1, 4, 1, 5, 9)
('hello', 'world', 3, 4, 5)
```

To create an iterator that contains only one element, be sure to put a comma afther the element like following:

```
(3,)
```

An expression `(3)` is recognized as an ordinary value of number `3`.
Operator `..` creates an iterator that generates a sequence of numbers. An expression `x..y` creates an iterator that generates a sequence starting from `x` and being increased by one until `y`.

```
1..10
```

An expression `x..` creates an iterator that generates a sequence starting from `x` and being increased by one indefinitely.

```
1..
```

Lists and iterators are convertible to each other. For instance, a list can be converted to an iterator by using `list#each` method like following.

```
[3, 1, 4, 1, 5, 9].each()
```

An iterator can be converted to an list by surrounding it with square brackets.

```
[1..10]
```

In many cases, an iterator is generated as a value returned from a function, which represents a series of multiple results. The most commonly used function may be `readlines`, which creates an iterator that reads a stream and returns strings splitted by line.

4.3.3 Dictionary

`dict` is a dictionary that contains key-value pairs as its elements where a key is one of `number`, `string` or `symbol` and a value is of any type.
You can create a dictionary by surrounding key-value pairs by `%{` and `}`.
There are several ways to describe the pairs. The most recommended way is to use `=>` operator between each key and value like following.

```
%{
    `symbol1 => 'value 1'
    `symbol2 => 'value 2'
    `symbol3 => 'value 3'
}
```

A pair can also be described as a list containing a key and a value.

```
%{
    [`symbol1, 'value 1']
    [`symbol2, 'value 2']
    [`symbol3, 'value 3']
}
```

You can also describe keys and values alternately in one-dimentional format.

```
%{
    `symbol1, 'value 1'
    `symbol2, 'value 2'
    `symbol3, 'value 3'
}
```

4.3.4 Expression

Any expression preceded by a back quote instantiates a value of `expr` type.

```
`(x + y)  `func(x)  `{ println('hello'), x += 1 }
```

4.3.5 Binary

A string literal preceded by `b` instantiates a value of `binary` type.

```
b'\x00\x01\0x02\0x03'
```

Chapter 5

Operator

5.1 Overview

There are three types of Operators.

- **Prefixed Unary Operator** takes an input value specified after it.
- **Suffixed Unary Operator** takes an input value specified before it.
- **Binary Operator** takes two input values specified on both sides of them.

An Operator has a table of procedures that are indexed by Data Types of given values, one Data Type indexing for Unary Operators and two Data Types for Binary Operators. For instance, operator `+` has a procedure to calculate between values of `number` and `number` and also a procecure beween values of `string` and `string` . These procedures are isolated each other as long as combination of the given Data Types is different.
Users can overload operators' procedures through `operator` instance. If combination of Data Types of the overloading procedure is the same as that of existing one, it would override the registered procedure. Otherwise, it would add a new procedure to the operator.

5.2 Precedence

The following table shows operators' precedence order from the lowest to the highest.

Precedence	**Operators**
Lower	`=>`
	`||`
	`&&`
	`!`
	`in`
	`< > <= >= <=> == !=`
	`..`
	`|`
	`^`
	`&`
	`<< >>`
	`+ -`
	`* / % ?`
Higher	`**`

5.3 Calculation Operators

Basically, Operators are used for mathematical and logical calculation. This subsection explains such functions of operators.

5.3.1 Prefixed Unary Operators

Operation `+x` returns the value of `x` itself.

Operation	Result Data Type
`+number`	`number`
`+complex`	`complex`
`+rational`	`rational`
`+matrix`	`matrix`
`+timedelta`	`timedelta`

Operation `-x` returns a negaive value of `x` .

Operation	Result Data Type
`-number`	`number`
`-complex`	`complex`
`-rational`	`rational`
`-matrix`	`matrix`
`-timedelta`	`timedelta`

Operation `x` returns a bit-inverted value of `x` .

Operation	Result Data Type
`number`	`number`

Operation `!x` returns a logically inverted value of `x` after evaluating it as a boolean value.

Operation	Result Data Type
`!any`	`boolean`

5.3.2 Suffixed Unary Operators

Operation `x..` returns an infinite iterator that starts from `x` and is increased by one.

Operation	Result Data Type
`number..`	`iterator`

Operation `x?` returns `false` if `x` is `false` or `nil` , and `true` otherwise. This operator is not affected by Implicit Mapping and returns `true` if `x` is of `list` or `iterator` type.

Operation	Result Data Type
`any?`	`boolean`

5.3.3 Binary Operators

Operation `x + y` returns an added result of `x` and `y`.

Operation	Result Data Type
`number + number`	`number`
`number + complex`	`complex`
`number + rational`	`rational`
`complex + number`	`complex`
`complex + complex`	`complex`
`complex + rational`	(error)
`rational + number`	`rational`
`rational + complex`	(error)
`rational + rational`	`rational`
`matrix + matrix`	`matrix`
`datetime + timedelta`	`datetime`
`timedelta + datetime`	`datetime`
`timedelta + timedelta`	`timedelta`

If `x` and `y` are of `string` or `binary` type, Operation `x + y` returns concatenated result of `x` and `y`.

Operation	Result Data Type
`string + string`	`string`
`binary + binary`	`binary`
`string + binary`	`binary`
`binary + string`	`binary`
`string + any`	`string` ('any' will be converted to 'string' before concatenation)
`any + string`	`string` ('any' will be converted to 'string' before concatenation)

Operation `x - y` returns a subtracted result of `x` and `y`.

Operation	Result Data Type
`number - number`	`number`
`number - complex`	`complex`
`number - rational`	`rational`
`complex - number`	`complex`
`complex - complex`	`complex`
`complex - rational`	(error)
`rational - number`	`rational`
`rational - complex`	(error)
`rational - rational`	`rational`
`matrix - matrix`	`matrix`
`datetime - timedelta`	`datetime`
`datetime - datetime`	`timedelta`
`timedelta - timedelta`	`timedelta`

Operation `x * y` returns a multiplied result of `x` and `y`.

Operation	Result Data Type
`number * number`	`number`
`number * complex`	`complex`
`number * rational`	`rational`
`complex * number`	`complex`
`complex * complex`	`complex`
`complex * rational`	(error)
`rational * number`	`rational`
`rational * complex`	(error)
`rational * rational`	`rational`
`matrix * matrix`	`matrix`
`matrix * list`	`list`
`list * matrix`	`list`
`timedelta * number`	`timedelta`
`number * timedelta`	`timedelta`

Applying `*` operator between `string` /`binary` and `number` will join the `string` /`binary` for `number` times.

Operation	Result Data Type
`string * number`	`string`
`number * string`	`string`
`binary * number`	`binary`
`number * binary`	`binary`

Operation `x / y` returns a divided result of `x` and `y` .

Operation	Result Data Type
`number / number`	`number`
`number / complex`	`complex`
`number / rational`	`rational`
`complex / number`	`complex`
`complex / complex`	`complex`
`complex / rational`	(error)
`rational / number`	`rational`
`rational / complex`	(error)
`rational / rational`	`rational`
`matrix / matrix`	`matrix`

Operation `x % y` returns a remainder after dividing `x` by `y` .

Operation	Result Data Type
`number % number`	`number`

Operation `x ** y` returns a powered result of `x` and `y` .

Operation	Result Data Type
`number ** number`	`number`
`number ** complex`	`complex`
`complex ** number`	`complex`
`complex ** complex`	`complex`

Operation `x == y` returns `true` when `x` equals to `y` , and `false` otherwise.

Operation	Result Data Type
`any == any`	`boolean`

Operation `x < y` returns `true` when `x` is less than `y` , and `false` otherwise.

Operation	Result Data Type
`any any`	`boolean`

Operation `x > y` returns `true` when `x` is greater than `y` , and `false` otherwise.

Operation	Result Data Type
`any > any`	`boolean`

Operation `x <= y` returns `true` when `x` is less than or equal to `y` , and `false` otherwise.

Operation	Result Data Type
`any = any`	`boolean`

Operation `x >= y` returns `true` when `x` is greater than or equal to `y` , and `false` otherwise.

Operation	Result Data Type
`any >= any`	`boolean`

Operation `x <=> y` returns `0` when `x` is equal to `y` , `-1` when `x` is less than `y` and `1` when `x` is greater than `y` .

Operation	Result Data Type
`any <=> any`	`number`

Operation `x in y` checks if `x` is contained in `y` .
When Operator `in` takes a value of any type other than `list` and `iterator` at its left, it will check if the value is contained in the container specified at its right. If the right value is not of `list` or `iterator` , it would act in the same way as Operator `==` .

Operation	Result Data Type
`any in list`	`boolean`
`any in iterator`	`boolean`
`any in any`	`boolean`

When Operator `in` takes a value of `list` or `iterator` type at its left, it will check if each value of the container's element is contained in the container specified at its right, and return a list of `boolean` indicating the result of each containing check.

Operation	Result Data Type
`list in list`	`list`
`list in iterator`	`list`
`list in any`	`list`
`iterator in list`	`list`
`iterator in iterator`	`list`
`iterator in any`	`list`

When Operator `in` is used in an argument of `for()` and `cross()` function, it would work as an iterable assignment. See Chapter.8. Flow Control for detail.
Operation `x & y` returns an AND calculation result of `x` and `y` .

- If `x` and `y` are of `number` type, it calculates bitwise AND between them.
- If `x` and `y` are of `boolean` type, it calculates logical AND between them.
- If either `x` or `y` is `nil, it returns` nil.

Operation	**Result Data Type**
`number number`	`number`
`boolean boolean`	`boolean`
`nil any`	`nil`
`any nil`	`nil`

Operation `x | y` returns an OR calculation result of `x` and `y` .

- If `x` and `y` are of `number` type, it calculates bitwise OR between them.
- If `x` and `y` are of `boolean` type, it calculates logical OR between them.
- If either `x` or `y` is `nil, it returns one of their values that is not` nil.

<table>
<tr><th>Operation</th><th>Result Data Type</th></tr>
<tr><td><code>number | number</code></td><td><code>number</code></td></tr>
<tr><td><code>boolean | boolean</code></td><td><code>boolean</code></td></tr>
<tr><td><code>nil | any</code></td><td><code>nil</code></td></tr>
<tr><td><code>any | nil</code></td><td><code>nil</code></td></tr>
</table>

Operation `x ∧ y` returns a XOR calculation result of `x` and `y` .

- If `x` and `y` are of `number` type, it calculates bitwise XOR between them.
- If `x` and `y` are of `boolean` type, it calculates logical XOR between them.

Operation	**Result Data Type**
`number ∧ number`	`number`
`boolean ∧ boolean`	`boolean`

Operation `x << y` returns a value of `x` shifted left by `y` bits.

Operation	**Result Data Type**
`number number`	`number`

Operation `x >> y` returns a value of `x` shifted right by `y` bits.

Operation	**Result Data Type**
`number >> number`	`number`

Operation `x && y` returns a conditional AND result of `x` and `y` as described below:

- If `x` is not of `list` nor `iterator` type, it would return the value of `x` when `x` is determined as `false`, and return the value of `y` otherwise. It won't evaluate `y` when `x` comes out to be in `false` state.
- If `x` is of `list` type, it applies the above operation on each value of the list's elements and returns a list containing the results.
- If `x` is of `iterator` type, it returns an iterator that is to apply the above operation on each value of the iterator's elements.

Operation	Result Data Type
`any && any`	`any`
`list && any`	`list`
`iterator && any`	`iterator`

Operation `x || y` returns a conditional OR result of `x` and `y` as described below:

- If `x` is not of `list` nor `iterator` type, it would return the value of `x` when `x` is determined as `true`, and return the value of `y` otherwise. It won't evaluate `y` when `x` comes out to be in `true` state.
- If `x` is of `list` type, it applies the above operation on each value of the list's elements and returns a list containing the results.
- If `x` is of `iterator` type, it returns an iterator that is to apply the above operation on each value of the iterator's elements.

Operation	Result Data Type
`any || any`	`any`
`list || any`	`list`
`iterator || any`	`iterator`

Operation `x..y` creates an iterator that returns `number` value that starts from `x` and is increased by one until `y`.

Operation	Result Data Type
`number..number`	`iterator`

Operation `x => y` returns a list `[x, y]`.

Operation	Result Data Type
`number => any`	`list`
`string => any`	`list`
`symbol => any`	`list`

When Operator `=>` is used in an argument declaration of any function definition, it would work as an assignment for a default value. And, when it is used in an argument list of any function call, it would work as a named argument. See Chapter.7. Function for their detail.

5.4 Other Operators

Operation `string % any` returns a result formatted by the string containing specifiers of `printf` format. The value of `any` must be a list if more than one argument are necessary.

```
'Name: %s, Age: %d' % [name, age]
```

The code above has the same result as the following.

```
format('Name: %s, Age: %d', name, age)
```

Operation `function` $*$ `any` applies the function on `any` .
Operation `stream << any` outputs `any` to the `stream` .

```
sys.stdout << 'Hello World\n'
```

5.5 Operator Overload

You can assign your own functions to operators through `operator` instance. The example below assings `string - string` operation by using `operator#assign()` method.

```
op = operator(`-)
op.assign(`string, `string) {|x, y|
    x.replace(y, '')
}
```

After this assignment, the following code results in `'Hello, world'` .

```
'Hello, 1234world' - '1234'
```

If you want to assign a function of a unary operator, specify one argument in `operator#assign()` method like below.

```
op = operator(`-)
op.assign(`string) {|x|
    x.each().reverse().join()
}
```

Then, the code below has a result `'987654321'` .

```
-'123456789'
```

You can also override existing operators.
You can use `operator#entries()` method to get all of the functions registered in the operator.

```
op = operator(`-)
println(op.entries())
```

The method returns entries registered as binary operators. Specifying a symbol `` `unary`` as its argument would return a list of unary operators.

```
op = operator(`-)
println(op.entries(`unary))
```

Chapter 6

Environment

6.1 Overview

Environment is a container to store maps associating symbols and values and maps associating symbols and value types.
Module, Class, and Object are all inherited from Environment.
scope problems

```
x = 0
if (true) {
    x = 3
}
println(x)
```

6.2 Frame

Frame contains:

- value map
- value type map

Frame stack
Frame cache
Environment type:

- root
- local
- block
- class
- object
- lister

When the Interpreter starts, it runs with an Environment containing a frame of `root` type.

```
+--------------------+
|       root         |
+--------------------+
```

In a function call, the Interpreter creates a new Environment with cloned frames and pushes a new frame of `local` type.

```
+--------------------+
|       local        |
+--------------------+
|        root        |
+--------------------+
```

When a block is evaluated, the Interpreter creates a new Environment with cloned frames and pushes a frame of `block` type.

```
+--------------------+
|       block        |
+--------------------+
|        root        |
+--------------------+



+--------------------+
|       block        |
+--------------------+
|       local        |
+--------------------+
|       local        |
+--------------------+
|        root        |
+--------------------+


+--------------------+
|       class        |
+--------------------+
|        root        |
+--------------------+


+--------------------+
|       object       |
+--------------------+
|       class        |
+--------------------+
|        root        |
+--------------------+
```

Chapter 7

Interpreter

7.1 How Interpreter Works

The Interpreter looks up and modifies content of Environment in accordance with Expressions that has been generated by parsing source codes.
The execution of the Interpreter consists of two stages **evaluation** and **assignment**. In an evaluation stage, it looks up variables in Environment and do evaluation depending on the current expression. In an assingment stage, the Interpreter will add new variables or modify existing variables in Environment.
In the Interpreter, Evaluation stage always occurs on each Expression while Assignment stage only does when `Assign` expression is executed.

```
      +---------------+
      |  Expressions  |
      +---------------+
             | control
             V
+----------------------------+     look-up     +---------------+
|             evaluation <--|-----------------|               |
|  Interpreter      |        |   assignment    |  Environment  |
|                   `--------|---------------->|               |
+----------------------------+                 +---------------+
```

7.2 Evaluation Stage

7.2.1 Overview

This section explains how each Expression acts in the Interpreter's evaulation stage.

7.2.2 Evaluation of Value

Evaluation result of a `Value` expression will be the value that it owns in itself.
Consider the following expressions:

- `3.141`

  Returns an instance of `number` type.

- `'hello'`

  Returns an instance of `string` type

- `b'\x00\x01\x02\x03'`

  Returns an instance of `binary` type.

7.2.3 Evaluation of Identifier

An `Identifier` expression will look up a variable whose name matches the expression's symbol in an Environment and return the result value. If no variable is found, it occurs an error.
Consider the following expression:

- `foo`

  Looks up a symbol `foo` in the current Environment and returns the associated value if found. If the symbol does not exist, occurs an error.

7.2.4 Evaluation of Suffixed

A `Suffixed` expression will look up an entry in Suffix Manager that matches its suffix symbol and execute the entry with its body string.
Consider the following expressions:

- `123.45foo`

  1. Looks up a handler associated with a symbol `foo` in the Suffix Manager.
  2. If found, it evaluates the handler by passing it a string `'123.45'` and returns the result. If no handler is found, occurs an error.

- `'hello world'bar`

  1. Looks up a handler associated with a symbol `bar` in the Suffix Manager.
  2. If found, evaluates the handler by passing it a string `'hello world'` and returns the result. If no handler is found, occurs an error.

7.2.5 Evaluation of UnaryOp

A `UnaryOp` expression evaluates the child expression it owns, and then evaluate the value with its associated unary operator.
Consider the following expressions:

- `-123.45`

  1. Evaluates the child expression and gets a value `123.45` of `number` type.
  2. Looks up a unary operator function of `-` that can calculate `number` type.
  3. Evaluates the function by passing it a number `123.45` and returns the result.

7.2.6 Evaluation of Quote

A `Quote` expression

```
`X
```

7.2.7 Evaluation of BinaryOp

A `BinaryOp` expression evaluates both of the two child expressions it owns, and then evaluate the value with its associated Binary Operator.

```
X + Y
```

Binary Operator `&&` and `||` are exceptional.
With operator `&&` , it first evaluates the child expression on the left. If the value is determined as **false**, that value is the result. Otherwise, it then evaluates the child expression on the right and returns the result.
With operator `||` , it first evaluates the child expression on the left. If the value is determined as **true**, that value is the result. Otherwise, it then evaluates the child expression on the right and returns the result.

7.2.8 Evaluation of Assign

Execution of an `Assign` expression triggers Assignment Stage. See the next section.

```
X = Y
```

7.2.9 Evaluation of Member

A `Member` expression

```
X.Y
```

Class, Module and Object

7.2.10 Evaluation of Lister

A `Lister` expression

```
[A, B, C]
```

7.2.11 Evaluation of Iterer

An `Iterer` expression

```
(A, B, C)
```

7.2.12 Evaluation of Block

A `Block` expression

```
{A, B, C}
```

7.2.13 Evaluation of Root

A `Root` expression

7.2.14 Evaluation of Indexer

An `Indexer` expression

```
X[A, B, C]

x[2]

x[1, 2, 3]

x['foo']
```

How an `Indexer` expression behaves in Interpreter's evaluation and assignment stage depends on what instance the car element returns.
If car's instance is of `list` type:

- **Evaluation:** the expression seeks the list's content at specified positions by indices.
- **Assignment:** modifies or adds the list's content at specified positions by indices.

In these cases, indices values are expected to be of `number` type.
If car's instance is of `dict` type:

- **Evaluation:** the expression seeks the dictionary's content using indices as the keys.
- **Assignment:** modifies or adds the dictionary's values associated with specified keys by indices.

In these cases, indices values are expected to be of `number` , `string` or `symbol` type.

7.2.15 Evaluation of Caller

A `Caller` expression evaluates expressions listed as its arguments.

```
X(A, B, C)

f(a, b, c, d)

f(a, b):foo:bar

f {}
```

If the argument is declared as Quoted, it doesn't evaluates its argument.
How a `Caller` expression behaves in Interpreter's evaluation stage depends on what instance the car element returns.
If car's instance is of `function` type the expression calls the function with specified arguments.
If the `Caller` expression is specified as a target in Interpreter's assignment stage, it always creates `function` instance and assigns it in a specific Environment.

7.3 Assignment Stage

7.3.1 Overview

In an operation `X = Y` , the target expression `X` may be one of `Identifer` , `Lister` , `Member` , `Indexer` and `Caller` .
If the target expression is `Identifier` , `Lister` or `Member` , the source expression is evaluated at first before the result is assigned to the target.
If the target expression is `Caller` , the source expression itself is assigned to the target without any evaluation.

7.3.2 Assignment for Identifier

An assignment for an `Identifier` expression

```
X = Y
```

If a type name is specified as the `Identifier` 's attribute, the source value will be casted to the type before assignment.

```
a:number = '3'
```

This works in the same way as a data type casting in an argument list of function call. See Chapter.7. Function for more detail.

7.3.3 Assignment for Lister

When the assignment destionation is a `Lister` expression, assignment operation is applied to each expression described as its element. Elements in the `Lister` must be `Identifier` expressions.

```
[A, B, C] = X
```

If assignment source is a scalar, that value is assigned to each element.

```
[a, b, c] = 3          // a = 3, b = 3, c = 3
```

If assignment source is a list, each value in the list is assigned to each element.

```
[a, b, c] = [1, 2, 3]  // a = 1, b = 2, c = 3
```

It would be the same with an iterator.

```
[a, b, c] = (1, 2, 3)  // a = 1, b = 2, c = 3
```

If the assignment source has more elements than the destination requires, remaining elements are simply ignored. If the source has insufficient number of elements, it would occur an error.

```
[a, b, c] = [1, 2, 3, 4, 5]  // a = 1, b = 2, c = 3
[a, b, c] = [1, 2]           // error!
```

7.3.4 Assignment for Member

A `Member` expression

```
X.Y = Z
```

Class, Module and Object

```
obj.var1 = 3
obj.f(x) = { }
```

7.3.5 Assignment for Indexer

An `Indexer` expression

```
X[A] = Y

X[A, B, C] = Y

x[n] = y

x[n] = 3

x[0, 2, 5] = 3

x[0, 2, 5] = [1, 2, 3]
```

7.3.6 Assignment for Caller

A `Caller` expression

```
X(A, B, C) = Y
```

Assignments for other expressions than what are described above are invalid and occurs an error.

7.3.7 Operator before Assignment

An Assignment operator can be combined with one of several other operators.

<table>
<tr><th>Assignment Form</th><th>Equivalent Code</th></tr>
<tr><td>x += y</td><td>x = x + y</td></tr>
<tr><td>x -= y</td><td>x = x - y</td></tr>
<tr><td>x *= y</td><td>x = x * y</td></tr>
<tr><td>x /= y</td><td>x = x / y</td></tr>
<tr><td>x %= y</td><td>x = x % y</td></tr>
<tr><td>x **= y</td><td>x = x ** y</td></tr>
<tr><td>x &= y</td><td>x = x & y</td></tr>
<tr><td>x |= y</td><td>x = x | y</td></tr>
<tr><td>x ∧= y</td><td>x = x ∧ y</td></tr>
<tr><td>x = y</td><td>x = x y</td></tr>
<tr><td>x >>= y</td><td>x = x >> y</td></tr>
</table>

Chapter 8

Function

8.1 Definition and Evaluation

The figure below shows an example of function definition with each part's designation.

```
    +-- declaration
    |
----------
f(x, y, z) = printf('x = %s, y = %s, z = %s\n', x, y, z)
- -------    -------------------------------------------
|   |                             |
|   +-- argument list             +-- body
+-- identifier
```

It composes of a declaration and a body with an assignment operator, and the declaration is made up with an identifier and an argument list.
The body must be a single expression. If you want to describe more than one expression, you have to use a Block expression embracing them like following.

```
f(x, y, z) = {
    printf('x = %s, y = %s, z = %s\n', x, y, z)
}
```

After defining a function, a `function` instance is assigned in the scope environment with the identifier. If the same identifier already exists in the environment, the existing one is overwritten no matter whether it's a `function` instance or other.
You can see a function's declaration by simply printing the instance like following.

```
println(f)
```

The argument list is a list of Identifier expressions. If no argument is necessary, specify an empty list.

```
g() = { /* body */ }
```

You can evaluate a `function` instance by passing it values as its arguments. The number of passed values must be the same as that of declared arguments.

```
f(1, 2, 3)    // OK
f(1, 2, 3, 4) // Error; too many arguments
f(1, 2)       // Error; insufficient arguments
```

If the Caller doesn't pass any argument for evaluation, specify an empty list.

