berry

Release 4.9.0

Guan Wenliang & Stephan Hadinger

Jul 29, 2023

CONTENTS

About

- genindex
- search

Berry is a ultra-lightweight dynamically typed embedded scripting language. It is designed for lower-performance embedded devices. The Berry interpreter-core's code size is less than 40KiB and can run on less than 4KiB heap (on ARM Cortex M4 CPU, Thumb ISA and ARMCC compiler).

The interpreter of Berry include a one-pass compiler and register-based VM, all the code is written in ANSI C99. In Berry not every type is a class object. Some simple value types, such as int, real, boolean and string are not class object, but list, map and range are class object. This is a consideration about performance. Register-based VM is the same meaning as above.

Berry has the following advantages:

- Lightweight: A well-optimized interpreter with very little resources. Ideal for use in microprocessors.
- Fast: optimized one-pass bytecode compiler and register-based virtual machine.
- Powerful: supports imperative programming, object-oriented programming, functional programming.
- Flexible: Berry is a dynamic type script, and it's intended for embedding in applications. It can provide good dynamic scalability for the host system.
- Simple: simple and natural syntax, support garbage collection, and easy to use FFI (foreign function interface).
- RAM saving: With compile-time object construction, most of the constant objects are stored in read-only code data segments, so the RAM usage of the interpreter is very low when it starts.

CHAPTER

ONE

FEATURES

Base Type

- Nil: nil
- Boolean: true and false
- Numerical: Integer (int) and Real (real)
- String: Single quotation-mark string and double quotation-mark string
- Class: Instance template, read only
- Instance: Object constructed by class
- Module: Read-write key-value pair table
- List: Ordered container, like [1, 2, 3]
- Map: Hash Map container, like { 'a': 1, 2: 3, 'map': {} }
- Range: include a lower and a upper integer value, like 0..5

Operator and Expression

- Assign operator: =, +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
- Relational operator: <, <=, ==, !=, >, >=
- Logic operator: &&, ||, !
- Arithmetic operator: +, -, *, /, %
- Bitwise operator: &, |, ~, ^, <<, >>
- Field operator: .
- Subscript operator: []
- Connect string operator: +
- Conditional operator: ? :
- Brackets: ()

Control Structure

- Conditional statement: if-else
- Iteration statement: while and for
- Jump statement: break and continue

Function

- Local variable and block scope
- Return statement
- Nested functions definition
- Closure based on Upvalue
- Anonymous function
- Lambda expression

Class

- Inheritance (only public single inheritance)
- Method and Operator Overload
- Constructor method
- Destructive method

Module Management

- Built-in module that takes almost no RAM
- Extension module support: script module, bytecode file module and shared library (like *.so, *.dll) module

GC (Garbage collection)

- Mark-Sweep GC

Exceptional Handling

- Throw any exception value using the raise statement
- Multiple catch mode

Bytecode file support

- Export function to bytecode file
- Load the bytecode file and execute

1.1 Welcome to the berry wiki!

1.1.1 The Berry Script Language Reference Manual

Preface

A few years ago, I tried to port the Lua scripting language to STM32F4 microcontroller, and then I experienced a Lua-based firmware on ESP8266: NodeMCU. These experiences made me experience the convenience of script development. Later, I came into contact with some scripting languages, such as Python, JavaScript, Basic, and MATLAB. At present, only a few languages are suitable for transplanting to the microcontroller platform. I used to pay more attention to Lua because it is a very compact embedded scripting language, and its design goal is to be embedded in the host program. However, for the microcontroller, the Lua interpreter may not be small enough, and I cannot run it on a 32-bit microcontroller with a relatively small memory. To this end, I started to read Lua code and developed my own scripting language-Berry on this basis.

This is an ultra-lightweight embedded scripting language, it is also a multi-paradigm dynamic language. Supports object-oriented, process-oriented and functional (less support) styles. Many aspects of this language refer to Lua, but its syntax also borrows from the design of other languages. If the reader already knows a high-level language, Berry's grammar should be very easy to grasp: the language has only some simple rules and a very natural scope design.

