Gura Language Manual

Updated: September 12, 2013

Official site: http://www.gura-lang.org/

Introduction

We often see a process that applies some operation or transformation on multiple data stored in lists and put the result into another list. For instance, it includes plotting the result of a mathematical function fed with sequence of numbers as its parameter and tansforming multiple record 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 elements up from lists subsequently and then create lists that contain result values. When you use a functional language, it might be a familiar approach that you prepare a higher-order function of mapping with which you apply a certain function on list elements.

Either way, you've had to explicitly program "repeat" operation with existing languages. However, when you provide n numbers to a function that takes one argument and returns one result, it's obvious that you want n answers from it. If a programming language itself holds a functionality to expand elements in a list or an iterator as a function argument and automatically repeat calling the function, users, or programmers, just have to call the function without repeating syntax. I calls this "implicit mapping" as it implicitly does mapping process.

In order to realize this idea, I had origianally considered to make expansion of an existing script language. But you couldn't see the real power of "implicit mapping" if it only works with a limited number of functions. Beyond number processing, it should cover all the process like string operation, image drawing and network access and should become a part of a daily programming style. That means that I had to create a completely new world instead of an expansion of an exsiting one. So I decided to create a script language from a scratch.

As for the creation of a new script language, I've been taking notice the following points:

Introduces a Familiar Syntax

> I don't think it's necessary to bother creating a new syntax if it has same function as the exsting one. I made it a policy that I decide syntax and symbol assign as in silimar way to familiar languages as possible. The language makes blocks with brace symbols, so a written script may look like Java or JavaScript program. It's also introduced rules of namespace handling by module and function naming policy from Python.

Makes It Handy and Practical

> Programming language is expected to solve problems that exist around us. And you can't call it practical if you must take much effort to such problems. So the new language will be shipped with graphical interfaces and functions that can handle any type of files so that an usual user can easily program what he wants to realize.

Under such principles, I've developed a script language Gura that comes with functions and methods reflectingng "implicit mapping" policy, and registered its first release in SourceForge.JP on March 15, 2011.

Development of a new programming language is amazing because creating a language doesn't instantly mean that the creater is an expert programmer of it. This may be similar to coming up with an idea of a new game: even if you make a rule of it, 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 myself to know how to make programs of Gura, which comes with an innovative rule "implicit mapping." And, through that process, I've confidently realized that Gura's various features like "implicit mapping" are really practical in actual programming fields.

I, as one user, recommend this script language for you.

Writer

September 27, 2012

Table of Contents

1. About This Reference

This reference explains about the execution method, the syntax, the basic data type and the processing functionality of script language Gura. Refer to "Gura Library Reference" if you want to see specification of implemented functions, methods and modules shipped with it.

2. Execution Method

2.1. Executable File

Gura's executable files for Windows are gura.exe and guraw.exe while that for Linux is gura.

An executable file guraw.exe doesn't show command prompt window and you should use it when running GUI program with wxWidgets, Tcl/Tk and SDL.

2.2. Interrupt Mode

When you run gura.exe or gura without specifying a script file, it shall enter in an interrupt mode and wait for user inputs.

```
Gura 0.3.1 [MSC v.1600, May 31 2012] copyright (c) 2011-2012 Y.Saito
>>>
```

When you input Gura's script and then an enter key after a prompt >>>, it shall execute the script and show its result.

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

To exit from an interrupt mode, enter Ctrl+C from keyboard or execute a script `sys.exit()`.

2.3. Script File

When you run the executable with a script file specified, it shall execute the content of the file.

Suffix names for Gura script files are .gura or .guraw. In a Windows environment, these are associated with executables gura.exe and guraw.exe respectively.

Script files are in free format text, so you can write Gura program from top of a text file without any declarations. However, when you use UNIX-lilke OS such as Linux, it would be convenient to write shebang at the top line. Below is a Hello World script that can run under both Windows and Linux.

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

OS's shell couldn't recognize shebang correctly when each line in the file is ended with CR-LF code. It must be ended with LF code.

If a script file contains non-ASCII characters like Japanese in script files, you'd better save it in UTF-8 character code, which is a default code set that Gura can handle. When you need to save the file in other character codes, use `-d` option from command line or describe a magic comment as below to execute it. Refer to 3.15.2 to see the detail of magic comment.

```
#!/usr/bin/env gura
# coding: shift_jis
println('こんにちは、世界')
```

2.4. Composite File

A composite file is an archive file in ZIP format that contains Gura scripts and other resource files, and it has a file name suffixed by .gurc or .gurcw. A composite file is a convenient way to distribute an application as it can handle more than one script files and image files by archiving them in a single file.

You can create a composite file by any archiving tool that support ZIP format. Notice that most of such tools determine a file format by its filename's suffix, so you have to create a file with a suffix .zip at once and then rename it to have Gura's composite file's suffix. Below is an example of using 7z as a tool.

```
> 7z a hoge.zip hoge.gura image1.png image2.png
> ren hoge.zip hoge.gurc
```

Gura package includes module `gurcbuild` that helps you create a composite file without any programs other than Gura itself. Below is a Gura script to create a composite file that consists of sames files as above.

```
#!/usr/bin/env gura
import(gurcbuild)
gurcbuild.build(['hoge.gura', 'image1.png', 'image2.png'])
```

A composite file created by this method has a shebang at its top and also an executable attribute, so you can handle it as an executable file under UNIX-like OS.

2.5. Command Line Options

Command line options below are acceptable.

オプション	説明
`-c cmd`	Recognizes the content of `cmd` described in an argument list a script and executes it.
`-t`	Enters into an interactive mode after evaluation of script files and so forth.
`-i module[,…]`	Specifies modules to be imported before an execution of a script. You can specify more than one module. This option has a same effect as evaluating `import` function in a script.
`-I dir`	Specifies directories in which modues are searched.
`-C dir`	Changes current directory to `dir` before an execution of a script.
`-d encoding`	Specifies character encoding of a script. Default is UTF-8.

`-T template`	Evaluates scripts embedded in the specified template file and then puts out the result to standard output.
`-v`	Prints Gura's version number.

2.6. Template Engine

You can specify a template file by a command line option `-T`. A template file is a file embedding Gura's script in a normal text document and used to alter contents of it dynamically. When an template file is evaluated, Gura script embedded in it would be evaluated and then the result be reflected to standard output.

3. Script Elements

3.1. Number Literal

There are two types of number literal: real number and imaginary one.

Real number is an instance of `number` type, which has a floating-point as its content. Regular expression for a real number is represented as below.

```
-?[0-9]*(\.[0-9]*)?([e|E][-+]?[0-9]+)?
```

Imaginary number is an instance of `complex` type, which is represented by a real number literal followed by "`j`". If you want to represent an imaginary number j, you should described it as "`1j`" because a token "`j`" without any number prefixed would be determined as a symbol for variables and so forth. For the same reason, number –j should be described as "`-1j`".

Below are valid examples of number literals.

```
43    3.1415    23e5      32j    1j     -3j
```

3.2. String Literal

A string surrounded by single quotations "`'`" or double quotations "`"`" is a string literal. String literal is an instance of `string` type, which has internal data in UTF-8 format.

Within a string literal, you can use the following escape characters following a back slash "`\`".

Escape Character	Note
`\\`	back quote
`\'`	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*	a character of code hexadecimal number *hh*

When you surround a string with single quotations, you need to describe single quotation characters within the string as escaped one. And you need to describe double quotation characters as escaped one within a string surrounded by double quotations. Although surrounding of single and double quotations have no difference in handling other than that, Gura script manner recommends you to surround a string with single quotations.

You can write a string that contains line-feed characters if you surround it with triple sequences of single or double quotation. It would enable you to describe multiple-line string as below.

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

You can align top of each line by escaping and disabling the first line-feed character in the string as described below.

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

A string literal prefixed by "r" or "R" can contain the escape character "\" without any escaping. This is convenient when you describe regular expressions or something like that that often use escape character as their elements. Be careful that, with this expression, you can't put an escape character at the end of the string.

The difference between prefix "r" and "R" is how a line-feed code that appears first should be treated in a multi-line string. Below is an example of prefix "r".

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

This is evaluated as a string `'\nABCD\nEFGH\nIJKL\n'`.

In contrast, an example below, which has prefix "R", eliminates the first line-feed code and evaluates it as `'ABCD\nEFGH\nIJKL\n'`.

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

It's assumed that string description in multi-line form is used to embed normal text in a script, so, using expression with "R" prefix , you can treat text data in a natural way.

3.3. Binary Literal

A string literal prefixed by "b" would become a binary literal. It has the same expression rule as a string literal.

A binary literal is treated as an instance of `binary` type, which contains a sequence of byte-sized data . It would always be process by byte unit.

3.4. Identifier

An identifier is a string that starts with an alphabet, an underscore "_", a dollar symbol "$", an at symbol "@" or a first code of UTF-8 and contains following characters of those elements as well as numbers and UTF-8 byte sequence. It can accept UTF-8 so that you can specify an identifier that contains Japanese characters and so forth.

An identifier would be replaced with a defined value in the current scope after an evaluation process. The replaced value would not be changed as long as the definition is not modified.

3.5. List

An expression surrounded by bracket characters "[" and "]" is a list. A list is an instance of `list` class.You can describe Gura's any types of data as elements of a list. Each element is separated by comma ",", and a line-feed is also recognized as a separator between elements. It means that, as for elements in different lines, a comma character between them can be eliminated.

When a list expression is evaluated, it would evaluate each element in sequence and return a list object that has evaluated results as its elements. Below are valid expressions of list.

```
[]
[1, 2, 3, 4, 5]
['Hello', 2, 3, 'World']
[[1, 2, 3], 4, 5, [6, 7, [8, 9, 10]]]
```

A list can also be described with characters "@{" and "}" instead of brackets. In this case, if lists are contained in an outer list as its elements, they can be embraced with brace characters "{" and "}". With those syntaxes, you can write a program in a similar way to C language's array declaration and easily import/export data between such a language and Gura. With brace characters, list declarations above can be rewritten as follows.

```
@{}
@{1, 2, 3, 4, 5}
@{'Hello', 2, 3, 'World'}
@{{1, 2, 3}, 4, 5, {6, 7, {8, 9, 10}}}
```

If a list has an element that is evaluated as an iterator, it would expand the iterator and incorporate the elements.

```
[1..10]       # equivalent to [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1..3, 8..10] # equivalent to [1, 2, 3, 8, 9, 10]
```

If a list contains infinite iterators, it would occur an error.

You can refer to an element of a list when you specify an index number surrounded by bracket

characters after a value that is evaluated as a list. And also, you can modify a value of an element when you specify a list value that is suffixed by an index number surrounded by bracket characters and then put assign operator "=" and a value.

3.6. Iterator

括弧記号 "(" および ")" で囲んだ領域はイテレータになります。イテレータは iterator クラスのインスタンスです。イテレータ中には Gura で認識できる任意のデータを要素として記述することができます。要素間はカンマ "," で区切りますが、改行も要素間の区切りとして認識されます。これは、要素を複数の行に分けて記述する場合、行末のカンマを省略できることを意味します。

イテレータを評価すると、内部の要素を順に評価し、その結果を要素としてもつイテレータオブジェクトを返します。以下は有効なイテレータ表記の例です。

```
()
(1, 2, 3, 4, 5)
('Hello', 2, 3, 'World')
([1, 2, 3], 4, 5, [6, 7, [8, 9, 10]])
```

要素の中に、評価結果がイテレータになるものがあると、そのイテレータを展開したものを要素として追加します。

```
(1..10)       # (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) と等価
(1..3, 8..10) # (1, 2, 3, 8, 9, 10) と等価
```

要素中に無限イテレータが含まれていてもエラーにはなりません。

3.7. Matrix

An expression surrounded by tokens "@@{" and "}" is a matrix. A matrix is an instance of matrix class.

A matrix has a set of column elements as each row element that is surrounded by brace characters "{" and "}", or parenthesis characters "[" and "]". Values after evaluation of each element expression will be stored in matrix instance. Every row must have the same number of column elements. Otherwise, it would occur an error.

Below are valid examples of matrix expression.

```
@@{{2, 5, -1}, {1, 3, 1}, {3, -1, -2}}
@@{{math.cos(t), -math.sin(t)}, {math.sin(t), math.cos(t)}}
```

3.8. Block

An expression surrounded by brace characters "{" and "}" is a block. Inside it, you can describe any data Gura can recognize as elements. Elements are separated by comma "," and line-feed as well. This means that you can omit a comma character between elements when they are described in different lines.

As long as the explanation above, you'll notice that it look the same as the description for list. But

they are different in evaluation process. When processing a block, it would evaluate each element sequentially just like a list, and the last evaluated element would be treated as a value of the block itself. Below are valid examples of block expression.

```
{}
{1, 2, 3, 4, 5}
{'Hello', 2, 3, 'World'}
{{1, 2, 3}, 4, 5, {6, 7, {8, 9, 10}}}
{print(), x += 2}
```

A block is used to describe a process in a function declaration and to pass a block expression in a function call.

3.9. Dictionary

An expression surrounded by tokens "`%{`" and "`}`" creates a dictionary. A dictionary is an instance of `dict` class. Each element in a dictionary definition consists of a key and a value and can be described in three ways as below. In a view of readability, the first one is recommended.

1. A key and a value are joined by assignment operator "=>". Each assignment expression is separated by a comma "`,`" or a line-feed character.
2. Express a pair of key and value as elements of a list or a block. In a list expression, they are surrounded by bracket characters "`[`" and "`]`". In a block expression, they are surrounded by brace characters "`{`" and "`}`".
3. Express a key and a value each other in a list of one dimentional.

You can use a number, a string or a symbol as a dictionary key. As for data, any types are assignable.

A declaration of a dictionary is actually a call of a function named "`%`" with a block expression. As for a detail of function and block, refer to エラー! 参照元が見つかりません。.

You can refer to an element of a dictionary when you specify a key surrounded by bracket characters after a value that is evaluated as a dictionary. And also, you can modify a value of an element when you specify a dictionary value that is suffixed by a key surrounded by bracket characters and then put assign operator "=" and a value.

3.10. Quoted Value

If an expression is prefixed by a back quotation "`` ` ``", it would be treated as a Quoted value, which is an instance of `expr` class.

3.11. Symbol

An identifier prefixed by a back quotation "`` ` ``" is treated as a symbol value. As a symbol is handled as a unique number in a program, symbols can quickly be compared each other. Using this characteristic, it's often used as a key of dictionaries and an enumeration value.

3.12. Function

When you append an argument list surrounded by parenthesis characters "(" and ")" after a value evaluated as a function instance, it would be recognized as a function call. Even when you don't need to use any arguments, you must specify a pair of parenthesis without any content as an argument list to explicitly declare it's a function call.

A function that belongs to a class is specifically called "method."

As a unique point compared with other languages, Gura doesn't have any specific statement to realize control sequences and class declarations. These are all implemented as functions.

3.13. Attribute

An identifier that follows after a colon character ":" is called an attribute. An attribute is described after an identifier or an argument list of a function and is used in the following cases.

- Type conversion in assignment to an identifier.
- Type declaration in an argument list of a function definition.
- Cutomizes a function behavior by being described after an argument list of a function call.
- Specifies a default behavior of a function by being described after an argument list of a function definition.

An attribute used in function call works in a similar way as an argument of boolean type. The different point is that an argument can take dynamically modified value while an attribute is statically specified.

3.14. Operator

Operators are special form of functions. There are an unary operator that takes only one argument and a binary operator that takes two arguments.

3.15. Comment

3.15.1. Line Comment and Block Comment

In a script, when a token "//" or a hash character "#" appears, following characters until its end of line would be treated as a comment. This is called line comment.

A region between tokens "/*" and "*/" would also become a comment. This is is called block comment. A block comment can contain line-feed characters in it. And, within a block comment, you can nest and describe other block or line comments.

Below are valid examples of comment.

```
// line comment
# line comment again
x = 10 // line comment after some code
x = 10 # line comment after some code again
```

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

3.15.2. Magic Comment

In the case that a script contains characters that are not in ASCII code, you have to describe an encoding name as a magic comment. A magic comment is described at the first or the second line in a script and has a format "`coding: XXXXXX`", in which `XXXXXX` is an encoding name and comes to something like `utf-8` and `shift_jis`.

A magic comment must be describe as a line comment. At first, the parser would check if a line comment appears at the first line and contains a magic comment. If the first line is a shebang, which is a comment including command line declaration for UNIX shell script that begins with a token "`#!`", the parser would search a magic comment at the second line.

Below is an example of specifying a shebang and a magic comment.

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

If you want it to work as a coding declaration for Emacs as well, describe it as below.

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

4. Class and Instance

4.1. Abstraction

All of data that Gura works with belong to certain classes. Classes are categorized in basic data type and object type. They have diffent manner to manage memory region occupied their data itself.

Any data generated from class is called instance. An instance inherits methods and member variables provided by its class.

Name of classes belogs to different name space from that of variables. It means that you can assign the same name to both a class and a variable.