```
g()
```

8.2 Returned Value

An evaluation result of the last expression in a function body becomes its returned value. The function below returns a string `'hello'` as its result:

```
f() = {
    // any process
    'hello'
}
```

The function below returns a returned value of `g()` as its result:

```
f() = {
    // any process
    g()
}
```

A function can return any types of value including `list`. This feature enables a function to return more than one value.

```
f() = {
    // any process
    [3, 4, 5]
}

[a, b, c] = f()  // a = 3, b = 4, c = 5
```

You can also use a function `return()` to explicitly specify the returned value even though its use is not recommended unless you need to quit a process in the middle.

```
f() = {
    // any process
    return('hello')
}
```

An attribute `:void` indicates that the function always return `nil` no matter what value is resulted at last in the process. A call for a function below returns `nil`, not a string `'hello'`.

```
f():void = {
    'hello'
}
```

You should put `:reduce` attribute if the function is supposed to return a unchanged value. Attributes `:void` and `:reduce` have a significant effect with Implicit Mapping.

8.3 Arguments

8.3.1 Type Name Declaration

You can specify a type name by describing it as an attribute after an Identifier's symbol.

```
f(x:number) = {
    // any process
}
```

When calling a function that has arguments with type name, the Interpreter first check the data type of the given value and try to cast it into specified data type if possible. If the type doesn't match and also fails to be casted correctly, it would occur an error.
If you expect an argument to take a list, specify a pair of square brackets that has no content after an Identifer's symbol.

```
f(x[]) = {
    // any process
}
```

A type name can be described after the bracket pair.

```
f(x[]:number) = {
    // any process
}
```

In this case, the interpreter checks types of all the items in the list and applies casting on them if possible.
You can also specify how many elements the list should contain by declaring the number in the square brackets.

```
f(x[8]) = {
    // any process
}
```

In the example above, only a list that contains eight elements could be accepted and an error would occur otherwise.

8.3.2 Data Type Casting

If the data type of a value given as an argument doesn't match with that that is specified in an argument list, the value will be casted to the expected data type if possible.
For instance, a value of `string` type can be casted to `number` if the string contains a valid text of number.

```
f(n:number) = {
    // any process
}
f('100')  // string will be casted to number
```

Casting feature can also be applied to other data types. Consider the following function:

```
f(in:stream) = {
    // process to read data from in
}
```

Since it expects to take a `stream` instance as its argument, you can call it with the instance created by `open()` function like below.

```
f(open('foo.txt'))
```

Now, you can also call it much easily using a casting feature that converts from `string` to `stream`.

```
f('foo.txt')
```

If a `string` value is passed to an argument that expects a `stream` value, the Interpreter opens a stream with a path name specified by the string and creates a `stream` instance for it.
In default, casting opens a stream with reading mode. You need to append `:w` attribute in a function declaration to get a stream with writing mode.

```
g(out:stream:w) = {
    // process to write data to out
}
```

An attribute `:r` is also prepared to explicitly indicate the stream is to be opened for reading.

```
f(in:stream:r) = {
    // process to read data from in
}
```

Let's see another case of casting. Consider a function that takes a value of `image` type, which also has a casting ability from `stream` data type.

```
f(img:image) = {
    // process on img
}
```

A function `image()` takes a value of `stream` data type and creates an `image` instance. With the most explicit way, the function above can be called as below.

```
f(image(open('foo.jpg')))
```

An `image` data type can be casted from a value of `stream` type.

```
f(open('foo.jpg'))
```

Using a feature to convert `string` to `stream`, it will be rewritten like following.

```
f('foo.jpg')
```

This means that, if a function expects `image` data type, you can call it with a value of either `image`, `stream` or `string` data type.
You can find information about what data type can be casted from which data type in **Gura Library Reference**.

8.3.3 Optional Argument

You can declare an optional argument by putting `?` right after an Identifier's symbol.

```
f(x?) = { /* body */ }
```

If you want to declare a type name for an optional argument, specify it like following.

```
f(x?:number) = { /* body */ }
```

For such a function, you can call it like following.

```
f(3)
f()
```

If the Caller omits a value for an optional argument, a variable for the argument would be in undefined state. You can use `isdefined()` function to check if the variable is defined or not.

```
f(x?) = {
    if (isdefined(x)) {
        // when x is specified
    } else {
        // when x is omitted
    }
}
```

You can specify more than one optional argument. Note that it's inhibited to declare any non-optional arguments following after optional one.

```
f(x?, y?, z?) = { /* body */ }  // OK
f(x, y?, z?)  = { /* body */ }  // OK
f(x?, y?, z)  = { /* body */ }  // Error
```

8.3.4 Argument with Default Value

An argument with a default value can be declared with an operator `=>` .

```
f(x => -1) = { /* body */ }
```

If you want to declare a type name for an argument with a default value, specify it like following.

```
f(x:number => -1) = { /* body */ }
```

For such a function, you can call it like following.

```
f(3)
f()
```

If the Caller omits a value for an argument with a default value, a variable for the argument would be set to the specified default value.
You can specify more than one arguments with default value. Note that any arguments that don't have a default value can not follow after one with a default value.

```
f(x => 1, y => 2, z => 3) = { /* body */ }  // OK
f(x, y => 2, z => 3)      = { /* body */ }  // OK
f(x => 1, y => 2, z)      = { /* body */ }  // Error
```

Optional arguments and arguments with default value follow the same positioning rule each other in an argument list.

```
f(x => 1, y => 2, z?) = { /* body */ }  // OK
```

8.3.5 Variable-length Argument

You can declare a variable-length argument by putting `+` or `*` right after an Identifier's symbol.

```
f(x+) = { /* body */ }
g(x*) = { /* body */ }
```

For the first one, the Caller can call it with **one** or more values. If it doesn't specify any value for the argument, it would occur an error.

```
f(1)          // OK
f(1, 2, 3, 4) // OK
f()           // Error
```

For the second one, the Caller can call it with **zero** or more values. It can even call it without any argument.

```
g(1)          // OK
g(1, 2, 3, 4) // OK
g()           // OK
```

If you want to declare a type name for a variable-length argument, specify it like following.

```
f(x+:number) = { /* body */ }
```

The variable-length argument can only be declared once and must be placed at the last.

```
f(x, y, z+)  = { /* body */ }  // OK
f(x, y+, z+) = { /* body */ }  // Error
f(x, y+, z)  = { /* body */ }  // Error
```

In the function body, a variable of variable-length argument takes a list of values.

```
f(x*) = {
    println('number of arguments: ', x.len())
    for (item in x) {
        sum += item
    }
}
```

If there are other arguments before a variable-length one, variables of those arguments are assigned in order before the rests are stored in a variable-length argument. For instance, consider the code below:

```
f(x, y, z+) = { /* body */ }
f(1, 2, 3, 4, 5)
```

In function `f`, variables `x`, `y` and `z` are set to `1`, `2` and `[3, 4, 5]` respectively.

8.3.6 Named Argument

Consider the following function:

```
f(x, y, z) = { /* body */ }
```

To evaluate it, you can explicitly specify variable names in the argument list like below:

```
f(x => 1, y => 2, z => 3)
```

Such arguments are called named arguments, which are useful when you want to specify only relevant one among many optional arguments.
If a function declaration contains an argument suffixed by `%` , it can take a all the values of named arguments that are not assigned to other arguments.
Consider the following function:

```
f(a, b, x%) = { /* body */ }
```

When you evaluate it like below:

```
f(a => 1, b => 2, c => 3, d => 4)
```

Then, variables `a` , `b` and `x` are set to `1` , `2` and `%{c => 3, d => 4}` .

8.3.7 Argument Expansion

```
f(x*)

f(1, 2, 3, 4)


f(x%)
```

8.3.8 Quoted Argument

Sometimes, there's a need to pass a function a procedure, not an evaluated result. For such a purpose, you can use a Quote operator that creates `expr` instance from any code,
See an exmple below:

```
f(x:expr) = {
    x.eval()
}

x = `println('hello')
f(x)
```

The variable `x` that holds an `expr` instance contaning expression of `println('hello')` will be passed to function `f` as its argument, which then actually evaluates it.
Of course, you can also specify the quoted value directly in the argument.

```
f(`println('hello'))
```

There's another way to pass an expression in a function call, and that is to put a Quoted operator in an argument list of a function definition like below.

```
g(`x) = {
    x.eval()
}
```

For such a function, the Caller doesn't have to put a Quote operator for the expression that you want to pass.

```
g(println('hello'))
```

8.4 Block

A block can be seen as a special form of an argument. It appears after an argument list and contains a procedure embraced by a pair of curly braces.
A function definition with a block comes like below:

```
f() {block} = { /* body */ }
```

And you can call the function like folowing:

```
f() { /* block procedure */ }
```

The function `f` takes a function instance of a block procedure with a variable named `block` , and it can call the procedure just like an ordinary function.
Consider the following function:

```
three_times() {block} = {
    block()
    block()
    block()
}
```

Then, you can call it like following:

```
three_times() {
    println('hello world')
}
```

This results in three print-outs of `'hello world'` .
As for a function that is declared to take a mandatory block, a call without specifying a block procedure would occur an error.

```
three_times()  // Error because of lacking block
```

The block procedure can have a list of block parameters that appears right after the opening curly brace and is embraced by a pair of vertical bars.

```
f() {|/* block parameters */| /* block procedure */ }
```

A declaration of block parameters is almost the same with that of function arguments. In fact, a function created from a block procedure has an argument list that are specified as block parameters.
Consider the following function:

```
three_times() {block} = {
    block(0, 'zero')
    block(1, 'one')
    block(2, 'two')
}
```

The function provides two block parameters, values of `number` and `string` type.
The function can be called like below:

```
three_times() {|idx, str|
    println(idx, ' ', str)
}
```

The caller can also specify a value type for each block parameter just like function's arguments.

```
three_times() {|idx:number, str:string|
    println(idx, ' ', str)
}
```

The caller doesn't have to declare all the block parameters that are provided by the function if it doesn't require them. In the case of calling the above function, declaraing only one block parameter like below is permitted:

```
three_times() {|idx|
    println(idx)
}
```

And having no block parameter like below is also allowed:

```
three_times() {
    println('hello')
}
```

You can specify an optional block by putting `?` after an identifier for the block procedure.

```
f() {block?} = { /* body */ }
```

You can call such a function either with or without a block.

```
f() {}  // OK
f()     // OK
```

If a block is not specified, the variable `block` takes `nil` value.

```
f() {block?} = {
    if (block) {
        // block is specified
    } else {
        // block is not specified
    }
}
```

If an Identifer for the block procedure is prefixed by Quote operator, the variable takes the procedure as an `expr` instance, not an `function` one.

```
f() {`block} = { /* body */ }
```

You need to use `expr#eval()` method to evaluate the block.

```
f() {`block} = {
    block.eval()
}
```

This feature is useful when you need to delegate a block to other function. If a Caller specifies a block that only has a block parameter containing a value of `expr` type, that value would be passed as a block procedure.
See a sample code below:

```
repeat_delegate(n) {`block} = {
    println('begin')
    repeat(n) {|block|}
    println('end')
}
```

Function `repeat_delegate()` takes a block procedure in `expr` type, which is passed to `repeat()` function in a delegation manner.
In general, a function call must be accompanied with an argument list even if it's empty. Though, if the call doesn't have any argument but has a block procedure, you can omit the list like below.

```
f { /* body */ }
```

8.5 Attribute

8.5.1 User-defined Attribute

An attribute works as another way to pass information to a function. In a function definition, acceptable attributes are listed within a pair of square brackets that follow after an argument list and a colon character.

```
f():[attr1,attr2,attr3] = { /* body */ }
```

You can call such a function like below. You can specify any number of attributes in any order.

```
f():attr1
f():attr2
f():attr1:attr3
```

In a function body, a variable named `__args__` of `args` type is defined, and you can use `args#isset()` method to check if an attribute is set.

```
f():[foo,bar] = {
    if (__args__.isset(`foo)) {
        // :foo is specified
    }
    if (__args__.isset(`bar)) {
        // :bar is specified
    }
}
```

8.5.2 Predefined Attributes

Attribute	Note
`:map`	
`:nomap`	
`:flat`	
`:noflat`	
`:list`	
`:xlist`	
`:iter`	
`:xiter`	
`:set`	
`:xset`	
`:void`	
`:reduce`	
`:xreduce`	
`:static`	
`:dynamic_scope`	
`:symbol_func`	
`:leader`	
`:trailer`	
`:finalizer`	
`:end_marker`	
`:public`	
`:private`	
`:nonamed`	
`:closure`	

8.6 Help Block

You can add a help block to a function by appending `%` and a block containing help information to a function declaration.

```
add(x, y) = {
    x + y
} % {
    `en, 'markdown', 'Takes two numbers and returns an added result.'
}
```

The content in the block has a format of {`lang:symbol, format:string, help:string`} which takes following values.

- `lang` .. Specifies a symbol of language that describes the help document: `en` for English and `ja` for Japanese, etc.
- `format` .. Specifies a syntax format. Only `'markdown'` is available so far.
- `help` .. Help string formatted in a syntax specified by `format` .

You can access the help information by following ways:

- In the interactive mode, evaluating the operator with a function instance would print its help information on the console.
- Calling function `function.gethelp()` would return a `help` instance that provides information about the used language, syntax format and help string.

A function may have multiple help blocks that contain explanatory texts written in different languages. Below is a function example that has helps written in English and Japanese:

```
add(x, y) = {
    x + y
} % {`en, 'markdown', R'''

(.. help document in English ..)

'''
} % {`ja, 'markdown', R'''

(.. help document in Japanese ..)

'''
} % {`de, 'markdown', R'''

(.. help document in German ..)

'''
}
```

A predefined variable `sys.langcode` determines which help should be printed by default. If a function doesn't have a help in the specified language, what appears at first in the declaration will be used.
You can also pass a language symbol to `function.gethelp` function as below.

```
function.gethelp(add, `en)
function.gethelp(add, `ja)
function.gethelp(add, `de)
```

8.7 Anonymous Function

A function `function()` creates an anonymous function instance from an argument list and a block that contains its function body.

```
function(x, y, z) { /* body */ }
```

When the function instance is assigned to a variable, that symbol is bound to the instance. The following two codes are equivalent each other.

```
f = function(x, y, z) { /* body */ }
f(x, y, z) = { /* body */ }
```

If you create a function that doesn't have arguments, you can call `function()` without an argument list like below.

```
function { /* body */ }
```

Since a special symbol `&` is also bound to the `function()` function, you can create a function instance as below.

```
&{ /* body */ }
```

When `function()` creates a function instance, it seeks variable symbols in the function body that start with `$` character, which are used as argument variables. For instance, see the following code:

```
function { printf('x = %s, y = %s, z = %s\n', $x, $y, $z) }
```

This is equivalent with the function creation shown below:

```
function($x, $y, $z) { printf('x = %s, y = %s, z = %s\n', $x, $y, $z) }
```

The order of arguments is the same with the order in which the variables appear in the body.

8.8 Closure

You can define a function inside another function body. In that case, the inner function can access variables in the outer function.

```
f() = {
    x = 3
    g() = {
        println('x = ', x)
    }
    g()  // evaluate the function
}
```

A function can also return a `function` instance that it creates as its result. The environment of the outer function will be held in the inner function.

```
f():closure = {
    x = 3
    g() = {
        println('x = ', x)
    }
    g
}

h = f()
h()
```

Make sure that a function that returns a `function` instance must be declared with `:closure` attribute.

8.9 Leader-trailer Relationship

When a Caller expression is described at the same line with the end of a preceding one, they have a leader-trailer relationship with the preceding one as a leader and the following one as a trailer.

```
f() g()
--- ---
 |   |
 |   +-- trailer
 +-- leader
```

In an ordinary case, these functions are evaluated sequentially in the same way that they're described in different lines.
The leader function has a right to control whether the trailer one should be evaluated or not. A method `args#quit_trailer()` will quit its trailer from being evaluated. Take a look at the following simple function to see how a trailer is controlled.

```
do_trailer(flag:boolean) = {
    if (!flag) {
        __args__.quit_trailer()
    }
}
```

Then the following code will print `hello` but no `good-bye`.

```
do_trailer(true) println('hello')
do_trailer(false) println('good-bye')
```

Some functions that govern sequence flow like `if-elsif-else` and `try-catch` utilizes this trailer control mechanism.

Chapter 9

Flow Control

9.1 Branch

Branch may be the most common flow-control in a program. Just like other programming language, Gura also provides `if - elsif - else` sequence. However, they're realized as functions, not as statements.
These elements are implemented by the following functions.
Function `if()` :

```
if (`cond):leader {block}
```

Function `elsif()` :

```
elsif (`cond):leader:trailer {block}
```

Function `else()` :

```
else():trailer {block}
```

They are concatenated with leader-trailer relationship, which means that a closing curly bracket of the preceding function must be in the same line as the top of the succceding one.

```
if (x) { /* branch 1 */ } elsif (y) { /* branch 2 */ } else { /* branch 3 */ }
```

Of course, content in a block embraced by a pair of curly brackets may contain multiple lines. This enables you to write a script in a similar syntax as other languages.

```
if (x) {

    /* branch 1 */

} elsif (y) {

    /* branch 2 */

} else {

    /* branch 3 */

}
```

Function `if()` and `elsif()` check the evaluated result of the expression `cond`. If it's determined as `true`, the block procedure will be evaluated, otherwise, the trailing function will be evaluated. Function `else()` always evaluates its block procedure.
Branch sequence has a result value as well. Consider the following code:

```
result = if (x < 0) {
    'less than zero'
} elsif (x > 0) {
    'greater than zero'
} else {
    'equal to zero'
}
```

In this case, if `x` is less than zero, the sequence would have a string `'less than zero'` as its result. It would have `'greater than zero'` for `x` with number greater than zero and `'equal to zero'` otherwise.
If function `if()` and `elsif()` have no following `else()` and their conditions are not evaluated as `true`, the result value will be `nil`.

```
x = 3
result = if (x < 0) {
    'less than zero'
}
// result is nil
```

9.2 Repeat

9.2.1 Repeating Functions

This subsection explains about some representative functions that evaluate a procedure repeatedly while it meets a given condition.
A function `repeat()` repeats a procedure for a specific number of times.

```
repeat (n?:number) {block}
```

If argument `n` is omitted, it will repeat the procedure indefinitely.
A function `while()` repeats a procedure while the condition is evaluated as `true`.

```
while (`cond) {block}
```

As a variable `cond` is an expression, it will be evaluated each time in the loop. In the following example, the function is given with an expression `n < 10`, which is to be evaluated during the repeating process.

```
n = 0
while (n < 10) {
    println('hello')
    n += 1
}
```

A function `for()` takes one or more expressions of iterable assignments, where an iterable means what can iterate elements including a list and an iterator instance.

```
for (`expr+) {block}
```

An iterator assignment is expressed with an operator `in` like below.
symbol `in` *iterable* `[`*symbol1* `,` *symbol2* `..]` `in` *iterable*
In the first format, it assigns `symbol` with a value in `iterable` each time in the loop. Below is an example.

```
for (name in ['apple', 'grape', 'banana']) {
    // any process
}
```

In the second format, if each element in the iterable is a list, corresponding values in the list are assigned to `symbol1` , `symbol2` , and so on. An example is shown below.

```
for ([name, yen] in [['apple', 100], ['grape', 200], ['banana', 90]]) {
    // any process
}
```

When a function `for()` takes more than one iterable assignment, it advances all the iterables one by one at each loop and repeats a procedure until one of the iterables reaches to the end. This means that the loop count is limited up to the smallest length of the iterables. The example below repeats the process three times.

```
for (x in [1, 2, 3, 4], y in [1, 2, 3], z in [1, 2, 3, 4, 5]) {
    // any process
}
```

A function `cross()` takes one or more expressions of iterable assignment and repeats a procedure with all the conceivable combination of elements from the iterables.

```
cross (`expr+) {block}
```

In `cross()` function, an iterable on the right advances at each loop and, when it reaches to its end, it will be rounded up to its first and causes an iterable on its left advance.
See the example below:

```
cross (x in ['A', 'B', 'C'], y in [1, 2, 3, 4]) {
    print(x, '-', y, ' ')
}
```

The result is:

```
A-1 A-2 A-3 A-4 B-1 B-2 B-3 B-4 C-1 C-2 C-3 C-4
```

Using `for()` function, the above code can be writen like below.

```
for (x in ['A', 'B', 'C']) {
    for (y in [1, 2, 3, 4]) {
        print(x, '-', y, ' ')
    }
}
```

Of course, you can specify any number of iterable assignments.

```
cross (x in ['A', 'B', 'C'], y in [1, 2, 3, 4], z in ['a', 'b', 'c']) {
    print(x, '-', y, '-', z, ' ')
}
```

9.2.2 Block Parameter

When calling `for()` , `while()` and `for()` , you can specify a block parameter in a format of `|i:number|` that takes an index number of loop starting from zero. In the following example, the parameter `i` takes `0` , `1` , `2` , `3` and `4` at each loop.

```
repeat (5) {|i|
    // any process
}
```

A block parameter for `cross()` function has a format of `|i:number, i1:number, i2:number, ..|` where `i` indicates an index number of loop, and each of `i1` , `i2` and so on takes an index number of corresponding iterable.

```
cross (x in ['A', 'B', 'C'], y in [1, 2, 3, 4], z in ['a', 'b', 'c']) {|i, ix, iy, iz|
    // any process
}
```

If you don't need indices information, you can omit whole the block parameter or part of its parameters.

9.2.3 Result Value of Repeat

Like a branch sequence, a repeat sequence also has a result value that comes from an evaluation of the last expression in the block procedure.
In default, among result values that have been generated from each loop, only the last one becomes the final result.

```
x = repeat (10) {|i|
    // any process
    i * 10
}
// x is 90
```

When you call a repeat function with `:list` attribute, it will return a list that contains result values in the loop.

```
x = repeat (10):list {|i|
    // any process
    i * 10
}
// x is [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
```

With an attribute `:xlist` , you can remove `nil` value from the created list.

```
x = repeat (10):xlist {|i|
    // any process
    if (i % 2 == 0) {
        i * 10
    }
}
// x is [0, 20, 40, 60, 80]
```

Using this feature, you can create a list that only contains elements that suit some conditions. Attributes `:set` and `:xset` work in a similar way with `:list` and `:xlist` respectively, but they would create a list that contains unique values by rejecting a value that already exists in the list.

9.2.4 Flow Control in Repeat Sequence

If you want to quit a repeat sequence, you can use `break()` function. Aiming for a similar appearance with C and Java, you can call `break()` without a pair of parenthesis for an argument list.

```
repeat (10) {|i|
    // any process
    if (i == 5) {
        break
    }
    // not evaluated when break() is called
}
```

The function `break()` takes an argument of any type that affects a result value of the repeat. When `break()` is called without an argument, the repeat's result doesn't contain a value of the last loop.

```
x = repeat (10):list {|i|
    if (i == 5) {
        break
    }
    i
}
// x is [0, 1, 2, 3, 4]
```

If you call `break()` with a valid argument, that will be included in the repeat's result.

```
x = repeat (10):list {|i|
    if (i == 5) {
        break(99)
    }
    i
}
// x is [0, 1, 2, 3, 4, 99]
```

If you need to go to the next turn of the loop after skipping remaining procedure, you can use `continue()` function. As with the function `break` , you can omit a pair of parentheses for an argument list when calling it.

```
repeat (10) {|i|
    // any process
    if (i % 2 == 0) {
        continue
    }
    // not evaluated when continue() is called
}
```

When you call `continue()` with no argument, the repeat's result doesn't contain a value of that loop.

```
x = repeat (10):list {|i|
    if (i % 2 == 0) {
        continue
    }
    i
}
// x is [1, 3, 5, 7, 9]
```

If you call `continue()` with a valid argument, that value will be included in the repeat's result.

```
x = repeat (10):list {|i|
    if (i % 2 == 0) {
        continue(99)
    }
    i
}
// x is [99, 1, 99, 3, 99, 5, 99, 7, 99, 9]
```

9.2.5 Generate Iterator

As you've already seen in the above, appending an attribute `:list` causes the repeating process to create a list that contains evaluated result of each loop as its element. In the following example, `x` will be a list of `[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]` .