The main application scenario I consider is embedded systems with low performance. The memory of these systems may be very small, so it is very difficult to run a full-featured scripting language. This means that we may have to make a choice. Berry may only provide the most commonly used and basic core functions, while other unnecessary features are only used as optional modules. This will inevitably lead to the language's standard library being too small, even the language There will also be uncertain designs (such as the implementation of floating-point numbers and integers, etc.). The benefits of these trade-offs are more room for optimization, while the disadvantage is obviously the inconsistency of language standards. Berry's interpreter refers to the implementation of Lua interpreter, which is mainly divided into two parts: compiler and virtual machine. Berry's compiler is a one-pass compiler to generate bytecode. This solution does not generate an abstract syntax tree, so it saves memory. The virtual machine is a register type. Generally speaking, the register type virtual machine is more efficient than the stack type virtual machine. In addition to implementing language features, we also hope to optimize the memory usage and operating efficiency of the interpreter. At present, the performance indicators of Berry interpreter are not comparable to those of mainstream languages, but the memory footprint is very small.

It wasn't until later that I learned about the MicroPython project: a simplified Python interpreter designed for microcontrollers. Nowadays, Python is very hot, and this Python interpreter designed for microcontrollers is also very popular. Compared with the current mature technology, Berry is a new language without a sufficient user base. Its advantage is that it is easy to master the grammar and may have advantages in terms of resource consumption and performance.

If you need to port the Berry interpreter, you need to ensure that the compiler you use provides support for the C99 standard (I previously fully complied with C89, and some optimization work later made me give up this decision). At present, most compilers provide support for C99, and the common compilers such as ARMCC (KEIL MDK), ICC (IAR) and GCC in the development of ARM processors also support C99. This document introduces Berry's grammar rules, standard library and other facilities, and finally guides readers to transplant and extend Berry. This document does not explain the implementation mechanism of the interpreter, and may be explained in other documents if we have time.

The author's level is limited, and the writing is in a hurry. If there are any omissions or errors in the article, I hope the readers will not hesitate.

Guan Wenliang

April 2019

Basic Information

1.1 Introduction

Berry is an ultra-lightweight dynamic type embedded scripting language. The language mainly supports procedural programming, as well as object-oriented programming and functional programming. An important design goal of Berry is to be able to run on embedded devices with very small memory, so the language is very streamlined. Nevertheless, Berry is still a feature-rich scripting language.

1.2 Start using

Get Interpreter

Readers can go to the project's GitHub page https://github.com/berry-lang/berry to get the source code of the Berry interpreter. Readers need to compile the Berry interpreter by themselves. The specific compilation method can be found in the README.md document in the root directory of the source code, which can also be viewed on the GitHub page of the project.

First, you must install software such as GCC, git, and make. If you do not use version control, you can download the source code directly on GitHub without installing git. Readers can use search engines to retrieve information about

these software. Readers using Linux and macOS systems should also install the GNU Readline library[1]. Use `git` command [2] Clone the interpreter source code from the remote warehouse to the local:

```
git clone https://github.com/berry-lang/berry
```

Enter the berry directory and use the `make` command to compile the interpreter:

```
cd berry
make
```

Now you should be able to find the executable file of the interpreter in the *berry* directory (in Windows systems, the file name of the interpreter program is "*berry.exe*", while in Linux and macOS systems the file name is "*berry*"), you can run the executable file directly [3] To start the interpreter. In Linux or macOS, you can use the command `sudo make install` to install the interpreter, and then you can start the interpreter with the `berry` command in the terminal.

REPL environment

REPL (Read Eval Print Loop) is generally translated as an interactive interpreter, and this article has also become the interactive mode of the interpreter. This mode consists of four elements: **Read**, read the source code input by the user from the input device; **evaluate**, compile and execute the source code input by the user; **print**, output the result of the evaluation process; **cycle**, cycle the above operations.

Start the interpreter directly (enter `berry` in the terminal or command window without parameters, or double-click berry.exe in Windows) to enter the REPL mode, and you will see the following interface:

```
Berry 1.0.0 (build in Feb 1 2022, 13:14:04)
[GCC 8.1.0] on Windows (default)
>
```

The first two lines of the interface display the version, compilation time, compiler and operating system of the Berry interpreter. The symbol ">" in the third line is called the prompt, and the cursor is displayed behind the prompt. When using the REPL mode, after inputting the source code, pressing the "Enter" key will immediately execute the code and output the result. Press the `Ctrl+C` key combination to exit the REPL. In the case of using the Readline library, use the `Ctrl+D` key combination to exit the REPL when the input is empty. Since there is no need to edit script files, REPL mode can be used for quick development or idea verification.