4.2. Member Access

You can refer to a content of a member variable by specifying an instance followed by a dot character "`.`" and the variable name. And, specifying a method call after an instance and a dot character would execute the method. Within a method, a variable named `this` is defined, which refer to the instance itself.

You can modify a content of a member variable by specifying an assign operator "=" and a value after a member variable expression.

Also, you can define a method by an assign operator "=" from outer scope just in the same way as a normal function definition. This means that you can append methods to an existing instance. Below is an example that defines a method named `introduce` in an instance of `string` type.

```
str = ''
str.introduce() = { println('this string is ', this) }
str.introduce()
```

It's also possible to append methods to a class. In that case, use a function `classref` to get a reference to the class. Below is a general format of the function.

```
classref(type:expr):map
```

An example to append a method to `string` class is shown below.

```
classref(`string).introduce() = { println('this string is ', this) }
str = ''
str.introduce()
```

Methods that are appended to a class can be used for every instance of the class. They are also available even for instances that have already been realized before the method definition. This feature is often used for modules to expand specification of existing classes. For instance, import of regular expression module `re` would append methods like `match` to `string` class.

4.3. Basic Data Type

Basic data type is the most primitive data type. In function arguments and variable assignment, any instance of basic data type would be passed by value. Basic data types built in the language are listed below.

Type	Notes
`symbol`	Data type to represent a symbol value. A symbol value is an indentifier prefixed by a back quotation "`". As each symbol value is identical and is distinguished by numbers, comparation of symbols is quickly processed.
`boolean`	Data type to represent truth-value. As predefined variables of `boolean` type, there are `true` and `false`. A variable `nil` is also recognized as false value while all of the other values are treated as true. Beware that an empty list and zero number are also recognized as true. When you convert a value of `boolean` to `number` type, `true` would come to `1` and `false` to `0`.
`number`	Data type to represent number.
`complex`	Data type to represent complex number.

4.4. Object Type

In function arguments and variable assignment, an instance of object type would be passed by reference. Object types built in the language are listed below.

Type	Notes
`function`	Function
`string`	String
`binary`	Binary data
`list`	List
`matrix`	Matrix
`dict`	Dictionary
`stream`	Stream
`datetime`	Date and time
`timedelta`	Time difference
`iterator`	Iterator
`expr`	Quoted value
`environment`	Scope
`error`	Error
`image`	Image

`color`	Color data
`palette`	Palette
`codec`	Character codec

5. Operator

5.1. Built-in Operator

The list below summarizes operators that are built in Gura and their behaviors.

Operator	Behavior
`+x`	When `x` is of `number`, `complex` or `matrix` type, returns the value of `x` itself. Other type of values would occur an error.
`-x`	When `x` is of `number`, `complex` or `matrix` type, returns a negative value of `x`. Other type of values would occur an error.
`~x`	When `x` is of `number` type, returns a bit-inverted value in `number` type. The value of `x` is rounded into an integer before calculation. Other type of values would occur an error.
`!x`	Regarding `x` as a truth-value, returns a logical inverted value in `boolean` type.
`x + y`	Returns a result of the following calculation. `number + number` Returns an added result in `number` type. `complex + complex` Returns an added result in `complex` type. `number + complex` Returns an added result in `complex` type. `complex + number` Returns an added result in `complex` type. `matrix + matrix` Returns an added result in `matrix` type. `datetime + timedelta` Returns an added result in `datetime` type. `timedelta + datetime` Returns an added result in `datetime` type. `timedelta + timedelta` Returns an added result in `timedelta` type. `string + string` Returns a joined result in `string` type. `binary + binary` Returns a joined result in `binary` type. `binary + string` Returns a joined result in `string` type. `sting + binary` Returns a joined result in `string` type. `string + any` Returns a joined result in `string` type after converting `any` into `string`. `any + string` Returns a joined result in `string` type after converting `any` into `string`.
`x - y`	Returns a result of the following calculation. `number - number` Returns an subtract result in `number` type. `complex - complex` Returns an subtract result in `complex` type. `number - complex` Returns an subtract result in `complex` type. `complex - number` Returns an subtract result in `complex` type. `matrix - matrix` Returns an subtract result in `matrix` type. `datetime - timedelta` Returns an subtract result in `datetime` type.

	`datetime - datetime` Returns an subtract result in `timedelta` type. `timedelta - timedelta` Returns an subtract result in `timedelta` type.
`x * y`	Returns a result of the following calculation. `number * number` Returns a product result in `number` type. `complex * complex` Returns a product result in `complex` type. `number * complex` Returns a product result in `complex` type. `complex * number` Returns a product result in `complex` type. `matrix * matrix` Returns a product result in `matrix` type. `matrix * list` Returns a product result in `list` type. `list * matrix` Returns a product result in `list` type. `timedelta * number` Returns a product result in `timedelta` type. `number * timedelta` Returns a product result in `timedelta` type. `function * any` Works as function binder. See 6.2 for detail. `string * number` Returns a joined result of `number` times of repetition in `string` type. `number * string` Returns a joined result of `number` times of repetition in `string` type. `binary * number` Returns a joined result of `number` times of repetition in `binary` type. `number * binary` Returns a joined result of `number` times of repetition in `binary` type.
`x / y`	Returns a result of the following calculation. `number / number` Returns a division result in `number` type. `complex / complex` Returns a division result in `complex` type. `number / complex` Returns a division result in `complex` type. `complex / number` Returns a division result in `complex` type. `matrix / matrix` Returns a division result in `matrix` type.
`x % y`	Returns a result of the following calculation. `number % number` Returns a remainder result in `number` type. `string % any` Returns a formatted `string`. See 5.3 for detail.
`x ** y`	Returns a result of the following calculation. `number ** number` Returns a powered result in `number` type. `complex ** complex` Returns a powered result in `complex` type. `number ** complex` Returns a powered result in `complex` type. `complex ** number` Returns a powered result in `complex` type.
`x == y`	Returns `true` when `x` equals to `y`, `false` otherwise.
`x != y`	Returns `true` when `x` differs from `y`, `false` otherwise.
`x > y`	Returns `true` when `x` is greater than `y`, `false` otherwise.
`x < y`	Returns `true` when `x` is smaller than `y`, `false` otherwise.

`x >= y`	Returns `true` when `x` is greater than or equals to `y`, `false` otherwise.
`x <= y`	Returns `true` when `x` is smaller than or equals to `y`, `false` otherwise.
`x <=> y`	Returns `-1` when `x` is smaller than `y`, `0` when equals and `1` when is greater.
`x in y`	When used as `for` function's argument: This would be treated as an iterator-assigning operator. For details, see the explation of `for` function. When used in other cases: If `y` is a list of an iterator, this returns `true` when `x` equals to one of the elements in `y`, and `false` otherwise. If `y` is of other types that them, this returns the same result as operator =. In other words, it returns `true` when `x` equals to `y` and `false` otherwise.
`x | y`	When both of x and y are of `number` type, returns a result of OR calculation for each bit of them in `numbe` type. Before the calcukation, `x` and `y` are rounded into integer numbers. When both of x and y are of `boolean` type, returns a logical add result in `boolean` type. In other words, it returns `false` when both of x and y are `false`, and `true` otherwise. If `x` is `nil`, it returns the value of `y`. If `y` is `nil`, it returns the value of `x`. Other type of values would occur an error.
`x & y`	When both of x and y are of `number` type, returns a result of AND calculation for each bit of them in `numbe` type. Before the calcukation, `x` and `y` are rounded into integer numbers. When both of x and y are of `boolean` type, returns a logical product result in `boolean` type. In other words, it returns `true` when both of x and y are `true`, and `false` otherwise. If `x` or y is `nil`, it returns `nil`. Other type of values would occur an error.
`x ^ y`	When both of x and y are of `number` type, returns a result of XOR calculation for each bit of them in `numbe` type. Before the calcukation, `x` and `y` are rounded into integer numbers. When both of x and y are of `boolean` type, returns an exclusive logical add result in `boolean` type. In other words, it returns `false` when x and y are the same truth value, and `true` otherwise. Other type of values would occur an error.
`x << y`	When both of x and y are of `number` type, it returns a value of `x` shifted left by `y` bits in `number` type. Before the calculation, `x` and `y` are rounded into integer numbers. Other type of values would occur an error.

`x >> y`	When both of x and y are of `number` type, it returns a value of `x` shifted right by `y` bits in `number` type. Before the calculation, `x` and `y` are rounded into integer numbers. Other type of values would occur an error.
`x && y`	If `x` is determined as a false value, it returns `false` as its result. In this case, `y` is not evaluated. If `x` is determined as a true value, it would evaluate `y` as well. It returns the value of `y` when `y` is also determined as a true value. If `y` is determined as a false value, it returns `false` as its result.
`x || y`	If `x` is determined as a true value, it returns the value of `x`. In this case, `y` is not evaluated. If `x` is determined as a false value, it would evaluate `y` as well. It returns the value of `y` when `y` is determined as a true value. If `y` is determined as a false value as well, it returns `false` as its result.
`x = y`	Assigns the value of `y` to `x`. If you specify an operator right before the symbol "=", it assigns the value of calculation between `x` and `y`. For instance, when an expression "`x += y`" is evaluated, it would calculate the result of "`x + y`" and then assign it to `x`. Among operations with this format, there are "`+=`", "`-=`", "`*=`", "`/=`", "`%=`", "`**=`", "`|=`", "`&=`", "`^=`", "`<<=`" and "`>>=`". For detail of assignment operator, see 5.4.

5.2. About Logical Operation

As logical operations `&&` and `||` would check condition on left side to determine if it's necessary to evaluate an expression on right side, they can be used as a branch sequence in place of `if` function.

5.3. String Formatter

When you combine a string and a list with a percent symbol '`%`', it would treat the string as formatter directive and convert values in the list into string. Each specifier in the format comes like `%[flags][width][.precision]specifier`.

You can specify one of the following as `[specifier]`.

specifier	説明
`d, i`	An integer decimal number with sign.
`u`	An integer decimal number without sign.
`b`	An integer binary number.
`o`	An integer octal number without sign.
`x, X`	An integer hex number without sign.
`e, E`	A floating number (any alphabet would be capitalized with specifier `E`)

f, F	A floating number (any alphabet would be capitalized with specifier F)
g, G	Favors e- or f- format (any alphabet would be capitalized with specifier G)
s	A string.
c	A character.

You can specify one of the following as [flags].

flags	説明
+	Appends + character at the head for positive numbers.
-	Arranges converted string on right align.
(space)	Appends a space character at the head for positive numbers.
#	Appends "0b", "0" and "0x" at the head for a converted result of a binary, an octent and a hex number respectively.
0	Fills lacking pact of columns with 0.

You should specify a decimal number for [width] as a minimum width. If length of a converted string is shorter than this number, the rest width will be filled with space characters of code 0x20. When the length is longer than or equal to the number, nothing will be done. If you specify an asterisk "*" instead of a number for [width], the minmum width will be retrieved from arguments.

As for [precision], you can specify a number of digits below a floating point.

5.4. Assign Operator

5.4.1. Assign to a Symbol

Assign operator "=" has a similar format with binary operators. However, it's different from other operators in terms of having a side effect to modify content of a variable scope.

Using assign operator, you can define new values to variables, indexed elements or functions. Also, you can assign values to multiple elements at once using brackets.

You can modify a content of variable by evaluating an expression "symbol = value". When you specify an attribute after the symbol that indicates a variable type, it would assign a value converted to the type. For instance, an expression "foo:string = 3" converts 3, a value of number type, to string type, and then assign it to a variable named foo.

5.4.2. Assignment by Index Access

You can modify a content of index element by evaluating an expression "obj[index] = value". Here, obj is an instance of classes that provide a method for index access, which are represented by class list and dictionary class dict.

You should specify an indexing value between bracket characters "[" and "]". Available data types for indexing value is different for each class of instance. list instance accepts only number as its index and occurs error with other types. For dict instances, you can specify any number, string

or `symbol` type of insance as an index.

You can specify multiople indices in brackets.

```
obj[5, 6, 7, 8, 9] = 10
```

When you specify a list or iterator as an index, elements of those would be treated as index values.

```
obj[5..9] = 10
```

5.4.3. Assignment to Function

Combining a function's generic expression and process body with an assign operator means a definition of a function. A simple example looks like "`func() = print('hello')`". Function's generic expression and how to define it will be described later.

5.4.4. Assignment at Once to Multiple Symbols

When you specify targets, which are variable symbol or index element, on left side of assign operator with brackets characters "`[`" and "`]`" surrounding, it will assign value to each target.

If the defining value is a list, each target is defined with a value at corresponding position in the list. The example below defines variable `a`, `b` and `c` with `1`, `2` and `3` respectively.

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

If a number of targets is less than that of assigning values, assignment will be done just for the number of targets. In contrast, if a number of targets is greater than that of values, it would occur an error.

You can specify an iterator for assigning value. The example above can also be described as follows.

```
[a, b, c] = 1..3
```

If a number of targets is less than that of elements of the iterator, assignment will be done just for the number of targets. In contrast, if a number of targets is greater than that of elements, it would occur an error.

When you specify an infinite iterator for assigning value, assignment will be done just for the number of targets. An example is shown below.

```
[a, b, c] = 1..
```

This format can be used like enum declaration in C.

If assigning value is not a list nor an iterator, targets in the brackets will be defined with the same value. For instance, the example below assigns `3` to variables `a`, `b` and `c` respectively.

```
[a, b, c] = 10
```

5.5. 演算子のオーバーロード

operatorクラスのメソッドassign()を使うと、演算子の処理内容を追加または上書きすることができます。このメソッドは以下のように実行します。

- 単項演算子の定義: `operator(op).assign(type) {|value| …}`
- 二項演算子の定義: `operator(op).assign(type_l, type_r) {|value_l, value_r| …}`

引数opは、演算子シンボルをバッククオート ` でシンボル化したものを指定します。引数type、type_lおよびtype_rは、型名シンボルの先頭にバッククオート ` をつけたものを指定します。ブロック内には演算子が評価されたときに実行する内容を記述します。このとき、演算対象の値がブロックパラメータとして渡されます。ブロックの最終的な評価値をその演算子の結果として扱います。

以下は、単項演算子 - に文字列を与えたとき、順序を逆にした文字列を返すようにする例です。

```
operator(`-).assign(`string) {|x| x[-(1..)].join()}
```

実行例を以下に示します。

```
>>> -'hello world'
'dlrow olleh'
```

6. Function

6.1. Call of Function

6.1.1. Elements

Evaluating a function instance with an argument list becomes a function call. An argument list is a list of more than zero arguments for a function that are separated by a comma character and surrounded by parentheses "`(`" and "`)`".
The most basic way to get a function instance is to evaluate an identifier that is assigned with a function instance. For example, an identifier, `println`, is assigned with a function instance that has a functionality to output strings and a line-feed character to standard output. Evaluation is done with an argument list attached like follows.

```
println('hello world')
```

Elements that composes a function call are listed below.

- Function instance
- Argument list
- Attributes
- Block

Documents like Gura Library Reference provides generic expression for each function that explains how it's supposed to be called. With generic expression, you can get information like follows.

- Function name
- Class it belongs to
- Name. type and default value for each argument
- Acceptable attributes
- Block existence

Below is an example of generic expression.

```
open(name:string, mode:string => 'r', encoding:string => 'utf-8'):map {block?}
```

This is a function named `open` that takes string values, `name`, `mode` and `encoding`, as arguments, among which `mode` and `encoding` have default values. It's also specified with an attribute `:map` and is able to accept a block optionally.

6.1.2. Function Instance

For a function call, you can get a function instance by other than evaluating an identifier. For example, if a function `foo()` returns a function instance, you can directly call the returned instance as follows.

```
foo()()
```

If a function instance is a method, the function instance contains a reference to a receiver instance. Consider the following example.

```
f = 'Hello'.mid
```

In this case, `f` is a function instance resulted from evaluation of symbol `string#mid`, which also contains a reference to a receiver instance `"Hello"` in it. Using the definition, a calling process `f(1, 2)` is equivalent to `"Hello".mid(1, 2)`.

6.1.3. Argument Specifier

In a general expression, a declation of argument without any attributes means it accepts any types of value. For example, if an argument list contains an argument that simply declares `variable`, it can take number, string and any types of instance as well as `nil` value.

If a variable name is followed by a pair of brackets `"[]"` with no element, it would accept a list. It also can take an iterator, in which case that iterator would be converted to a list. Other type would occur an error.

If a variable name is followed by a colon `":"` and a type name, it would declare an acceptable type. For instance, in a function `func(x:number, y:string)`, the first argument takes `number` and the second `string`. When a variable of other types is passed to them, type conversion would be tried first. If it fails, that occurs an error.