```
x = repeat (10):list {|i|
    // any process
     i * 10
}
```

An attribute `:iter` would have a more interesting result. Take a look at the code below:

```
x = repeat (10):iter {|i|
    // any process
     i * 10
}
```

In this case, repeating process is not executed when the `repeat` function is evaluated. `x` is an *iterator* that generates values of 0, 10, 20, 30, 40, 50, 60, 70, 80 and 90, and these values are available only when the iterator is actually evaluated.
The following code shows how to get values from the iterator using Implicit Mapping:

```
println(x)
```

Following code evaluates `x` step by step to confirm that it actually works as an iterator.

```
println(x.next())
println(x.next())
println(x.next())
println(x.next())
println(x.next())
```

An attribute `:xiter` works as `:iter` except that it will eliminate `nil` value from its element.

```
x = repeat (10):xiter {|i|
    // any process
    if (i % 2 == 0) {
        i * 10
    }
}
```

In the above case, `x` is an *iterator* that generates values of 0, 20, 40, 60 and 80.
You can also use `break()` and `continue()` in an iterator created by a repeating function. Such an iterator yields elements in the same way as a repeating process that creates a list.
An iterator created by a repeat function and a closure generated within a function are similar in that they postpone their actual jobs. They also have similarity in a manner to handle variable environments. Consider the following code.

```
f() = {
    n = 0
    while (n < 5):iter {
        n += 1
        n
    }
}
x = f()
```

The function `f` returns an iterator created by `while` , which is expected to generate values of 1, 2, 3, 4 and 5. In this case, the repeat body has a reference to a variable named `n` that belongs to the scope of function `f` . Can an iterator refer to a variable that may be destroyed at the end of a function?
Actually, it's OK. An iterator created by a repeating function owns an environment in which that function has been called. In the above example, the variable `n` is owned by the returned iterator.
You'll see more practical usage of this feature in this.
You can also implement a nested loop in an iterator created by a repeat function.

```
x = for (a in ['A', 'B', 'C']):iter {

    // any process

    for (b in [0, 1, 2]):iter {
        a + b
    }
}
// x will generate 'A0', 'A1', 'A2', 'B0', 'B1', 'B2', 'C0', 'C1' and 'C2'
```

A nested loop with an iterator generation must be placed at the last in the repeat procedure.
You can also place any iterators in the repeat function that are to be iterated when the outer iterator is evaluated. But, there's one point you have to be careful with. See the following code:

```
x = repeat (2):iter {
    range(3)
}
```

It's expected that the iterator `x` will generate numbers of 0, 1, 2, 0, 1 and 2 after the outer iterator iterates an iterator created by `range(3)` for twice. But, in reality, it will just generate two iterator instances without iterating them.
Iterators created by repeat functions have a "repeater" flag that enable them to be iterated in a nested block. Since other ordinary interators don't have this flag, you have to call `iterator#repeater()` method to turn it on as shown below.

```
x = repeat (2):iter {
    range(3).repeater()
}
```

9.2.6 Repeat Process with Function that Creates Iterator

Many of functions that creates an iterator as their result may take an optional block procedure. For such functions, you can specify a block that is to be evaluated repeatedly while iterating values in the created iterator.
For instance, consider a function `readlines()` , which creates an iterator that reads content of a stream and returns strings of each line. Without a block, it simply returns the created iterator.

```
x = readlines('foo.txt')
```

Specifying a block would evaluate the block procedure repeatedly.

```
readlines('foo.txt') {
    // any process
}
```

You can get each value from the iterator by specifying a block parameter.

```
readlines('foo.txt') {|line|
    print(line)
}
```

A second argument in the block parameter takes an index number of the loop.

```
readlines('foo.txt') {|line, i|
    printf('%d: %s', i + 1, line)
}
```

When you specify a block procedure to an iterator creating function, it behaves in the same way as repeating functions such as `for()` and `repeat()`. This means that you can use flow control functions `break()` and `continue()` in that loop.

```
readlines('foo.txt') {|line|
    // any process
    if (line.chop() == '') {
        break
    }
    // any process
}
```

You can also specify attributes `:list`, `:xlist`, `:set` and `:xset` to indicate it to create a list.

```
x = readlines('foo.txt'):list {|line|
    line.upper()
}
// x is a list containing each line's string in uppercase.
```

And attributes `:iter` and `:xiter` that create an iterator are also available.

```
x = readlines('foo.txt'):iter {|line|
    line.upper()
}
// x is an iterator that generates each line's string in uppercase.
```

9.3 Error Handling

You can use `try-catch` sequence to capture errors. Any process that may occur errors is written in a block of `try()` function and error handling processes are written in blocks of `catch()` function trailing after that.

```
try {
    // any process
} catch (error.ValueError) {
    // handling ValueError
} catch (error.IndexError, error.IOError) {
    // handling IndexError and IOError
} catch {
    // handling of other errors
}
```

A function `catch()` takes one or more arguments that specify `error` instances that are to be handled. If no argument is specified, any type of errors are handled in the function.
Here are some of the `error` instances that can be specified for `catch()` argument.

Error Instance	**Note**
`error.ValueError`	Invalid argument is specified.
`error.IndexError`	Invalid value for indexing.
`error.IOError`	Error occurs while accessing I/O devices.

A block in the `catch()` function has a block parameter in a format of `|err:error|` where `err` takes a value of `error` type that contains error information such as an error message and a file name and a line position at which the error occurs.

Property	**Data Type**	**Note**
`error#lineno`	`number`	Line number
`error#source`	`string`	Source of the code that occurs an error
`error#text`	`string`	Error message

An example code is shown below:

```
try {
    // any process
} catch {|err|
    printf('%s at %s:%d\n', err.text, err.source, err.lineno)
}
```

Chapter 10

Object Oriented Programming

10.1 Class and Instance

A **class** is a kind of environment that contains properties such as functions and variables, and has an ability to create **instances** that share these properties. A class is associated with a data type one by one, which means that all the values in a script are bound to certain classes. For example, a value `3.14` is associated with `number` class, and `'hello world'` with `string` class.
A class contains functions called **method** that operate with a class or an instance. A method that belongs to a class is called **class method** and is described as below, where `Foo` and `func` are names of the class and the class method respectively.

```
Foo.func()
```

A method that works on an instance is called **instance method** and is described as below, where `Foo` and `func` are names of the class and the instance method respectively.

```
Foo#func()
```

The symbol `#` is not used for an actual instance operation and only appears in documentation to describe instance methods. You can call an instance method like below, where `foo` is an instance of class `Foo` .

```
foo.func()
```

A class also owns variables called **class variable**, which are shared by instances from the class. Each instance can contain its own variables that are called **instance variable**.
A class variable is described as below, where `Foo` and `value` are names of the class the class variable respectively:

```
Foo.value
```

An instance variable is described as below, where `Foo` and `value` are names of the class the instance variable respectively:

```
Foo#value
```

You can use `dir()` function to see what methods and variables are available with a value.

```
>>> x = 3.14
3.14
>>> dir(x)
[‘__call__, ‘__iter__, ‘clone, ‘getprop!, ‘is, ‘isinstance, ‘isnil, ‘istype, ‘nomap, ‘roundoff, ‘setprop
```

10.2 User-defined Class

You can use `class` function to create a user-defined class. The code below creates a class named `A` with empty properties.

```
A = class {}
```

This assignment would create a class named `A` and also define a constructor function `A()` that generates an instance of the class. You can call the constructor function like below:

```
a = A()
```

A block of the `class` function should contain Assign and Caller expressions. Existance of other expressions would cause an error. They're evaluated just one time when the class is created. Actual jobs in these expressions are summarized below:

- **Assign expression**

  A function assigned in the block becomes a method that belong to the class. If the function is declared with `:static` attribute appended right after the argument list, it would become a class method that you can call along with the class name. Otherwise, it would become an instance method that works with an instance created from the class.

  A variable assigned in the block are registered as a class variable that belong to the class itself, not to an instance.

  The assigned value is actually evaluated at the timing of assignment, which means you can even call a function to get the value.

- **Caller expression**

  A function or another callable is evaluated within the class as its environment.

Here's a sample script to see details about factors in the block.

```
Person = class {

    __init__(name:string, age:number, role:string) = {
        this.name = name
        this.age = age
        this.role = role
    }

    fmt = 'name: %s, age: %d, role:%s\n'  // class variable

    Print() =  {
        // A class variable doesn't need 'this' or class name when accessing it
        // while an instance variable does.
        printf(fmt, this.name, this.age, this.role)
    }

    Test():static = {
        println('test of class method')
    }

}
```

In an instance method, a variable named `this` is defined, which contains a reference to the instance itself. You always need to specify `this` variable to access instance variables and instance methods.
As for a class variable, a method can refer to it without specifying `this` or the class name.

An instance method `__init__()` is a special one that defines a constructor function. Its arguments are reflected on that of the constructor. The sample above creates a function named `Person` that has a declaration shown below:

```
Person(name:string, age:number, role:string) {block?}
```

You can create an instance by calling it like below:

```
p = Person('Taro Yamada', 27, 'engineer')
```

If you specify an optional block, the block procedure will be evaluated with a block parameter that takes the created instance.

```
Person('Taro Yamada', 27, 'engineer') {|p|
    // any process
}
```

After an instance is created, you can call an instance method with it. Below is an example to call an instance method `Print()`, where `p` is the created instance:

```
p.Print()
```

You can call a class method `Test()` like below:

```
Person.Test()
```

You can also call a class method by specifying an instance.

```
p.Test()
```

10.3 Inheritance

10.3.1 Basic

You can create an inherited class by specifying a super class in an argument of `class()`.

```
Person2 = class(Person) {
    // class variable and methods
}
```

If you don't declare `__init__()` method in the derived class, it would inherit a constructor of the super class.

10.3.2 Constructor in Derived Class

When you declare `__init__()` method in the derived class, you have to specify block parameters that satisfies the argument declaration of the super class's constructor.

```
Teacher = class(Person) {
    __init__(name:string, age:number) = {|name, age, 'teacher'|
    }
    Work() = {
        println('give lectures to others')
    }
```

```
}

Student = class(Person) {
    __init__(name:string, age:number) = {|name, age, 'student'|
    }
    Work() = {
        println('learn about something')
    }
}
```

As block parameters are just like oridinary arguments in a function call, you can describe any expressions in them. Though, take notice that you have to surround an expression including bitwise OR operation "`|` " with parentheses to avoid it from being confused with border characters around block parameters. See the example below:

```
A = class {
    __init__(name:string, bitflags:number) = {
        // any jobs
    }
}

B = class(A) {
    __init__() = {|'hello', (1 | 4 | 8)|
        // any jobs
    }
}
```

10.3.3 Method Override

Take a look at a behavior of instance methods in an inherited class. Consider the following script:

```
A = class {
    func() = {
        println('A.func()')
    }
}

B = class(A) {
    func() = {
        println('B.func()')
    }
    test() = {
        this.func()  // calls B#func()
    }
}

b = B()
b.test()
```

Both class A and B have a method with the same name `func()` . When the method `B#test()` evaluates `this.func()` , it actually calls `B#func()` .
You can use `super()` function to call a method that belongs to a super class. Below is a sample code to show how to use it.

```
B = class(A) {
    func() = {
        println('B.func()')
    }
```

```
    test() = {
        super(this).func()  // calls A#func()
    }
}
```

10.4 Encapsulation

By default, instance and class variables are only accessible through `this` variable. Such variables are called **private variable**. You can make them accessible through other instance variables by specifying `:public` attribute in their assignment expressions. Those variables are called **public variable**.

```
X = class {

    c = 3
    d:public = 4

    __init__() = {
        this.a = 1
        this.b:public = 2
    }
}

x = X()
println(x.a)    // private instance variable .. Error
println(x.b)    // public instance variable  .. OK
println(x.c)    // private class variable    .. Error
println(x.d)    // public class variable     .. OK
```

You can also call `public()` function within a block of `class()` function that indicates which variables are to be publicized. The `public()` function takes a block that contains two types of expressions: Identifier and Assign. An Identifer expression only declares a variable symbol for publication. An Assign expression creates a public class variable with the specified value. The script below is equivalent with the above but uses `public()` function.

```
X = class {

    c = 3
    public {
        b
        d = 4
    }
    __init__() = {
        this.a = 1
        this.b = 2
    }
}
```

Different from variables, methods are accessible through variables other than `this` by default. Such methods are called **public method**. You can make them only accessible through `this` variable by specifying `:private` attribute in their assignment expressions. Those methods are called **private method**.

10.5 Structure

A structure is a kind of a class, but offers a simple way to implement a constructor. Function `struct` takes variable declarations as its arguments that are reflected on the generated constructor. Below is an example:

```
Point = struct(x:number, y:number)
```

This generates a constructor shown below:

```
Point(x:number, y:number)
```

You can call it like below:

```
pt = Point(3, 4)
```

A created instance from this class will have members named `x` and `y` .

```
printf('%d, %d\n', pt.x, pt.y)
```

The code above that uses `struct` can be written using `class` like below:

```
Point = class {
    __init__(x:number, y:number) = {
        this.x:public = x
        this.y:public = y
    }
}
```

A structure can also have methods by describing them in a block of `struct` function.

```
Point = struct(x:number, y:number) {
    Move(xdiff:number, ydiff:number) = {
        this.x += xdiff
        this.y += ydiff
    }
    Print() = {
        printf('%d, %d\n', this.x, this.y)
    }
}
```

10.6 Creation of Multiple Instances

How can we create a list of instances from a certain class? Below is an example to create a list of `Person` instances.

```
people = [
    Person('Kikuo Ochiai',     24, 'teacher')
    Person('Seiji Miki',       33, 'engineer')
    Person('Haruka Nakao',     28, 'sales')
    Person('Takashi Sugimura', 21, 'student')
]
```

Obviously, it's cumbersome to describe a function name `Person()` for each item. Using a list creation function `@` enables you to write more simple code.

```
people = @(Person) {
    { 'Kikuo Ochiai',     24, 'teacher'  }
    { 'Seiji Miki',       33, 'engineer' }
    { 'Haruka Nakao',     28, 'sales'    }
    { 'Takashi Sugimura', 21, 'student'  }
}
```

Function `@` takes a function such as a constructor, and its block contains a set of argument lists fed into that function.

10.7 Forward Declaration

Within a block of the `class` function, it would be no problem for argument declarations to refer to its own class being currently declared.

```
A = class {
    func(a:A) = {     // This is OK.
        // ...
    }
}
```

It's not allowed to refer to a class which declaration appears afterwards.

```
A = class {
    func(b:B) = {     // *** error ***
        // ...
    }
}

B = class {

}
```

For such a case, you need to prepare a forward declaration of the referenced class before the referencing point by creating an empty class like below:

```
B = class()           // B's forward declaration.

A = class {
    func(b:B) = {     // This is OK.
        // ...
    }
}

B = class {           // B's actual declaration.
    // ...
}
```

Chapter 11

Mapping Process

11.1 About This Chapter

This chapter explains about Gura's mapping process, Implicit Mapping and Member Mapping. In the documentation, following terms are used to describe species of values.

- scalar an instance of any type except for `list` and `iterator`
- list an instance of `list`
- iterator an instance of `iterator`
- iterable list or iterator

11.2 Implicit Mapping

11.2.1 Overview

Implicit Mapping is a feature that evaluates a function or an operator repeatedly when it's given with a list or an iterator.
A function that is capable of Implicit Mapping is marked with an attribute `:map` . Consider a function `f(n:number):map` that takes a number value and returns a squared number of it. You can call it like `f(3)` , which is expected to return a number `9` . Here, using Implicit Mapping, you can call it with a list of numbers like below:

```
f([2, 3, 4])
```

This will result in a list `[4, 9, 16]` after evaluating each number in the list.
Implicit Mapping also works with operators. The example below applies an operation that adds three to each value in the list using Implicit Mapping:

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

This will result in `[5, 6, 7]` . Of course, you can also apply Implicit Mapping on an operation between two lists. See the following example:

```
[2, 3, 4] + [3, 4, 5]
```

As you might expect, it returns a list `[5, 7, 9]` .
The above example may just look like a vector calculation. Actually, this type of operation, which applies some operations on a set of numbers at once, is known as "vectorization", and has been implemented in languages and libraries that compute vectors and matrices.
Implicit Mapping enhances that idea so that it has the following capabilities:

1. Implicit Mapping can handle any type of objects other than number.

   Consider a function `g(str:string):map` that takes a string and returns a result after converting alphabet characters in the string to upper case. When you call it with a single value, it will return one result.

   ```
   str = 'hello'
   x = g(str)
   // x is 'HELLO'
   ```

   You can call it with a list of string to get a list of results by using Implicit Mapping feature.

   ```
   strs = ['hello', 'Gura', 'world']
   x = g(strs)
   // x is ['HELLO', 'GURA', 'WORLD']
   ```

2. Implicit Mapping can operate with an iterator as well.

   Consider the function `g(str:string):map` that has been mentioned above. If you call it with an iterator, it will return an iterator as its result.

   ```
   strs = ('hello', 'Gura', 'world') // creates an iterator
   x = g(strs)
   // x is an iterator that equivalent with ('HELLO', 'GURA', 'WORLD')
   ```

   It means that the actual evaluation of the function `g()` will be postponed by the time when the created iterator is evaluated or destroyed. This is important because using an iterator will enable you to avoid unnecessary calculation and memory consumption.

3. You can use Implicit Mapping to repeat a function call without an explicit repeat procedure.

   A built-in function `println():map` prints a content of the given value before putting out a line-feed. Consider a case that you need to print each value in the list `strs` that contains `['hello', 'Gura', 'world']` . With an ordinary idea, you may use `for()` to process each item in a list.

   ```
   for (str in strs) {
       println(str)
   }
   ```

   Using Implicit Mapping, you can simply do it like below:

   ```
   println(strs)
   ```

4. Implicit Mapping can work on any number of lists and iterators given in an argument list of a function call.

   Consider that there's a function `f(a:string, b:number, c:string):map` , and you give it lists as its arguments like below:

   ```
   as = ['first', 'second', 'third', 'fourth']
   bs = [1, 2, 3, 4]
   cs = ['one', 'two', 'three', 'four']
   f(as, bs, cs)
   ```

   This has the same effect as below:

```
f('first', 1, 'one')
f('second', 2, 'two')
f('third', 3, 'three')
f('fourth', 4, 'four')
```

11.2.2 Mapping Rule with Operator

Implicit Mapping works on most of the operators even though there are several exceptions. In the description below, a symbol `o` is used to represent a certain operator.
With a Prefixed Unary Operator, species of the result is the same as that of the given value. Below is a summary table:

Operation	**Result**
`o scalar`	scalar
`o list`	list
`o iterator`	iterator

Examples are shown below:

Example	**Result**
`!true`	`false`
`![true, true, false, true]`	`[false, false, true, false]`
`!(true, true, false, true)`	`(false, false, true, false)`

With a Suffixed Unary Operator, species of the result is the same as that of the given value. Below is a summary table:

Operation	**Result**
`scalar o`	scalar
`list o`	list
`iterator o`	iterator

With a Binary Operator, the folloiwing rules are applied.

- If both of left and right values are of scalar species, the result becomes a scalar.
- If either of left or right value is of iterator species, the result becomes an iterator.
- Otherwise, the result becomes a list.

Below is a summary table:

Operation	**Result**
`scalar o scalar`	scalar
`scalar o list`	list
`scalar o iterator`	iterator
`list o scalar`	list
`list o list`	list
`list o iterator`	iterator
`iterator o scalar`	iterator
`iterator o list`	iterator
`iterator o iterator`	iterator

If both of left and right values are iterable and they have different length, Implicit Mapping would be applied on a range of a shorter length.
Some operators expect lists or iterators in their own operations and inhibit Implicit Mapping. See the table below:

Operation	Note
`x?`	It deters Implicit Mapping because it needs to check if `x` itself can be determined as `true` or not.
`x*`	It expects `x` may take an iterator or a list.
`x * y` where `x` is `function`	It may take a list as a value of `y`.
`x % y` where `x` is `string`	It may take a list as a value of `y`.
`x in y`	It expects `x` and `y` may take list values.
`x => y`	It expects `y` may take a list value.

11.2.3 Mapping Rule with Function

A function with `:map` attribute in its declaration is capable of Implicit Mapping.
Here are function definitions that return a square value of the given number to see the effect of `:map` attribute.

```
f_nomap(x:number) = x * x
f_map(x:number):map = x * x
```

The function delcared with `:map` attribute is capable of Implicit Mapping and can take a list for an argument that expects a `number` value.

```
f_nomap([1, 2, 3]) // Error
f_map([1, 2, 3])   // Implicit Mapping works on each item and returns [1, 4, 9]
```

As for a function `f(x):map` that takes one argument, the mapping rule is the same as that of Unary Operator. See the following summary table:

Operation	Result
`f(scalar)`	scalar
`f(list)`	list
`f(iterator)`	iterator

A function `f(x, y):map` that takes two arguments behaves in the same manner with Binary Operator. Below is a summary table:

Operation	Result
`f(scalar, scalar)`	scalar
`f(scalar, list)`	list
`f(scalar, iterator)`	iterator
`f(list, scalar)`	list
`f(list, list)`	list
`f(list, iterator)`	iterator
`f(iterator, scalar)`	iterator
`f(iterator, list)`	iterator
`f(iterator, iterator)`	iterator

In general, if a function takes more than one argument, the following rules are appplied.

- If all of the argument values are of scalar species, the result becomes a scalar.
- If one of the argument values is of iterator species, the result becomes an iterator.
- Otherwise, the result becomes a list.

Here are some example cases with a function `f(x, y, z):map` :

Operation	**Result**
`f(scalar, scalar, sholar)`	scalar
`f(scalar, scalar, list)`	list
`f(scalar, scalar, iterator)`	iterator
`f(scalar, list, iterator)`	iterator

If an argument list contains iterables that have different length each other, Implicit Mapping would be applied on a range of the shortest length. For example, the code below repeats the process three times.

```
f([1, 2, 3], ['a', 'b', 'c', 'd'], [4, 5, 6])
```

Implicit Mapping does not work with arguments that match the following case:

- If a function contains an argument that expects `list` or `iterator` , Implicit Mapping would not work with that argument. In the following example, putting a list or an iterator to argument `z` , which expects a list or an iterator as its value, is not considered as a criteria for Implicit Mapping.

  ```
  f(x, y, z:list):map     = { /* body */ }
  f(x, y, z:iterator):map = { /* body */ }
  f(x, y, z[]):map        = { /* body */ }
  ```

- Putting an attribute `:nomap` to an argument declaration would exclude it from Implicit Mapping criteria. In the example below, specifying a list or an iterator to argument `z` is not considered as a criteria for Implicit Mapping.

  ```
  f(x, y, z:nomap):map = { /* body */ }
  ```

11.2.4 Result Control on List

Consider a function `f(n:number):map` that is defined as below:

```
f(n:number):map = println('n = ', n)
```

It takes a number value and just prints it.

```
f(3)  // prints 'n = 3'
```

Here, function `println()` is defined with an attribute `:void` that is meant to always return `nil` as its result. So, the function `f()` that evaluates `println()` at last would return `nil` as well.
As the function `f()` is capable of Implicit Mapping, you can call it with a list for repeating process.

```
f([1, 2, 3])  // prints 'n = 1', 'n = 2' and 'n = 3'
```

As you've already seen above, when a function with `:map` attribute takes a list, it will evaluate each value in the list immediately and returns a list containing the results. Considering that rule, you may think the calling it as above could return `[nil, nil, nil]` .
But, in reality, it returns a single `nil` .
Implicit Mapping is designed to work as a generic repeating mechanism. If a function is expected to always return some meaningless value such as `nil` , creating a list that contains such values through a repeating process absolutely makes no sense. To avoid that vain process, Implicit Mapping would only create a list when a valid value appears in the result.
Consider a function below that simply returns the given value as its result.