Hello World Program

Take the classic "Hello World" program as an example. Enter `print('Hello World')` in the REPL and execute it. The results are as follows:

```
Berry 1.0.0 (build in Feb 1 2022, 13:14:04)
[GCC 8.1.0] on Windows (default)
> print('Hello World')
Hello World
>
```

The interpreter output the text "`Hello World`". This line of code realizes the output of the string `'Hello World'` by calling the `print` function. In REPL, if the return value of the expression is not `nil`, the return value will be displayed. For example, entering the expression `1 + 2` will display the calculation result `3`:

```
> 1 + 2
3
```

Therefore, the simplest "Hello World" program under REPL is to directly enter the string `'Hello World'` and run:

```
> 'Hello World'
Hello World
```

More usage of REPL

You can also use the interactive mode of the Berry interpreter as a scientific calculator. However, some mathematical functions cannot be used directly. Instead, use the `import math` statement to import the mathematical library, and then use the functions in the mathematical library. "`math.`" as a prefix, for example `sin` function should be written as `math.sin`:

```
> import math
> math.pi
3.14159
> math.sin(math.pi / 2)
1
> math.sqrt(2)
1.41421
```

Script file

Berry's script file is a file that stores Berry code, and the script file can be executed by an interpreter. Usually, the script file is a text file with the extension ".be". The command to execute the script using the interpreter is:

```
berry script_file
```

`script_file` is the file name of the script file. Using this command will run the interpreter to execute the Berry code in the `script_file` script file, and the interpreter will exit after execution.

If you want the interpreter to enter the REPL mode after executing the script file, you can add the `-i` parameter to the command to call the interpreter:

```
berry -i script_file
```

This command will first execute the code in the `script_file` file and then enter the REPL mode.

1.3 Words

Before introducing Berry's syntax, let's take a look at a simple code (you can run this code in REPL mode):

```
def func(x) # a function example
    return x + 1.5
end
print('func(10) =', func(10))
```

This code defines a function `func` and calls it later. Before understanding how this code works, we first introduce the syntax elements of the Berry language.

In the above code, the specific classification of grammatical elements is: `def`, `return` and `end` are keywords of Berry language; and "`# a function example`" in the first line is called a comment; `print` , `func` and `x` are some identifiers, they are usually used to represent a variable; `1.5` and `10` these numbers are called numerical literals, they are equivalent

to the numbers used in daily life; `'func(10) ='` It is a string literal, they are used in places where you need to represent text; + is an addition operator, here the addition operator can be used to add the variable `x` and the value `1.5`.

The above classification is actually done from the perspective of a lexical analyzer. Lexical analysis is the first step in Berry source code analysis. In order to write the correct source code, we start with the most basic lexical introduction.

Comment

Comments are some text that does not generate any code. They are used to make comments in the source code and be read by people, while the compiler will not interpret their content. Berry supports single-line comments and cross-line block comments. Single-line comments start with the character "#' until the end of the newline character. The quick note starts with the text "#-" and ends with the text "-#". The following is an example of using annotations:

```
# This is a line comment
#- This is a
   block comment
-#
```

Similar to C language, quick comments do not support nesting. The following code will terminate the analysis of comments at the first "-#" text:

```
#- Some comments -# ... -#
```

literal value

The literal value is a fixed value written directly in the source code during programming. Berry's literals are integers, real numbers, booleans, strings, and nil. For example, the value `34` is an integer denomination.

Numerical Literal Value

Numerical literals include **Integer** (integer) literals and **Real number** (real) literals.

```
40 # Integer literal
0x80 # Hexadecimal literal (integer)
3.14 # Real literal
1.1e-6 # Real literal
```

Numeric literals are written similarly to everyday writing. Berry supports hexadecimal integer denominations. Hexadecimal literals start with the prefix `0x` or `0X`, followed by a hexadecimal number.

Boolean literal value

Boolean values (boolean) are used to represent true and false in the logic state. You can use the keywords `true` and `false` to represent Boolean literals.