In an argument list of a general expression, a variable name followed by `"?"` is treated as an optional argument and can be omitted when calling. For example, a function `func(x?, y?, z?)` can be called as follows.

```
func(1, 2, 3)
func(1)
func()
```

You can provide `nil` value for an optional argument. This feature is useful when you have to specify argument values after some optional arguments.

```
func(nil, 2, 3)
```

Some of Gura functions take argument list of variable length. In a general expression of such a function, an argument of variable length is declared with a symbol `"*"` or `"+"` appending.

A good example of a function with variable argument is `printf`. The function, which derives from a famous C function having the same name, has a general expression as follows.

```
printf(format:string, values*):map:void
```

When you call `printf` function, you can specify a value of `string` type and then any number of values as following arguments. Below is an example.

```
printf('Hello world\n')
printf('Current number: %d\n', x)
printf('%d + %d = %d\n', x, y, z)
```

A variable argument with a symbol "*" takes zero or more values. It means that it would not occur an error even if no value is specified. Meanwhile, an argument with a symbol "+" takes one or more values, which means that it expects at least one value and it would occur an error if the argument doesn't take any values.

If an argument is followed by a token "=>" and a specified value, it would be treated as a default value. In a function call, if an argument is not specified, the default value is used instead. For example, with a function `func(x => "yes")`, if you call it as `func()`, the argument `x` is initialized with `"yes"`.

As for the evaluation timing, the default value is evaluated when the function is called, not when the function is defined. This means that, if the result of an expression followed after "=>" is different at each evaluation, the default value is different at each calling timing.

In an argument list, if the variable name is prefixed with a back quote "`", the expression itself will be passed to the function without any evaluation. This feature is used in functions that control flow sequences such as `if` and `while`. For example, function `while` has a general expression below.

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

In general, elements in the argument list shall be evaluated before being passed to a function. However, an expression passed to the argument `cond` is passed to the function body without evaluation. `while` function itself is responsible of evaluating it. Using this mechanism, any functions may work just like statements in other languages.

6.1.4. Argument List Expansion

In a function call, if an argument is followed by symbol "*", it will be recognized as a list and expanded to element values for each argument after that. This is called argument list expansion. For example, considering a list `x = [1, 2, 3]`, a function call `func(x*)` is equivalent to `func(1, 2, 3)`.

You can specify any number of argument list expansions and can also mix them with other normal arguments. Considering variables `x = [1, 2, 3]` and `y = [5, 6, 7]`, a function call `func(x*, 4, y*)` will be expanded to `func(1, 2, 3, 4, 5, 6, 7)`.

6.1.5. Named Argument and Argument Dictionary Expansion

In function's general expression, each argument is associated with a symbol name. For example, in a function that has a general expression `func(a, b, c)`, symbols for those argument are `a`, `b` and `c`. Using a feature called named argument, you can specify those symbol names explicitly in a function call. A named argument is described by a symbol name of an argument and an assigned value

combined with a dictionary assignment operator "=>". The following three calls are equivalent each other.

```
func(1, 2, 3)
func(a => 1, b => 2, c => 3)
func(b => 2, a => 1, c => 3)
```

You don't need to put backquotations before symbol names of named arguments.

Named arguments are often used when there are many arguments or when you want to improve readability by explicityly specifying each argument with complicated meaning. It would also be useful to set a value of selected argument among many arguments being optional.

If an argument is prefixed by a symbol "`%`", the value would be treated as a dictionary and be expanded to keyword argument elements in the argument list. For example, considering a dictionary `x = %{`foo => 3, `bar => 4}`, a function call `func(x%)` is equivalent to `func(foo => 3, bar => 4)`.

You can specify any number of dictionary expantion and also use them with mixture of other ordinary argument specifiers. If there are variables `x = %{`foo => 1, `bar => 2}` and `y = %{`hoge => 5}`, a function call `func(x%, 4, y%)` is treated as `func(foo => 1, bar => 2, 4, hoge => 5)`.

6.1.6. アトリビュート指定

You can specify attributes that are prefixed by a colon character "`:`" after an argument list. Using attributes, you can customize a function's behavior.

If a function supports attributes customization, such attributs are represented as a list of symbols surrounded by brackets "`[`" and "`]`" in its general expression. As an example, below is a general expression of function `tonumber`, which converts any type of value into `number` type.

```
tonumber(value):map:[nil,zero,raise,strict]
```

This function takes attributes like `:nil` and `:zero` and customizes its behavior in accordance with these specifiers.

6.1.7. Block Specifier

Some functions take a list of elements that is surrounded by brace characters "`{`" and "`}`" after a list of arguments and attributes. This list is called a block. A general expression of a function that takes a block has a declaration like "`{block}`" and "`{block?}`" after an argument list. For a function with the first declaration, you must always specify a block. For the second one, you can optionally specify a block when calling.

Each function has different manners in the way of how to evaluate elements in the block. Below are major ways of evaluation.

- Handles it as a procedure and evaluate its elements sequentially.
- Handles it as a list of data and stores evaluated value into a container.

You can pass a list of arguments to a block by specifying it embraced with two bars "`|`" right after a brace character "`{`". This is called a block argument.

A number of arguments and data types for block arguments are different for each function that evaluates the block. For example, a function `repeat` that evaluates the content of the block repeatedly for a specified count passes a block argument in a format `|idx:number|` to the block where `idx` is a loop count that begins from 0. And a function `readlines` with a block would evaluate the block for each line read. A block argument for it is `|line:string, idx:number|` where `line` is a line string and `idx` is an index number that begins from 0. Below are examples for those functions call.

```
repeat (10) {|n| println(n)}
readlines('hoge.txt') {|line, idx| print(idx, ' ', line)}
```

A description rule of a block argument is the same as that of an ordinary argument list for a function definition. An argument with an attribute to specify its type would convert a passed value to that type. A variable length specifier "`*`" and dictionary specifier "`%`" are also available. The only different point is the strictness in checking the number of the argument. For an ordinary function call, it would cause an error if declared number of arguments is not passed. For a block argument, arguments declaration that don't appear in the list would simply be ignored. Unnecessary argument can be omitted and a block argument list itself can be eliminated if no block argument is necessary. In contrast, it would cause an error if you declare more arguments than the function provides.

A block expression contains its own block procedure body and block argument information. It may sometimes happen that you want to pass the information to other functions intact. In such a case, you can pass a content of a block to a called function by describing a variable of `expr` type that contains a block expression with "`{|`" and "`|}`" as below.

```
block = `{|x| println(x)}
repeat(10) {|block|}
```

6.1.8. Omissible Argument List

When a function takes a block expression and you don't need any arguments, you can omit an argument list for it. For example, a function `repeat`, which takes an argument for repeating count, would execute an inifite loop without any argument as below.

```
repeat () {
    // some process
}
```

In this case, you can omit the parenthesises for the argument list as below.

```
repeat {
    // some process
}
```

A function declared with an attribute :symbol_func can instantaneously be evaluated when only the symbol is specified. Among such functions, there are return, break and continue.

6.1.9. Scope

A scope is a mechanism to limit the reference to variables and functions that are stored in different spaces. This is a necessary idea for structural programming and would allow more effective programming with a proper handling. In languages like C and Java, which statically define variable types, a code location where variables are declared becomes scope space. Meantime, Gura is a script language that can use variables without declaration and it has a different manner of how to create scope spaces.

Gura では、「環境」と呼ぶ構造によってスコープを実現されています。環境は、「フレーム」と呼ばれる層を積み重ねたフレームスタックを内部に持ちます。フレームは、フレームの性質を定義する属性と、変数、関数の実体や型名とシンボル値とを結びつける辞書を持っています。関数呼び出しをすると、新たな環境を生成してそれまで実行していた環境のフレームの参照を引き継いでフレームスタックを作り、その上に新しいフレームを積み重ねます。プログラムの中で変数や関数の参照を行うと、そのときに属している環境のフレームスタックを順に探索していきます。フレームスタックの最上位に配置したフレームの属性によって、このときの探索ルールが変わります。

Gura implements scope using a structure called "environment". An environment contains a stack of layer called "frame" internally. A frame contains a dictionary that associates attributes, bodies, types and symbols.

スクリプトを実行すると、一つのフレームを持った環境が用意されます。このときの環境をルート環境、中に用意したフレームをルートフレームと呼びます。ルートフレーム内に定義した変数や関数は、そのスクリプト内の任意の位置から参照が可能です。

関数を呼び出すと、環境を用意して新たなフレームを一つ積み重ねます。このフレームを関数呼出フレームと呼びます。関数に渡した引数の内容は関数呼出フレーム内に定義されます。また、関数内部で評価した代入操作の結果もこのフレームに反映します。

関数呼出のたびに関数呼出フレームを積み重ねるので、関数はそれぞれ独立したフレームを持つことになります。変数や関数の代入操作は、この独立したフレームに対して行われ、外部に影響を与えることはありません。変数や関数の参照は、積み重ねたフレームを順に探索していきます。つまり、初めに独自のフレーム内を探索し、そこで見つからなければ一つ下のフレームという具合です。

あるシンボルが外部のフレームの定義内容を指している場合、そのシンボルを関数内部から一度でも参照すると、その後の同じシンボルへの代入は外部定義へのアクセスになります。以下の例を考えて見ます。

```
x = 3
func() = { x = x + 1 }
```

```
func()
```

関数 func を呼び出すと、まず x + 1 を評価するためにシンボル x の参照を行いますが、このとき参照されるのは関数外部のフレームで定義されている変数 x です。関数 func はシンボル x への代入をこの変数に割り付けますので、評価結果の x への代入は同じ変数へのアクセスになります。

参照をしないで外部への代入処理をするには、extern 関数を使うか、アクセスする変数に :extern アトリビュートをつけます。

extern 関数の一般式は以下の通りです。

```
extern(`syms+)
```

引数 syms には、アクセスする変数のシンボル名を列挙します。この関数を実行すると、syms で指定されたシンボル名を現在の環境のフレームスタックから探索し、見つかったものを現在の環境で書き込みできるよう設定します。指定のシンボルが見つからないとエラーになります。

シンボルに代入するときにアトリビュート:extern をつけると、そのシンボルに対して extern 関数と同じ処理してから代入を行います。書式は以下のとおりです。

```
symbol:extern = value
```

一度参照がされていれば、extern 関数や:extern アトリビュートを使う必要はないのですが、そのような場合でも明示することによってスコープの範囲を明確にすることができます。

6.1.10. レキシカルスコープとダイナミックスコープ

関数の外部参照のスコープは、プログラム中における関数の記述位置を基点にした、いわゆるレキシカルスコープになります。これにより、プログラムの見た目がそのままスコープの内外関係になるので、処理内容を把握するのが容易になります。以下のプログラムで、関数 f が表示する x は、呼出元である関数 caller 内部の x ではなく、プログラムの「見た目」どおりの「外側」にある x です。この結果は "root" を表示します。

```
x = 'root'
f() = println(x)
caller() = {
    x = 'caller'
    f()
}
g()
```

上記のレキシカルスコープが関数のデフォルトのふるまいになりますが、関数を定義するときにアトリビュート :dynamic_scope をつけると、その関数はダイナミックスコープで動作するようになります。以下の例は "caller" を表示します。

```
x = 'root'
f():dynamic_scope = println(x)
caller() = {
    x = 'caller'
    f()
```

```
}
g()
```

ダイナミックスコープはどのような場面で役立つのでしょうか。想定されるもののひとつは、引数に式を渡したときの評価です。

例として、式を引数に受け取り、その評価結果を表示する tester という関数の定義を考察します。以下のようなコードを書いたとしましょう。

```
tester(test:expr) = printf('result .. %s\n', eval(test))
x = 1
tester(`(x + 2))
```

この場合、関数 tester は x + 2 という式を引数で受け取り、これを関数 eval で評価します。関数 eval はレキシカルスコープのルールに基づいて変数 x を参照し、結果を得ることができます。

しかし、関数 tester を以下のように呼出したらどうなるでしょうか。

```
tester(test:expr) = printf('result .. %s\n', eval(test))
hoge() = {
    x = 1
    tester(`(x + 2))
}
hoge()
```

結論から言うと、これはエラーになります。変数 x は関数 hoge のローカル変数なので、関数 tester のレキシカルスコープの範囲にないからです。これを以下のようにダイナミックスコープに切り替えると、関数 tester の「外側」は呼び出し元である関数 hoge の環境になるので、期待どおりの結果を得られます。

```
tester(test:expr):dynamic_scope = printf('result .. %s\n', eval(test))
hoge() = {
    x = 1
    tester(`(x + 2))
}
hoge()
```

一般的な用途では、関数呼出でダイナミックスコープを使うことはごくまれです。式を関数に渡す処理が必要になったとき、この機能を思い出してください。

6.1.11. ブロック式とスコープ

Gura は関数呼び出しの際にブロック式を渡すことができます。ブロック式はそれを評価する関数の中で関数インスタンスとして扱われますが、これをブロック関数と呼びます。ブロック関数内のスコープの性質は通常の関数とは少々異なります。以下、例をあげて考察していきます。

```
func() {block} = {
    block()
}
```

関数 func はブロック式をとります。ブロック式の内容は関数インスタンスとして変数 block に代入されるので、

それを内部で呼び出しています。

ここで、以下のような処理を考えてみます。最後の println で表示される内容は 1 でしょうか、それとも 2 でしょうか。

```
x = 1
func() { x = 2 }
println(x)
```

Gura でこれを実行すると、2 が表示されます。上記の func のようにブロック式をとる関数は制御構文のように使われることが多く、実際 if や while などの関数はまさにその用途のために存在します。この観点からすると、ブロックの中に記述した x = 2 という式は呼び出しもとのスコープに対する処理とするのが自然な発想といえます。

しかし、ブロックの内容は関数インスタンスとなって func に渡され、関数呼出で評価がされています。これを通常の関数呼出フレームで評価してしまうと、その内部における代入処理は外部のフレームに影響しません。

これを解決するため、ブロック関数の呼び出し時は通常の関数呼出フレームではなく「ブロック関数呼出フレーム」を使います。このフレームがスタックフレームの最上位に積まれていると、代入処理を評価したときにこのフレームではなくその下のフレームに対して処理を行うようになります。

場合によっては、ブロック式の中で有効な変数を使いたいことがあります。そのような時は、代入する識別子にアトリビュート :local をつけ、ローカル変数として宣言します。これは、アトリビュート :extern とは逆に、変数の代入操作を最上位のフレームに限定するものです。以下の例では、ブロック内の変数 x がローカル変数になり、2 を代入する操作はこのローカル変数に対して行われるので、1 を表示します。

```
x = 1
func() { x:local = 2 }
println(x)
```

あるスコープ内で、変数にいったんローカル変数として代入操作をすると、以降はアトリビュート:local をつけなくてもローカル変数として扱われます。

ローカル変数は local 関数を使っても宣言することができます。local 関数の一般式は以下の通りです。

```
local(`syms+)
```

引数 syms には、ローカル変数として宣言する変数のシンボル名を列挙します。

6.2. 関数バインダ

関数インスタンスとリストを演算子 "*" でつなげると、リストの内容を引数リストにして関数を実行することができます。このときの演算子 "*" のふるまいを関数バインダと呼びます。例えば、func * [1, 2, 3] という式は func(1, 2, 3) という呼び出しと同じです。

ところで、関数バインダと同じ効果は、引数リスト中で "*" を使ったリスト展開でも得られます。上の例は func([1, 2, 3]*) と評価しても結果は同じです。しかし、関数バインダを使うと「引数となるリスト」のリストやイテレータをとって、複数の関数評価ができるようになります。例えば、x = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] というリストがあったとします。これに対して func * x と評価すると、各要素を引数リストとみなして func(1, 2, 3)、func(4, 5, 6)、func(7, 8, 9) という呼び出しになります。右辺の内容として、リスト

のかわりにイテレータを渡すこともできます。

この機能が役立つケースのひとつは、CSV やデータベースアクセスで得られた結果を構造体に収める処理です。CSV を例題にとりあげて考察してみます。CSV ファイルは複数の文字列をカンマで区切って一行ずつ配置したテキストフォーマットで、列ごとに意味を持たせています。例えば、一列目に名前、二列目に性別、三列目に年齢を格納した以下のような CSV ファイル people.csv を考えてみます。

```
Honma Chise,female,46
Kawahata Nana,female,47
Kikuchi Takao,male,35
Iwai Michiko,female,36
Kasai Satoshi,male,24
```

Gura では csv モジュールの関数 csv.read を使って複数の文字列を要素に持つリストを、一行ごとに生成するイテレータを得ることができます。上のファイルを使い、csv.read('people.csv') を実行すると、以下のような要素を生成するイテレータを返します。

```
['Honma Chise', 'female', '46']
['Kawahata Nana', 'female', '47']
['Kikuchi Takao', 'male', '35']
['Iwai Michiko', 'female', '36']
['Kasai Satoshi', 'male', '24']
```

リストのままだと要素のアクセスがしづらいので、構造体を使うことを考えます。上のデータ構造を表現する構造体は、struct 関数を使って以下のように作ることができます。

```
Person = struct(name:string, gender:string, age:number)
```