```
g(n):map = n
```

The table below summarizes what result you get from `g()` when it's given with a list containing valid and `nil` values.

Script	Result
g([])	[]
g([nil])	nil
g([nil, nil])	nil
g([nil, nil, 3])	[nil, nil, 3]
g([nil, nil, 3, 5])	[nil, nil, 3, 5]
g([nil, nil, 3, 5, 3])	[nil, nil, 3, 5, 3]
g([nil, nil, 3, 5, 3, nil])	[nil, nil, 3, 5, 3, nil]

Note that, when you give an empty list to a function with Implicit Mapping, it would return an empty list as its result.
There are some attributes that control how Implicit Mapping generates the result even when it's expected to generate a list by default.

- Attribute `:list` always creates a list even if it only contains `nil` values.

Script	Result
g([]):list	[]
g([nil]):list	[nil]
g([nil, nil]):list	[nil, nil]
g([nil, nil, 3]):list	[nil, nil, 3]
g([nil, nil, 3, 5]):list	[nil, nil, 3, 5]
g([nil, nil, 3, 5, 3]):list	[nil, nil, 3, 5, 3]
g([nil, nil, 3, 5, 3, nil]):list	[nil, nil, 3, 5, 3, nil]

- Attribute `:xlist` always creates a list after excluding `nil` value from the result.

Script	Result
g([]):xlist	[]
g([nil]):xlist	[]
g([nil, nil]):xlist	[]
g([nil, nil, 3]):xlist	[3]
g([nil, nil, 3, 5]):xlist	[3, 5]
g([nil, nil, 3, 5, 3]):xlist	[3, 5, 3]
g([nil, nil, 3, 5, 3, nil]):xlist	[3, 5, 3]

- Attribute `:set` always creates a list after excluding duplicated values.

Script	Result
`g([]):set`	`[]`
`g([nil]):set`	`[nil]`
`g([nil, nil]):set`	`[nil]`
`g([nil, nil, 3]):set`	`[nil, 3]`
`g([nil, nil, 3, 5]):set`	`[nil, 3, 5]`
`g([nil, nil, 3, 5, 3]):set`	`[nil, 3, 5]`
`g([nil, nil, 3, 5, 3, nil]):set`	`[nil, 3, 5]`

- Attribute `:xset` always creates a list after excluding `nil` and duplicated values.

Script	Result
`g([]):xset`	`[]`
`g([nil]):xset`	`[]`
`g([nil, nil]):xset`	`[]`
`g([nil, nil, 3]):xset`	`[3]`
`g([nil, nil, 3, 5]):xset`	`[3, 5]`
`g([nil, nil, 3, 5, 3]):xset`	`[3, 5]`
`g([nil, nil, 3, 5, 3, nil]):xset`	`[3, 5]`

- Attribute `:iter` creates an iterator.

Script	Result
`g([]):iter`	equivalent of `[].each()`
`g([nil]):iter`	equivalent of `(nil,)`
`g([nil, nil]):iter`	equivalent of `(nil, nil)`
`g([nil, nil, 3]):iter`	equivalent of `(nil, nil, 3)`
`g([nil, nil, 3, 5]):iter`	equivalent of `(nil, nil, 3, 5)`
`g([nil, nil, 3, 5, 3]):iter`	equivalent of `(nil, nil, 3, 5, 3)`
`g([nil, nil, 3, 5, 3, nil]):iter`	equivalent of `(nil, nil, 3, 5, 3, nil)`

- Attribute `:xiter` creates an iterator that excludes `nil` value from the result.

Script	Result
`g([]):xiter`	equivalent of `[].each()`
`g([nil]):xiter`	equivalent of `[].each()`
`g([nil, nil]):xiter`	equivalent of `[].each()`
`g([nil, nil, 3]):xiter`	equivalent of `(3,)`
`g([nil, nil, 3, 5]):xiter`	equivalent of `(3, 5)`
`g([nil, nil, 3, 5, 3]):xiter`	equivalent of `(3, 5, 3)`
`g([nil, nil, 3, 5, 3, nil]):xiter`	equivalent of `(3, 5, 3)`

- Attribute `:void` indicates the function always returns `nil` regardless of its original result.

Script	Result
`g([]):void`	`nil`
`g([nil]):void`	`nil`
`g([nil, nil]):void`	`nil`
`g([nil, nil, 3]):void`	`nil`
`g([nil, nil, 3, 5]):void`	`nil`
`g([nil, nil, 3, 5, 3]):void`	`nil`
`g([nil, nil, 3, 5, 3, nil]):void`	`nil`

- Attribute `:reduce` indicates the function returns the last evaluated value and doesn't create a list.

Script	Result
`g([]):reduce`	`nil`
`g([nil]):reduce`	`nil`
`g([nil, nil]):reduce`	`nil`
`g([nil, nil, 3]):reduce`	`3`
`g([nil, nil, 3, 5]):reduce`	`5`
`g([nil, nil, 3, 5, 3]):reduce`	`3`
`g([nil, nil, 3, 5, 3, nil]):reduce`	`nil`

- Attribute `:xreduce` indicates the function returns the last evaluated value and doesn't create a list. The returned value is updated only when the result is valid.

Script	Result
`g([]):xreduce`	`nil`
`g([nil]):xreduce`	`nil`
`g([nil, nil]):xreduce`	`nil`
`g([nil, nil, 3]):xreduce`	`3`
`g([nil, nil, 3, 5]):xreduce`	`5`
`g([nil, nil, 3, 5, 3]):xreduce`	`3`
`g([nil, nil, 3, 5, 3, nil]):xreduce`	`3`

11.2.5 Result Control on Iterator

Consider a function below that simply returns the given value as its result.

```
g(n):map = n
```

When you give it an iterator, it would return an iterator as well following after Implicit Mapping rule.

Script	Result
`g([].each())`	equivalent of `[].each()`
`g((nil,))`	equivalent of `(nil,)`
`g((nil, nil))`	equivalent of `(nil, nil)`
`g((nil, nil, 3))`	equivalent of `(nil, nil, 3)`
`g((nil, nil, 3, 5))`	equivalent of `(nil, nil, 3, 5)`
`g((nil, nil, 3, 5, 3))`	equivalent of `(nil, nil, 3, 5, 3)`
`g((nil, nil, 3, 5, 3, nil))`	equivalent of `(nil, nil, 3, 5, 3, nil)`

There are some attributes that control how Implicit Mapping generates the result even when it's expected to generate an iterator by default.

- Attribute `:list` creates a list.

Script	Result
`g([].each()):list`	`[]`
`g((nil,)):list`	`[nil]`
`g((nil, nil)):list`	`[nil, nil]`
`g((nil, nil, 3)):list`	`[nil, nil, 3]`
`g((nil, nil, 3, 5)):list`	`[nil, nil, 3, 5]`
`g((nil, nil, 3, 5, 3)):list`	`[nil, nil, 3, 5, 3]`
`g((nil, nil, 3, 5, 3, nil)):list`	`[nil, nil, 3, 5, 3, nil]`

- Attribute `:xlist` creates a list after excluding `nil` value from the result.

Script	Result
`g([].each()):xlist`	`[]`
`g((nil,)):xlist`	`[]`
`g((nil, nil)):xlist`	`[]`
`g((nil, nil, 3)):xlist`	`[3]`
`g((nil, nil, 3, 5)):xlist`	`[3, 5]`
`g((nil, nil, 3, 5, 3)):xlist`	`[3, 5, 3]`
`g((nil, nil, 3, 5, 3, nil)):xlist`	`[3, 5, 3]`

- Attribute `:set` creates a list after excluding duplicated values.

Script	Result
`g([].each()):set`	`[]`
`g((nil,)):set`	`[nil]`
`g((nil, nil)):set`	`[nil]`
`g((nil, nil, 3)):set`	`[nil, 3]`
`g((nil, nil, 3, 5)):set`	`[nil, 3, 5]`
`g((nil, nil, 3, 5, 3)):set`	`[nil, 3, 5]`
`g((nil, nil, 3, 5, 3, nil)):set`	`[nil, 3, 5]`

- Attribute `:xset` creates a list after excluding `nil` and duplicated values.

Script	Result
`g([].each()):xset`	`[]`
`g((nil,)):xset`	`[]`
`g((nil, nil)):xset`	`[]`
`g((nil, nil, 3)):xset`	`[3]`
`g((nil, nil, 3, 5)):xset`	`[3, 5]`
`g((nil, nil, 3, 5, 3)):xset`	`[3, 5]`
`g((nil, nil, 3, 5, 3, nil)):xset`	`[3, 5]`

- Attribute `:iter` creates an iterator. This is a default behavior of Implicit Mapping for an iterator.

- Attribute `:xiter` creates an iterator that excludes `nil` value from the result.

Script	Result
`g([].each()):xiter`	equivalent of `[].each()`
`g((nil,)):xiter`	equivalent of `[].each()`
`g((nil, nil)):xiter`	equivalent of `[].each()`
`g((nil, nil, 3)):xiter`	equivalent of `(3,)`
`g((nil, nil, 3, 5)):xiter`	equivalent of `(3, 5)`
`g((nil, nil, 3, 5, 3)):xiter`	equivalent of `(3, 5, 3)`
`g((nil, nil, 3, 5, 3, nil)):xiter`	equivalent of `(3, 5, 3)`

- Attribute `:void` indicates the function will always return `nil` regardless of its original result.

Script	Result
`g([].each()):void`	`nil`
`g((nil,)):void`	`nil`
`g((nil, nil)):void`	`nil`
`g((nil, nil, 3)):void`	`nil`
`g((nil, nil, 3, 5)):void`	`nil`
`g((nil, nil, 3, 5, 3)):void`	`nil`
`g((nil, nil, 3, 5, 3, nil)):void`	`nil`

- Attribute `:reduce` indicates the function returns the last evaluated value and doesn't create an iterator.

Script	Result
`g([].each()):reduce`	`nil`
`g((nil)):reduce`	`nil`
`g((nil, nil)):reduce`	`nil`
`g((nil, nil, 3)):reduce`	`3`
`g((nil, nil, 3, 5)):reduce`	`5`
`g((nil, nil, 3, 5, 3)):reduce`	`3`
`g((nil, nil, 3, 5, 3, nil)):reduce`	`nil`

- Attribute `:xreduce` indicates the function returns the last evaluated value and doesn't create an iterator. The returned value is updated only when the result is valid.

Script	Result
`g([].each()):xreduce`	`nil`
`g((nil)):xreduce`	`nil`
`g((nil, nil)):xreduce`	`nil`
`g((nil, nil, 3)):xreduce`	`3`
`g((nil, nil, 3, 5)):xreduce`	`5`
`g((nil, nil, 3, 5, 3)):xreduce`	`3`
`g((nil, nil, 3, 5, 3, nil)):xreduce`	`3`

An iterator created by Implicit Mapping has a special feature; it will be evaluated automatically when it's destroyed. Consider the following function:

```
f(n:number):map = println('n = ', n)
```

And call it as below:

```
f((3, 1, 4))
```

In Implicit Mapping rule, the call above would simply return an iterator and is supposed not do any process unless the iterator is actually evaluated. But usually, the above case is expected to print values in the iterator at the timing of the function call.
Actually, the code above works as expected because the returned iterator loses any reference from others and is evaluated before destroyed. The script below shows what happens in the above.

```
x = f((3, 1, 4))
x = nil  // iterator is destroyed after printing 'n = 3', 'n = 1' and 'n = 4'.
```

However, the timing to destroy an instance is sometimes unpredictable. It's recommended that you specify `:void` attribute for an instant evaluation.

```
f((3, 1, 4)):void
```

Attributes `:void` , `:reduce` and `:xreduce` don't return an iterator, which cause the actual process on given values done immediately.
It may be the best that you specify `:void` , `:reduce` or `:xreduce` attribute in the function definition if you know beforehand that the function always returns `nil` or other unchanged value.

```
f(n:number):map:void = println('n = ', n)
```

Then, you can call the function with an iterator through Implicit Mapping without any worry.

```
f((3, 1, 4))
```

11.2.6 Suspend Implicit Mapping

A function call with an attribute `:nomap` would suspend Implicit Mapping.
Consider a case that you need to print a content of `x` that contains `[1, 2, 3, 4]` as a list instance. Simply executing `println(x)` would just print each value in the list through Implicit Mapping. To avoid it, put `:nomap` for the call as below.

```
println(x):nomap
```

11.3 Member Mapping

11.3.1 Overview

Member Mapping is a feature to access members of instances that are stored in a list or are generated from an iterator.
There's an instance method `string#len()` that retrieves a length of a string. With a single instance, you can call it like below:

```
x = 'first'
n = x.len()
// n is 5
```

Using a member accessor `::` , you can apply the method on multiple instances in a list.

```
xs = ['first', 'second', 'third', 'fourth']
ns = xs::len()
// ns is [5, 6, 5, 6]
```

A member accessor `:*` creates an iterator that gets results of member access.

```
xs = ['first', 'second', 'third', 'fourth']
ns = xs:*len()
// ns is an iterator that generates (5, 6, 5, 6)
```

11.3.2 Mapping Rule

There are three member accessors in Member Mapping as shown below:

Member Accessor	Name
`::`	map-to-list
`:*`	map-to-iterator
`:&`	map-along

A map-to-list accessor `::` applies a member method or looks up a member variable on instances in an iterable, a list or an iterator, and creates a list of the results. Below shows examples:

```
xs::variable
xs::func()
```

A map-to-iterator accessor `:*` creates an iterator that applies a member method or looks up a member variable on instances in an iterable, a list or an iterator. Below shows examples:

```
xs:*variable
xs:*func()
```

A map-along accessor `:&` only has effect with a member method. It iterates the iterable on the left along with iterables in its argument list following after Iterator Mapping rule. See the following example:

```
xs = [x1, x2, x3]
as = ['first', 'second', 'third']
bs = [3, 1, 4]
xs:&func(as, bs)
```

This has the same effect with shown below:

```
[x1.func('first', 3), x2.func('second', 1), x3.func('third', 4)]
```

The mapping rule with map-along accessor is summarized below:

- If the target iterable or one of the argument values is of iterator species, the result becomes an iterator.
- Otherwise, the result becomes a list.

Chapter 12

Module

12.1 Module as Environment

A **module** is a kind of environment and capable of containing variables and functions inside it. You can use `module()` function that takes a block procedure containing expressions of variable and function assignments. Below is an example:

```
foo = module {
    var:public = 'hello'
    func() = { /* body */ }
}
```

Then, you can call functions and read/modify variables in the module with a member accessing operator `.` specifying the module on its left.

```
foo.func()
println(foo.var)
```

By default, functions defined in a module are marked as public and are accessible from outside. On the other hand, variables in a module are marked as private and would cause an error for an access from outer scope. You have to put `:public` attribute in a variable assignment to make it public.

You can use modules to isolate variables and functions from the current scope by giving them an independent name space. But its main purpose is to provide a mechanism to load external files that extend the language's capability.

12.2 Importing Module File

Gura language has a policy that the interpreter itself should provide functions that are less dependent on external libraries, operating systems and hardware specifications. So, variety of functions such as handling regular expressions, image processing and GUI are realized by dynamically loadable files called **module files**.

There are two types of module files: script module file and binary module file.

Module File	**Suffix**	**Content**
script module file	`.gura`	a usual Gura script file
binary module file	`.gurd`	a dynamic link library that has been compiled from C++ source code

A process of loading a module file and registering its properties to the current environment is called "import". You can use `import()` function in your script to import a module like below:

```
import(re)
```

This loads a module file `re.gurd` and creates a module `re` in the current scope. After importing, functions like `re.match()` and `re.sub()` that the module provides become available.
You can import module properties into the current scope by specifying their symbols in a block of `import()` function.

```
import(re) { match, sub }
```

Then, you can call these functions like `match()` instead of `re.match()`. Specifying `*` in the block will import all of the module properties into the current scope.

```
import(re) { * }
```

Usually, this is not a recommended manner because there's a risk that symbols in a module conflict with ones that already exist. However, it may be a practical way to import some modules like `opengl`, which guarantees all the properties have distinguishable symbols.
You can also import modules at the timing launching the interpreter by specifying a command line option `-i` with module names. Below is an example that imports a module `re` before parsing the script file `foo.gura`.

```
$ gura -i re foo.gura
```

You can specify multiple module names by separating them with a comma character.

```
$ gura -i re,http,png foo.gura
```

Under Windows, the interpreter searches module files in the following path, where `GURA_VERSION` and `GURA_DIR` represent the interpreter's version and the path name in which the program has been installed respectively.

1. Current directory.
2. Directories specified by `-I` option in the command line.
3. Directories specified by environment variable `GURAPATH`.
4. Directory: `%LOCALAPPDATA%\Gura\GURA_VERSION\module`.
5. Directory: `GURA_DIR\module`.
6. Directory: `GURA_DIR\module\site`.

Under Linux, the interpreter searces module files in the following path.

1. Current directory.
2. Directories specified by `-I` option in the command line.
3. Directories specified by environment variable `GURAPATH`.
4. Directory: `$HOME/.gura/GURA_VERSION/module`.
5. Directory: `/usr/lib/gura/GURA_VERSION/module`.
6. Directory: `/usr/lib/gura/GURA_VERSION/module/site`.

A variable `sys.path` is assigned with a list that contains path names to search module files. You can add path names into the list while a script is running.

12.3 Creating Module File

Any script file can be a script module file, which you can import in other scripts. But there are several points you need to know concerning access controls. Consider the following script file named `foo.gura` :

```
var:public = 'hello'
func() = { /* body */ }
```

Then, you can import it to make its properties available.

```
import(foo)
println(foo.var)
foo.func()
```

As with a module created by `module()` function, following rules are applied:

- Functions defined in a module file are marked as public and are accessible from outside. If necessary, you can put `:private` attribute in a function assignment to encapsulate it inside the file.
- Variables defined in a module file are marked as private and would cause an error for an access from outer scope. You have to put `:public` attribute in a variable assignment to make it public.

As a script module file is not different to a usual script file, it can contain any expressions as well other than assignment expressions of function and variable. These expressions are evaluated once, when `import()` function is called.
If a script file is imported as a module, a global variable `__name__` holds its own module name. For instance, a script in `foo.gura` sees the variable with a value `'foo'` when imported. If a script file is parsed by the interpreter firsthand, the variable is set to `'__main__'` . Utilizing this feature, you can write a script in a module file to test its own functions like below:

```
func() = { /* body */ }

if (__name__ == '__main__') {
    func()  // test func()
}
```

Since the body of `if()` function would only be evaluated when the script runs as a main one, you can write codes inside it that wouldn't be evaluated when imported as a module.

12.4 Extensions by Module

Modules don't only provide functions but could enhance various capabilities.

- **Extensions of Existing Class**

  Some modules would provide additional methods to classes that already exists. For example, module `re` would add some methods to `string` class like `string#match()` .

- **Operator**

  Some modules would enhance operators so that they can handle objects the modules provide. For example, a module named `gmp` provides operators on arbitrary precision numbers.

- **Image Format**

  You can use a function `image()` to read a image file. Importing modules that handle image data would expand the function's capability to support additional image formats. For example, after importing `jpeg` module, the function can read a file in JPEG format like following:

```
import(jpeg)
img = image('foo.jpg')
// .. any jobs
```

- **Path Name for Stream**

  You can use a stream instance to access a file stored in a certain storage. While a stream is opened by specifying a path name associated with it, some modules would expand the path name handler so that it can recognize its specific name format. For example, importing a module named `curl` would allow access to a file stored in networks and enhance the path name handler to be able to recognize names that begin with '`http:`'.

```
import(curl)
print(readlines('http://example.com/index.html'))
```

  For another example, module `zip` provides functions to read and write content of ZIP files. and it would make the path name accessible in a ZIP file. The example below prints a content of `doc/readme.txt` that is stored in `foo.zip`.

```
import(zip)
print(readlines('foo.zip/doc/readme.txt'))
```

- **Path Name for Directory**

  Path names in functions that handle directories could also be enhanced by modules.

  A function `path.walk()` recursively retrieves entries in a storage with a specified path name. After importing module `zip`, you can seek entries in a ZIP file using that function.

```
import(zip)
println(path.walk('foo.zip/src'))
```

- **Suffix Handler**

  There's a case that a module will provide additional suffix handlers. For example, module `gmp` can handle suffix `L` that creates an instance of arbitrary precision number from a number literal.

```
import(gmp)
x = 3.1415L * 2 * r
```

- **Character Codec**

  Modules can provide additional handlers for character codec.

12.5 List of Bundled Modules

This section describes a list of modules that are bundled with the interpreter.
Image file format:

Module	Note
`bmp`	handles BMP image file
`gif`	handles GIF image file
`jpeg`	handles JPEG image file
`msico`	handles Microsoft Icon file
`png`	handles PNG image file
`ppm`	handles PPM image file
`tiff`	handles TIFF image file
`xpm`	handles XPM image file

Compression/depression/archiving/hash:

Module	Note
`bzip2`	provides compressor/decompressor functions for bzip2 format
`gzip`	provides compressor/decompressor functions for gzip format
`tar`	provides function to read/write tar archive file
`zip`	provides function to read/write ZIP archive file
`hash`	

Image operation:

Module	Note
`cairo`	provides APIs of Cairo, a 2D graphic library
`freetype`	provides APIs of FreeType, a library to render fonts
`opengl`	provides APIs of OpenGL, a library to render 2D/3D graphics
`glu`	Utility functions for OpenGL

GUI operation:

Module	Note
`sdl`	provides APIs of SDL, a library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D
`tcl`	provides APIs of Tcl interpreter
`tk`	provides APIs of Tk using `tcl` module
`wx`	provides APIs of wxWidgets, a cross-platform GUI library
`show`	provides `image#show()` method that displays image on a window

Audio operation:

Module	Note
`midi`	provides APIs to control MIDI hardware and to create MIDI files

Network operation:

Module	Note
`curl`	provides APIs to access to network using CURL library
`http`	provides APIs for HTTP server and client functions

OS specific:

Module	Note
`conio`	controls console I/O
`mswin`	provides APIs for OLE interface registry access
`msxls`	provides simple classes that handle MS Excel documents
`uuid`	generates UUID

Text file operation:

Module	Note
`csv`	Read/write CSV file
`markdown`	parser of Markdown syntax
`re`	Regular expression
`tokenizer`	provides APIs that tokenize strings
`xml`	XML parser
`xhtml`	XHTML composer
`yaml`	provides APIs to read/write document in YAML format

Mathematical:

Module	Note
`gmp`	provides APIs of GMP, a library for arbitrary precision arithmetic, operating on signed integers, rational numbers, and floating-point numbers.

Database:

Module	Note
`sqlite3`	provides APIs to access to database of sqlite3

Helper to build modules:

Module	Note
`gurcbuild`	provides APIs to create a composite file
`modbuild`	used in a script to build a binary module
`modgen`	generates template files to build a binary module

Utilities:

Module	Note
`argopt`	provides APIs to handle argument options
`calendar`	generates a specified year's calendar
`sed`	replaces strings using regular expression across multiple files
`testutil`	utilities for tester script
`units`	definition of units
`utils`	utilities

12.6 Creating Binary Module File

Gura has a mechanism to support users who create binary modules. This document shows how to create an original binary module hoge.
At first, execute the following command.

```
$ gura -i modgen hoge
```

This would generate a builder script, build.gura, and a template source file of module, Module_hoge.cpp. Although the file Module_hoge.cpp is just a C++ source file that consists of less than 40 lines of codes, it already has an implementation for a Gura function named test.
Executing build.gura would create the module by launching a proper C++ compiler. If you try it in Windows, you need to install Visual Studio 2010 in advance. You may use Express version that is available for free of charge.

```
$ gura build.gura --here
```

If you find a binary module file hoge.gurd has successfully been built in the current directory, import it into Gura's script and test it.

```
$ gura
>>> import(hoge)
>>> dir(hoge)
['__name__, 'test]
>>> hoge.test(3, 5)
8
```

Congratulations! It's ready to edit Module_hoge.cpp for implementations as you like. If you get what you want, execute the following command to install the module into Gura's environment.

```
$ sudo gura build.gura install
```

By the way, you need to get some information about C++ functions and classes provided by Gura for actual programming. The best way for it is to see source files of other binary modules. At first, find out a module from those provided by Gura, which has a function similar to what you want to create. You can find module source files in a directory gura/src/Module_module in a source package. Each module is so simple that consists of one to two source files. I'm sure it's relatively easy to know how to realize your purpose by investigating them, because they have been developed in the same coding policy.

Chapter 13

String and Binary

13.1 Overview

A string is a sequence of character codes in UTF-8 format and is represented by `string` class. Class `string` is a primitive type, which means there's no operation that could modify the content of `string` instances. This leads to the following principles:

- It's not allowed to edit each character in a string content through index access.
- Modification methods are supposed to return a new `string` instance with modified result.

The interpreter itself provides fundamental operations for strings. Importing module named `re` expand the capability so that it can process string data using regular expressions.
Meanwhile, a binary is a byte sequence of data that has any format and is represented by `binary` class. Class `binary` is an object type, so you can modify the content of the instance. A `binary` instance can be used as a plain memory image capable of containing any data.

13.2 Operation on String

13.2.1 Character Manipulation

You can specify an index number starting from zero embraced by a pair of square brackets to retrieve a character as a sub string at the specified position. Multiple numbers for indexing can also be specified to get a list of sub strings.