String literal

A string is a piece of text, and its literal writing is to use a pair of ' or " to surround the string text:

```
'this is a string'
"this is a string"
```

String literals provide some escape sequences to represent characters that cannot be input directly. The escape sequence starts with the character '\', and then follows a specific sequence of characters to achieve escape. The escape sequences specified by Berry are

Escape character	significance	Escape character	significance
\a	Ring the bell	\b	Backspace
\f	Form feed.	\n	Newline
\r	Carriage return	\t	Horizontal tab
\v	Vertical tab	\\	Backslash
\'	apostrophe	\"	Double quotes
\?	question mark	\0	Null character

Escape character sequence

Escape sequences can be used in strings, for example

```
print('escape character LF\n\tnew line')
```

The result of the operation is

```
escape character LF
        new line
```

You can also use generalized escape sequences, in the form of \x followed by 2 hexadecimal digits, or \ 3 octal digits, using this escape sequence can represent any character. Here are some examples of using the ASCII character set:

```
'\115' #-'M' -#'\x34' #- '4' -#'\064' #- '4' -#
```

Nil literal value

Nil represents a null value, and its literal value is represented by the keyword nil.

identifier

Identifier (identifier) is a user-defined name, which starts with an underscore or letter, and then consists of a combination of several underscores, letters or numbers. Similar to most languages, Berry is case-sensitive, so identifiers A and identifiers a will be resolved into two identifiers.

```
a
TestVariable
Test_Var
_init
baseCass
_
```

Keywords

Berry reserves the following tokens as language keywords:

```
if elif else while for def
end class break continue return true
false nil var do import as static
```

The specific usage of keywords will be introduced in the relevant chapters. Note that keywords cannot be used as identifiers. Because Berry is case sensitive, `If` can be used for identifiers.

[1] For GNU Readline, the installation command for the Debian series of Linux distributions is `sudo apt install libreadline-dev`, and the installation command for the RedHat series of Linux distributions is `yum install readline-devel`, under macOS The installation command is `brew install readline`. In addition, it is easy to find GNU Readline documentation and related materials in search engines.

[2] commands need to be used in the "command line interface" after the preparation work is completed. The command line environment in Windows systems is usually a command prompt (CMD) window, while the command line environment in Unix-like systems is usually Called "Terminal" (Terminal). This is not very accurate, but it will not be expanded here.

[3] In Windows, you can directly double-click to run the executable file. In Linux or macOS, use the terminal to run it. You can also run the interpreter in the Windows command prompt window. Please refer to the README.md file for specific usage.

2. Types and Variables

Type is an attribute of data, which defines the meaning of the data and the operations that can be performed on the data. Types can be divided into built-in types and user-defined types. Built-in types refer to some basic types built into the Berry language, among which types that are not based on class definitions are called **Simple type**. Types based on class definitions are called **Class type**, some of the built-in types are class types, and user-defined types are also class types.

2.1 Built-in type

2.1.1 Simple Type

`nil`

The Nil type is the null type, which means that the object has an invalid value, or it can be said that the object has no meaningful value. This is a very special type. Although we might say that a variable is `nil`, in fact the nil type has no value, so what we are talking about here is that the type of the variable is nil (not a value).

The default value of a variable before assignment is `nil`. This type can be used in logic operations. In logic operations, `nil` is equivalent to `false`.

Integer type

The integer type (integer) represents a signed integer, referred to as an integer. The number of bits of the integer represented by this type depends on the specific implementation, and usually consists of a 32-bit signed integer on a 32-bit platform. Integer is an arithmetic type and supports all arithmetic operations. Pay attention to the value range of integers when using this type. The typical value range of 32-bit signed integers is between 2147483648 and 2147483647.

Any value can be converted to `int` using the `int()` function; however `int(nil) == nil`. If the argument is an instance and if it contains a member `toint()` it we be called and the return value converted to `int`.

Real Number Type

The real type (real), to be precise, is a floating-point type. Real number types are usually implemented as single-precision floating-point numbers or double-precision floating-point numbers. The real number type is also an arithmetic type. Compared with the integer type, the real number type has higher precision and a larger value range, so this type is more suitable for mathematical calculations. It should be noted that the real number type is actually a floating point number, so there are still precision problems. For example, it is not recommended to compare two values of type `real` for equality.

When integers and real numbers participate in operations at the same time, the integers are usually converted to real numbers.

Boolean type

The Boolean type (boolean) is used for logical operations. It has two values `true` and `false`, which represent the two true values (true and false) in logic and Boolean algebra. The Boolean type is mainly used for conditional judgment. The operands and return values of logical expressions and relational expressions are all boolean types, and statements such as `if` and `while` all use boolean types as conditional checks.