Person は構造体を生成する関数インスタンスです。person = Person('Honma Chise', 'female', 46) のように評価すると、構造体インスタンス person を作り、person.name に "Honma Chise"、person.gender に "female"、person.age に 46 が入ります。この関数インスタンスと、前述の CSV アクセス関数を関数バインダで組み合わせると、CSV ファイルを読み込んで構造体に格納する処理は以下のように記述できます。

```
Person = struct(name:string, gender:string, age:number)
people = Person * csv.read('people.csv')
```

people は Person 構造体インスタンスを要素に持つイテレータになります。後述するメンバマッピングを使うと、各フィールドは people:*name, people:*gender, people:*age というようにアクセスできます。

6.3. 関数定義

6.3.1. 構成要素

関数の一般式を記述して、代入演算子 "=" とそれに続く関数本体の式を記述すると、関数インスタンスを生成して識別子に関連付けます。

最も簡単な例として、引数をひとつもとらない関数 hoge の定義を考えます。この場合の一般式は hoge() とな

るので、関数定義は以下のように書けます。

```
hoge() = println('Hello world')
```

関数本体が複数の式から成る場合、式をカンマ "," で区切って列挙したものをブレース記号 "{" および "}" で囲んだブロック式で表記します。以下に例を示します。

```
hoge() = { println('first line'), println('second line') }
```

ブロック式の内容を複数の行に分けて記述することもできます。その際、行末はカンマと同じ意味を持つので、行ごとにカンマを書く必要はありません。上の例は以下のように書くことができます

```
func() = {
    println('first line')
    println('second line')
}
```

関数は以下の要素で構成されます。

- 関数名
- 引数定義リスト
- アトリビュート定義
- ブロック定義
- ヘルプ文字列

以下、関数定義の一般式で指定される要素の詳細について説明します。

6.3.2. 関数名

関数定義で指定する関数名は、識別子として認識できる任意のシンボル名です。これは、変数名として扱えるものと同じです。

6.3.3. 引数定義リスト

引数定義リストは、0 個以上の引数定義を括弧 "(" および ")" で囲ったものです。引数定義の間はカンマ記号 "," で区切ります。

引数定義の最も簡単なものは、単に識別子を記述したものです。関数が呼び出されると、呼び出し時の引数位置に対応する識別子に値を代入し、関数を評価します。

識別子の後にアトリビュートをつけると、それを引数の型として扱います。異なる型の値をこの引数に渡すと、最初に型変換を試み、それに失敗するとエラーになります。

引数定義に型指定がなければ、任意の型の値を受け取ることができます。これには nil 値も含まれます。型指定がされると nil 値はエラーになりますが、ケースによっては無効値として引数に渡したいことがあります。そのような場合は引数のアトリビュートに ":nil" を指定します。

引数のアトリビュートに ":nomap" を指定すると、その引数にリストやイテレータが指定されても暗黙的マッピングで展開されないようになります。型指定のアトリビュートと ":nomap" は併記が可能です。

識別子の後に、辞書代入演算子 "=>" と値を指定すると、それが引数のデフォルト値になります。指定の位置の引数を省略すると、定義されたデフォルト値がかわりに変数に設定されます。

識別子の前にバッククオート "`" をつけると、その位置に指定した引数は、評価前の式が関数に渡されます。

識別子の後に対になった角括弧 "[]" をつけると、その引数はリストを受け取ります。関数呼び出しでイテレータがこの引数に渡されると、リストに変換されます。リストとして扱えない要素を渡すと型エラーになります。

識別子の後にクエスチョンマーク "?" をつけると、その引数はオプショナルになります。

識別子の後にアスタリスク "*" やプラス記号 "+" をつけると、可変長引数を受け付けるようになります。アスタリスクをつけた場合、引数は 0 個以上の値を受け付けます。つまり、対応する位置に引数がなくてもエラーにはなりません。一方、プラス記号をつけると、引数は 1 個以上の値を受け付けます。対応する位置に引数がひとつも指定されていないとエラーになります。

引数リストの中に、識別子に続いてパーセント記号 "%" が記述された要素があると、名前つき引数の内容がこの識別子に辞書として格納されます。

6.3.4. 関数のアトリビュート定義

関数定義の引数リストの後、コロンに続いて角カッコでシンボルのリストを列記すると、その関数がオプションでうけとるアトリビュートの定義になります。

```
f():[a,b,c] = {
    if (__args__.isset(`a)) { … }
    if (__args__.isset(`b)) { … }
    if (__args__.isset(`c)) { … }
}
```

関数のアトリビュート指定は、主に関数のふるまいを静的に決定したいときに使います。

6.3.5. ブロック定義

関数にブロックを渡せるようにするには、引数定義リストとアトリビュート定義に続いて、ブロック要素を受け取る識別子をブレース記号 "{" および "}" で囲んだものを指定します。識別子は、慣例的に block という名前をつけることが多いですが、任意の名前をつけることができます。

ブロック式定義がない関数に、ブロックをつけて呼び出すとエラーになります。

逆に、ブロック式定義された関数は、呼び出しの際、必ずブロックを記述しなければいけません。ブロックを記述しないとエラーになります。ただし、ブロック式定義中の識別子の後にクエスチョンマーク "?" をつけると、そのブロックはオプショナルになります。関数は、ブロックなしでもありでも呼び出すことができるようになります。

ブロックは、関数インスタンスとして識別子に代入されます。ブロック式をオプショナル指定にした関数を、ブロックなしで呼び出すと、この識別子には nil が代入されます。

識別子の前にバッククオート "`" をつけると、ブロックは関数インスタンスでなく quoted 値として代入されます。

ブロック定義をした関数宣言の例を以下に示します。

```
f(x:number) {block} = {
    block(x)
    block(x + 1)
```

```
    block(x + 2)
}
```

以下のように呼び出します。

```
f(2) {|x|
    print(x)
}
```

ブロックの引数は外部のスコープと独立しています。

```
x = 0
f(2) {|x|
    print(x)
}
println(x)
```

ブロック式に割り当てる名前は何でもかまいません。

```
f(x:number) {yield} = {
    yield(x)
    yield(x + 1)
    yield(x + 2)
}
```

ブロックをオプション指定で宣言したとき、ブロックをつけないで関数を呼び出すとブロック式のシンボルには `nil` が渡されます。

```
f_opt() {block?} = {
    if (block == nil) {
        println('not specified')
    } else {
        block()
    }
}
f_opt()
f_opt() {
    println('message from block')
}
```

ブロックは通常、それを実行している関数の「外側」の環境にアクセスできる変数スコープで動作します（「外側」とはレキシカルスコープのそれになりますが、ダイナミックスコープに切り替えることもできます)。関数外部の変数の値を変更することができます。

```
g() {block} = {
    block()
}
n = 2
g() {
```

```
    n = 5
}
printf('n = %d\n', n) # n = 5
```

ブロックシンボルにアトリビュート :inside_scope をつけると、関数内部のスコープに切り替わり、関数の内部処理で設定される変数などにアクセスできるようになります。関数外部の変数は、参照できた値に対しての改変が可能です。

```
h() {block:inside_scope} = {
    m = 'local in h()'
    block()
}
h() {
    printf('%s\n', m)    # h()'s local variable m is accessible
}
n = 2
h() {
    n = 5
}
printf('n = %d\n', n)    # n = 2
h() {
    n += 5
}
printf('n = %d\n', n)    # n = 7
```

quoted value にしたブロックを設定した変数を {|..|} で囲って関数に渡すと、それがブロック本体として扱われます。ブロックパラメータも記述できます。例えば:

```
f() {block} = {
    block(1, 2, 3, 4)
}
```

という関数があった場合、以下のふたつの呼び出しは等価です。

```
block = `{|a, b, c, d|
    printf('%d %d %d %d\n', a, b, c, d)
}
f() {|block|}

f() {|a, b, c, d|
    printf('%d %d %d %d\n', a, b, c, d)
}
```

6.3.6. ヘルプ文字列

関数インスタンスのプロパティ help にヘルプ文字列を登録することができます。

```
f() = {
```

```
    println('Hello, World!')
}
f.help = R'''
This function just prints out Hello, World.
'''
```

6.4. 関数定義の例

関数の引数には、オプション引数・デフォルト値・可変長引数を指定できます。

```
f1(a, b?, c?) = printf('%s, %s, %s\n', a, b, c)
f1(2)                       # 2, nil, nil

f2(a, b => 10, c => 'abc') = printf('%s, %s, %s\n', a, b, c)
f2(2)                       # 2, 10, abc

f3(a, b, c*) = printf('%s, %s, %s\n', a, b, c):nomap
f3(2, 3, 4, 5, 6, 7)        # 2, 3, [4, 5, 6, 7]
f3(2, 3)                    # 2, 3, []

f4(a, b, c+) = printf('%s, %s, %s\n', a, b, c):nomap
f4(2, 3, 4, 5, 6, 7)        # 2, 3, [4, 5, 6, 7].
f4(2, 3)                    # error. c has to get at least one value.
```

関数呼び出しの際は、キーワード引数指定ができます。キーワードと値は、辞書演算子（=>）で対応づけます。

```
g1(a, b, c) = printf('%s, %s, %s\n', a, b, c)
g1(2, b => 3, c => 4)               # 2, 3, 4
```

引数リストの中に、% を後尾につけたシンボルを加えておくと、引数リストに合致しないキーワード引数指定の組を辞書にした値をそのシンボルに割り当てます。

```
g2(a, b, dict%) = printf('%s, %s, %s\n', a, b, dict)
g2(2, b => 3, c => 4, d => 5)       # 2, 3, %{c => 4, d => 5}
g2(2, 3, c => 4, d => 5)            # 2, 3, %{c => 4, d => 5}
```

引数宣言のシンボル名の先頭にバッククオートをつけると、未評価の式（quoted value）を値として渡せます。この機能を使って、制御構文を実現することができます。以下は、quoted value を使って、C の for ステートメントのような動作をする関数を作成している例です。

```
c_like_for(`init, `cond, `next):dynamic_scope {block:inside_scope} = {
    env = outers()
    env.eval(init)
    while (env.eval(cond)) {
        block()
        env.eval(next)
    }
}
```

```
n = 0
c_like_for (i = 1, i <= 10, i += 1) {
    n += i
}
printf('i = %d, sum = %d\n', I, n)
```

6.5. 関数の戻り値

関数の本体で、一番最後に評価された式の値が関数の戻り値になります。
また、return 関数を使って戻り値を指定することもできます。一般式は以下のとおりです。

```
return(value?):symbol_func
```

return 関数を呼ぶと、関数の処理を中断して処理を呼び出しもとに戻します。このとき、引数 value を指定すると、その値を関数の戻り値として扱います。引数を省略すると、nil を戻り値とします。
この関数は :symbol_func アトリビュートをつけて定義されているので、引数が必要ない場合、引数リストの括弧を省略して実行することができます。

6.6. 暗黙的マッピング

この節では暗黙的マッピングと関係のある関数アトリビュート定義について説明します。暗黙的マッピングの詳細は後の章を参照ください。

関数アトリビュートとして :map をつけると、その関数は暗黙的マッピングが有効であることを表します。

アトリビュート :void は、関数が常に nil 値を返すことを宣言するものです。通常、関数というものはある入力を受け取って何らかの処理をし、その結果として値を返します。戻り値が常に nil ということは、その関数の処理結果が戻り値としてでなく、なにがしかの状態または外部 I/O への働きとして現れることを示唆します。例えば、画面に文字列を表示するprintln関数は:voidアトリビュートをつけて定義されていますが、この関数の処理結果は標準出力 I/O へのアクセスという形で現れます。
この宣言は、暗黙的マッピングを適切に働かせるために重要です。暗黙的マッピングでは、引数にイテレータが渡されたとき、関数の処理を含めたイテレータを返すというルールがあります。つまり、引数にイテレータが入っていると、関数の処理が即座に行われないのです。例えば、println(1..10) という記述があったとき、ユーザは 1 から 10 までの数字を即座に表示することを期待しています。しかし、関数 println に渡されているのはイテレータなので、暗黙的マッピングのルールに基づくと、この呼び出しでは所定の処理を行うイテレータが返され、表示処理そのものは遅延されることになります。
しかし、アトリビュート :voidがついていると、スクリプトはその関数が値を返す類のものでないことを知ることができます。これは、関数が内部でデータを「消費」していると見ることができますが、そういった処理のため、イテレータを渡しても即時実行するように動作を切り替えるわけです。

アトリビュート :reduce をつけた関数は、常に同じ実体を返すことを表します。このアトリビュートの用途として想定しているものの一つは、this 参照を返すメソッドの定義です。
クラス Hoge があり、メソッド Hoge#foo(x)とHoge#bar(y) が実装されていると仮定します。このとき、これらのメソッドがインスタンスへの参照 this を戻り値として返すように作られていると、クラス Hoge のインスタンス

hoge へのメソッド呼び出しを hoge.foo(1).bar(2) のように続けて記述する、いわゆるメソッドチェーンが可能になります。これは、プログラムを簡潔に表記するのに便利ですが、暗黙的マッピングのルールを適用したときに不都合が起こります。例えば、hoge.foo([1, 2, 3]) のようにメソッドを呼び出すと、暗黙的マッピングによってリスト要素ごとの処理を行い、戻り値が [this, this, this] というリストになります。これは同じインスタンスへの参照を含むリストが呼び出しごとに生成されることになり、非効率的です。さらに、このような値が帰ってきてしまうと、前述のようなメソッドチェーンが記述できなくなります。

関数定義のときにアトリビュート :reduce をつけておくと、暗黙的マッピングで繰り返し処理を行う際、最初に評価した値を常に返すようになります。前の例で hoge.foo([1, 2, 3]) という呼び出しがされても、この戻り値はリスト [this, this, this] ではなく this になります。これにより、メソッドチェーン中に暗黙的マッピングを働かせて hoge.foo([1, 2, 3]).bar(2) というような記述が可能になります。

6.7. 関数呼び出しの連結関係

ブロックの終端ブレース '}' の後、同じ行に関数呼び出しの式が続くと、二つ目の関数呼び出しは前の関数と連結関係を持つようになり、二つ目の関数が評価されるか否かは最初の関数の実行内容によって制御されます。例えば、一行の間に func1(){}func2(){} と記述すると、func1 が func2 の評価をするか否かを決めることができるようになります。関数はいくつでも連結することができます。

この機能を使う代表的な例が if-elsif-else シーケンスと、try-except シーケンスです。

例えば、if と elsif を使った条件文は以下のようになります。

```
if (cond1) { process1 } elsif (cond2) { process2 }
```

この文は、最初に if 関数を評価します。if 関数は引数 cond1 の結果を真値と判断すると自身のブロック内容 process1 を評価し、続く連結式を評価しません。逆に cond1 の結果を偽値と判断すると、連結されている elsif 関数の呼び出しを評価します。elsif 関数は引数 cond2 の結果を真値と判断すると自身のブロック内容 process2 を評価します。

同一行に書く必要があるのは終端ブレース '}' と関数インスタンスの式の間だけなので、あとの要素は行を分けて記述することができます。前述の if-elsif の文は以下のように記述できます。

```
if (cond1) {
    process1
} elsif (cond2) {
    process2
}
```

連結関係を認識しない関数に連結式をつなげると、単に無視されて評価されません。

6.8. 名前なし関数

関数 function を使うと、名前なし関数を生成することができます。関数 function の一般式は以下の通りです。

```
function(`args*) {block}
```

args に引数指定、block に関数本体のコードを記述します。引数指定は、通常の関数定義と同じ文法で記

述することができます。

引数指定が必要ない場合、function のかわりに "&{…}" という形式を使って関数を生成することもできます。一般式は以下の通りです。

```
&{block}
```

どちらの形式でも、通常の関数定義にはない機能があります。それは、関数本体のコードの中に、先頭がドル記号 "$" で始まる識別子があると、その識別子が出現した順に引数リストに追加するというものです。これは、"&{…}" の形式を使って簡易的に関数インスタンスを生成したいときに便利です。以下の 2 つの表記は同じ機能を持つ関数の定義になります。

```
&{println($foo, $bar)}
function (foo, bar) {println(foo, bar)}
```

名前なし関数は、クロージャを実現するのに使われます。

```
new_counter(n:number) = {
    function() { n += 1}
}
cnt = new_counter(2)
printf('%d\n', cnt())
printf('%d\n', cnt())
printf('%d\n', cnt())
```

7. 制御構文

Gura の実行要素はすべて関数であり、制御構文という特別な要素は存在しません。しかし、他のプログラミング言語でおなじみの条件分岐や繰り返しといった処理によく似た形式で実行することができる関数を提供しています。ブロックを使っているので、外見は Java や C++などと差異が見つからないかもしれません。

この章ではそれらの関数の動作内容を見ていきます。あわせて Gura に特有の、リスト・イテレータ生成の方法も説明します。

7.1. 条件分岐

条件分岐を行う if-elsif-else シーケンスの一般式は以下のようになります。

```
if (`cond) {block} elsif (`cond) {block} elsif (`cond) {block} else {block}
```