```
str = 'abcdefghijklmnopqrstuvwxyz'
str[6]           // returns 'g'
str[20]          // returns 'u'
str[17]          // returns 'r'
str[0]           // returns 'a'
str[6, 20, 17, 0] // returns ['g', 'u', 'r', 'a']
```

You can also specify iterators and lists to get a list of sub strings. Numbers and iterators can be mixed together as indexing items.

```
str = 'The quick brown fox jumps over the lazy dog'
str[10..14]       // returns ['b', 'r', 'o', 'w', 'n']
str[4..8, 35..38] // returns ['q', 'u', 'i', 'c', 'k', 'l', 'a', 'z', 'y']
```

If you specify an infinite iterator as an indexing item, you would get sub strings within an available range.

```
str = 'The quick brown fox jumps over the lazy dog'
str[35..]       // returns ['l', 'a', 'z', 'y', ' ', 'd', 'o', 'g']
```

An index with a negative number points the position from the bottom, where `-1` is the last position.

```
str = 'The quick brown fox jumps over the lazy dog'
str[-3]        // returns 'd'
str[-2]        // returns 'o'
str[-1]        // returns 'g'
```

Function `chr()` returns a string that contains a character of the given UTF-8 character code.

```
chr(65)        // returns 'A'
```

Function `ord()` takes a string and returns UTF-8 character code of its first character.

```
ord('A')       // returns 65
```

13.2.2 Iteration

Method `string#each()` creates an iterator that returns each character as a sub string.

```
str = 'The quick brown fox jumps over the lazy dog'
x = str.each()
// x is an iterator that returns 'T', 'h', 'e' ...
```

A call of `string#each()` with attribute `:utf8` or `:utf32` would create an iterator that returns character code numbers in UTF-8 or UTF-32 instead of sub strings.

```
str = 'XXX'  // assumes it contains kanji characters 'ni-hon-go'
x = str.each():utf8
// x is an iterator that returns 0xe697a5, 0xe69cac and 0xe7aa9e

x = str.each():utf32
// x is an iterator that returns 0x65e5, 0x672c and 0x8a9e
```

Method `string#eachline()` creates an iterator that splits a string by a newline character and returns strings of each line.

```
str = R'''
1st
2nd
3rd
'''
lines = str.eachline()
// lines is an iterator that returns '1st\n', '2nd\n' and '3rd\n'
```

Method `string#chop()` is useful when you want to remove a newline character appended at the bottom.

```
x = str.eachline()
lines = x:*chop()  // an iterator to apply string#chop() to each value in x
// lines is an iterator that returns '1st', '2nd' and '3rd'
```

Method `string#eachline()` and others that split a multi-lined text into strings of each line like `readlines()` are equipped with an attribute `:chop` that applies the same process as `string#chop()` .

```
lines = str.eachline():chop
// lines is an iterator that returns '1st', '2nd' and '3rd'
```

Method `string#split()` creates an iterator that splits a string by a separator string specified in the argument.

```
str = 'The quick brown fox jumps over the lazy dog'
x = str.split(' ')
// x is an iterator that returns 'The', 'quick', 'brown', 'fox' ...
```

If you want to split a string into segments with the same length, use `string#fold()` method.

```
str = 'abcdefghijklmnopqrstuvwxyz'
x = str.fold(5)
// x is an iterator that returns 'abcde', 'fghij', 'klmno', 'pqrst', 'uvwxy' and 'z'
```

13.2.3 Modification and Conversion

Applying an operator `+` between two `string` instances would concatenate them together.

```
str1 = 'abcd'
str2 = 'efgh'
str1 + str2   // returns 'abcdefgh'
```

An operator $*$ between a `string` and a `number` value would concatenate the string the specified number of times.

```
str = 'abcd'
str * 3      // returns 'abcdabcdabcd'
```

Method `list#join()` joins all the string in the list and returns the result. If it contains elements other than `string` , they're converted to strings before joined.

```
['abcd', 'efgh', 'ijkl'].join()    // returns 'abcdefghijkl'
```

The method can take a separator string as its argument that is inserted between elements.

```
['abcd', 'efgh', 'ijkl'].join(', ') // returns 'abcd, efgh, ijkl'
```

Method `string#capitalize()` returns a string with the top alphabet converted to uppper case.

```
str = 'hello, WORLD'
str.capitalize()  // returns 'Hello, WORLD'
```

Methods `string#upper()` and `string#lower()` return a string after converting all the alphabet characters to upper and lower case respectively.

```
str = 'hello, WORLD'
str.upper()       // returns 'HELLO, WORLD'
str.lower()       // returns 'hello, world'
```

Method `string#binary()` returns a `binary` instance that contains a binary sequence of the string in UTF-8 format.

```
str = 'XXX'    // assumes it contains kanji characters 'ni-hon-go'
str..binary()  // returns a binary b'\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e'
```

You can use `string#encode()` to get a binary sequence in other codec other than UTF-8.

```
str = 'XXX'              // assumes it contains kanji characters 'ni-hon-go'
str.encode('shift_jis')  // returns a b'\x93\xfa\x96\x7b\x8c\xea'
```

Method `string#reader()` returns a `stream` instance that reads a binary sequence of the string in UTF-8 format.

```
str = 'The quick brown fox jumps over the lazy dog'
x = str.reader()
// x is a stream instance for reading
```

Method `string#encodeuri()` converts characters that can not be described in URI by a percent-encoding rule, while method `string#decodeuri()` converts such encoded string into normal characters.
Method `string#escapehtml()` escapes characters that can not be described in HTML with character entities prefixed by an ampersand, while method `string#unescapehtml()` converts such escaped ones into normal characters.

13.2.4 Extraction

Method `string#strip()` removes space characters that exist on both sides of the string. Attributes `:left` and `:right` would specify the side to remove spaces.

```
str = '    hello   '
str.strip()        // returns 'hello'
str.strip():left   // returns 'hello   '
str.strip():right  // returns '    hello'
```

Method `string#left()` returns a sub string that has extracted specified number of characters from the left side, while method `string#right()` extracts from the right side.

```
str = 'The quick brown fox jumps over the lazy dog'
str.left(3)  // returns 'The'
str.right(3) // returns 'dog'
```

Method `string#mid()` returns a sub string that has extracted specified number of characters from the specified position.

```
str = 'The quick brown fox jumps over the lazy dog'
str.mid(10, 5)  // returns 'brown'
```

13.2.5 Search, Replace and Inspection

To see the length of a string, `string#len()` is available. Note that `string#len()` returns the number of characters, not the size in byte.

```
str = 'abcdefghijklmnopqrstuvwxyz'
n = str.len()
// n is 26
```

Method `string#find()` searches the specified sub string in the target string and returns the found position starting from zero. If not found, it returns `nil` .

```
str = 'The quick brown fox jumps over the lazy dog'
str.find('fox')  // returns 16
str.find('cat')  // returns nil
```

Method `string#replace()` replaces the sub string with the specified one.

```
str = 'The quick brown fox jumps over the lazy dog'
str.replace('fox', 'cat') // returns 'The quick brown cat jumps over the lazy dog'
```

Method `string#startswith()` returns `ture` if the string starts with the specified sub string, and returns `false` otherwise. Method `string#endswith()` checks if the string ends with the specified sub string.

```
str = 'abcdefghijklmnopqrstuvwxyz'
str.startswith('abcde') // returns true
str.startswith('hoge')  // returns false
str.endswith('vwxyz')   // returns true
str.endswith('hoge')    // returns false
```

Specifying an attribute `:rest` indicates that these functions return a string excluding the specified sub string when that matches the head or the bottom part. If the sub string doesn't match, they would return `nil` .

```
str.startswith('abcde):rest // returns 'fghijklmnopqrstuvwxyz'
str.startswith('hoge'):rest // returns nil
str.endswith('vwxyz'):rest  // returns 'abcdefghijklmnopqrstu'
str.endswith('hoge'):rest   // returns nil
```

13.3 Formatter

13.4 Functions Equipped with Formatter

You can use format specifiers in some functions that are similar to what are realized in C language's `printf` to convert objects like numbers into readable strings.
Function `printf()` takes a string containing format specifiers and values you want to print in its argument list and put the result out to `sys.stdout` stream.

```
printf('x = %d, y = %d\n', x, y)
```

Method `stream#printf()` has the same argument declaration with `printf()` and puts the result to the target stream capable of writing instead of `sys.stdout` stream.

```
open('foo.txt', 'w').printf('x = %d, y = %d\n', x, y)
```

Method `list#printf()` is another form of `printf()` , which takes values to print in the list of the target instance, not in the argument list.

```
[x, y].printf('x = %d, y = %d\n')
```

Function `format()` takes arguments in the same way as `printf()` but it returns the result as a `string` instance.

```
str = format('x = %d, y = %d\n', x, y)
```

You can also use `%` operator to get the same result with `format()` , which takes a format string on the left and a list containing values for printing on the right.

```
str = 'x = %d, y = %d\n' % [x, y]
```

If there's only one value for printing, you can even give the operator the value without a list.

```
str = 'x = %d\n' % x
```

13.5 Syntax of Format Specifier

A format specifier begins with a percent character and has the syntax below, where optional fields are embraced by square brackets:

```
%[flags][width][.precision]specifier
```

You always have to specify one of the following characters for the `specifier` field.

specifier	Note
`d` , `i`	decimal integer number with a sign mark
`u`	decimal integer number wihout a sign mark
`b`	binary integer number without a sign mark
`o`	octal integer number without a sign mark
`x`	hexadecimal integer number in lower character without a sign mark
`X`	hexadecimal integer number in upper character without a sign mark
`e`	floating number in exponential form
`E`	floating number in exponential form (in upper character)
`f`	floating number in decimal form
`g`	better form between e and f
`G`	better form between E and F
`s`	string
`c`	character

You can specify one of the following characters for the optional `flags` field.

flags	Note
`+`	`+` precedes for positive numbers
`-`	adjust a string to left
(space)	space character precedes for positive numbers
`#`	converted results of binary, octdecimal and hexadecimal are preceded by `'0b'` , `'0'` and `'0x'` respectively
`0`	fill lacking columns with `'0'`

The optional field `width` takes a decimal number that specifies a minimum width for the corresponding value. If the value's length is shorter than the specified width, the rest would be filled with space characters. If you specify `*` for that field, the formatter would try to get the minimum width from the argument list.
The optional field `precision` has different meanings depending on the specifier as below:

specifier	Note
`d` , `i` , `u` , `b` , `o` , `x` , `X`	It specifies the minimum number of digits. If the value is shorter than this number, lacking digits are filled with zero.
`e` , `E` , `f`	It specifies the number of digits after a decimal point.
`g` , `G`	It specifies the maximum number of digits for significand part.
`s`	It specifies the maximum number of characters to print.

13.6 Regular Expression

You can import module `re` to use regular expression for string search and substition, which supports a syntax based on POSIX Extended Regular Expression.
Importing module `re` would equip `string` class with methods that can handle regular expressions. See the sample code below:

```
import(re)

str = '12:34:56'

m = str.match(r'(\d\d):(\d\d):(\d\d)')
if (m) {
    printf('hour=%s, min=%s, sec=%s\n', m[1], m[2], m[3])
} else {
    println('not match')
}
```

Method `string#match()` that is provided by `re` module takes a regular expression pattern. It would return `re.match` instance if the pattern matches, and return `nil` otherwise. As regular expressions often contain back slash as a meta character, it would be convenient to use an expression `r' ... '` for a pattern string to avoid recognizing a backslash as an escaping character.
An instance of `re.match` contains information about matching result. It supports indexing access where `m[0]` has a string that matches the whole pattern and `m[1]` , `m[2]` ... returns a string of each group. You can specify a string instead of a number to index each group when you use `?<name>` specifier for the group in a regular expression pattern.

```
m = str.match(r'(?<hour>\d\d):(?<min>\d\d):(?<sec>\d\d)')
if (m) {
    printf('hour=%s, min=%s, sec=%s\n', m['hour'], m['min'], m['sec'])
} else {
    println('not match')
}
```

Although you can pass a string containing a pattern to method `string#match()` , it actually takes `re.pattern` instance in its argument that is capable of accepting a `string` instance through casting feature. Above example is equivalent with below:

```
pat = re.pattern(r'(\d\d):(\d\d):(\d\d)')
m = str.match(pat)
```

When you give a string to a function or a method that expects `re.pattern`, it always compile the string to create `re.pattern` instance, which may cause some overhead in a process of huge amount of data. In such a case, it may be a good idea to call a function with a `re.pattern` instance that has explicitly been created by `re.pattern()` function in advance like shown above.
Method `string#sub()` takes a `re.pattern` instance and replaces the matched part with the given substitution value.
A subsitution item can be either a string or a function. When you give a string for it, the method replaces the matched part with the string.

```
str = 'The quick brown fox jumps over the lazy dog'
str.sub(r'[Tt]he', 'THE') // returns 'THE quick brown fox jumps over THE lazy dog'
```

You can specify a group reference \ *n* in a subsitution string where *n* indicates the group index.
If you specify a function for the substitution value, which takes a `re.match` value as its argument and to return a substitution string, it would be called when the matching succeeds.

```
str = '### #### ##### ## #####'
f(m:re.match) = format('%d', m[0].len())
str.sub('#+', f)                              // returns '3 4 5 2 5'
```

An anonymous function would make the code more simple.

```
str = '### #### ##### ## #####'
str.sub('#+', &{format('%d', $m[0].len())})  // returns '3 4 5 2 5'
```

13.7 Operation on Binary

13.7.1 Creation of Instance

You can create a `binary` instance by put a prefix `b` to a string literal.

```
b'AB\x01\x00\xff'
```

The example above is a `binary` instance that contains a sequence of byte data: 0x41, 0x42, 0x01, 0x00 and 0xff. As an instance created by a string literal prefixed by `b` can not be modified, it would occur an error when you try some modification operations on such an instance.
There are several ways to create a `binary` instance that accepts modification.

- Constructor function `binary()` creates an empty `binary` instance.

  ```
  buff = binary()
  ```

- Class method `binary.alloc()` creates a `binary` instance of the specified size.

  ```
  buff = binary.alloc(1000)
  // buff has a memory of 1000 bytes
  ```

- Class method `binary.pack()` packs values into a binary sequence according to the packing specifier.

  ```
  buff = binary.pack('Bl', 0xaa, 0x12345678)
  // buff has a byte sequence: 0xaa, 0x78, 0x56, 0x34, 0x12.
  ```

You can use method `binary#dump()` to print out a content of a `binary` in a hexadecimal dump format.

13.7.2 Byte Manipulation

An index access on a `binary` would return a number value at the specified position.

```
buff = b'\xaa\xbb\xcc\xdd\xee'
buff[0] // returns 0xaa
buff[1] // returns 0xbb
```

You can also specify an iterator as an indexing item for a binary just like a string.

```
buff[1..3] // returns [0xbb, 0xcc, 0xdd]
```

Modification through an indexer on a writable binary is also possible.

```
buff = binary.alloc(8)
buff[0] = 0x12
buff[1] = 0x34
buff[3..] = 0..4
// buff has a byte sequence: 0x12, 0x34, 0x00, 0x00, 0x01, 0x02, 0x03, 0x04.
```

Method `binary#each()` creates an iterator that returns each 8-bit number value in the binary.

```
buff = b'\xaa\xbb\xcc\xdd\xee'
x = buff.each()
// x is an iterator that returns 0xaa, 0xbb, 0xcc, 0xdd and 0xee.
```

13.7.3 Pack and Unpack

Using an indexer and `binary#each()` method, you can retrieve and modify the content of a binary by a unit of 8-bit number. To store and extract values such as number that consits of multiple octets and string that contains a sequence of character codes, the following methods are provided.

- Class method `binary.pack()` to create a binary sequence that contains numbers and strings.
- Method `binary#unpack()` to extract numbers and strings from a binary sequence.

Class method `binary.pack()` takes a formatter string containing specifiers and values to store as its argument. For example:

```
rtn = binary.pack('H', 0x1234)
```

The specifier `H` means an unsigned 16-bit number, so the result `rtn` is a `binary` instance that contains a binary sequence of 0x34 and 0x12.
You can write any number of specifiers in the format.

```
rtn = binary.pack('HHH', 0x1234, 0xaabb, 0x5678)
```

The result contains a binary sequence of 0x34, 0x12, 0xbb, 0xaa, 0x78 and 0x56.
If there's a sequence of the same specifier like above, you can brackets them together by specifying the number ahead of that specifier.

```
rtn = binary.pack('3H', 0x1234, 0xaabb, 0x5678)
```

This has the same result as the previous example.
Meanwhile, method `binary#unpack()` takes a formatter string and returns a list containing unpacked result. For example:

```
buff = b'\x34\x12'
rtn = buff.unpack('H')
```

The result `rtn` is a list `[0x1234]` . Note that you always get a list as the result even when it contains only one value.
Below is an example of a format that contains multiple specifiers:

```
buff = b'\x34\x12\xbb\xaa\x78\x56'
rtn = buff.unpack('HHH')
// rtn is [0x1234, 0xaabb, 0x5678]
```

Just like the packing rule, you can specify the number of succeeding specifiers.

```
buff = b'\x34\x12\xbb\xaa\x78\x56'
rtn = buff.unpack('3H')
```

Using an assignment to lister expression may often be helpful, since you can assign extracted values to independent variables.

```
buff = b'\x34\x12\xbb\xaa\x78\x56'
[x, y, z] = buff.unpack('3H')
```

The table below summarizes specifiers that are used to pack or unpack number values.

Specifier	**Unit Size**	**Note**
`b`	1 byte	Packs or unpacks a signed 8-bit number (-128 to 127).
`B`	1 byte	Packs or unpacks an unsigned 8-bit number (0 to 255)
`h`	2 bytes	Packs or unpacks a signed 16-bit number (-32768 to 32767)
`H`	2 bytes	Packs or unpacks an unsigned 16-bit number (0 to 65535)
`i`	4 bytes	Packs or unpacks a signed 32-bit number (-2147483648 to 2147483648)
`I`	4 bytes	Packs or unpacks an unsigned 32-bit number (0 to 4294967295)
`l`	4 bytes	Packs or unpacks a signed 32-bit number (-2147483648 to 2147483648)
`L`	4 bytes	Packs or unpacks an unsigned 32-bit number (0 to 4294967295)
`q`	8 bytes	Packs or unpacks a signed 64-bit number (-9223372036854775808 to 9223372036854775807)
`Q`	8 bytes	Packs or unpacks an unsigned 64-bit number (0 to 18446744073709551615)
`f`	4 bytes	Packs or unpacks a single precision floating point number.
`d`	8 bytes	Packs or unpacks a double precision floating point number.

By default, byte order of numbers in 16-bit, 32-bit and 64-bit size is a little endian. You can change the order by using the following specifiers:

Specifier	**Note**
`@`	Turns to a system-dependent endian.
`=`	Turns to a system-dependent endian.
	Turns to a little endian.
	Turns to a big endian.
`!`	Turns to a big endian.

```
rtn = binary.pack('H>H', 0x1234, 0x1234)
// rtn contains 0x34, 0x12, 0x12, 0x34.
```

Specifier `x` only advances pointer ahead for specified size without packing or unpacking of values. When packing, the skipped area would be filled with zero.

```
rtn = binary.pack('H3xH', 0x1234, 0x1234)
// rtn contains 0x34, 0x12, 0x00, 0x00, 0x00, 0x34, 0x12.
```

Specifiers `c` and `s` are prepared to pack or unpack string data.

Specifier	**Note**
`c`	Packs a first character code in a string, or unpack a 8-bit number as a chracter code and returns a string containing it.
`s`	Packs character codes in a string according to the specified codec, or unpack 8-bit numbers as character codes according the specified codec and returns a string containing them.

You can specify a codec for `s` specifier by surrounding its name with { and } .

13.7.4 Pointer

`binary#pointer()`
`pointer#unpack()`
`pointer#pack()`

13.7.5 Binary as Stream

`binary#writer()`
`binary#reader()`
cast from `binary` to `stream`

Chapter 14

Iterator/List Operation

14.1 Overview

An iterator and a list are quite similar in terms of handling multiple values in a flat structure. In fact, many of their methods share the same names and functions each other.
The difference is that a list is a container that actually owns its element values while an iterator only provides a method that retrieves a "next" value of a sequence and doesn't necessarily have to own values. This feature leads to the following principles:

- An iterator can handle a sequence of data that continues indefinitely because it doesn't need to keep all the values in it.
- An iterator consumes less memory than a list in many cases.
- A list provides an indexing method that enables random access for its elements.
- A list provides methods to append or remove values.

Note that Gura makes it a rule to implement most functions to return an iterator by default if they have multiple values as its result. Even with such functions, you can easily get a list as their result by calling it with `:list` attribute.

14.2 Iteration on Iterators and Lists

Consider a task that prints elements in the list shown below:

```
words = ['one', 'two', 'three', 'four', 'five', 'six', 'seven', 'eight', 'nine', 'ten']
```

There are several ways to iterate elements in an iterator or a list.

- As you've already seen a previous chapter, iterators and lists can work with functions, methods and operators through Implicit Mapping. You can simply call `printf()` function with iterators or lists that causes a repetitive evaluation of the function.

  ```
  printf('%s\n', words)
  ```

  A function with Implicit Mapping is capable of iterating multiple iterables provided as its arguments. In addition to the list of words, you can specify an iterator that generates numbers starting from zero to print indexing numbers as shown below.

  ```
  printf('%d: %s\n', 0.., words)
  ```

- Using `for()` function, you can iterate a list or an iterator in a way that you may have been familiar with in other languages.

```
for (word in words) {
    printf('%s\n', word)
}
```

  You can get a loop index starting from zero by specifying a block parameter.

```
for (word in words) {|i|
    printf('%d: %s\n', i, word)
}
```

- You can also use method `iterator#each()` or `list#each()` to iterate elements on them. In this case, the block parameter contains an iterated element as its first value.

```
words.each {|word|
    printf('%s\n', word)
}
```

  It provides a loop index as the second value in the block parameters as below.

```
words.each {|word, i|
    printf('%d: %s\n', i, word)
}
```

Most functions and methods that return an iterator as their result are designed to iterate elements when they take a block. Actually, methods `iterator#each()` and `list#each()` , which are mentioned above, simply return an iterator when they're called without a block.

```
rtn = words.each()
// rtn is an iterator that iterates each element in words
```

To see other examples that have the same feature, consider methods `iterator#filter()` and `list#filter()` , which returns an iterator that pick up elements satisfying a criteria specified in the argument.

```
rtn = words.filter(&{$word.startswith('t')})
// rtn is an iterator that generates 'two', 'three' and 'ten'
```

Specifying a block with the method would repetitively evaluate it while iterating elements of the result.

```
words.filter(&{$word.startswith('t')}) {|word, i|
    printf('%d: %s\n', i, word)
}
```

The result comes as below:

```
0: two
1: three
2: ten
```

14.3 Iterator-specific Manipulation

14.3.1 About This Section

This section explains about methods and ohter manipulation that are specific to iterators.

14.3.2 Finite Iterator vs. Infinite Iterator

Iterators that generate a limited numer of elements are called Finite Iterator. An iterator `0..5` is a representative one that is definitely expected to generate 6 elements. It's possible that you convert a Finite Iterator into a list.
Iterators that generate elements indefinitely or couldn't predict when elements drain out are called Infinite Iterator. Among them, there's an iterator `0..` that generates numbers starting from 0 and increasing for ever. It would occur an error if you try to convert Infinite Iterator into a list.
You can use method `iterator#isinfinite()` to check if an iterator is an infinite one or not.

```
(0..5).isinfinite()  // returns false
(0..).isinfinite()   // returns true
```

Some functions may possibly create either Finite or Infinite Iterator depending on their arguments. The second argument in function `rands()` specify how many random values it should generate, and, if omitted, the function would generate values without end.

```
rands(100)     // returns an Infinite Iterator
rands(100, 80) // returns a Finite Iterator that is expected to generate 80 elements
```

Infinity of the result of function `readlines()` depends on the attribute of the source stream: it would be an Infinite Iterator if the stream is infinite while it would be a Finite Iterator for a finite stream.
An iterator's infinity may be derived from one to another. This happens with iterators that are designed to manipulate values after retrieving them from other source iterator. For example, method `iterator#filter()` returns an iterator that picks up elements based on the given criteria. In the following code, `y` is a Finite Iterator that generates numbers 0, 2, 4, 6, 8 and 10.