In many cases, non-boolean values can also be used as boolean types. This is because the interpreter will implicitly convert the parameters. This is also the reason that conditional check expressions such as `if` statements can use any type of parameters. The rules for converting various types to Boolean types are:

- `nil`: converted to `false`.
- **Integer**: when the value is `0`, it is converted to `false`, otherwise it is converted to `true`.
- **Real number**: when the value is `0.0`, it is converted to `false`, otherwise it is converted to `true`.
- **String**: when the value is "" (empty string) it is concerted to `false` otherwise it is converted to `true`.
- **Comobj** and **Comptr**: when the internal pointer is `NULL` it is converted to `false`, otherwise it is converted to `true`.
- **Instance**: if the instance contains a method `tobool()`, the return value of the method will be used, otherwise it will be converted to `true`.
- All other types: convert to `true`.

Any value can be converted to `bool` using `bool()` function.

String

A string is a sequence of characters. In terms of storage, Berry divides strings into long strings and short strings. There is only one instance of the same short character string in memory, and all short character strings are linked in a hash table. This design helps to improve the performance of string equality comparison and can reduce memory usage. Since the use frequency of long strings is low, and the overhead of hash operation is quite high, they are not linked to the hash table, so there may be multiple identical instances in the memory. The string is read-only after it is created. Therefore, "modifying" the string will generate a new string, and the original string will not be modified.

Berry does not care about the format or encoding of characters. For example, the string `'abc'` is actually the ASCII code of the characters `'a'`, `'b'` and `'c'`. Therefore, if there are wide characters in the string (the character length is greater than 1 byte), the number of characters in the string cannot be directly counted. In fact, using the `size()` function can only get the number of bytes in the string. In addition, in order to facilitate interaction with the C language, Berry's string always ends with `'\0'` characters. This feature is transparent to the Berry program.

The string type can be compared in size, so it can be used in relational operations.

Function

A function is a piece of code that is encapsulated and available for call, generally used to implement a specific function. Function is actually a big category, which includes several subtypes such as closures, native functions, and native closures. For Berry code, all function subtypes have the same behavior. Functions belong to the first type of value in Berry, so they can be passed as values. In addition, it can be directly used in expressions through the "literal" form of "anonymous functions".

A function is a read-only object and cannot be modified once defined. You can compare whether two functions are equal (whether they are the same function), but the function type cannot be compared.**Native function** (native function) and **Native closure** (native closure) refer to functions and closures implemented in C language. One of the main purposes of native functions and native closures is to provide functions that the Berry language does not provide, such as IO operations and low-level operations. If a piece of code is used frequently and has performance requirements, it is also recommended to rewrite it as a native function or native closure.

Class

In object-oriented programming, a class is an extensible program code template. Classes are used to create instance objects, so the class can be said to be the "type" of the instance. All instance objects are of type `instance` and they all have a corresponding class, which is called instance **Class type**. To put it simply, a class is a value representing the type of an instance object, and a class is an abstraction of the characteristics of an instance. A class is also a read-only object, once defined, it cannot be modified.

Classes can only compare equals and unequals, but cannot compare sizes.

Examples

An instance is a materialized object generated by a class, and the process of generating an instance from a class is called `Instantiate`. In object-oriented programming, "instance" is usually synonymous with "object". However, in order to distinguish from non-instance objects, we do not use the term "object" alone, but use "instance" or "instance object". Berry instances are always allocated dynamically and need to be used with a garbage collector. In addition to memory allocation, the process of instantiation also needs to initialize the instance, this process is completed by `Constructor`. In addition, you can complete the destruction of the object through `Destructor` before reclaiming the object's memory.

In the internal implementation, the instance will contain a reference to the class, and the instance itself only stores member variables and not methods.

2.1.2 Class Type

Some of the built-in types are class types, they are `list`, `map` and `range`. Unlike custom types, built-in class types can be constructed using literals, for example `[1, 2, 3]` is a literal of type `list`.

List

The List class is a container that provides support for list data types. Berry's list is an ordered collection of elements, and each element in the list has a unique integer index, and each element can be accessed directly according to the index. List supports inserting or deleting elements at any position, and the element can be of any type. In addition to using indexes, you can also use iterators to access elements in the list.

The implementation of List is a dynamic array, and this data structure has good random access performance. The efficiency of adding and deleting elements at the end of the list is very high, but the efficiency of adding and deleting elements in the middle of the list is low.