ひとつの `if` に対して、0 個以上の任意の数の `elsif` を記述できます。`else` はひとつのみです。ブロック内に記述する式がひとつだけであっても、ブロックを囲むブレース記号 `"{"` および `"}"` は省略できないので注意してください。

if-elsif-else シーケンスを評価すると、条件に合致したブロックの評価値を全体の値として返します。この性質を使って、C 言語でおなじみの三項演算子、すなわち `result = flag? a : b` という形式を以下のように記述することができます。

```
result = if (flag) {a} else {b}
```

7.2. 繰り返し

繰り返しを実現する関数には `repeat`、`while`、`for` および `cross` があります。

Gura の繰り返し関数は、単にリピート処理をするだけではありません。ループが一回まわるごとに、評価した値をリストの要素として残していく機能を使うと、リストの生成をシンプルに記述できます。また、繰り返し処理をその場で評価せず、処理を内包したイテレータを生成するという機能もあるので、クロージャの生成機構としてふるまわせることも可能になります。

7.2.1. `repeat` 関数

`repeat` 関数の一般式は以下のようになります。

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

`repeat` 関数は、引数で指定した回数だけ `block` の処理を繰り返します。引数は省略可能で、省略した場合無限ループになります。

ブロックを評価するとき、ブロックパラメータを `|idx:number|` という形式で渡します。`idx` は 0 から始まるループカウンタです。

7.2.2. `while` 関数

`while` 関数の一般式は以下のようになります。

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

while 関数は、引数で指定した式が条件を満たす間だけ block の処理を繰り返します。
ブロックを評価するとき、ブロックパラメータを |idx:number| という形式で渡します。idx は 0 から始まるループカウンタです。

7.2.3. for 関数

for 関数の一般式は以下のようになります。

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

for 関数は、一つ以上のイテレータ代入式を引数にとり、イテレータが終了するまで block の処理を繰り返します。
ブロックを評価するとき、ブロックパラメータを |idx:number| という形式で渡します。idx は 0 から始まるループカウンタです。
イテレータ代入式の形式は以下のようになります。

symbol in *iterator*
[*symbol1, symgol2* ..] in *iterator*

最初の形式では、イテレータの要素が *symbol* で表される変数に代入されます。もし要素がリストであれば、*symbol* に代入される値はそのリストそのものになります。二番目の形式では、イテレータの要素がリストであればリストの要素ごとに対応する位置にあるシンボルの変数に値を代入します。要素がリストでない場合、全てのシンボルの変数に同じ値が代入されます。
イテレータ代入式が二つ以上指定された場合、一回のループで引数中のイテレータを一つずつ評価していきます。こうして、いずれかのイテレータが終了するまで処理が繰り返されます。つまり、イテレータの要素数が異なるときは、ループの回数は一番短いイテレータの要素数にあわせられます。
イテレータ要素をひとつずつ評価するには、イテレータの each メソッドを使う方法もあります。以下に for 関数を使った場合と、イテレータの each メソッドを使った場合の例を示します。

```
tbl = [1, 2, 3, 4, 5]
for (x in tbl) { … }
tbl.each() {|x| … }
```

イテレータの each メソッドを使った記述の方が簡潔にかけることが多いですが、二つ以上のイテレータを同時に評価したりする場合は for 関数を使うと便利です。

7.2.4. cross 関数

cross 関数の一般式は以下のようになります。

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

cross 関数は、一つ以上のイテレータ代入式を引数にとり、イテレータが終了するまで block の処理を繰り返します。イテレータ代入式が一つのとき、処理内容は for 関数に一つの引数を渡したのと同じです。二つの

イテレータ代入式を指定すると多重ループになり、一つ目のイテレータが外側、二つ目のイテレータが内側のループを構成します。イテレータ代入式を複数指定することも可能で、n 個の代入式を指定すると n 重の多重ループになります。

ブロックを評価するとき、ブロックパラメータを |idx:number, i0:number, i1:number, …| という形式で渡します。idx は 0 から始まるループカウンタで、それに続く i0、i1、… は指定したイテレータそれぞれのインデクス値です。

cross 関数の実行例を以下に示します。

```
>>> cross (x in ['Taro', 'Hanako'], y in 1..3) { println(x, ' ', y) }
Taro 1
Taro 2
Taro 3
Hanako 1
Hanako 2
Hanako 3
```

7.2.5. 繰り返し中のフロー制御

繰り返し関数を途中で抜けるために、break 関数が用意されています。この関数を評価すると、一番内側の繰り返し関数の処理を中断します。一般式は以下のとおりです。

```
break(value?):symbol_func
```

アトリビュート:symbol_func は、この関数が単独のシンボルで記述したときでも、関数呼び出しとして評価することを指定するものです。引数として value を渡すと、中断した繰り返し関数の戻り値をその値に設定します。省略すると、繰り返し関数の戻り値は nil になります。

例を以下に示します。

```
for (str in strList) {
    if (str == 'end') { break }
}
```

繰り返し処理の続きをスキップして先頭に戻るには continue 関数を使います。一般式は以下のとおりです。

```
continue(value?):symbol_func
```

この関数も break 関数と同じように、シンボルのみで関数呼び出しになります。引数として value を渡すと、ループのその回の評価値をその値に設定します。省略すると、その回の評価値は nil になります。

7.2.6. 繰り返し関数によるリストの生成

繰り返し関数 repeat, while, for, cross は、デフォルトでは一番最後のループで評価した値をその関数自体の戻り値とします。しかし、アトリビュート :list または :xlist を指定すると、ループごとの評価値を要素にもつリストを返すようになります。アトリビュート :list を指定すると、すべての評価値を要素に持つリストになります。アトリビュート:xlist では、評価値が nil になるものを要素から除外します。

7.2.7. 繰り返し関数によるイテレータの生成

繰り返し関数 repeat、while、for、cross は、デフォルトでは繰り返し条件に基づいて即座にブロックの内容を評価します。

しかし、アトリビュート :iter または :xiter を指定すると、その場で評価することはせず、ループの内容を一度ずつ評価するイテレータを返すようになります。アトリビュート :iter を指定すると、すべての評価値を返すイテレータになります。アトリビュート:xiter では、評価値が nil になるものをスキップするイテレータを生成します。

この機能を使って、ユーザ定義のイテレータを作成することができます。詳細は「イテレータ」の章を参照ください。

7.3. 例外処理

例外処理を行う try-except シーケンスの一般式は以下のようになります。

```
try {block} except (error*:error) {block} except (error*:error) {block}
```

ひとつの try に対して 1 個以上の任意の数の except を記述することができます。

通常、スクリプトの実行中に例外が発生するとスクリプトが中断されます。しかし、try 関数のブロック中で発生した例外はこの関数が捕捉し、それから後続する except 関数にエラー内容を順番に渡していきます。except 関数は、引数で指定されたエラーインスタンスと渡されたエラー内容を比較し、等しいと判断したときは自身のブロックの内容を実行し、この try-except シーケンスを終了します。もしいずれの except 関数の条件にも合致しないときは、通常どおりのエラー処理が行われます。

except 関数は、0 個以上のエラーインスタンスを引数にとることができます。引数がなにも指定されないと、それまでのexcept 関数で合致しなかった残りのすべての例外をその場で捕捉します。1 個以上指定された場合は、いずれかのエラーインスタンスに合致すれば捕捉することになります。

except 関数に渡すエラーインスタンスは以下のようなシンボル名で定義されています。

```
SyntaxError、ArithmeticError、TypeError、ZeroDivisionError、ValueError
SystemError、IOError、IndexError、KeyError、ImportError、AttributeError
StopIteration、RuntimeError、NameError、NotImplementedError、IteratorError
CodecError、CommandError、MemoryError、FormatError、ResourceError
```

except 関数のブロックは、|error:error| という形式のブロック引数を受け取ります。引数 error は実際に発生したエラーに対応するエラーインスタンスで、エラー種別やメッセージなどをメンバに含みます。ユーザは、この情報をもとに適切な処理を実装することができます。

関数 raise を使って、ユーザが意図的に例外を発生させることもできます。一般式は以下の通りです。

```
raise(error:error, msg:string => 'error', value?)
```

8. 暗黙的マッピング

8.1. 実装のきっかけ

数式 $y = x^2$ のグラフを描画する処理を考えてみます。座標値は、x に-5 から 5 までの数値を 1 きざみで代入したときの y の値を求め、各座標値に対応する画面位置にプロットすることにしましょう。

従来のプログラミング言語でこのような処理を行うには、ループ構文を記述して繰り返し処理するというのが常套手段でした。C 言語であれば、以下のようなプログラムを思い浮かべることができます。

```
const float x[] = {-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5 };
float y[11];
for (int i = 0; i < 11; i++) {
    y[i] = x[i] * x[i];
}
```

関数プログラミングを提唱している言語ならば、同じ処理をするのに高階関数を適用することを思いつくでしょう。LISP の場合、写像処理を行う map を使って以下のように記述できます。

```
(map (lambda (x) (* x x)) '(-5 -4 -3 -2 -1 0 1 2 3 4 5))
```

かなりエレガントに書くことができました。LISP に限らず、高階関数という概念はものごとを抽象的にとらえる強力な武器になります。しかし抽象的な思考というものは、得てしてその道の入門者にとってはとっつきづらいものです。そもそも、ここで実際に解決したいのは、x の数列に対応する x^2 の値を求めるという単純な課題です。以下のような記述で、答えが求まらないものでしょうか。

```
x = [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]
y = x * x
```

x * x という表記は、x にひとつの数値を受け取ることを期待しています。これに対して、x に数値のリストを与えたとき、暗黙的に写像すなわちマッピングを行うようにすれば、ユーザは繰り返し処理を意識することなく結果を得られるようになります。

「暗黙的マッピング」の実装はこのような発想からスタートしました。

8.2. コンセプト

Gura の演算子は、ほとんどすべて暗黙的マッピングが有効になります。これは、数式を構成する四則演算だけでなく、大小比較などの演算子を含みます。ユーザが書いた演算式は、そのまま数列を処理する機能を持つことになります。

演算子に加え、関数（組込み関数とユーザ定義関数）も、暗黙的マッピングの宣言をしていればこの機能が働くようになります。つまり、引数にデータ列を渡すと、データ列の要素ごとにくりかえし関数が実行されるのです。Gura が提供する組込み関数や標準モジュールの関数のほとんどは、暗黙的マッピングの宣言がされています。

8.3. 適用ルール

暗黙的マッピングはまた、アトリビュート `:map` をつけて宣言された関数に対しても働きます。Gura が標準で提供する関数の多くは、この宣言をつけて提供されています。

Gura において、データ列を表現するデータ型はリストとイテレータです。リストやイテレータが、演算子や関数に渡されると、暗黙的マッピングが行われるようになります。

以下説明のため、データ型を 3 つのカテゴリに分類します。つまり、リスト、イテレータ、そしてそれ以外のスカラーです。

暗黙的マッピングによって得られる結果は、関数のアトリビュート指定や、引数のカテゴリがリスト、イテレータまたはスカラーのどれなのかによって異なります。デフォルトの動作では、引数にイテレータがひとつでも含まれると、結果はイテレータになります。以下に戻り値の条件をまとめます。

引数カテゴリ	戻り値
スカラーのみ	スカラーについて関数を実行し、結果を返します
スカラーかリスト (イテレータは無い)	リストの要素ごとに関数を実行し、その結果をリストとして返します
イテレータが含まれる	イテレータを結果として返します

暗黙的マッピング宣言された関数 `func(a, b)` の呼び出しを例にとって考察します。引数 a, b にスカラー、リスト、イテレータを渡したときの戻り値は以下のようになります。

```
func(scalar, scalar)          scalar
func(scalar, list)            list
func(list, list)              list
func(scalar, iterator)        iterator
func(iterator, list)          iterator
func(iterator, iterator)      iterator
```

戻り値の型を変えたい場合は、関数呼び出しでアトリビュートを指定します。暗黙的マッピングの戻り値を変更するアトリビュートの一覧を以下に示します。

アトリビュート	説明
`:list`	リストを返します。
`:xlist`	`nil` 値を要素から除外したリストを返します。
`:set`	重複する値を要素から除外したリストを返します。
`:xset`	`nil` 値と重複する値を要素から除外したリストを返します。
`:iter`	イテレータを返します。
`:xiter`	`nil` 値をスキップするイテレータを返します。

8.4. ケーススタディ

8.4.1. 演算子と暗黙的マッピング

Gura の演算子に暗黙的マッピングを適用した例を以下に示します。

```
>>> [1, 2, 3, 4] + [5, 6, 7, 8]
[6, 8, 10, 12]
>>> [1, 2, 3, 4] + 5
[6, 7, 8, 9]
>>> ([1, 2, 3, 4] + [5, 6, 7, 8]) / 2
[3, 4, 5, 6]
>>> [3, 8, 0, 4] < [4, 5, 3, 1]
[true, false, true, false]
```

演算子の暗黙的マッピングと、リスト・イテレータ操作とを組み合わせるといろいろな処理が簡潔に表現できます。

リスト x, y の内積を計算 ……(x * y).sum()
数値リスト x の中で、10 未満の要素をカウント……(x < 10).count()
数値リスト x の中で、3 以上 10 以下の要素をカウント……(3 <= x && x <= 10).count()

8.4.2. 文字列出力との組み合わせ

暗黙的マッピング処理をさまざまなデータ入出力関数や処理関数と組み合わせると、制御構文を記述することなく多くの課題を解決することができます。以下に例をあげます。

```
>>> x = [1, 2, 3, 4]
>>> printf('result = %2d, %2d, %2d, %f\n', x, x * x, x * x * x, math.sqrt(x))
result =  1,  1,  1, 1.000000
result =  2,  4,  8, 1.414214
result =  3,  9, 27, 1.732051
result =  4, 16, 64, 2.000000
```

`x * x` や `math.sqrt(x)` などの式で暗黙的マッピング処理が働いてリスト要素ごとの演算をしています。さらに関数 `printf` の実行でも、リストが引数として与えられたことによってやはりこの機能が作動し、要素ごとの表示処理をします。 `printf` は値を持たない関数なので、結果としてのリストは生成しません。

8.4.3. ファイル入力との組み合わせ

行番号をつけてファイルを表示するプログラムは以下のように書けます。

```
printf('%7d %s', (1..), open('hoge.txt').readlines())
```

`1..` と `stream#readlines` はリストではなくイテレータを返します。`1..` は 1 から始まる無限数列を表しますが、長さの異なるリストやイテレータが与えられた場合は短い方にあわせられるので表示する行数は `stream#readlines` が終了するまでになります。

8.4.4. パターンマッチングとの組み合わせ

以下は正規表現を使ってファイルから情報を抽出し、表示する例です。

```
import(re)
lines = readlines('hoge.h')
println(re.match(r'class (\w+)', lines).skipnil():*group(1))
```

stream#readlines で生成したイテレータ lines を受け取った関数 re.match は、結果として re.match_t インスタンスを要素にするイテレータを返します。 関数 re.match は、パターンに合致しない場合は nil を返すので、イテレータのインスタンスメソッド iterator#skipnil を使って nil 値をスキップするイテレータを生成します。":*" は後述するメンバマッピングオペレータで、上の例ではイテレータの各要素に対して re.match_t#group メソッドを実行しています。

9. メンバマッピング

暗黙的マッピングは、関数の引数にリストやイテレータが渡されたときに、それらを展開して関数を評価する機能でした。メンバマッピングは、メンバアクセスのレシーバになった対象がリストやイテレータだったとき、その要素に対して一つずつメンバアクセス処理をするものです。

メンバマッピングには、マッピングの結果をリストとして得る map-to-list、マッピングの結果をイテレータとして得る map-to-iterator、そして暗黙的マッピングのルールに基づいて要素を走査する map-along という 3 つのモードがあります。モードはレシーバとメンバを結合する演算子によって切り替えます。

モード	演算子	説明
map-to-list	::	リスト中のオブジェクトごとにメンバを評価し、その結果をリストとして返します。例えば、`objs::method()` は以下のコードと同じ結果になります。 `for (obj in objs):list { obj.method() }`
map-to-iterator	:*	リスト中のオブジェクトごとにメンバを評価するイテレータを返します。例えば、`objs:*method()` は以下のコードと同じ結果になります。 `for (obj in objs):iter { obj.method() }`

map-along	:&	引数の値をもとに暗黙的マッピングを行いますが、このときレシーバであるリストの要素も順に走査していきます。例えば、as, bs, cs を何らかのリストと仮定すると、`objs:&method(as, bs, cs)` は以下のコードのように要素を走査します（結果は異なります）。
		`for (obj in objs, a in as, b in bs, c in cs) {`
		`  obj.method(a, b, c)`
		`}`
		この形式は、暗黙的マッピングとメンバマッピングがくみあわさった形と見ることができます。