```
tbl = 0..10
rtn = tbl.filter(&{$x % 2 == 0})
// rtn is finite
```

If the source iterator is infinite, the result iterator will be infinite too. In the code below, `y` is an Infinite Iterator that generates even numbers indefinitely.

```
tbl = 0..
rtn = tbl.filter(&{$x % 2 == 0})
// rtn is infinite
```

14.3.3 Conversion into List

Embracing iterators with a pair of square brackets would make a list from them.

```
[0..5]               // creates [0, 1, 2, 3, 4, 5]
```

You can specify any numbers of iterators in it as below.

```
[0..2, 5..7, 8..10]  // creates [0, 1, 2, 5, 6, 7, 8, 9, 10]
```

It would occur an error if you try to create a list from Infinite Iterators.

```
[0..]                // error!
```

Another way to create a list from an iterator is to use `iterator#each()` method with `:list` attribute.

```
tbl = 0..5
tbl.each():list      // returns [0, 1, 2, 3, 4, 5]
```

14.3.4 Operation on Elements

You can retrieve elements from an iterator by using method `iterator#next()` .

```
tbl = 0..5
tbl.next()   // returns 0
tbl.next()   // returns 1
tbl.next()   // returns 2
```

14.4 List-specific Manipulation

14.4.1 About This Section

This section explains about methods and ohter manipulation that are specific to lists.

14.4.2 Indexing Read from List

You can specify an index number starting from zero embraced by a pair of square brackets to retrieve an element at the specified position. Multiple numbers for indexing can also be specified to get a list of elements.

```
tbl = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven']
tbl[2]        // returns 'two'
tbl[4]        // returns 'four'
tbl[1, 3, 5]  // returns ['one', 'three', 'five']
```

You can also specify iterators and lists to get a list of elements. Numbers and iterators can be mixed together as indexing items.

```
tbl = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven']
tbl[2..4]        // returns ['two', 'three', 'four']
tbl[1..3, 5..7]  // returns ['one', 'two', 'three', 'five', 'six', 'seven']
```

If you specify an infinite iterator as an indexing item, you would get elements within an available range.

```
tbl = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven']
tbl[5..]        // returns ['five', 'six', 'seven']
```

An index with a negative number points the position from the bottom, where `-1` is the last position.

```
tbl = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven']
tbl[-1]         // returns 'seven'
tbl[-2]         // returns 'six'
```

Method `list#first()` returns the first item in the list and method `list#last()` the last item. These have the same effect with index accesses by numbers 0 and -1 respectively.

```
tbl = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven']
tbl.first()     // returns 'zero'
tbl.last()      // returns 'seven'
```

You can use method `list#get()` for index access, which would be useful when used with Member Mapping.

```
tbl = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
tbl::get(0)     // returns [1, 4, 7]
```

14.4.3 Indexing Modification on List

An assignment to elements in a list through indexing access is also available.
If an indexing item is a single number, the element at the specified position will be modified.

```
tbl = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven']
tbl[2] = '2'
tbl[4] = '4'
// tbl is ['zero', 'one', '2', 'three', '4', 'five', 'six', 'seven']
```

Multiple numbers can also be specified for indexing. In this case, if the assigned value is an iterable, each element in the iterable will be stored at the specified positions in the target list.

```
tbl = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven']
tbl[1, 3, 5] = ['1', '3', '5']
// tbl is ['zero', '1', 'two', '3', 'four', '5', 'six', 'seven']
```

If the assigned value is a scalar, the same value is stored at the positions.

```
tbl = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven']
tbl[1, 3, 5] = '1'
// tbl is ['zero', '1', 'two', '1', 'four', '1', 'six', 'seven']
```

You can also specify an iterator as indexing item.

```
tbl = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven']
tbl[1..3, 5..7] = ['1', '2', '3', '5', '6', '7']
// tbl is ['zero', '1', '2', '3', 'four', '5', '6', '7']
```

When you specify an Infinite Iterator for an indexing item, all the elements in the assigned iterable are stored at the specified position.

```
tbl = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven']
tbl[5..] = ['5', '6']
// tbl is ['zero', 'one', 'two', 'three', 'four', '5', '6', 'seven']
```

Negative number can also be specified for indexing.

```
tbl = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven']
tbl[-1] = '7'
tbl[-2] = '6'
// tbl is ['zero', 'one', 'two', 'three', 'four', 'five', '6', '7']
```

You can use method `list#put()` for index modification, which would be useful when used with Member Mapping.

```
tbl = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
tbl::put(2, 99)
// tbl is [[1, 2, 99], [4, 5, 99], [7, 8, 99]]
```

14.4.4 Conversion into Iterator

Method `list#each()` returns an iterator that generates values based on the list's elements.

```
tbl = ['one', 'two', 'three', 'four']
rtn = tbl.each()
// rtn is an iterator that generates 'one', 'two', 'three' and 'four'.
```

14.4.5 Operation on Elements

Method `list#isempty()` will check if a list is empty or not.

```
tbl = []
tbl.isempty()    // returns true
```

Both of methods `list#add()` and `list#append()` will add values to the target list. They have the same behavior when they try to add a scalar value. Below is a sample of `list#add()` :

```
tbl = ['one', 'two', 'three']
tbl.add('four')
// tbl is ['one', 'two', 'three', 'four']
```

And a sample of `list#append()` is shown below:

```
tbl = ['one', 'two', 'three']
tbl.append('four')
// tbl is ['one', 'two', 'three', 'four']
```

They have different results when they're given with a list as an element to add. Method `list#add()` adds the list itself to the target list as one of its elements.

```
tbl = ['one', 'two', 'three']
tbl.add(['four', 'five', 'six'])
// tbl is ['one', 'two', 'three', ['four', 'five', 'six']]
```

Method `list#append()` adds each of the list's element to the target list.

```
tbl = ['one', 'two', 'three']
tbl.append(['four', 'five', 'six'])
// tbl is ['one', 'two', 'three', 'four', 'five', 'six']
```

Method `list#clear()` will create all the contet of the target list.

```
tbl = ['one', 'two', 'three']
tbl.clear()
// tbl is []
```

Method `list#erase()` will erase elements at positions specified by its arguments. You can specify multiple indices at which elements are erased.

```
tbl = ['zero', 'one', 'two', 'three', 'four', 'five', 'six', 'seven']
tbl.erase(2, 4, 6)
// tbl is ['zero', 'one', 'three', 'five', 'seven']
```

Method `list#shift()` erase the first element before it returns that value.

```
tbl = ['one', 'two', 'three']
rtn = tbl.shift()  // returns 'one'
// tbl is ['two', 'three']
```

`list#flat()`
`list.zip()`

14.5 Common Manipulation for Iterator and List

14.5.1 About This Section

This section explains about methods that are prepared for both iterators and lists. To make descriptions simple, a pseudo class name `iterable` is used to represent `list` or `iterator` class. For example, `iterable#len()` is an inclusive term for `list#len()` and `iterator#len()` .

14.5.2 Inspection and Reduce

Method `iterable#len()` return the number of elements in the iterable.
Method `iterable#count()` takes an optional argument `criteria` with which elements would be filtered out, and return the number of elements matching it. The method behaves differently depends on a value given to `criteria` .

- If no value is specified for `criteria` , it would return the number of elements that can be determined as `true` .

  ```
  [true, false, true, true}.count()          // returns 3
  ```

- If it takes a `function` , which takes one argument and returns a boolean value, it would call the given function with each element's value and count the number of `true` returned from it.

  ```
  [3, 1, 4, 1, 5, 9, 2, 6].count(&{$x < 4}) // returns 4
  ```

- If it takes a value other than `function` , it would return the number of elements that equals to the given value.

  ```
  [3, 1, 4, 1, 5, 9, 2, 6].count(1)         // returns 2
  ```

Method `iterable#contains()` checks if the iterable contains the specified value in it.

```
tbl = [3, 1, 4, 1, 5, 9, 2, 6]
tbl.contains(1)  // return true
tbl.contains(7)  // return false
```

Methods `iterable#and()` and `iterable#or()` calculate logical AND and OR on the iterable's elements repectively. It regards `false` and `nil` as a false state, and other values as a true.

```
[true, true, true].and()   // returns true
[true, false, true].and()  // returns false
[3, 1, 4, 1, 5].and()      // returns true
[true, false, true].or()   // returns true
[nil, false, nil].or()     // returns false
```

Classes `list` and `iterator` are equipped with some statistical operations described below:

- `iterable#sum()` calculates summation of elements in the iterable.
- `iterable#average()` calculates an average of elements in the iterable.
- `iterable#stddev()` calculates a standard deviation value of elements in the iterable.
- `iterable#variance()` calculates a variance value of elements in the iterable.
- `iterable#max()` and `iterable#min()` returns maximum and minimum value in the iterable.

Method `iterable#join()` would join all the strings in the iterable and returns the result. If an element is not a `string` instance, it would be converted to a string before joined. It takes an optional argument that specifies a string inserted between adjacent elements.

```
['abc', 'def', 'ghi'].join()      // returns 'abcdefhij'
['abc', 'def', 'ghi'].join('#')   // returns 'abc#def#hij'
```

Method `iterable#reduce()` is a generic one to summarize information from elements. It takes a block procedure that is evaluated for each element with block parameters `|x, accum|`, where `x` takes each element value and `accum` the result of the previous evaluation of the block. The initial value of `accum` is specified by the method's argument. For example, you can use `iterable#reduce()` to implement a function that works similar with `iterable#sum()` as below.

```
my_sum(iter) = iter.reduce(0) {|x, accum| x + accum }
```

`iterable#find()`

14.5.3 Mapping Method

Method `iterable#nilto()` returns an iterator that replaces `nil` existing in the iterable into a specified value. Note that this method doesn't modify the target list.

```
rtn = [nil, 1, 2, nil, 3, 4].nilto(99)
// rtn is an iterator that generates 99, 1, 2, 99, 3, 4.
```

Method `iterable#replace()` returns an iterator that replaces elements matching to its first argument with the value of its second argument. Note that this method doesn't modify the target list.

```
rtn = [3, 1, 4, 1, 5, 9, 2, 6].replace(1, 99)
// rtn is an iterator that generates 3, 99, 4, 99, 5, 9, 2, 6.
```

Method `iterable#rank()` returns an iterator that generates ranked number for each element after sorted. The argument `directive` specifies sorting rule, which is described in a document of `iterable#sort()` .

```
rtn = ['apple', 'grape', 'orange', 'banana'].rank()
// rtn is an iterator that generates 0, 2, 3, 1
```

Method `iterable#map()` returns an iterator that applies a function on each element. In general, you can use Implicit Mapping to get the same result with this method.

14.5.4 Element Manipulation

This subsection explains about methods that changes the order of elements and extracts elements by a certain condition.
Method `iterable#align()` creates an iterator that picks up the specified number of elements from the iterable.

```
rtn = [3, 1. 4, 1, 5, 9].align(3)
// rtn is an iterator that generates 3, 1, 4.
```

If the specified number is more than the length of the source iterable, the rests are filled with `nil` value.

```
rtn = [3, 1. 4, 1, 5, 9].align(10)
// rtn is an iterator that generates 3, 1, 4, 1, 5, 9, nil, nil, nil, nil.
```

Method `iterable#fold()` creates an iterator that segments the iterable into group of lists containing the specified number of elements.

```
rtn = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3].fold(3)
// rtn is an iterator that generates [3, 1, 4], [1, 5, 9], [2, 6, 5], [3].
```

Method `iterable#filter()` returns an iterator that picks up elements where the given argument `criteria` , a function or an iterable, is evaluated as `true` .
A function for `criteria` has a single argument that takes each element value and returns `true` when it wants the value picked up.

```
f(x) = x < 4
tbl = [3, 1, 4, 1, 5, 9, 2]
rtn = tbl.filter(f)
// rtn is an iterator that generates 3, 1, 1, 2.
```

Using an anonymous function would make the code more simple.

```
tbl = [3, 1, 4, 1, 5, 9, 2]
rtn = tbl.filter(&{$x < 4})
// rtn is an iterator that generates 3, 1, 1, 2.
```

Method `iterable#filter()` can also take an iterator or a list of `boolean` elements as the `criteria` . Using this feature, you can call the function as below:

```
tbl = [3, 1, 4, 1, 5, 9, 2]
rtn = tbl.filter(tbl < 4)
// rtn is an iterator that generates 3, 1, 1, 2.
```

Implicit Mapping works on the expression `tbl < 4` that creates a list `[true, true, false, true, false, false, true]` . Then, the method picks up elements of which corresponding boolean value is `true` .
Method `iterable#skipnil()` creates an iterator that skips `nil` value.

```
rtn = [3, 1, nil, 4, 1, nil, nil, 5].skipnil()
// rtn is an iterator that generates 3, 1, 4, 1, 5.
```

Method `iterable#skip()` creates an iterator that skip the specified number between elements.

```
rtn = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3].skip(2)
// rtn is an iterator that generates 3, 1, 2, 3.
```

Method `iterable#head()` creates an iterator that picks up the specified number of elements from the top.

```
rtn = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3].head(4)
// rtn is an iterator that generates 3, 1, 4, 1.
```

Method `iterable#tail()` creates an iterator that picks up the specified number of elements from the bottom.

```
rtn = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3].tail(4)
// rtn is an iterator that generates 2, 6, 5, 3.
```

Method `iterable#offset()` creates an iterator that skip the specified number of elements from the top.

```
rtn = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3].offset(5)
// rtn is an iterator that generates 9, 2, 6, 5, 3.
```

Method `iterable#pingpong()` creates an iterator that seeks elements from the top to the bottom, then from the bottom to the top, and repeats.

```
rtn = [1, 2, 3, 4, 5].pingpong()
// rtn is an iterator that generates 1, 2, 3, 4, 5, 4, 3, 2, 1, 2, 3, ...
```

Method `iterable#cycle()` creates an iterator that repeatedly seeks elements from the top to the bottom.

```
rtn = [1, 2, 3, 4, 5].cycle()
// rtn is an iterator that generates 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, ...
```

Method `iterable#reverse()` creates an iterator that seeks elements from the bottom to the top.

```
rtn = [1, 2, 3, 4, 5].reverse()
// rtn is an iterator that generates 5, 4, 3, 2, 1.
```

Method `iterable#runlength()` examines how many times the same values continue. It creates an iterator that generates a pair containing the number of how many times a value appears and the value itself.

```
rtn = ['A', 'A', 'B', 'B', 'B', 'C', 'D', 'D'].runlength()
// rtn is an iterator that generates [2, 'A'], [3, 'B'], [1, 'C'], [2, 'D']
```

Method `iterable#sort()` sorts iterable's elements in an ascending order.

```
rtn = [3, 1, 4, 1, 5, 9, 2, 6].sort()
// rtn is an iterator that generates 1, 1, 2, 3, 4, 5, 6, 9.
```

Specifying a symbol `` `descend `` in an argument of the method will sort elements in a descending order.

```
rtn = [3, 1, 4, 1, 5, 9, 2, 6].sort(`descend)
// rtn is an iterator that generates 9, 6, 5, 4, 3, 2, 1, 1.
```

Methods `iterable#after()` , `iterable#since()` , `iterable#before()` , `iterable#until()` and `iterable#while()` create an iterator that picks up elements within a certain range. They take an argument `criteria` that prompts where the range begins and ends. The `criteria` is the same as that of `iterable#filter()` and may take a function or an iterable.

- An iterator by `iterable#after()` starts extraction of elements right after the `criteria` is evaluated as `true` .

```
tbl = [3, 1, 4, 1, 5, 9, 2, 6, 5]
rtn = tbl.after(&{$x >= 5})
// rtn is an iterator that generates 9, 2, 6, 5.
```

- An iterator by `iterable#since()` starts extraction of elements at the point where the `criteria` is evaluated as `true` .

```
tbl = [3, 1, 4, 1, 5, 9, 2, 6, 5]
rtn = tbl.since(&{$x >= 5})
// rtn is an iterator that generates 5, 9, 2, 6, 5.
```

- An iterator by `iterable#before()` carrys on extraction of elements until right before the `criteria` is evaluated as `true` .

```
tbl = [3, 1, 4, 1, 5, 9, 2, 6, 5]
rtn = tbl.before(&{$x >= 5})
// rtn is an iterator that generates 3, 1, 4, 1.
```

- An iterator by `iterable#until()` carrys on extraction of elements until the point where the `criteria` is evaluated as `true` .

```
tbl = [3, 1, 4, 1, 5, 9, 2, 6, 5]
rtn = tbl.until(&{$x >= 5})
// rtn is an iterator that generates 3, 1, 4, 1, 5.
```

- An iterator by `iterable#while()` carrys on extraction of elements while the `criteria` is evaluated as `true` .

```
tbl = [3, 1, 4, 1, 5, 9, 2, 6, 5]

rtn = tbl.while(&{$x < 5})
// rtn is an iterator that generates 3, 1, 4, 1.
```

Method `list#combination()` creates an iterator that returns a group of all combinations of elements extracted from the target list. It takes an argument that specifies the number of elements to extarct.

```
rtn = [1, 2, 3, 4].combination(3)
// rtn is an iterator that generates [1, 2, 3], [1, 2, 4], [1, 3, 4], [2, 3, 4]
```

Method `list#permutation()` creates an iterator that returns a group of all permutations of elements extracted from the target list. It takes an argument that specifies the number of elements to extarct.

```
rtn = [1, 2, 3].permutation(2)
// rtn is an iterator that generates [1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]
```

If it omits the argument, all the elements would be extracted.

```
rtn = [1, 2, 3].permutation()
// rtn is an iterator that generates [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]
```

Method `list#shuffle()` returns a list in which elements are shuffled in a random order.

14.6 Iterator Generation

Function `range()` returns an iterator that generates numbers within the specified range.

```
range(10)          // 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
range(4, 10)       // 4, 5, 6, 7, 8, 9, 10, 11, 12, 13
range(0, 10, 2)    // 0, 2, 4, 6, 8
```

Function `interval()` returns an iterator that generates the specified number of `number` values between the prescribed range.

```
interval(1, 3, 5)  // 1, 1.5, 2, 2.5, 3
```

Function `consts()` returns an iterator that generates the specified number of a constant value of any type.

```
consts('foo', 3)   // 'foo', 'foo', 'foo'
```

Function `rands()` returns an iterator that generates random number values.

```
rands(100)         // random numbers between 0 and 99
```

Chapter 15

File Operation

15.1 Overview

Gura provides mechanism of **Stream** and **Directory** to work on files: Stream is prepared to read and write content of a file and Directory to retrieve lists of files stored in some containers. Here, a term "file" is not limited to what is stored in a file system of an OS. You can also use Stream and Directory to access files through networks and even ones stored in an archive files. Gura provides a generic framework to handle these resources so that you can expand the capabilities by importing Modules.
Each of Streams and Directories is associated with a uniquely identifiable string called **pathname**. A framework called **Path Manager** is responsible of recognizing pathname for Stream and Directory and dispatching file operations to appropriate processes that have been registered by built-in and imported Modules.

15.2 Pathname

15.2.1 Acceptable Format of Pathname

A pathname is a string that identifies Stream and Directory, which should be handled by Path Manager.
By default, built-in module `fs` has been registered to Path Manager, which tries to recognize a pathname as what is for ones stored in a file system. Below are some of such examples:

```
/home/foo/work/example.txt
C:\Users\foo\source\main.cpp
```

You can use both a slash or a backslash as a directory separator for a file in file systems, which is to be converted by `fs` module to what the current OS can accept. You can see variable `path.sep_file` to check what character is favorable to the OS.

15.2.2 Utility Functions to Parse Pathname

Function `path.dirname()` extracts a directory part by eliminating a file part from a pathname.

```
rtn = path.dirname('/home/foo/work/example.txt')
// rtn is '/home/foo/work/'
```

If the pathname ends with a directory separator, the function determines it doesn't contain a file part and returns the whole string.

```
rtn = path.dirname('/home/foo/work/')
// rtn is '/home/foo/work/'
```

Function `path.filename()` extracts a file part from a pathname.

```
rtn = path.fileame('/home/foo/work/example.txt')
// rtn is 'example.txt'
```

When given with a pathname that ends with a directory separator, the function determines it doesn't contain a file part and returns a null string.

```
rtn = path.filename('/home/foo/work/')
// rtn is ''
```

Function `path.split()` splits a pathname by a directory separator and returns a list containing its directory part and file part. This works the same as a combination of `path.dirname()` and `path.filename()` .

```
rtn = path.split('/home/foo/work/example.txt')
// rtn is ['/home/foo/work/', 'example.txt']
```

Function `path.cutbottom()` eliminates the last element in the directory hierarchy. This works the same as `path.dirname()` when the pathname ends with a file part.

```
rtn = path.cutbottom('/home/foo/work/example.txt')
// rtn is '/home/foo/work/'
```

Note that it would have a different result if the pathname ends with a directory separator.

```
rtn = path.cutbottom('/home/foo/work/')
// rtn is '/home/foo/'
```

Function `path.bottom()` splits a pathname and returns the last element. This works the same as `path.filename()` when the pathname ends with a file part.

```
rtn = path.bottom('/home/foo/work/example.txt')
// rtn is 'example.txt'
```

Note that it would have a different result if the pathname ends with a directory separator.

```
rtn = path.bottom('/home/foo/work/')
// rtn is 'work'
```

Function `path.splitext()` splits a pathname by a period existing last and returns a list containing its preceding part and suffix part.

```
rtn = path.splitext('/home/foo/work/example.txt')
// rtn is ['/home/foo/work/example', 'txt']
```

Function `path.absname()` takes a relative path name in a file system and returns an absolute name based on the current directory.

15.3 Stream

15.3.1 Stream Instance

A Stream is a data object to read and write content of a file and represented by a `stream` instance created by a constructor function named `stream()` . Below shows a declaration of the constructor function:

```
stream(pathname:string, mode?:string, codec?:codec):map {block?}
```

In many platforms and languages, people are accustom to using a term `open` as a function name for opening a file, or a stream. So, function `open()` is provided as a complete synonym for `stream()` , which has the same declaration with it.

```
open(pathname:string, mode?:string, codec?:codec):map {block?}
```

In many cases, this document uses function `open()` instead of `stream()` to create a `stream` instance.
Function `open()` takes a pathname string as its argument and returns a `stream` instance.

```
fd = open('foo.txt')
// fd is a stream to read data from "foo.txt"
```

When it is called with its second argument `mode` set to `'w'` , the function would create a new file and returns a `stream` instance to write data into it.

```
fd = open('foo.txt', 'w')
// fd is a stream to write data into "foo.txt"
```

A `stream` instance will be closed when method `stream#close()` is called on it.

```
fd.close()
```

When a stream for writing is closed, all the data stored in some buffer would be flushed out before the instance is invalidated.
Method `stream#close()` would also be called automatically when the instance is destroyed after its reference count decreases to zero. At times, it may be ambiguous about when the instance is destroyed, so it may be better to use `stream#close()` explicitly when you want to control the closing timing.
Another way to create and utilize a `stream` instance is to call `open()` function with a block procedure that will take a `stream` instance through its block parameter.

```
open('foo.txt') {|fd|
    // any jobs here
}
```

Using this description, you can access the created instance only within the block, which will be automatically destroyed at the end of the procedure.

15.3.2 Cast to Stream Instance

If a certain function has an argument that expects a `stream` instance, you can pass it a string of a pathname, which will automatically be converted to a `stream` instance by a casting mechanism. The `stream` instance would be created as one for reading.

```
f(fd:stream) = {
    // fd is a stream instance for reading
    // any jobs here
}
f('foo.txt')   // same as f(open('foo.txt'))
```

If the argument is declared with `:w` attribute, the `stream` instance would be created for writing.

```
f(fd:stream:w) = {
    // fd is a stream instance for writing
    // any jobs here
}
f('foo.txt')   // same as f(open('foo.txt', 'w'))
```

Attribute `:r` is also prepared to explicitly declara that the stream is to be opened for reading.

15.3.3 Stream Instance for Standard Input/Output

There are three `stream` instances for the access to standard input and output, which are assigned to variables in `sys` module.

- `sys.stdin` ...Standard input that retrieves data from key board.
- `sys.stdout` ...Standard output that outputs texts to console screen.
- `sys.stderr` ...Standard error output that outputs texts to console screen without interference of pipe redirection.

Functions `print()` , `printf()` and `println()` output texts to the stream `sys.stdout` . This means that the following two codes would cause the same result.