The literal initialization method of the List container is to use a list of objects surrounded by square brackets, and multiple objects are separated by commas, for example:

```
[]
['string']
[0, 1, 2,'list']
```

Operations: see chapter 7.

Map

Map is also a kind of container, map is a collection of key-value pairs, and each possible key appears at most once in the collection. The Map container provides the following basic operations:

- Add key-value pairs to the collection
- Remove key-value pairs from the collection
- Modify the value corresponding to an existing key
- Find the corresponding value by key

Map is implemented using a hash table and has high search efficiency. The operation of adding and deleting key-value pairs will consume more time if "re-hashing" occurs.

The Map container can also be initialized using literal values, written in curly braces to enclose a list of key-value pairs, separated by colons between keys and values, and separated by commas between key-value pairs. E.g:

```
{}
{'str':'hello'}
{'str':'hello','int': 45, 78: nil}
```

Operations: see chapter 7.

Range

The Range container represents an integer range, which is usually used to iterate in an integer range. This type has a `__lower__` member and `__upper__` member, which represent the lower and upper bounds of the range, respectively. The literal value of Range is a pair of integers connected using the `..` operator:

```
0 .. 10
-5 .. 5
```

When the Range class is used for iteration, the elements of the iteration are all integer values from the lower bound to the upper bound, including boundary values. For example, the iteration result of `0..5` is:

```
0 1 2 3 4 5
```

Therefore, it should be noted that for a range of *x* .. (*x*+*n*), the number of iterations is *n*+1. A common construct to iterate through elements of a list by item is:

```
for i: 0..size(l)-1
```

Open range: if you omit the last range, it is implicitly replaced with MAXINT.

```
> r = 10..
> r
(10..9223372036854775807)
```

Bytes

Bytes object denote a bytes buffer which can be used to manipulate bytes buffers or to read/write some C memory areas or structures.

See Chapter 7.

2.2 Variables

A variable is a storage space with a name, and the data or information stored in the storage space is called the value of the variable. Variable names are used to refer to variables in source code. In different scopes, a variable name can bind multiple independent variables, but variables have no aliases. The value of the variable can be accessed or changed at any time during the running of the program. Berry is a dynamically typed language, so the type of variable value is determined at runtime, and the variable can store any type of value.

2.2.1 define variables

The first way to define a variable is to use an assignment statement to assign a value to a new variable name:

```
'var' = expression
```

variable is the name of the variable, and the variable name is an identifier (see section identifier). **expression** is the expression to initialize the variable.

```
a = 1
b ='str'
```

However, this method of defining variables has some limitations. Take the following code as an example:

```
i = 0
do
    i = 1
    print(i) # 1
end
print(i) # 1
```

The do statement in the routine constitutes the inner scope. We modified the value of the variable i at line 3, and the value of i is still 1 after leaving the inner scope at line 6 . If we want the variable i of the inner scope to be an independent variable, the method of defining the variable by directly assigning to the new variable name cannot meet the requirement, because the identifier i already exists in the outer scope. In this case, the variable can be defined by the var keyword:

```
'var' variable
'var' variable = expression
```

There are two ways of using var to define a variable: The first is to follow the variable name **variable** after the keyword var, in this case the variable will be initialized to nil, and the other is written in The variable is initialized at the same time as the variable is defined. In this case, an initial value expression **expression** is required. Using var to define a variable has two possible results: if the current scope does not define the variable of **variable**, define and initialize the variable, otherwise it is equivalent to reinitialize the variable. Therefore, the variable defined with var will shield the variable with the same name in the outer scope.

Now we change the previous example to use the var keyword to define variables:

```
i = 0
do
    var i = 1
    print(i) # 1
end
print(i) # 0
```

From the modified routine, it can be found that the value of the variable i in the inner scope is 1, and its value in the outer scope is 0. This proves that after using the var keyword, a new variable i is defined in the inner scope and the variable with the same name in the outer scope is blocked. After the inner scope ends, the identifier i is once again bound to the variable i in the outer scope.

When using the var keyword to define a variable, you can also use a list of multiple variable names, separated by commas. You can also initialize one or more variables when defining variables:

```
var a = 0, b, c ='test'
```

2.2.2 Scope and Life Cycle

As mentioned earlier, variable names can be bound to multiple variable entities (storage spaces), and variable names are bound to only one entity at each position. The entity bound by the variable name needs to be determined according to the position where the variable name appears.