9.1. ケーススタディ

簡単なクラスを宣言して、メンバマッピングの用例を見ていきます。

以下は、名前と値段を表示する `Print()` というメソッドを持った `Fruit` 構造体を作った後、`Fruit` 構造体のインスタンスのリスト `fruits` を生成しています。

```
Fruit = struct(name:string, price:number) {
    Print() = printf('name:%s  price:%d\n', this.name, this.price)
}
fruits = @(Fruit) {
    { 'apple', 100 }, { 'orange', 80 }, { 'grape', 120 }
}
```

`fruits` の要素について `Print` を実行するには、メンバマッピングを使って以下のように記述します。

```
fruits::Print()
```

値段の合計と平均を計算します。

```
printf('sum = %.1f, average = %.1f\n',
          fruits::price.sum(), fruits::price.average())
```

一番長い名前にそろえて一覧表示します。一見簡単そうなこの処理は、制御構文を使うと意外と煩雑になります。メンバマッピング処理で簡潔な記述が可能になります。

```
printf('%-*s %d\n',
          fruits::name::len().max(), fruits::name, fruits::price)
```

上と同じですが、イテレータとしてメンバマッピングを処理しています。要素数が多いときは、こちらの方が実行速度が速くなります。

```
printf('%-*s %d\n',
          fruits:*name:*len().max(), fruits:*name, fruits:*price)
```

関数インスタンスを使って、値段が 100 円未満のものを表示します。

```
fruits.filter(&{$f.price < 100})::Print()
```

以下の例は、上と同じ処理を、暗黙的マッピングと組み合わせて処理しています。

```
fruits.filter(fruits:*price < 100)::Print()
```

値段や名前をキーにしてソートをします。

```
fruits.sort(&{$f1.price <=> $f2.price})::Print()
fruits.sort(&{$f1.name <=> $f2.name})::Print()
```

10. ユーザ定義クラス

10.1. class 関数

ユーザ定義のクラスを作成するには class 関数を使います。class 関数の一般式は以下のとおりです。

```
class(superclass?:function) {block?}
```

10.2. 基本的なクラス定義

下のスクリプトは、A という名前のクラスを作成する例です。

```
A = class {
    Hello() = {
        println('Hello, ')
    }
}
```

class 関数のブロック内で定義される関数は「メソッド」と呼ばれ、このクラスのインスタンスを操作するための関数として働きます。

変数 A には、クラス A のインスタンスを生成するための関数が代入されます。この関数のことを、クラス A の「コンストラクタ関数」と呼びます。クラス A のインスタンスを生成してメソッドを呼び出す例を以下に示します。

```
a = A()
a.Hello()
```

コンストラクタ関数は、ブロックをとることができます。コンストラクタ関数をブロックをつけて評価すると、|obj| という形式でブロックパラメータを渡してブロックを評価します。obj は生成したインスタンスです。この場合、ブロックで最後に評価した値が、コンストラクタ関数の戻り値になります。ブロックを使うと、上の例は以下のように書くことができます。

```
A() {|a|
    a.Hello()
}
```

この表記では、生成したインスタンスはブロックの評価が終わった時点で消滅します。インスタンスの寿命を限定するときに便利です。

コンストラクタ関数は、インスタンスを生成するだけでなくインスタンスの内部状態を初期化する役目ももっています。以下のように __init__ という名前のメソッドを定義すると、コンストラクタを実行した際、インスタンス生成の後にこのメソッドの内容を実行します。

```
B = class {
    __init__(name:string) = {
        this.name = name
    }
    Hello() = {
```

```
        println('Hello, ', this.name)
    }
}
```

メソッド __init__ には引数を指定することができ、コンストラクタ関数も同じ引数リストを持ちます。上のクラス B を生成する例を以下に示します。

```
b = B('Gura')
```

ところで、上の例において this という名前の変数がメソッド内部で使われています。これはメソッドが属しているインスタンス自身への参照になっています。メソッド__init__では this.name に値を代入していますが、これは B のインスタンス内の変数 name への代入になります。メソッド Hello における this.name の値参照は、同じく B インスタンスの name 変数を参照しています。今後、クラスのインスタンス内で定義される変数を「プロパティ」と呼ぶことにします。

変数 this を使ってメソッドを呼び出すこともできます。以下に例を示します。

```
C = class {
    __init__(name:string) = {
        this.name = name
    }
    Hello() = {
        println('Hello, ', this.DuplicateName(4))
    }
    DuplicateName(n:number) = {
        this.name * n
    }
}
```

10.3. コンストラクタ関数についての詳細

この節では、クラスとコンストラクタ関数の生成について詳しく見ていきます。以下に例をあげます。

```
D = class {}
```

これは D という名前のクラスを生成している例ですが、詳しく見ると二つの処理が行われています。ひとつは、「D という名前のクラス」の作成であり、もう一つは「D という名前の関数」の作成です。

まず class 関数を実行すると、class 型のデータを生成して返します。このとき、class 関数自体はクラス名に関する情報を与えられていませんから、生成する class 型データは名前なしクラスになります。

クラスに名前がつけられるのは、代入演算子 = を評価するときです。この演算子は、右辺が class 型のデータで、また名前がついていない場合、左辺のシンボル値をもとにこのクラスに名前をつけます。さらに、演算子 = はこのクラスを生成するコンストラクタ関数を作成し、シンボル D に割り当てます。

10.4. クラスメソッドとインスタンスメソッド

メソッドの定義をするとき、引数リストの括弧に続いてアトリビュート :static をつけると、そのメソッドはクラスメソッドになります。

クラスメソッドは、クラス名の名前空間内に作成した通常関数としてふるまいます。呼び出すときはクラス名とドット記号 "." に続いてメソッド名と引数リストをつけます。

クラスメソッドの一般名を表記するときは *class*.*method*() のようにクラス名とメソッド名を "." でつなげて表します。これは実際の呼び出し方法のときの記述と同じです。

それに対し、インスタンスメソッドの一般名は *class*#*method*()のようにクラス名とメソッド名を "#" でつなげたもので表記します。これはドキュメントやヘルプなど、メソッドのふるまいを説明する資料でのみ使われる表記方法です。実際の呼び出しでは、例えばインスタンスの変数名が obj だとすると、obj.method() のようになります。

10.5. 継承

クラスを継承する場合は、引数 superclass にスーパークラスのコンストラクタ関数を指定します。省略したときは、Gura のルートクラス object をスーパークラスとします。スーパークラスのコンストラクタに渡す引数は、メソッド __init__ のブロック引数に記述します。

```
Person = class {
    __init__(job:string, name:string, age:number) = {
        this.job = job
        this.name = name
        this.age = age
    }
    Print() = {
        println(this.job, ' : ', this.name, ' : ', this.age)
    }
}
Teacher = class(Person) {
    __init__(name:string, age:number) = {|'teacher', name, age|
    }
}
Student = class(Person) {
    __init__(name:string, age:number) = {|'student', name, age|
    }
}
```

10.6. 特別なメソッド

定義するメソッドの中には、すでに出てきた __init__ メソッドを含め、以下のように特殊な働きをするものがあります。

__init__(…)

コンストラクタ関数の定義をします。この関数で定義した引数やブロック式が、class 関数で返される関数インスタンスの引数になります。

__del__()

インスタンスが削除されるときに呼ばれるメソッドです。

__getprop__(symbol:symbol)

インスタンスに対してプロパティ参照をした際、指定のプロパティがインスタンス内で定義されていないときに呼ばれます。引数 symbol にプロパティ名が渡されるので、対応するプロパティ値を返します。

例えば、foo.bar という式が評価され、foo インスタンスの中にプロパティ bar が存在しないと __getprop__が呼ばれ、symbol に`bar が入ります。

__putprop__(symbol:symbol, value)

インスタンスに対してプロパティ代入をしたときに呼ばれるメソッドです。引数 symbol に設定するプロパティのシンボル、value に値が渡されます。このメソッドで代入処理をした場合は true、しなかった場合は false を返します。

例えば、foo.bar = 3 という式が評価されると __putprop__ が呼ばれ、symbol に`bar、value に数値 3 が入ります。

__getitem__(key)

インスタンスに対してインデクス参照をしたときに呼ばれるメソッドです。引数 key には、キーとして指定された値が渡されます。

例えば、foo['hoge'] という式が評価されると __getitem__ が呼ばれ、key に文字列 "hoge" が渡されます。

__getitemx__()

インスタンスに対して中身が空のインデクス参照をしたときに呼ばれるメソッドです。

例えば、foo[] という式が評価されると __getitemx__ が呼ばれます。

__setitem__(key, value)

インスタンスに対してインデクス代入をしたときに呼ばれるメソッドです。引数 key には、キーとして指定された値、value には代入値が渡されます。

例えば、foo['hoge'] = 3 という式が評価されると __setitem__ が呼ばれ、key に文字列 "hoge" が、value に数値 3 が入ります。

__setitemx__(value)

インスタンスに対して中身が空のインデクス代入をしたときに呼ばれるメソッドです。value には代入値が渡されます。

例えば、foo[] = 3 という式が評価されると __setitemx__ が呼ばれ、value に数値 3 が入ります。

__str__()

インスタンスを文字列として評価するときに呼ばれるメソッドです。

また、以下のメソッドを定義すると、オペレータをオーバーライドすることができます。

メソッド	オーバーライドするオペレータ
__pos__(a)	+a
__neg__(a)	-a
__invert__(a)	~a

__not__(a)	!a
__add__(a, b)	a + b
__sub__(a, b)	a - b
__mul__(a, b)	a * b
__div__(a, b)	a / b
__eq__(a, b)	a == b
__ne__(a, b)	a != b
__ge__(a, b)	a >=b
__le__(a, b)	a <= b
__cmp__(a, b)	a <= > b
__or__(a, b)	a | b
__and__(a, b)	a & b
__xor__(a, b)	a ^ b
__shl__(a, b)	a << b
__shr__(a, b)	a >> b
__seq__(a, b)	a .. b
__seqinf__(a)	a..

10.7. 構造体のユーザ定義

Guraにおける構造体は、クラスの特殊な形式として実装されています。構造体はstruct関数で作成することができます。struct関数の一般式は以下のとおりです。

```
struct(`args+):[loose] {block?}
```

blockにはclass関数のblockと同じように、メソッド定義やクラス変数の定義を記述します。アトリビュート :loose を指定すると、引数すべてがオプショナルになります。

10.8. 既存のクラスへのメソッド追加

代入演算子を使い、クラス宣言をした後にインスタンスやクラスにメソッドを追加することもできます。

```
x = 'hello'
x.hoge() = println('This string is: ', this)
x.hoge()
```

クラスにメソッドを追加する場合も同様です。以下は、文字列クラスにメソッド print を定義する例です。classref関数は組込みクラスの参照を得る関数です。

```
classref(`string).hoge() = println('This string is: ', this)
```

11. モジュール

モジュールは、関数やクラスを提供するファイルです。モジュールには、通常の Gura スクリプトで記述されたスクリプトモジュール（*.gura）と、C++で記述してビルドしたバイナリモジュール（*.gurd）があります。スクリプトをモジュールとして使用する場合、コード中に特別な記述をする必要はありません。

モジュールを現在実行しているスクリプト中にとりこむには、import 関数を使用します。import 関数は、引数としてモジュール名を受け取り、そのモジュール名にサフィックス（.gura または .gurd）をつけたファイルを指定のパスから探索します。探索パスは sys モジュール中の変数 sys.path に配列の形式で指定します。この変数の内容を書き換えると、モジュールの探索処理に反映されます。Windows 環境では、デフォルトで以下の順にモジュールを探索します（gura.exe が存在するディレクトリを %GURA_DIR% で表しています)。

1. カレントディレクトリ
2. %GURA_DIR%\module
3. %GURA_DIR%\module\site

Linux 環境では以下のようになります（ディレクトリのプレフィックスが /usr/local になるか /usr になるかは、インストール時のコンフィグレーションによって決まります)。

1. カレントディレクトリ
2. /usr/local/lib/gura/ または /usr/lib/gura
3. /usr/local/lib/gura/site または /usr/lib/gura/site

import 関数の最も基本的な使い方は、単にモジュール名を引数として渡すものです。例えば、CSV フォーマットの読み書きをするモジュール csv をインポートするには、以下のようにします。

```
import(csv)
```

これで csv モジュールが読み込まれ、csv という名前でモジュール内のシンボルを参照できるようになります。例えば、csv モジュール内の read という関数を呼び出すには、csv.read(stream)のように記述します。

場合によっては、モジュール内のシンボルを現在の名前空間にとりこんで、モジュール名なしに参照したいこともあります。そのような場合は、import 関数の後にブロックを記述し、とりこむシンボル名を列挙します。例えば、csv モジュールの read および write 関数をとりこむには以下のように記述します。

```
import(csv) {read, write}
```

これで、プログラムからは read(stream) のように呼び出すことができます。モジュール内のシンボルをすべて取り込むこともでき、その場合はアスタリスク "*" をブロック内に記述します。以下は、opengl モジュールのすべてのシンボルをとりこむ例です。

```
import(opengl) {*}
```

ただし、シンボルを現在の名前空間にとりこむと、すでにあるシンボル名と衝突してエラーになる可能性があります。モジュール内で定義されているシンボル名がユニークなときだけこの表記を利用してください。

import 関数に二つ目の引数を指定すると、モジュールを別名で取り込むことができます。この機能は、長い名前のモジュールを短い名前で参照する場合などに便利です。以下は、sqlite3 モジュールを sq という名前で参照する例です。

```
import(sqlite3, sq)
```

import 関数をアトリビュート :binary をつけて実行すると、バイナリモジュールのみをインポート対象にします。これは、同じ名前のスクリプトモジュールとバイナリモジュールを用意しておき、スクリプトモジュールから対応するバイナリモジュールをインポートするときに使用します。この機構により、基本機能をバイナリモジュールで提供しておき、それをスクリプトモジュールで拡張することが可能になります。

import 関数をアトリビュート :mixin_type をつけて実行すると、現在の名前空間にモジュール中の型シンボルをとりこみます。

12. リストとイテレータ

12.1. 概要

リストは、任意の要素を集めたものです。数値や文字列、オブジェクトなどをカンマで区切り、ブラケットで囲むと、それはリストになります。

イテレータは、コンテナ内の要素を順に取得または評価するための機構です。イテレータのもっとも一般的な用途は、繰り返し処理を行う for 構文などに渡して、コンテナ内の要素に順にアクセスするというものです。リストは代表的なコンテナのひとつです。

リストとイテレータは、操作方法が非常によく似ています。また、リストからイテレータに変換したり、イテレータからリストに変換したりすることは、ごく普通に行われる操作です。このため、両者の違いは普段あまり意識する必要がないかもしれません。しかし、要素としてのデータの存在期間が問題になるとき、注意が必要になります。

Gura におけるリストやイテレータの役割は、他の言語よりもずっと重要です。なぜなら、これらは暗黙的マッピングや、メンバマッピングに適用する基本的なデータだからです。そのため、Gura では豊富な種類のイテレータを容易しています。これらを組み合わせると、今まで制御構文で行っていた処理がもっと簡潔な記法で実現できるようになります。

12.2. 有限イテレータと無限イテレータ

イテレータには、有限イテレータと無限イテレータがあります。

有限イテレータは、走査に先立って要素の総数があらかじめ分かっているイテレータです。例えば、数列 `1..10` は代表的な有限イテレータです。

一方、無限イテレータは、要素の数が不明なものを指します。実際に走査を始めたら有限な個数で終了したという場合でも、あらかじめ要素数を知る手段が得られないものは無限イテレータと呼ばれます。無限数列 `1..` は代表的な無限イテレータです。

このような区別をつけるのは、イテレータ操作の中には要素数があらかじめ分かっていなければいけないものがあるからです。例えば、要素数を返す `iterator#count` メソッドや、要素を逆順に操作するイテレータを生成する `iterator#reverse` などがこれにあたります。無限イテレータにこれらの操作を行うとエラーになります。

また、イテレータをリストに変換するような操作を無限イテレータに適用すると、エラーになります。

12.3. イテレータ操作とブロック式

イテレータを返す関数は、オプションでブロック式を受け付けます。関数呼び出しの際にブロックが指定されると、イテレータの要素ごとに繰り返しブロックの内容を評価します。このとき、`|value, idx:number|` という形式でブロック引数が渡されます。`value` は要素の値、`idx` はループのインデクス数値です。

12.4. リストの生成

イテレータをブラケットで囲むと、イテレータを展開した結果得られる要素を持つリストになります。複数のイテレータをカンマで区切ってブラケットで囲むと、それぞれのイテレータを展開して要素に追加します。

イテレータを返す関数にアトリビュート :list をつけて実行するとリスト生成をすることができます。例えば、指定の範囲の数列を出力する range 関数にリストを出力するよう指示するには以下のようにします。

```
range(10):list
```

アトリビュート :list といっしょにブロック式の指定がされると、ループごとのブロックの評価値を要素に持つリストが生成されます。以下の例は、二乗値を要素に持つリストの生成になります。

```
range(10):list {|x| x * x}
```