```
println('Hello world')
sys.stdout.println('Hello world')
```

You can also assign a `stream` instance you create to these variables. Assignment to `sys.stdout` would affect the behavior of functions such as `println()` .

```
sys.stdout = open('foo.txt', 'w')
println('Hello world')   // result will be written into 'foo.txt'.
```

15.3.4 Stream with Text Data

There are fundamental functions that print texts out to standard output stream.

- Function `print()` takes multiple values that are to be printed out to `sys.stdout` in a proper format.
- Function `println()` works the same as `print()` but also puts a line feed at the end.
- Function `printf()` works similar with C language's `printf()` function and prints values to `sys.stdout` based on format specifiers. See chapter String Operation for more details about formatter.

Below is a sample code using above functions to get the same result each other.

```
n = 3, name = 'Tanaka'
print('No.', n, ': ', name, '\n')
println('No.', n, ': ', name)
printf('No.%d: %s\n', n, name)
```

Class `stream` is equipped with methods `stream#print()` , `stream#println()` and `stream#printf()` that correspond to functions `print()` , `println()` and `printf()` respectively, but output result to the target `stream` instread of `sys.stdout` . The code below outputs string to a file `foo.txt` .

```
n = 3, name = 'Tanaka'
open('foo.txt', 'w') {|fd|
    fd.print('No.', n, ': ', name, '\n')
    fd.println('No.', n, ': ', name)
    fd.printf('No.%d: %s\n', n, name)
}
```

Method `stream#readline()` returns a string containing one line of text from the stream. It will return `nil` when it reaches to end of the stream, so you can write a program that prints content of a file as below:

```
fd = open('foo.txt')
while (line = fd.readline()) {
    print(line)
}
```

Regarding that you often need to read multiple lines from a stream, method `stream#readlines()` may be more useful. It creates an iterator that returns each line's string as its element. A program to prints contet of a file comes as below:

```
fd = open('foo.txt')
lines = fd.readlines()
print(lines)
```

Using function `readlines()` that takes `stream` instance as its argument, you don't need to explicitly open a stream because of casting mechanism from `string` to `stream` . This is the simplest way to read text files.

```
lines = readlines('foo.txt')
print(lines)
```

If you want to eliminate a line feed character that exists at each line, specify `:chop` attribute.

```
lines = readlines('foo.txt'):chop
println(lines)
```

An iterator created by method `stream#readlines()` and function `readlines()` owns a reference to the `stream` instance because they're designed to read data from it while iteration. This means that the stream instance won't be released while such iterator is running.
Consider the following code that is expected to read text from `foo.txt` and write text back to the same file after converting alphabet characters to upper case.

```
lines = readlines('foo.txt')
open('foo.txt', 'w').print(lines:*upper())
```

Unfortunately, this program doesn't work correctly. The iterator `lines` owns a stream to read content from the file `foo.txt`, which conflicts with the attempt to open `foo.txt` for writing. To avoid this, you need to call `readlines()` function with `:list` attribute that reads whole the lines from the stream before storing them to a `list` instance. The function would release the stream and then return the `list` instance as its result.

```
lines = readlines('foo.txt'):list
open('foo.txt', 'w').print(lines:*upper())
```

Method `stream#readtext()` returns a string containing the whole content of the stream.

```
txt = fd.readtext()
```

As for the character sequence existing at each end of line in a file, two types of sequence are acceptable: LF (0x0a) and CR(0x0d)-LF(0x0a). Some systems like Linux that have inherited from UNIX uses LF code at line end while Windows uses CR-LF sequence. By default, the following policies are applied so that the string read from a file would only contain LF code.

- When reading, all the CR codes are removed.
- When writing, there's no modification about the sequence of end of line. This results in a file containing only LF code.

To change this behavior, use methods `stream#delcr()` and `stream#addcr()`. If you want to keep CR code from the read text, call `stream#delcr()` method with an argument set to `false`.

```
fd.delcr(false)
```

If you want to append CR code at each end of line in a file to write, call `stream#addcr()` method with an argument set to `true`.

```
fd.addcr(true)
```

15.3.5 Character Codecs

While a `string` instance holds string data in UTF-8 format, there are various character code sets to describe texts in files. To be capable of handling them, a `stream` instance may contain an instance of `codec` class that is responsible of converting characters between UTF-8 and those codes. You can specify a `codec` instance to a `stream` by passing it as a third argument of `open()` function.

```
fd = open('foo.txt', 'r', codec('cp932'))
```

Since there's a casting feature from `string` to `codec` instance, you can simply specify a codec name to the argument as well.

```
fd = open('foo.txt', 'r', 'cp932')
```

Below is a table that shows what codecs are available and what module provides them.

Module	**Available Codec Names**
`codecs.basic`	`base64`, `us-ascii`, `utf-8`, `utf-16`
`codecs.chinese`	`big5`, `cp936`, `cp950`, `gb2312`
`codecs.iso8859`	`iso8859-1`, .. `iso8859-16`
`codecs.japanese`	`cp932`, `euc-jp`, `iso-2022-jp`, `jis`, `ms_kanji`, `shift_jis`
`codecs.korean`	`cp949`, `euc-kr`

Codecs only have effect on methods to read/write text data that are summarized below:

```
stream#print(), stream#println(), stream#printf()
stream#readline(), stream#readlines(), stream#readtext()
```

The standard output/input streams, `sys.stdin` , `sys.stdout` and `sys.stderr` , must be equipped with a codec of the character code set that the console device expects. While the detection of a proper codec is done by a value of environment variable `LANG` or a result of some system API functions, it may sometimes happen that you want to change codec in these. In such a case, you can use `stream#setcodec()` like below:

```
sys.stdout.setcodec('utf-8')
```

15.3.6 Stream with Binary Data

In addition to methods to handle text data, class `stream` is equipped with methods to read/write binary data as well.
Method `stream#read()` reads specified size of data into a `binary` instance and returns it. When the stream reaches its end, the method returns `nil` .

```
open('foo.bin') {|fd|
    while (buff = fd.read(512)) {
        // some jobs with buff
    }
}
```

Method `stream#write()` writes content of a `binary` instance to the stream.

```
open('foo.bin', 'w') {|fd|
    fd.write(buff)
}
```

Method `stream#seek()` moves the current offset at which read/write operations are applied.
Method `stream#tell()` returns the current offset.
Methods `stream.copy()` , `stream#copyto()` and `stream#copyfrom()` are responsible of copying data from a stream to another stream. They have the same result each other but take `stream` instances in different ways. Below shows how they are called where `src` means a source stream and `dst` a destination.

```
stream.copy(src, dst)
src.copyto(dst)
dst.copyfrom(src)
```

These methods can take a block procedure that takes `binary` instance containing a data segment during the copy process. The size of a data segment can be specified by an argument named `bytesunit` .

```
stream.copy(src, dst) {|buff:binary|
    // any job during copying process
}
```

You can use the block procedure with the copying method to realize a indicator that shows how much process the methods have done.
Method `stream#compare()` compares contents between two streams and returns `true` if there's no difference and `false` otherwise.

15.3.7 Filter Stream

A Filter Stream is what is attached to other `stream` instance and applies data modification while reading or writing operation.
There are two types of Filter Stream: reader and writer.
A Filter Stream of reader type applies operation on methods for reading data including `stream#read()` , `stream#readline()` , `stream#readlines()` and `stream#readtext()` .

```
+--------+    +---------------+
| stream |--->| filter stream |---> (reading data)
|        |    |   (reader)    |
+--------+    +---------------+
```

A Filter Stream of writer type applies operation on methods for writing data including `stream#write()` , `stream#print()` , `stream#println()` and `stream#printf()` .

```
+--------+    +---------------+
| stream |<---| filter stream |<--- (writing data)
|        |    |   (writer)    |
+--------+    +---------------+
```

Module `gzip` provides functions to read and write files in gzip format, which usually have a suffix `.gz` . Importing the module would add following methods to `stream` class.

- `stream#gzipreader()` returns a stream from which you can read data after decompressing a sequence of gzip format from the attached stream.
- `stream#gzipwriter()` returns a stream to which you can write data that are to be compressed to a sequence of gzip format into the attached stream.

Module `bzip2` provides functions to read and write files in bzip2 format, which usually have a suffix `.bz2` . Importing the module would add following methods to `stream` class.

- `stream#bzip2reader()` returns a stream from which you can read data after decompressing a sequence of bzip2 format from the attached stream.
- `stream#bzip2writer()` returns a stream to which you can write data that are to be compressed to a sequence of bzip2 format into the attached stream.

Module `base64` provides functions to encode and decode files in Base64 format, which often appear in protocols of network. It's a build-in module that you can utilize without importing and makes following methods available in `stream` class.

- `stream#base64reader()` returns a stream from which you can read data after decoding a sequence of Base64 format from the attached stream.
- `stream#base64writer()` returns a stream to which you can write data that are to be encoded to a sequence of Base64 format into the attached stream.

Following code is an example to read content of a file in gzip format:

```
import(gzip)
open('foo.gz') {|fd_gzip|
    fd = fd_gzip.gzipreader()
    // reading process from fd
    fd.close()
}
```

These methods that generate a Filter Stream can accept a block procedure just like `open()` function, in which you can take the instance of Filter Stream as a block parameter.

```
import(gzip)
open('foo.gz') {|fd_gzip|
    fd_gzip.gzipreader {|fd|
        // reading process from fd
    }
}
```

Or simply, you can write it as below:

```
import(gzip)
open('foo.gz').gzipreader {|fd|
    // reading process from fd
}
```

The same goes with a writing process. In this case, the attached stream must have a writing attribute.

```
import(gzip)
open('foo.gz', 'w') {|fd_gzip|
    fd = fd.gzipwriter()
    // writing process to fd
    fd.close()
}
```

You can also attach a Filter Stream on yet another Filter Stream, which enables you to compose a chain of stream. Following is a code to decode a sequence in Base64 and then decompress it with gzip algorithm:

```
import(gzip)
open('foo.gz.hex') {|fd_hex|
    fd_hex.base64reader().gzipreader {|fd|
        // reading process from fd
    }
}
```

Below shows a diagram of the process:

```
+--------+    +-----------------+    +---------------+
| stream |--->|  filter stream  |--->| filter stream |---> (reading data)
|        |    | (base64 reader) |    | (gzip reader) |
+--------+    +-----------------+    +---------------+
```

You can construct a chain of stream for writing process, too.

```
import(gzip)
open('foo.gz.hex', 'w') {|fd_hex|
    fd_hex.base64writer().gzipwriter {|fd|
        // writing process to fd
    }
}
```

Below shows a diagram of the process:

```
+--------+    +-----------------+    +---------------+
| stream |<---|  filter stream  |<---| filter stream |<--- (writing data)
|        |    | (base64 writer) |    | (gzip writer) |
+--------+    +-----------------+    +---------------+
```

15.3.8 Stream with Archive File and Network

After importing `tar` module, you can create a stream that reads an item stored in a TAR archive file. When a pathname contains a filename suffixed with `.tar` , `.tgz` , `.tar.gz` or `tar.bz2` , it would decompress the content in accordance with TAR format. The example below opens an item named `src/main.cpp` in a TAR file `foo/example.tar.gz` .

```
import(tar)
open('foo/example.tar.gz/src/main.cpp') {|fd|
    // reading process from fd
}
```

Since all the works necessary to decompress content of archive files are encapsulated in Path Manager framework, you can access them just like an ordinary file in file systems. Below is an example to print content of `src/main.cpp` in `foo/example.tar.gz` .

```
import(tar)
print(readlines('foo/example.tar.gz/src/main.cpp'))
```

After importing `zip` module, you can create a stream that reads an item stored in a ZIP archive file. When a pathname contains a filename suffixed with `.zip` , it would decompress the content in accordance with ZIP format. The example below opens an item named `src/main.cpp` in a TAR file `foo/example.zip` .

```
import(zip)
open('foo/example.zip/src/main.cpp') {|fd|
    // reading process from fd
}
```

Importing `curl` module, which provides features to access network using curl library, or importing `http` module would make Path Manager able to recognize URIs that begin with protocol names like "http" and "ftp".

```
import(curl)
open('http://www.example.com/doc/index.html') {|fd|
    // reading process from fd
}
```

15.4 Directory

15.4.1 Operations

A Directory is a data object to seek a list of files and sub directories and is represented by `directory` class. But currently, there's few chance to utilize the `directory` instance explicitly since it is usually built in other objects like iterators and hidden from users. One thing you have to note about `directory` is that you can cast a `string` containing a pathname to `directory` instance, so you can pass a pathname to an argument declared with `directory` type.
There are three functions that searches items like files and sub directories: `path.dir()` , `path.glob()` and `path.glob()` . Consider the following directory structure to see how these functions work.

```
example
|
+--dir-A
|  +--file-4.txt
|  `--file-5.txt
```

```
+--dir-B
|  +--dir-C
|  |  +--file-6.doc
|  |  `--file-7.doc
|  `--dir-D
+--file-1.txt
+--file-2.doc
`--file-3.txt
```

Function `path.dir()` creates an iterator that returns pathname of items that exists in the specified directory. For example, a call `path.dir('example')` create an iterator that returns following strings.

```
example/dir-A/
example/dir-B/
example/file-1.txt
example/file-2.doc
example/file-3.txt
```

Function `path.glob()` creates an iterator that returns pathname of items matching the given pattern with wild cards. For example, a call `path.glob('example/*.txt')` create an iterator that returns following strings.

```
example/file-1.txt
example/file-3.txt
```

Function `path.walk()` creates an iterator that seeks directory structure recursively and returns pathname of items. For example, a call `path.walk('example')` create an iterator that returns following strings.

```
example/dir-A/
example/dir-B/
example/file-1.txt
example/file-2.doc
example/file-3.txt
example/dir-A/file-4.txt
example/dir-A/file-5.txt
example/dir-B/dir-C/
example/dir-B/dir-D/
example/dir-B/dir-C/file-6.doc
example/dir-B/dir-C/file-7.doc
```

15.4.2 Status Object

By default, functions `path.dir()` , `path.glob()` and `path.glob()` create an iterator that returns a string of pathname. Specifying `:stat` attribute would create an iterator generating an object called `stat` that contains more detail information about items.
There are several different `stat` instances depending on the container in which an item exists, which provide various properties for additional information as well as the item's name.
An item in file system returns `fs.stat` instance that has following properties.

Property Name	Data Type	Content
pathname	string	
dirname	string	
filename	string	
size	number	
uid	number	
gid	number	
atime	datatime	
mtime	datatime	
ctime	datatime	
isdir	boolean	
ischr	boolean	
isblk	boolean	
isreg	boolean	
isfifo	boolean	
islnk	boolean	
issock	boolean	

The code below shows an example that prints each filename and size of items under a directory `example` .

```
stats = path.dir('example'):stat
printf('%-16s %d\n', stats:*filename, stats:*size)
```

15.4.3 Directory in Archive File

After importing `tar` module, you can get a list of items stored in a TAR archive file. The code below prints all the items stored in `example.tar.gz` by `path.walk()` .

```
println(path.walk('example.tar.gz/'))
```

Note that you have to append a directory separator after the archive filename so that Path Manager recognize it as a container, not an ordinary file.
An item in TAR archive file returns `tar.stat` instance that has following properties.

Property Name	Data Type	Content
name	string	
filename	string	
linkname	string	
uname	string	
gname	string	
mode	number	
uid	number	
gid	number	
size	number	
mtime	datetime	
atime	datetime	
ctime	datetime	
chksum	number	
typeflag	number	
devmajor	number	
devminor	number	

After importing `zip` module, you can get a list of items stored in a ZIP archive file. The code below prints all the items stored in `example.tar.gz` by `path.walk()` .

```
println(path.walk('example.zip/'))
```

An item in ZIP archive file returns `zip.stat` instance that has following properties.

Property Name	Data Type	Content
`filename`	`string`	
`comment`	`string`	
`mtime`	`datetime`	
`crc32`	`number`	
`compression_method`	`number`	
`size`		number
`compressed_size`	`number`	
`attributes`		number

15.5 OS-specific Operations

15.5.1 Operation on File System

Module `fs` provides functions that work with file systems.
Function `fs.mkdir()` creates a directory. If there are non-existing directories in the specified pathname, it would occur an error. Specifying attribute `:tree` would create intermediate directories in the pathname if they don't exist.
Function `fs.rmdir()` removes a directory. If the specified directory contains files or sub directories, it would occur an error. Specifying attribute `:tree` would remove all such items before deleting the specified directory.
Function `fs.remove()` removes a file.
Function `fs.rename()` rename a file or a directory.
Function `fs.chmod()` modifies attribute of a file or a directory.
Function `fs.cpdir()` copies content of a directory to another directory.

15.5.2 Execute Other Process

Function `os.exec()` executes a process and waits until it finishes. While the process runs, its standard output and standard error are redirected to streams defined by variables `os.stdout` and `os.stderr`, and its standard input is redirected from `os.stdin`. By default, variables `os.stdin`, `os.stdout` and `os.stderr` are assigned with `sys.stdin`, `sys.stdout` and `sys.stderr` respectively. You can modify those variables to retrieve console output from the process and feed text data to it through standard input. Below is an example to run an executable `foo` after redirecting the standard output to a memory buffer.

```
buff = binary()
saved = os.stdout
os.stdout = buff.writer()
os.exec('foo')
os.stdout = saved
print(os.fromnative(buff))
```

Function `os.fromnative()` converts `binary` instance that contains a raw data from the process to a string in UTF-8 format.

Chapter 16

Network Operation

16.1 Overview

`curl` module
`http` module
client-side and server-side

16.2 Client-side Operation

You can download files via HTTP protocol using a generic stream-copy function copy. Below is the example.

```
import(http)
copy('http://sourceforge.jp/', 'sf.html')
```

If you want to use a proxy server, you need to specify a server setting using http.addproxy like follows.

```
import(http)
http.addproxy('xx.xx.xx.xx', 8080, 'username', 'password')
copy('http://sourceforge.jp/', 'sf.html')
```

16.3 Server-side Operation

Simple Example:

```
import(http)

text = R'''
<html>
<body>
Welcome to Gura server
</body>
</html>
'''

http.server(port => 8000).wait {|req|
    println(req.uri)
    req.response('200', nil, text.encode('utf-8'),
        'Cache-Control' => 'private'
        'Server'        => 'Gura_HTTP_Server'
        'Connection'    => 'Keep-Alive'
```

```
        'Content-Type'  => 'text/html')
}
```

The following example works as a HTTP server, which generates a graph that shows values in SQLite3 database temperature.sqlite3.

```
import(re)
import(cairo)
import(http)
import(png)
import(sqlite3)

makeGraph(iSites[]:number) = {
    Item = struct(day:number, temps*:number)
    tbl = Item * sqlite3.db('temperature.sqlite3').query('select * from sites')
    img = image(`rgba, 320, 320, `white)
    [wdAxis, htAxis] = [img.width * 0.9, img.height * 0.9]
    [xAxis, yAxis] = [(img.width - wdAxis) / 2, (img.height - htAxis) / 2]
    [dayMax, dayMin] = [tbl:*day.max(), tbl:*day.min()]
    dayRange = dayMax - dayMin
    [tempMax, tempMin] = [tbl:*temps:*max().max(), tbl:*temps:*min().min()]
    tempRange = tempMax - tempMin
    calcX(day) = xAxis + (day - dayMin) * wdAxis / dayRange
    calcY(temp) = yAxis + htAxis - (temp - tempMin) * htAxis / tempRange
    img.cairo {|cr|
        cr.set_line_width(img.height / 300)
        cr.rectangle(xAxis, yAxis, wdAxis, htAxis).stroke()
        cr.save {
            cr.set_dash([img.height / 200, img.height / 200], 0)
            cr.move_to(xAxis, calcY(0)).line_to(xAxis + wdAxis, calcY(0))
            cr.stroke()
        }
        for (iSite in iSites) {
            func = cr.move_to
            for (item in tbl) {
                func(calcX(item.day), calcY(item.temps[iSite]))
                func = cr.line_to
            }
            cr.stroke()
        }
    }
    img
}
http.server(port => 80).wait {|req|
    iSites = [0]
    query = req.query
    if (query.haskey('site')) {
        iSites = tonumber(query['site'].split(','):list)
    }
    buff = binary()
    makeGraph(iSites).pngwrite(buff)
    req.response('200', nil, buff,
        'Server' => 'Gura_HTTP_Server' 'Connection' => 'close')
}
```

After the script runs, it waits for HTTP requests. Launch a Web browser and access to it as like http://localhost/?site=0,1. If you try it on Linux, you have to run the script as a root user or replace the port number with one larger than or equal to 1024.

Chapter 17

Image Operation

17.1 Overview

17.2 Image Instance

An instance of `image` class contains image data and provides functions such as reading/writing image files, resizing and rotating.
An image instance can be created by a constructor function `image` . Calling `image` function with an argument that specifies a stream containing an image data would read that data. The code below reads a JPEG file and write it in PNG format.

```
import(jpeg)
import(png)
image('foo.jpg').write('foo.png')
```

Before `image` function, you have to import a module that can handle an image type. The following table shows image types and associated module names.

Image Type	Module	Added Methods to `image`
BMP	`bmp`	`bmpread` , `bmpwrite`
JPEG	`jpeg`	`jpegread` , `jpegwrite`
GIF	`gif`	`gifread` , `gifwrite`
PNG	`png`	`pngread` , `pngwrite`
Microsoft Icon	`msico`	`msicoread` , `msicowrite`
PPM	`ppm`	`ppmread` , `ppmwrite`
XPM	`xpm`	`xpmdata` , `xpmwrite`
TIFF	`tiff`	`tiffread`

Importing those modules also add methods to `image` class like `jpeg` module adding `image#jpegread` and `image#jpegwrite` .

17.3 Format-specific Operations

17.4 JPEG

EXIF

17.5 GIF

Here is a JPEG image file that contains animation frames: cat-picture.jpg.

(Any size of picture would be acceptable if only all the frames have the same size and are aligned at regular invervals.)
The program needs to do the following jobs.

- Reads a JPEG file as a source image.
- Reduces number of colors in the image down to 256 so that it suits GIF specification.
- Creates a GIF content.
- Divides the source image into frames and adds them to the GIF content.
- Writes the GIF content to a file.

And here is the script code:

```
import(jpeg)
import(gif)

delayTime = 12              // interval time in 1/100 seconds
[nx, ny] = [6, 2]           // number to divide a source image
img = image('cat-picture.jpg').reducecolor(`win256)
[w, h] = [img.width / nx, img.height / ny]
i = range(nx * ny)
xs = (i % nx) * w
ys = int(i / nx) * h
imgFrames = img.crop(xs, ys, w, h)
gif.content().addimage(imgFrames, delayTime).write('cat-anim.gif')
```

It utilizes Implicit Mapping feature to process frame images. If you're interested in what's running in the code, trace the variable `imgFrames` about how it's created by `image#crop()` and how it's processed in `gif.content#addimage()` .
cat-anim.gif

17.6 Cairo

17.6.1 Simple Example

Here is a simple example using Cairo.

```
import(cairo)
import(show)

img = image(`rgba, 300, 300)
img.cairo {|cr|
    cr.scale(img.width, img.height)
    cairo.pattern.create_linear(0, 0, 1, 1) {|pat|
        pat.add_color_stop_rgb(0, 0, 0, 0)
        pat.add_color_stop_rgb(1, 1.0, 1.0, 1.0)
        cr.set_source(pat)
    }
    cr.rectangle(0.1, 0.1, 0.8, 0.8)
    cr.fill()
}
img.show()
```

17.6.2 Render in Exisiting Image

The following is an example that performs reading a JPEG file, drawing something on it with Cairo APIs and writing it out as a JPEG file.

```
import(jpeg)
import(cairo)
I(filename:string) = path.join(sys.datadir, 'sample/resource', filename)
img = image(I('Winter.jpg'))
img.cairo {|cr|
    repeat (10) {|i|
        [x, y, r] = [128 + 30 * i, 128 + 30 * i, 60 - i * 4]
        pat = cairo.pattern_create_radial(
            x - r / 10, y - r / 6, r / 5, x - r / 6, y - r / 6, r * 1.2)
        pat.add_color_stop_rgba(0, 1, 1, 1, 1)
        pat.add_color_stop_rgba(1, 0, 0, 0, 1)
        cr.set_source(pat)
        cr.arc(x, y, r)
        cr.fill()
    }
}
img.write('result.jpg')
```

17.6.3 Output Animation GIF File Combining Multiple Image Files

You can create a GIF file that has a dynamically produced image. The example below shows how to output an animation GIF file that contains images created by Cairo APIs.