Scope refers to the code area where the name and the entity are uniquely bound. Outside the scope, the name may be bound to other entities, or not bound to any entity. The entity is only visible in the scope bound to the name, that is, the variable is only valid in its scope.A code block (see block) is a scope. A variable is only available inside the block, and names in different blocks may bind different variable entities. The following example demonstrates the scope of variables:

```
var i = 0
do
    var j ='str'
    print(i, j) # 0 str
end
# The variable j is not available here
print(i) # 0
```

The names `i` and `j` are defined in this routine. The name `i` is defined outside the `do` sentence, and the name defined in the outermost block has **Global scope** (global scope). The name with global scope is available in the entire program after customization. The name `j` is defined in the block in the `do` sentence, and the name of this type of definition in the non-outermost block has **Local scope** (local scope). A name with a local scope cannot be accessed outside the scope.

Berry has some built-in objects, which are all in the global scope. However, built-in objects and global variables defined in scripts are not in the same global scope. Built-in objects actually belong to **Built-in scope** (built-in scope). The scope is globally visible as the ordinary global scope, but can be covered by the ordinary global scope. Built-in objects include functions and classes in the standard library. These objects include `print` functions, `type` functions, and `map` classes. Different from other scopes, the variables in the built-in scope are read-only, so "assignment" to the variables in the built-in scope actually defines a variable with the same name in the global scope, which overrides The symbols in the built-in scope.

nested scope

Nested scope means that the scope contains another scope. We call the contained scope **Inner scope**, and the scope that contains the inner scope **Outer scope**. The name defined in the outer scope can be accessed in all inner scopes. The inner scope can also rebind the name already defined in the outer scope. The previous example using `var` to define variables describes this scenario.

Variable Life Cycle

There is no concept of variable names when the program is running, and variables exist in the form of entities at this time. The "validity period" of a variable during program execution is the variable's **Life cycle**. Variables at runtime are only valid within the scope. After leaving the scope, the variables will be destroyed to reclaim resources.

Variables defined in the global scope are called **Global variable** and have **Static life cycle**. Such variables can be accessed during the entire program running and will not be destroyed. Variables defined in the local scope are called **Local variable** and have **Dynamic life cycle**. Such variables cannot be accessed after leaving the scope and will be destroyed.

Due to the different life cycles, local variables and global variables use different ways to allocate storage space. Local variables are allocated on a structure called **Stack** (stack), and objects allocated based on the stack can be quickly reclaimed at the end of the scope. Global variables are allocated in **Global table** (global table). Objects in the global table will not be recycled once they are created, and the table can be accessed anywhere in the program.

2.2.3 Type of variable

Berry determines the type of the variable at runtime. In other words, the variable can store any type of value. Therefore Berry is a **Dynamic typing** language. The interpreter does not deduce the type of the variable at compile time, which may cause some errors to be exposed at runtime. For example, the error generated by executing the expression `'1' + 1` is a runtime error rather than a compiler error. The advantage of using dynamic types is that many designs can be simplified, and the program will be more flexible, not to mention the need to design a complex type inference system.

Due to the lack of type checking by the interpreter, user code may need to determine the type of value by itself, and this feature can also be used to implement some special operations. This feature also makes overloaded functions unnecessary. For example, the native function `type` accepts any type of parameter and returns a string describing the parameter type.

3. Expression

3.1 Basics

An expression (Statement) is composed of one to more operands and operators, and a result can be obtained by evaluating the expression. This result is called the value of the expression. The operand can be a literal value, variable, function call or sub-expression, etc. Simple expressions and operators can also be combined into more complex expressions. Similar to the four arithmetic operations, the precedence of operators affects the evaluation order of expressions. The higher the precedence of the operator, the earlier the expression is evaluated.

Operators and expressions

Berry provides some unary operators (Unary Operator) and binary operators (Binary Operator). For example, the logical AND operator `&&` is a binary operator, and the logical negation operator `!` is a unary operator. Some operators can be either unary operators or binary operators. The specific meaning of such operators depends on the context. For example, operator `-` is a unary symbol in expression `-1`, but it is a binary minus sign in expression `1-2`.