12.5. 要素操作ダイジェスト

リストの要素操作をするメソッドを使ったスクリプトとその実行結果を示します。メソッドの詳細については「Gura ライブラリリファレンス」を参照ください。

x が [A, B, C, D, E, F, G, H, I, J] というリストである場合:

スクリプト	実行結果
x.head(3):list	[A, B, C]
x.tail(3):list	[H, I, J]
x.offset(3):list	[D, E, F, G, H, I, J]
x.skip(2):list	[A, D, G, J]
x.fold(3):list	[[A, B, C], [D, E, F], [G, H, I], [J]]
x.reverse():list	[J, I, H, G, F, E, D, C, B, A]
x.shuffle():list	[H, D, F, E, I, B, G, A, J, C] (結果はランダム)

x が [A, B, C, D] というリストである場合:

スクリプト	実行結果
x.round(14):list	[A, B, C, D, A, B, C, D, A, B, C, D, A, B]
x.pingpong(10):list	[A, B, C, D, C, B, A, B, C, D]
x.combination(3):list	[[A, B, C], [A, B, D], [A, C, D], [B, C, D]]
x.permutation(3):list	[[A, B, C], [A, B, D], [A, C, B], [A, C, D], [A, D, B], [A, D, C], [B, A, C], [B, A, D], [B, C, A], [B, C, D], [B, D, A], [B, D, C], [C, A, B], [C, A, D], [C, B, A], [C, B, D], [C, D, A], [C, D, B], [D, A, B], [D, A, C], [D, B, A], [D, B, C], [D, C, A], [D, C, B]]

12.6. ユーザ定義イテレータ

新しいデータ型に対して独自のイテレータを定義したい場合は、以下の関数を使うことができます。

- 繰り返し関数 repeat、while、for および cross
- 汎用イテレータ関数 iterator

12.6.1. 繰り返し関数によるイテレータ定義

繰り返し関数 repeat は指定回数だけブロックを評価します。以下は "Hello" という文字列を 10 回画面に表示する例です。

```
repeat (10) {
    println('Hello')
}
```

Gura は繰り返し関数に:iter というアトリビュートをつけることで、イテレータを生成することができます。上の例で、repeat にアトリビュート:iter をつけたコードを以下に示します。

```
x1 = repeat (10):iter {
    println('Hello')
}
```

これはいったいどういう意味を持つのでしょうか。まず、repeat 関数を評価した時点では、ブロックの内容は実行されず、画面には何も表示されません。ここで行われているのは、「"Hello" を 10 回表示するイテレータ」を生成し、それを変数 x1 に代入することです。実際に評価を行うには、以下のように each メソッドを実行します。

```
x1.each() {}
```

イテレータというのは、通常、値を順次返すものを指しますので、上の例は「イテレータ」と呼ぶには少々難がありそうです。繰り返し関数によるイテレータ生成では、繰り返しブロック内の最後に評価された式の値が、イテレータの要素になります。以下で生成しているのは 10 未満の 2 の倍数を返すイテレータです。

```
x2 = repeat (5):iter {|i|
    i * 2
}
```

生成したイテレータをリスト化して内容を確かめてみます。

```
>>> [x2]
[0, 2, 4, 6, 8]
```

通常の繰り返し処理と同じように、break や continue を使ってフローを制御することもできます。
上と同じ処理をbreakを使って書いた例を以下に示します。ループ回数を大きく設定し、指定の回数になったら break でぬけるようにしています。

```
x3 = repeat (100000):iter {|i|
    (i == 5) && break
    i * 2
}
```

評価結果は以下のようになります。

```
>>> [x3]
[0, 2, 4, 6, 8]
```

同じく continue を使った例を考えてみます。10 までの数値を作り、条件に合致しない数値の場合は continue でスキップします。

```
x4 = repeat (10):iter {|i|
    (i % 2 == 1) && continue
    i * 2
}
```

これの評価結果は以下のようになります。

```
>>> [x4]
[0, nil, 2, nil, 4, nil, 6, nil, 8, nil]
```

期待した数値列の間に nil 値が入ってしまいました。これは continue を評価した際のループの評価値が nil になるので、これが要素として扱われるためです。

アトリビュート:xiter を指定すると、要素から nil 値をとりのぞくことができます。書きなおした例を以下に示します。

```
x5 = repeat (10):xiter {|i|
    (i % 2 == 1) && continue
    i * 2
}
```

これの評価結果は、以下のように期待どおりのものになります。

```
>>> [x5]
[0, 2, 4, 6, 8]
```

少し複雑な例として、多重ループのイテレータを生成してみます。以下は、1から3までの数値の3つの組み合わせを返すイテレータの例です（これと同じ処理はcross関数を使うともっと簡単に実現できますが、多重ループの例としてとりあげています)。

```
x6 = repeat (2):iter {|i|
    repeat (2):iter {|j|
        repeat (3):iter {|k|
            [i, j, k]
        }
    }
}
```

評価結果は以下のようになります。

```
>>> [x6]
[[0, 0, 0], [0, 0, 1], [0, 0, 2], [0, 1, 0], [0, 1, 1], [0, 1, 2], [1, 0,
0], [1, 0, 1], [1, 0, 2], [1, 1, 0], [1, 1, 1], [1, 1, 2]]
```

12.7. 汎用イテレータ関数によるイテレータ定義

関数 iterator を使うと、イテレータや値を結合し、任意のデータ列を返すイテレータを定義することができま

す。

以下は [3, 1, 4, 1, 5, 9, 3] というデータ列を返すイテレータです。

```
x7 = iterator(3, 1, 4, 1, 5, 9, 3)
```

イテレータを要素にすると、そのイテレータの内部の要素を展開します。例を以下に示します。

```
x8 = iterator(1..5, 3, 9, 2, 8..3)
```

評価結果は以下の通りです。

```
>>> [x8]
[1, 2, 3, 4, 5, 3, 9, 2, 8, 7, 6, 5, 4, 3]
```

13. 数学に関する機能

13.1. 複素数計算

四則演算、マトリクス演算に対応しています。

13.2. 統計処理

合計・分散値・平均値・標準偏差を算出します。

13.3. 順列

`list#permutation`および`list#combination`メソッドにより、順列および組み合わせによる要素抽出を行います。

13.4. 行列演算

`matrix`クラスを使い、以下の行列演算ができます。

- 加算・減算・乗算
- 逆行列
- 転置行列

行列の要素には、実数および複素数を入れることができます。

13.5. 式の微分演算

式そのものを微分することができます。合成式の微分は以下の公式に則って式を導き出しています。

$\{f(x) + g(x)\}' = f'(x) + g'(x)$
$\{f(x) - g(x)\}' = f'(x) - g'(x)$
$f(g(x))' = f'(u)g'(x)$
$\{f(x)g(x)\}' = f'(x)g(x) + f(x)g'(x)$
$\{f(x) / g(x)\}' = \{f'(x)g(x) - f(x)g'(x)\} / g(x)^2$
$\{f(x)^{g(x)}\}' = f'(x)g(x)f(x)^{g(x)-1} + g'(x)\{\log f(x)\}f(x)^{g(x)}$

微分式を得るには、`function` クラスの `diff` メソッドを使います。以下のように数式からなる関数を定義し、`diff` メソッドを実行すると、微分式からなる関数を返します。関数の定義内容は `expr` プロパティで確認できます。

```
>>> f(x) = math.sin(x) ** 2
f(x)
>>> g = f.diff()
g(x)
>>> g.expr
`(math.cos(x) * 2 * math.sin(x))
```

得られる結果の例を以下に示します。

式	微分結果
`x ** 2`	`2 * x`
`x ** 3`	`3 * x ** 2`
`x ** 4`	`4 * x ** 3`
`a ** x`	`math.log(a) * a ** x`
`math.sin(x)`	`math.cos(x)`
`math.cos(x)`	`-math.sin(x)`
`math.tan(x)`	`1 / math.cos(x) ** 2`
`math.exp(x)`	`math.exp(x)`
`math.log(x)`	`1 / x`
`math.log10(x)`	`1 / (x * math.log(10))`
`math.asin(x)`	`1 / math.sqrt(1 - x ** 2)`
`math.acos(x)`	`(-1) / math.sqrt(1 - x ** 2)`
`math.atan(x)`	`1 / (1 + x ** 2)`
`math.sqrt(x)`	`1 / (2 * math.sqrt(x))`
`math.sin(x) ** 2`	`math.cos(x) * 2 * math.sin(x)`
`math.sin(x ** 2)`	`math.cos(x ** 2) * (2 * x)`
`math.log(math.sin(x))`	`(1 / math.sin(x)) * math.cos(x)`
`x ** 2 * math.sin(x)`	`2 * x * math.sin(x) + x ** 2 * math.cos(x)`
`math.sin(x) / x ** 2`	`(math.cos(x) * x ** 2 - math.sin(x) * (2 * x)) / x ** 2 ** 2`
`3 ** 2 * x`	`2 * math.log(3) * 3 ** 2 * x`
`math.log((x ** 2 + 1))`	`(1 / (x ** 2 + 1)) * (2 * x)`
`(x - 1) ** 2 * (x - 2) ** 3 / (x - 5) ** 2`	`((2 * (x - 1) * (x - 2) ** 3 + (x - 1) ** 2 * (3 * (x - 2) ** 2)) * (x - 5) ** 2 - (x - 1) ** 2 * (x - 2) ** 3 * (2 * (x - 5))) / (x - 5) ** 2 ** 2`
`math.sin(2 * x - 3)`	`math.cos(2 * x - 3) * 2`
`math.cos(x) ** 2`	`-math.sin(x) * 2 * math.cos(x)`
`(2 * x - 1) ** 3`	`6 * (2 * x - 1) ** 2`
`math.sqrt(x ** 2 + 2 * x + 3)`	`(1 / (2 * math.sqrt(x ** 2 + 2 * x + 3))) * (2 * x + 2)`
`1 / x`	`(-1) / x ** 2`
`math.exp(x) + math.exp(-x)`	`math.exp(x) - math.exp(-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.cos(x + 2) + 1) * (math.sin(x + 3) + x + 3) + (math.sin(x + 2) + x + 2) * (math.cos(x + 3) + 1)`
`math.sin(math.sin(x ** 2 / 3))`	`math.cos(math.sin(x ** 2 / 3)) * (math.cos(x ** 2 / 3) * ((2 * x * 3) / 9))`

14. パス名の操作

14.1. Gura におけるパス名

一般にパス名というと、多くの場合ハードディスクやソリッドストレージデバイスなどのファイルやディレクトリの名前を指します。しかし、Gura における「パス名」はそれよりも適用範囲が広く、ファイルシステム上の資源はもちろん、それ以外にもネットワーク資源の名前や、アーカイブファイルの中身なども含まれます。以下に Gura で扱えるパスの種類をまとめます。

- ファイルシステム内のパス
- インターネットの URI パス
- アーカイブファイル内のパス

Gura は、こういったパス名で指し示される資源、すなわちファイルやディレクトリに対し、データの読み書きをするストリームの生成や、要素一覧の取得などを行う仕組みを提供します。また、モジュールをインポートすることで、まったく新しいプロトコルによるパス名の解釈や実際の資源操作を追加することもできます。

14.1.1. ファイルシステム内のパス

ファイルシステム上のファイルやディレクトリの例を以下に示します。

```
/home/yamada/work
C:\Windows\Media\chimes.wav
```

パス名を記述する際のセパレータは、スラッシュ "/" またはバックスラッシュ "\" を指定します。どちらを使って記述しても、実際にファイルをオープンしたりディレクトリを指定したりする際に、現在動作している OS に応じて適切に変換が行われます。バックスラッシュはエスケープする必要があるので、スラッシュ "/" を使って記述した方がすっきりとすることが多いでしょう。

14.1.2. インターネットの URI パス

インターネットの URI パスの例を以下に示します。

```
http://sourceforge.jp/
ftp://foo.hoge.com/dir1/dir2/
```

URI パスは、インターネットの通信プロトコル名に続いてコロン、スラッシュと資源へのパス名が記述されます。Gura は、通信プロトコルに対応するモジュールをインポートすることでこれらのパスにアクセスができるようになります。

14.1.3. アーカイブファイル内のパス

アーカイブファイルの中のパスの例を以下に示します。

```
hoge.zip/foo/bar.txt
footool.tar.gz/src/main.c
```

```
hoge.zip/
```

hoge.zip や footool.tar.gz はそれぞれ ZIP 形式および tar 形式でファイルをまとめたアーカイブファイルです。これらのアーカイブファイルに対応したモジュールをインポートすることで、アーカイブファイル名に続けて内部のパスを指定することができます。

言うまでもなく、アーカイブファイル自体はファイルシステムまたはインターネット上の資源として存在します。デフォルトではファイルシステム上にあるアーカイブファイルを扱えますが、必要なモジュールをインポートすることでインターネット上のアーカイブファイルを指定し、さらにその内部のパスを指定することができます。以下に例を示します。

```
/home/yamada/work/hoge.zip/foo/bar.txt
http://sourceforge.jp/hoge.zip/foo/bar.txt
ftp://foo.hoge.com/dir1/dir2/hoge.zip/foo/bar.txt
```

14.2. ディレクトリ操作

パス名が指すストレージやプロトコルがディレクトリサーチに対応していれば、ファイルの一覧や検索が可能になります。

ディレクトリ内の要素をサーチするため、以下の関数を用意しています。詳細は「Gura ライブラリリファレンス」を参照ください。

関数	説明
path.dir	パス名で指定したディレクトリに含まれるファイルまたはディレクトリをサーチします
path.walk	パス名で指定したディレクトリを基点として、含まれるファイルまたはディレクトリを再帰的にサーチします
path.glob	パターンに適合するファイルやディレクトリをサーチします

15. ストリーム

15.1. 概要

Gura では、ストレージ中などにあるファイルを「ストリーム」という抽象化されたインターフェースを使って読み書きします。この仕組みにより、データの実体が実際にどこに格納されているか、また、どのようなプロトコルでアクセスするかを言語が判断し、適切なモジュールを使って処理を行います。例えば、ストリームを使って以下のようなファイルにアクセスすることができます。

- ディスクストレージ中のファイル
- HTTP プロトコルで取得するファイル
- アーカイブファイル中のファイル

15.2. ストリームの種類

ストリームは以下のものに大別されます。

- ソースストリーム
- デスティネーションストリーム
- フィルタストリーム

ソースストリームはデータの入力元になるストリームです。`stream#read` をはじめとするデータ読み込みメソッドを使ってデータを取得します。

デスティネーションストリームはデータの出力先になるストリームです。`stream#write` などのデータ書き込みメソッドを使ってデータを出力します。

フィルタストリームは、ソースストリームまたはデスティネーションストリームにアタッチするストリームです。フィルタストリームに読み込み操作を行うと、アタッチしたストリームからデータを読み込み、それを加工した結果を返します。また、フィルタストリームに書き込み操作を行うと、書き込みデータを加工し、その結果をストリームに書き込みます。

15.3. ストリームの生成

ストリームを生成する代表的な関数は open です。一般式は以下のとおりです。

```
open(name:string, mode:string => 'r', encoding:string => 'utf-8'):map {block?}
```

引数 name に、ストリームを表すパス名を指定します。引数 mode はアクセス方法の指定で、読み込みのとき "r"、書き込みには "w"、追加は "a" を指定します。引数 encoding には、ストリームの内容をテキストデータとして入出力するときに使用する文字コーデックの名前を指定します。デフォルトでは utf-8 が使用されます。

また、open 関数で明示的にオープンしなくても、stream 型の値を指定した引数に文字列を渡すと、自動的に stream 型にキャストがされます。例えば、テキストファイルから行ごとに文字列を読み込む関数 readlines は、最初の引数に stream を受け取るので、open 関数を使って以下のように実行します。

```
lines = readlines(open('hoge.txt'))
```

型キャストを使うと、以下のように記述できます。

```
lines = readlines('hoge.txt')
```

15.4. コーデックの指定

Gura にあらかじめ組み込まれている文字コーデックを以下に示します。

モジュール	コーデック
codecs.basic	us-ascii, utf-8, utf-16, base64
codecs.iso8859	iso-8859-1, iso-8859-2, iso-8859-3, iso-8859-4, iso-8859-5, iso-8859-6, iso-8859-7, iso-8859-8, iso-8859-9, iso-8859-10, iso-8859-11, iso-8859-12, iso-8859-13., iso-8859-14, iso-8859-15, iso-8859-16
codecs.japanese	euc-jp, cp932, shift_jis, ms_kanji, jis, iso-2022-jp

モジュールを追加することで、新たな文字コーデックに対応させることができます。

15.5. 標準入出力

コンソールに対する入出力処理もストリームとして扱います。標準入力・標準出力・標準エラー出力に対応するストリームが、sys モジュールと os モジュールでそれぞれ以下の変数で定義されています。