```
import(cairo)
import(gif)
str = 'Hello'
img = image(`rgba, 64, 64, `white)
gifobj = gif.content()
img.cairo {|cr|
    cr.select_font_face('Georgia', cairo.FONT_SLANT_NORMAL, cairo.FONT_WEIGHT_BOLD)
    cr.set_font_size(64)
    te = cr.text_extents(str)
    cr.set_source_rgb(0.0, 0.0, 0.0)
    for (x in interval(64, -te.width, 30)) {|i|
        img.fill(`white)
        cr.move_to(x, 50)
        cr.show_text(str)
        gifobj.addimage(img.clone(), 10)
    }
}
gifobj.write('anim2.gif')
```

17.6.4 More Sample Scripts

You can find sample scripts using Cairo on GitHub repository.

17.7 OpenGL

17.7.1 Sample Script

Gura supports APIs of OpenGL 1.1.
The following example has been ported from one of the samples in http://www.wakayama-u.ac.jp/ tokoi/opengl/libglut.html.

```
import(glu) {*}
import(opengl) {*}
import(gltester)

vertex = [
    [0, 0, 0], [1, 0, 0], [1, 1, 0], [0, 1, 0]
    [0, 0, 1], [1, 0, 1], [1, 1, 1], [0, 1, 1]
]

init(w:number, h:number) = {
    glClearColor(1, 1, 1, 1)
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glEnable(GL_DEPTH_TEST, GL_CULL_FACE)
    glEnable(GL_LIGHTING, GL_LIGHT0, GL_LIGHT1)
    glCullFace(GL_FRONT)
    glViewport(0, 0, w, h)
    glMatrixMode(GL_PROJECTION)
    glLoadIdentity()
    gluPerspective(30, w / h, 1, 100)
}

display(degree:number) = {
    glMatrixMode(GL_MODELVIEW)
    glLoadIdentity()
    gluLookAt(3, 4, 5, 0, 0, 0, 0, 1, 0)
    glRotated(degree, 1, 1, 0)
    glMaterialfv(GL_FRONT_AND_BACK,
            GL_AMBIENT_AND_DIFFUSE, [0.8, 0.2, 0.2, 1])
    glBegin(GL_QUADS) {
        glNormal3dv([ 0,  0, -1]), glVertex3dv(vertex[0, 1, 2, 3])
        glNormal3dv([ 1,  0,  0]), glVertex3dv(vertex[1, 5, 6, 2])
        glNormal3dv([ 0,  0,  1]), glVertex3dv(vertex[5, 4, 7, 6])
        glNormal3dv([-1,  0,  0]), glVertex3dv(vertex[4, 0, 3, 7])
        glNormal3dv([ 0, -1,  0]), glVertex3dv(vertex[4, 5, 1, 0])
        glNormal3dv([ 0,  1,  0]), glVertex3dv(vertex[3, 2, 6, 7])
    }
}

degree = 0
[width, height] = [300, 300]
gltester.mainloop(width, height, 0, `idle) {
    `onDraw => function {
        init(width, height)
        display(degree)
    }
    `onKeyPoll => %{
        `left => function { degree += 1 }
        `right => function { degree -= 1 }
    }
}
```

Execution result.

17.7.2 More Sample Scripts

You can find sample scripts using OpenGL on GitHub repository, which have been ported from SGI.

Chapter 18

Graphical User Interface

18.1 Overview

wxWidgets
Tk
SDL

18.2 wxWidgets

18.2.1 About wxWidgets

Gura's `wx` module uses libraries of wxWidgets 3.0.0.

18.2.2 Simple Example

The code below is the simplest example that shows an empty window.

```
import(wx)

MyApp = class(wx.App) {
    OnInit() = {
        frame = MyFrame('Button Test', size => wx.Size(200, 100))
        frame.Show()
        true
    }
}

MyFrame = class(wx.Frame) {
    __init__(title:string, pos:wx.Point => wx.DefaultPosition,
            size:wx.Size => wx.DefaultSize) = {|nil, wx.ID_ANY, title, pos, size|
        wx.Button(this, wx.ID_ANY, 'Push Me')
    }
}

wx.IMPLEMENT_APP(MyApp)
```

An application using wx module must create a class that derives from `wx.App` and implement `OnInit()` method in it. The method is responsible of initializing GUI-related resource and creating a main frame. It should return `true` at the end if no error occurs.
In the above example, the main frame is declared by a class `MyFrame` that derives from `wx.Frame`, which has a constructor function including an instance creation of `wx.Button` contol. You can create any necessary controls within the constructor.
An application class is realized by calling `wx.IMPLEMENT_APP`, which runs a main loop in it.

18.2.3 Event Handling

There are several ways to address event handling. The first one is to call `wx.Window#Bind` method to the control instance like below.

```
import(wx)

MyApp = class(wx.App) {
    OnInit() = {
        frame = MyFrame('Button Test', size => wx.Size(200, 100))
        frame.Show()
        true
    }
}

MyFrame = class(wx.Frame) {
    __init__(title:string, pos:wx.Point => wx.DefaultPosition,
            size:wx.Size => wx.DefaultSize) = {|nil, wx.ID_ANY, title, pos, size|
        ctrl = wx.Button(this, wx.ID_ANY, 'Push Me')
        ctrl.Bind(wx.EVT_BUTTON) {|event|
            wx.MessageBox('Button was pushed', 'Button Test', wx.OK, this)
        }
    }
}

wx.IMPLEMENT_APP(MyApp)
```

You need to specify an event type like `wx.EVT_BUTTON` as an argument for `wx.Window#Bind` method and also describe a procedure that will be evaluated when the event occurs as its block. You may specify a block parameter `event` , which will take an instance of `wx.CommandEvent` class at the block's evaluation. Even though the button controls doesn't offer much information with the event instance, more complicated controls could include more data in it.
Another approach is to assign unique identifiers to controls and let the parent window to handle events that are sent from them. The example comes like this:

```
import(wx)

MyApp = class(wx.App) {
    OnInit() = {
        frame = MyFrame('Button Test', size => wx.Size(200, 100))
        frame.Show()
        true
    }
}

MyFrame = class(wx.Frame) {
    [
        ID_BTN_PushMe
    ] = wx.NewIds()
    __init__(title:string, pos:wx.Point => wx.DefaultPosition,
            size:wx.Size => wx.DefaultSize) = {|nil, wx.ID_ANY, title, pos, size|
        ctrl = wx.Button(this, ID_BTN_PushMe, 'Push Me')
        this.Bind(wx.EVT_BUTTON, ID_BTN_PushMe) {|event|
            wx.MessageBox('Button was pushed', 'Button Test', wx.OK, this)
        }
    }
}

wx.IMPLEMENT_APP(MyApp)
```

The function `wx.NewIds` generates as many unique identifers as you want. You can specify one of them to the second argument of a control constructor and also the second argument of `window#Bind` method. The identifier is necessary because the parent window must determine what control has issued the event.

18.2.4 Layout Management

You can use classes derived from `wx.Sizer` to arrange controls' size and position.

```
import(wx)

MyApp = class(wx.App) {
    OnInit() = {
        frame = MyFrame('Button Test', size => wx.Size(200, 200))
        frame.Show()
        true
    }
}

MyFrame = class(wx.Frame) {
    __init__(title:string, pos:wx.Point => wx.DefaultPosition,
            size:wx.Size => wx.DefaultSize) = {|nil, wx.ID_ANY, title, pos, size|
        vbox = wx.BoxSizer(wx.VERTICAL)
        this.SetSizer(vbox)
        ctrl = wx.Button(this, wx.ID_ANY, 'First')
        vbox.Add(ctrl, wx.SizerFlags(1).Expand())
        ctrl = wx.Button(this, wx.ID_ANY, 'Second')
        vbox.Add(ctrl, wx.SizerFlags(1).Expand())
        ctrl = wx.Button(this, wx.ID_ANY, 'Third')
        vbox.Add(ctrl, wx.SizerFlags(1).Expand())
    }
}

wx.IMPLEMENT_APP(MyApp)
```

`wx.BoxSizer` is one the sizer classes that layouts controls in a direction, either vertical or horizontal. A top-level sizer must be associated to the window by `window#SetSizer` method. And then, you can put each control under the sizer's management by calling `wx.Sizer#Add` method. The method takes a `wx.SizerFlags` instance as its second argument, with which you can specify how the control's size is arranged.

18.2.5 More Sample Scripts

You can find sample scripts using wxWidgets on GitHub repository.

18.3 Tk

18.3.1 About Tk

Gura provides modules named `tcl` and `tk` that use Tcl/Tk library for GUI programming.

18.3.2 Simple Example

The following example creates a window that has one Button widget.

```
import(tk)

tk.mainwindow() {|mw|
    mw.Button(text => 'Push me') {|w|
        w.pack()
        w.bind(`command) {
            w.tk$MessageBox(title => 'event', message => 'hello')
```

```
        }
    }
}
tk.mainloop()
```

18.3.3 Sample Script

The code below is a drawing program. I have ported it from a sample in TkDocs.

```
import(tk)

tk.mainwindow() {|mw|
    mw.Canvas(bg => 'white') {|c|
        c.pack(fill => 'both', expand => true)
        [lastx, lasty] = [0, 0]
        color = 'black'
        c.bind('<1>') {|x:number, y:number|
            [lastx, lasty] = [x, y]
        }
        c.bind('<B1-Motion>') {|x:number, y:number|
            addLine(x, y)
        }
        addLine(x:number, y:number) = {
            extern(lastx, lasty)
            c.Line(lastx, lasty, x, y, fill => color, width => 3)
            [lastx, lasty] = [x, y]
        }
        setColor(colorNew:string) = {
            color:extern = colorNew
        }
        function(color:string, y:number):map {
            c.Rectangle(10, y, 30, y + 20, fill => color) {|item|
                item.bind('<1>') { setColor(color) }
            }
        }(['red', 'blue', 'black'], 10 + (0..) * 25)
    }
}
tk.mainloop()
```

Sample result.

18.3.4 More Sample Scripts

You can find sample scripts using Tk on GitHub repository.

18.4 SDL

18.4.1 About SDL

Gura provides a module named `sdl` that uses SDL library.
SDL, Simple DirectMedia Layer, is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL and Direct3D.

18.4.2 Simple Example

The following script only shows a blank window by using SDL.

```
import(sdl)

sdl.Init(sdl.INIT_EVERYTHING)
screen = sdl.SetVideoMode(640, 480, 16, sdl.SWSURFACE)
repeat {
    event = sdl.WaitEvent()
    (event.type == sdl.QUIT) && break
}
```

At first, you have to initialize SDL's status by calling `sdl.Init` . Then, calling `sdl.SetVideoMode` with screen size and depth in its arguments will show a window.
Unlike other GUI platform, SDL requires you to implement an event handling loop explicitly. The function `sdl.WaitEvent` would wait until some events come in and returns an instance of `sdl.Event` class that contains event type and related information.

18.4.3 More Sample Scripts

You can find sample scripts using SDL on GitHub repository.

Chapter 19

Mathematic Functions

19.1 Complex Number

suffix `j`

19.2 Rational Number

suffix `r`

```
4 / 7r + 3 / 10r
3 / 5r + 3 / 10r
2 / 3r - 3 / 5r
6 / 7r - 1 / 3r
```

19.3 Big Number

`gmp` module
suffix `q`
suffix `L`

19.4 Differential

`function#mathdiff`

```
>>> f(x) = math.sin(x ** 2)
>>> g = f.mathdiff()
>>> g.expr
`(math.cos(x ** 2) * (2 * x))
```

Function	**Derivative**
x ** 2	2 * x
x ** 3	
x ** 4	
a ** x	
math.sin(x)	
math.cos(x)	
math.tan(x)	
math.exp(x)	
math.log(x)	
math.log10(x)	
math.asin(x)	
math.acos(x)	
math.atan(x)	
math.sqrt(x)	
math.sin(x) ** 2	
math.sin(x ** 2)	
math.log(math.sin(x))	
x ** 2 * math.sin(x)	
math.sin(x) / (x ** 2)	
3 ** (2 * x)	
math.log(x ** 2 + 1)	
((x - 1) ** 2 * (x - 2) ** 3) / ((x - 5) ** 2)	
math.sin(2 * x - 3)	
math.cos(x) ** 2	
(2 * x - 1) ** 3	
math.sqrt(x ** 2 + 2 * x + 3)	
1 / x	
math.exp(x) + math.exp(-x)	
math.exp(x) - math.exp(-x)	
(math.sin(x + 2) + x + 2) * (math.sin(x + 3) + x + 3)	
math.sin(math.sin(x ** 2 / 3))	
(2 * x - 1) / x ** 2	

Chapter 20

Template Engine

20.1 Overview

Sometimes, you may want to daynamically generate text from a template that contains some variable fields. You can use Template Engine to embed Gura scripts within a text for such purposes.

20.2 How to Invoke Template Engine

There are two ways to invoke Template Engine as below:

- In a command line, launch Gura intepreter with `-T` option and a template file containing embedded scripts.
- In a script, create a `template` instance in a script with which you can control the engine.

20.2.1 Invoke from Command Line

Consider a template file `sample.tmpl` that contains the below text content containing an embedded script:
`[sample.tmpl]`

```
Current time is ${datetime.now().format('%H:%M:%S')}.
```

From a command line, execute the Gura interpreter with the option `-T` followed by the file name as below:

```
$ gura -T sample.tmpl
```

This would evaluate the file with the engine that renders the result to the standard output like below:

```
Current time is 12:34:56.
```

20.2.2 Invoke from Script

In a script, you can create a `template` instance to work with the engine. Below is an example to read the above sample file and create the instance:

```
tmpl = template('sample.tmpl')
```

Then, you can render the result of the template with `template#render()` method. Below is an example to put the result to standard output:

```
tmpl.render(sys.stdout)
```

If the method takes no argument, it would return the result as a string.

```
result = tmpl.render()
```

It may sometimes happen that you want to describe a template containing embedded scripts as a `string` value in a script. The `string` class provides method `string#template()` that create a `template` instance from the string.

```
str = 'Current time is ${datetime.now().format('%H:%M:%S')}.'
result = str.template().render()
```

As it's thought to be a common process to create a `template` instance from a string and then render it, a utility method called `string#embed()` is prepared. The above code can also be writen as below:

```
str = 'Current time is ${datetime.now().format('%H:%M:%S')}.'
result = str.embed()
```

20.3 Embedded Script

When the engine finds a region surrounded by borders "`${` " and "} " in a template, that would be recognized as an embedded script in which you can put any number and any type of expressions as long as the embedded script has a final result value of one of the following types:

- `string`
- `number`
- `nil`
- a list or iterator of `string`
- a list of iterator of `number`

An error occurs if the embedded script has any other types of value.
If the embedded script has no element in it, it would render nothing. Below is an example:
Template:

```
Hello${}World
```

Result:

```
HelloWorld
```

If the embedded script has a `string` value, it would render that string.
Template:

```
Hello ${'gura'} World
```

Result:

```
Hello gura World
```

As the content of the embedded script is an ordinary script, it can contain any number and any types of expressions including variable assignments and function calls.
Template:

```
Hello ${str = 'gura', str.upper()} World
```

Result:

```
Hello GURA World
```

The embedded script can be written in free format as for inserted spaces, indentations and line breaks. The format of the script doesn't affect the rendering result as long as they're described within borders of a embedded script.
Template:

```
Hello ${
    str = 'gura'
    str.upper()
} World
```

Result:

```
Hello GURA World
```

If the embedded script has a `number` value, the engine converts the result into a string before rendering.
Template:

```
Calculation: ${3 + 4 * 2}
```

Result:

```
Calculation: 11
```

If the embedded script has a value of `nil`, it would render nothing.
Template:

```
Hello${nil}World
```

Result:

```
HelloWorld
```

If the result is a list or iterator, the engine would render each element in it.
Template:

```
Hello ${['1st', '2nd', '3rd']} World
```

Result:

```
Hello 1st2nd3rd World
```

This feature would be useful when used in combination with iterator operations such as Implicit Mapping. Below is an example to render the content of an external text file with line numbers:
Template:

```
Here is the content of foo.txt:
----
${format('%d: %s', 1.., readlines('foo.txt'))}
----
```

20.4 Indentation

If an embedded script that has a string containing multiple lines appears first in a line and is preceded by white spaces or tabs, each line would be indented with the preceding spaces.
Template:

```
Lines:
  ${'1st\n2nd\n3rd\n'}
```

Result:

```
Lines:
  1st
  2nd
  3rd
```

When the embedded script has a list or iterator of string including line breaks, each element would also be indented.
Template:

```
Lines:
  ${['1st\n', '2nd\n', '3rd\n']}
```

Result:

```
Lines:
  1st
  2nd
  3rd
```

20.5 Rendering nil Value

An embedded script that has `nil` value would render nothing just like an empty string.
Template:

```
nil${nil}-ahead
----
empty${''}-ahead
```

Result:

```
nil-ahead
----
empty-ahead
```

If an embedded script that has `nil` value appears at the end of a line, it would defeat the trailing line break while an empty string would not.
Template

```
nil${nil}
-ahead
----
empty{''}
-ahead
```

Result:

```
nil-ahead
----
empty
-ahead
```

If an embedded script that has `nil` value is an only element in a line, nothing would be rendered for the line even if it's preceded by white spaces.
Template

```
Hello
  ${nil}
World
```

Result:

```
Hello
World
```

Utilizing these rules of `nil` , some functions and methods are designed to return `nil` value so that it doesn't affect the rendering result.
The `nil` rules may sometimes have to be applied when you describe embedded scripts. Consider the following template that has an embedded script to initialize variables `x` and `y` :
Template:

```
${x = 2, y = 3}
Hello World
```

Result:

```
3
Hello World
```

You would see an unexpected result that the embedded script renders "`3` " caused by the evaluation result of the last expression "`y = 3` ". To avoid this, put `nil` at the last of the embedded script as below:
Template:

```
${x = 2, y = 3, nil}
Hello World
```

Result:

```
Hello World
```

A symbol "`-` " is defined as `nil` so that it can be used as a terminator for such scripts.
Template:

```
${x = 2, y = 3, -}
Hello World
```

Result:

```
Hello World
```

20.6 Calling Function with Block

The engine can also call a function with a block that usually appears surrounded by "`{` " and "`}` " in an ordinary script.
In a template text, a block starts implicitly after a function call that expects a mandatory block and ends with a call of a function named `end` .
Consider a function `repeat()` that repeats the procedure of the given block for the specified times. A template that repeats a text "`repeated` " with a line-break for 4 times comes like below:
Template:

```
${repeat (4)}
repeated
${end}
```

Result:

```
repeated
repeated
repeated
repeated
```

Besides the function `end` , some functions declared with `:trailer` attribute such as `elsif` and `else` can work as a block terminator. A branch sequence of `if-elsif-else` could be described like below:
Template:

```
${if (..)}
if-clause
${elsif (..)}
elsif-clause
${else}
else-clause
${end}
```

Below is an example that uses repetitions and branches in a more practical context:
Template:

```
${for (i in 1..5)}
${if (i < 2)}
${i} is less than two
${elsif (i < 4)}
${i} is less than four
${else}
${i} is greater or equal to four
${end}
${end}
```

Result:

```
1 is less than two
2 is less than four
3 is less than four
4 is greater or equal to four
5 is greater or equal to four
```

With the function `repeat()` , you can take an index number during the repetition using a block paramter like below:

```
repeat(4) {|i|
    println('repeated #', i)
}
```

In a template, such block parameters should be described in a block containing only a block parameter list within an embedded script.
Template:

```
${repeat(4) {|i|}}
repeated #${i}
${end}
```

Result:

```
repeated #0
repeated #1
repeated #2
repeated #3
```

Some functions like `range()` can take an optional block, not a mandatory one, which doesn't give Template Engine any information on whether a block should be followed. To give such a function a block, specify an empty block "{} " in an embedded script.
Template:

```
${range(4) {}}
repeated
${end}
```

Result:

```
repeated
repeated
repeated
repeated
```

20.7 Template Directive

An embedded script that begins with a character "`=` " is called a template directive, which is categorized into the following types:

- Macro Definition and Call
- Inheritance
- Rendering Other Templates

20.7.1 Macro Definition and Call

Macros are used to define text patterns that can be applied for multiple times. They're defined and called with the following directives:

- `${=define(symbol:symbol, 'args*)}` .. `${end}`
- `${=call(symbol:symbol, args*)}`

Below is an example:
Template:

```
${=define('author)}Taro Yamada{end}
Author: ${=call('author)}
```

Result:

```
Author: Taro Yamada
```

20.7.2 Inheritance

Using Template Engine's inheritance feature, you can create a derived template that inherits the text content from a base template.

```
+------------------+
|  base template   |
+------------------+
         A
         |
+--------+---------+
| derived template |
+------------------+
```

Template Engine provides the following directives for the inheritance feature:

- `${=block(symbol:symbol)}` .. `${end}` .. In a base template, it defines a template block which content would be replaced by the derived template. In a derived template, it replaces the corresponding template block defined in its base template.
- `${=extends(template:template)}` .. Declares the current template derives from the specified one.
- `${=super(symbol:symbol)}` .. Used within a template block in a derived template to insert the content of a template block defined by its base template.

A base template provides basement text content including template blocks that are supposed to be replaced by a derived template.
`[base.tmpl]`

```
block1
------
${=block(‘block1)}
block1-content base
${end}

block2
------
${=block(‘block2)}
block2-content base
${end}

block3
------
${=block(‘block3)}
block3-content base
${end}
```

Result:

```
block1
------
block1-content base

block2
------
block2-content base

block3
------
block3-content base
```

A template that calls `${=extends}` directive becomes a derived template, which should only contain `${=block}` directive to replace the content of the base template.
`[derived.tmpl]`

```
${=extends(’base.tmpl’)}

${=block(‘block1)}
block1-content derived
${end}

${=block(‘block3)}
block3-content derived
${end}
```

Result:

```
block1
------
block1-content derived

block2
------
block2-content base

block3
------
block3-content derived
```

Using directive `${=super()}` , you can render the content of the template block defined in the base template.
`[derived.tmpl]`

```
${=extends('base.tmpl')}

${=block(‘block1)}
${=super(‘block1)}
block1-content derived
${end}

${=block(‘block3)}
block3-content derived
${end}
```

Result:

```
block1
------
block1-content base
block1-content derived

block2
------
block2-content base

block3
------
block3-content derived
```

20.7.3 Rendering Other Templates

The directive `${=embed()}` renders other templates from the current template.

- `${=embed(template:template)}`

Below is an example:
Template:

```
${=embed('header.tmpl')}
${=embed('body.tmpl')}
${=embed('footer.tmpl')}
```

20.7.4 How Does Directive Work?

A directive actually consists of two methods named like `template#xxxxx()` and `template#init_xxxxx()` where `xxxxx` is the directive name. They would work with the engine that has two phases of process: presentation and initialization phase. The presentation phase runs all the rendering and scripting process while the initialization phase only evaluates directive's methods `template#init_xxxxx()` .
When a parser in the engine finds a directive `${=xxxxx()}` , it will add parsed result of `this.init_xxxxx()` to the initialization phase and `this.xxxxx()` to the presentation phase.

20.8 Comment

The engine recognizes a region surrounded by "`${==` " and "`==}$` " as a comment and just skips it during parsing process.
Template:

```
1st line
2nd line
${== comment of single-line ==}$
3rd line
${==
comment of multi-lines
==}$
4th line
5th line${== comment at end of line ==}$
6th line
7th ${== comment in the middle of line ==}$line
8th line
```

Result:

```
1st line
2nd line
3rd line
4th line
5th line
6th line
7th line
8th line
```

20.9 Scope Issues

An embedded script in a template runs with a scope in which `template#render()` is evaluated. Consider the following template file including an embedded script that contains variable references named `fruit` and `price` :
`[sample.tmpl]`

```
The price of ${fruit} is ${price} yen.
```

Below is a script to render that template.
script:

```
func(tmpl:template, fruit:string, price:number) = {
    tmpl.render(sys.stdout)
}

tmpl = template('sample.tmpl')
func(tmpl, 'grape', 100)
```

Note that the template is evaluated with a scope in the context of `func` .