標準入力	sys.stdin	os.stdin
標準出力	sys.stdout	os.stdout
標準エラー出力	sys.stderr	os.stderr

これらのストリームは以下の用途で使用します。

- 関数 print、println、printf は出力先のストリームとして sys.stdout を参照します。
- 関数 os.exec は、起動した外部プロセスの標準出力の内容を os.stdout に、標準エラー出力の内容を os.stderr に出力します。

15.6. プロセス実行と標準入出力

標準入出力を設定する変数の内容をほかのストリームインスタンスで置き換えると、入出力がそのストリームに切り替わります。これは、外部プロセスの出力内容をとりこむときなどに便利です。以下は、外部プロセスの標準出力の内容を binary 型のバッファにとりこむ例です。

```
buff = binary()
os.stdout = buff.stream()
os.exec('program')
```

標準入出力を切り替える処理は比較的頻繁に行われますが、このとき変数の設定を上記のように直接行うと記述が煩雑になります。これを解決するため os.redirect という関数が用意されています。以下は上の処理をこの関数を使って書きなおした例です。

```
buff = binary()
os.redirect(nil, buff) {
    os.exec('program')
}
```

切り替えた効果がブロックの範囲内に限定されるので、処理が分かりやすくなります。

15.7. テキストアクセスとバイナリアクセス

ストリームで扱うデータは単なるバイト列です。この意味で言うとストリームにおけるデフォルトのアクセスフォーマットはバイナリデータであると考えることができます。ストリームで扱うデータをバイナリデータとして扱うか、テキストデータとして扱うかは、ストリームを操作するメソッドによって決まります。open 関数に渡すエンコーディング指定は、テキストアクセスのメソッドのみで有効になり、バイナリアクセスのメソッドには影響しません。つまり、ストリームをバイナリとして扱うことが分かっていれば、open 関数のエンコーディング指定は気にする必要はありません。

ストリームの内容をバイナリデータとして扱うメソッドには以下のものがあります。

```
stream#read(len?:number)
stream#write(buff:binary):reduce
stream#seek(offset:number, origin?:symbol):reduce
stream#tell()
stream#copyto(stream:stream):map:reduce
stream#copyfrom(stream:stream):map:reduce
stream#compare(stream:stream):map
```

ストリームの内容をテキストデータとして扱うメソッドには以下のものがあります。

```
stream#print(values*):map:void
stream#println(values*):map:void
stream#printf(format:string, values*):map:void
stream#readtext()
```

```
stream#readline():[chop]
stream#readlines(nlines:number):[chop] {block?}
```

その他にも、引数としてストリームを受け取る関数やメソッドがあり、それぞれストリームデータの扱いが異なります。例えば、JPEG ファイルの読み書きならばバイナリデータとして扱いますし、CSV ファイルならばテキストファイルとして見るでしょう。

15.8. ストリーム間のデータコピー

入力用ストリームから出力用ストリームにデータをコピーするための関数として copy が用意されています。

copy 関数の最も簡単な使用方法は、コピー元のファイル名とコピー先のファイル名を指定して内容をコピーするというものです。以下に例を示します。

```
copy('src.txt', 'dest.txt')
```

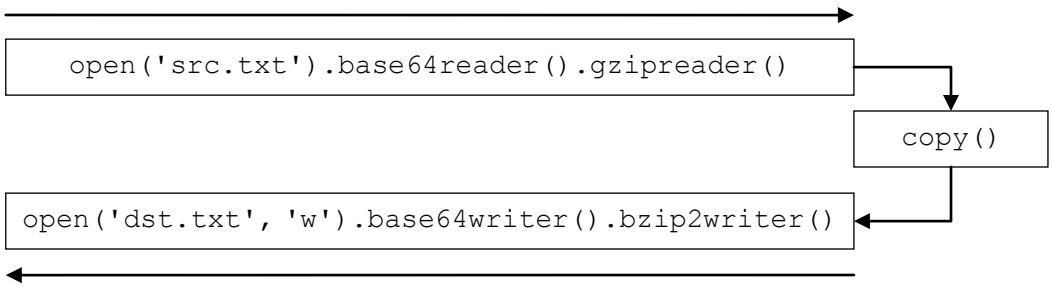
内部の処理では、ファイル名がストリームに変換されるので、上の処理は以下のように記述したのと同じです。

```
copy(open('src.txt', 'r'), open('dest.txt', 'w'))
```

最初の例の方が簡単ですが、二番目に示したものの方がより柔軟性に富んだ処理をすることができます。例えば、ファイル src.txt に base64 でエンコードされた gzip 形式のデータが格納されているとしましょう。このデータを展開し、今度は bzip2 形式で圧縮して再び base64 でエンコードした結果を dst.txt に格納する処理は以下のように記述することができます。

```
copy(open('src.txt').base64reader().gzipreader(),
     open('dst.txt', 'w').base64writer().bzip2writer())
```

データの流れを模式的に表すと以下のようになります。



15.9. スクリプトファイルの実行

15.9.1. アーカイブ中のスクリプトファイル

Gura はほとんどのデータ入出力をストリームとして扱いますが、これはスクリプトファイルそのものも例外ではありません。例えば、ZIP アーカイブの中にあるスクリプトファイルを、展開することなく以下のように直接実行することができます。

```
gura -i zip archive.zip/hello.gura
```

"-i" 引数により zip モジュールをインポートし、ZIP アーカイブ内のスクリプトファイルを指定して実行しています。

スクリプトファイルのサフィックスが .gurc または .gurcw のファイルをコンポジットファイルと呼び、これらにはスクリプトファイルや他のファイルを梱包することができますが、これは ZIP アーカイブ内のストリームにアクセスする機構を使って実現されています。Gura はコンポジットファイルのサフィックスを見つけると、自動的に zip モジュールをインポートし、アーカイブファイル内へのアクセスができるようにします。

15.9.2. HTTP 上のスクリプトファイル

HTTP サーバ上にあるスクリプトファイルも、以下のように実行できます。

```
gura -i net.http http://aaa.bbb.ccc/hello.gura
```

16. イメージ

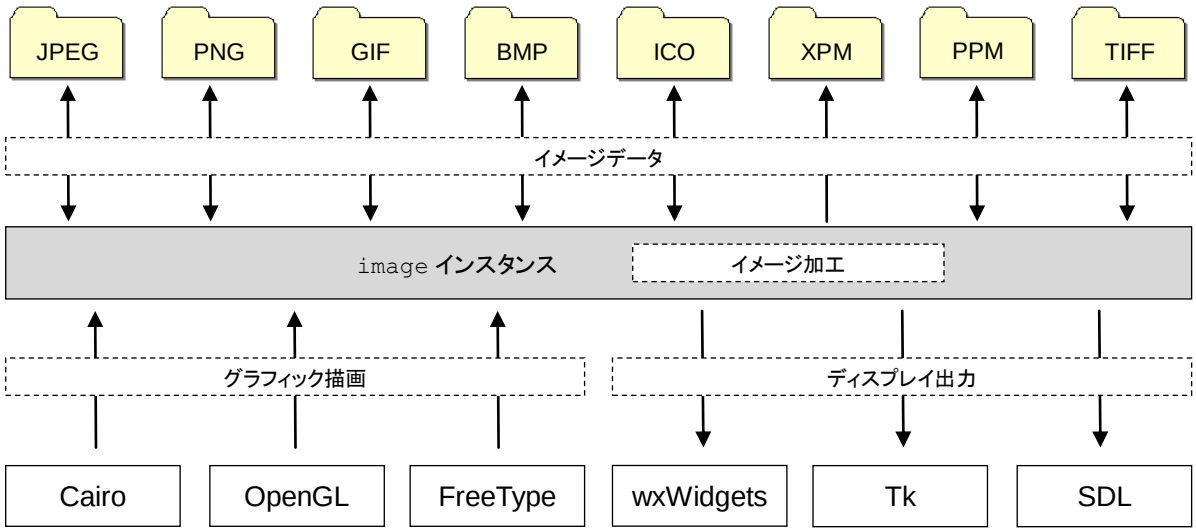
16.1. 概要

かつてのコンピュータ操作はテキストのやりとりが中心でしたが、今ではグラフィカルユーザインターフェースによるものに完全に移行しました。また、Web ブラウザを中心としたインターネットアクセスをぬきにしては今日のプログラミング技術は語れません。そのような中にあって最も重要な位置を占めるのが、グラフィックイメージ（以下、単にイメージと呼びます）の操作です。

イメージ操作には、イメージデータ入出力・フィルタ処理・グラフィック描画・ディスプレイ出力などがあり、それぞれの処理においてライブラリが発表されています。このため、ライブラリの処理の間でイメージデータのやりとりをすることは頻繁に行われることの一つです。イメージデータは赤・緑・青の三原色と、透明度をあらわすアルファ値で表現され、これらの要素を順番にメモリの中に格納するフォーマットが一般にとられます。このとき、ライブラリによって格納するバイト順やアラインメントが異なっているために、変換処理などの煩雑な手続きが必要な場合が少なくありません。

そこで Gura は、イメージデータを言語の標準的なデータ型と位置づけ、イメージを操作するモジュール群はこのデータ型を中心に実装する方針をとりました。これにより、いろいろな処理をするライブラリ・モジュール間のデータ交換を自然な形で実装することができるようになります。

例えば標準の Tcl/Tk ライブラリで扱えるイメージフォーマットは GIF と PPM だけです。しかし、Gura に組み込まれた tk モジュールは Gura のイメージ型を扱うように実装されているため、Gura のモジュールが対応している PNG や JPEG などのイメージも Tk のキャンバスに表示できます。また、Gura のイメージインスタンスを Cairo や OpenGL などのグラフィック描画ライブラリの描画対象にすることができるので、イメージへの重ね描きをしたり、描画結果を任意のイメージフォーマットで出力したりすることができます。



16.2. ブランクイメージを生成する

以下の形式で image 関数を呼び出すと、ブランクのイメージインスタンスを生成します。

```
image(format:symbol, width:number, height::number, color?:color) {block?}
```

format はイメージインスタンスのデータ内部表現で、RGB 要素のみを持つ`rgb かアルファ要素も含む`rgba を指定します。省略すると、`rgba が使われます。

width、height にはイメージの幅と高さをそれぞれピクセル単位で指定します。

color は生成時に塗りつぶす色指定です。省略すると黒になります。

16.3. ストリームからのイメージデータ読み込み

ストリームからイメージデータを読み込むときは、image 関数を以下の形式で呼び出します。

```
image(stream:stream:r, format?:symbol, imgtype?:string) {block?}
```

引数 stream は、イメージデータを読み込むストリームです。この引数に文字列を渡すと、それをパス名として解釈して stream 型にキャストし、イメージデータを読み込みます。

format はイメージインスタンスのデータ内部表現で、RGB 要素のみを持つ`rgb かアルファ要素も含む`rgba を指定します。省略すると、`rgba が使われます。

imgtype には、"jpeg" や"png" というようにイメージタイプ名を文字列で指定します。この引数が省略されると、イメージファイルのヘッダ情報やファイル名のサフィックスからイメージタイプを識別します。

イメージファイルの読み込みをするには、対応するモジュールをあらかじめインポートしておく必要があります。モジュールとサポートするイメージファイルは以下のとおりです。

モジュール名	サポートするイメージファイル	イメージタイプ名
bmp	Windows BMPファイル	bmp
msico	Windowsアイコンファイル	msico
ppm	PPMファイル	ppm
jpeg	JPEGファイル	jpeg
png	PNGファイル	png
gif	GIFファイル	gif

モジュールを新規に開発することで、新しいイメージタイプに対応させることが可能です。

イメージタイプによっては、アニメーション GIF のように複数のイメージデータをひとつのファイルに格納していたり、イメージ特有のプロパティデータを持っているものがあります。これらの情報は、各モジュールが提供する関数やクラスで操作することができます。詳細は、モジュールのリファレンスを参照してください。

ブロック式をつけると、|img:image| という形式でブロック引数を渡してブロックを評価します。img は生成したイメージのインスタンスです。

16.4. ストリームへのイメージデータ書き込み

ストリームにイメージデータを書き込みを行うメソッド image#write が用意されています。一般式は以下のとおりです。

```
image#write(stream:stream:w, imgtype?:string):map:reduce
```

引数 stream は、イメージファイルを書き込むストリームです。この引数に文字列を渡すと、それをパス名として解釈して stream 型にキャストし、イメージデータを書きこみます。

imgtype には、"jpeg" や"png" というようにイメージタイプ名を文字列で指定します。この引数が省略されると、イメージファイルのヘッダ情報やファイル名のサフィックスからイメージタイプを識別します。

イメージファイルの読み込みをするには、対応するモジュールをあらかじめインポートしておく必要があります。モジュールとサポートするイメージファイルは、「ファイルからの読み込み」の節を参照ください。

このメソッドは、一つの画像データのみの書き込みに対応しています。しかしイメージタイプによっては、アニメーション GIF のように複数のイメージデータをひとつのファイルに格納していたり、イメージ特有のプロパティデータを持っているものがあります。こういったファイルの書き込みは、各モジュールが提供する関数やクラスを使うことで可能になります。詳細は、モジュールのリファレンスを参照してください。

16.5. イメージ加工

image クラスには、以下の操作を行うメソッドが用意されています。

メソッド	操作
image#crop	イメージ切り出し
image#flip	左右・上下反転
image#paste	イメージ貼り付け
image#reducecolor	減色処理
image#resize	サイズ変更
image#rotate	任意の角度の回転
image#thumbnail	サムネイル画像生成

16.6. グラフィック描画

二次元グラフィックを描画したいときは、ライブラリ Cairo をサポートするモジュール cairo が便利です。

三次元グラフィックライブラリ OpenGL をサポートするモジュール opengl を使うと、Z バッファを使った高度な三次元グラフィック描画ができます。

テキストを扱いたいだけであれば、ライブラリ FreeType をサポートする freetype モジュールで手軽にテキストをイメージに埋め込むことができます。

16.7. ディスプレイ出力

GUI を構築する wxWidgets モジュール wx と、Tcl/Tk モジュール tk を用意しています。

また、高速な画面表示を可能にする SDL (Simple Direct Layer) のモジュール sdl が用意されています。

17. テンプレートエンジン

テンプレートエンジンを使うと、任意のテキスト文字列の中に Gura スクリプトを埋め込み、スクリプトの実行結果を文字列中に挿入することができます。テンプレートエンジンを起動には以下の方法があります。

コマンドライン

オプション–T を使ってテンプレートを記述したテキストファイルを指定すると、その内容を評価します。

関数呼び出し

メソッド `string#template()`、`stream#template()` または関数 `template()` を使い、文字列またはストリーム中に記述されているテンプレート文字列に対して評価を行います。

Gura スクリプトは `"${"` と `"}"` にはさんで記述します。この内部は、通常の Gura スクリプトとして扱われるので、改行などを含んでいてもかまいません。通常のテキストの中に `"${"` という文字の並びがあり、これをスクリプトでなく通常文書として扱う場合は `"$${"` と記述します。スクリプト中の空白や改行は結果に影響を与えません。

評価結果を出力するときのルールは以下の通りです。

- 結果が文字列のとき、その内容を出力します。
- リストやイテレータの場合、その要素を文字列に変換して結合した結果を出力します。
- それ以外の `nil` 以外の要素は、文字列に変換されて出力されます。
- 行の先頭からスクリプト開始の `"${"` の間に空白が存在し、スクリプトの結果が複数行にわたる場合は、先頭の空白分がすべての行の前に追加されます（オートインデント機能）
- 最後に現れる改行コードはとりのぞかれます

例を以下に示します。

テンプレート	結果
`Hello ${'gura'.capitalize()} World`	`Hello Gura World`
`Hello ${3 + 4 * 2} World`	`Hello 11 World`
`AB ${['1st', '2nd', '3rd']} CD`	`AB 1st2nd3rd CD`

`AB ${['1st\n', '2nd\n', '3rd\n']} CD`	`AB 1st`
	`2nd`
	`3rd CD`

`  ${['1st\n', '2nd\n', '3rd\n']}`	`  1st`
	`  2nd`
	`  3rd`

`Hello ${`
`'gura'.capitalize()`
`} World`
	`Hello Gura World`

評価結果が `nil` の場合は何も出力されません。このふるまいは、評価結果が空の文字列のときと同じですが、`"${"` と `"}"` の直後にある改行コードをとりのぞく点が異なります。例を以下に示します。

テンプレート	結果
`Hello ${''} World`	`Hello World`

Line1 ${''} Line2	Line1 Line2
Hello ${nil} World Line1 ${nil} Line2	Hello World Line1 Line2

"${" と "}" の中で最後に記述されている関数が常にブロックをとる関数で、スクリプトにはブロックが記述されていない場合、そのスクリプトの後から "${end}" が現れるまでの文字列がそのブロックの内容として扱われます。これにより、if-elsif-else などの制御構文やループなどを記述することができます。

テンプレート	結果

${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}
	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

最後の関数呼び出しに空のブロックが指定された場合も、継続する文字列がブロックに追加されます。

テンプレート	結果
${(1..3).each {|i|}} ${i} ${end}	1 2 3