Operator combination expression

Both the left and right sides of a binary operator can be subexpressions, so you can use binary operators to combine expressions. A more complex expression often has multiple operators and operands. At this time, the order of evaluation of each sub-expression in the expression may affect the value of the expression. The precedence and associativity of operators guarantee the uniqueness of the expression evaluation order. For example, a combined expression:

```
1 + 10/2 * 3
```

The four arithmetic operations in daily use will first calculate the division expression `10/2`, then the multiplication expression, and finally the addition expression. This is because multiplication and division have higher priority than addition.

operand type

In the operation of expressions, the operands may have types that do not match the operators. In addition, binary operators usually require the left and right operands to be of the same type. The expression `'10'+10` is wrong. You cannot add a string to an integer. The problem with the expression `-'b'` is that you cannot take a negative value on a string. Sometimes a binary operator has different operand types but can perform operations. For example, when adding an integer to a real number, the integer object will be converted to a real number and added to another real number object. The logical AND and logical OR operators allow the operands on both sides of the operator to be of any type. In logical expressions, they will always be converted to the `boolean` type according to certain rules.

Another situation is that operators can be overloaded when using custom classes. In essence, you can interpret this operator arbitrarily, and it is up to you to decide what type of its operand should be.

3.1.1 Priority and associativity

In a compound expression composed of multiple operators, the precedence and associativity of the operators determine the order of evaluation of the expressions. The precedence and associativity of each operator are given in Table 1.1.

The precedence specifies the order of evaluation between different operators, and expressions with higher precedence operators will be evaluated first. For example, the process of evaluating the expression `1+2*3` will first calculate the result of `2*3`, and then the result of the addition expression. Using parentheses can improve the evaluation order of low-priority expressions. For example, in the evaluation of expression `(1+2)*3`, the result of expression `1+2` in the parentheses is calculated first, and then the multiplication expression outside the parentheses is calculated.

Associativity refers to the evaluation order of the operands on both sides of the operator, where the operands may be subexpressions. For example, in the addition expression `expr1 + expr2`, the value of `expr1` is calculated first and then the value of `expr2`, because the addition operator is left-associative.

priority	Operator	Description	Associativity
1	`()`	Grouping symbol	•
2	`() [] .`	Field operation	left
3	`- ! ~`	Negative sign, logical negation, bit flip	left
4	`* / %`	M ultiplication, division, and remainder	left
5	`+ -`	Addition, subtraction	left
6	`<< >>`	Move left, move right	left
7	`&`	Bitwise AND	left
8	`^`	Bitwise XOR	left
9	`|`	Bitwise OR	left
10	`..`	Concatenation operator	left
11	`< <= > >=`	Greater than, greater than or equal to	left
12	`== !=`	Equal to, not equal to	left
13	`&&`	Logical AND	left
14	`||`	Logical OR	left
15	`? :`	Conditional operator	right
16	`&= | = ^= <<= >>=`	Assignment	right

Operator list

Use brackets to increase priority

Parentheses can be used when we need operators with lower precedence to be evaluated first. During expression evaluation, the value of the expression in parentheses is calculated first. In other words, for the entire expression, the expression in parentheses is equivalent to an operand, regardless of the composition of the expression in parentheses.

3.2 Operator

3.2.1 Arithmetic Operators

Arithmetic operators are used to implement arithmetic operations. These operators are similar to the mathematical symbols we usually use. The arithmetic operators provided by Berry are shown in Table 1.2.

Operator	Description	Example
-	Unary minus	`- expr`
+	Plus/string concatenation	`expr + expr`
-	Minus sign	`expr - expr`
*	Multiplication sign	`expr * expr`
/	Division sign	`expr / expr`
%	Take the remainder	`expr % expr`

Arithmetic Operator

Binary operator + In addition to being a plus sign, it is also a string concatenation. When the operand of this operator is a string, string concatenation will be performed to concatenate two strings into a longer string. To be precise, + as a string concatenation is no longer in the category of arithmetic operators.

The binary operator `%` is the remainder symbol. Its operands must be integers. The result of the remainder operation is the remainder after dividing the left operand by the right operand. For example, the result of `11%4` is `3`. The real number type cannot do divisible, so the remainder is not supported.

In general, arithmetic operators do not satisfy the commutative law. For example, the values of the expressions `2/4` and `4/2` are not the same.

All arithmetic operators can be overloaded in the class. The overloaded operators are not necessarily limited to their original functional design, but are determined by the programmer.

3.2.2 Relational operators

Relational operators are used to compare the magnitude of the operands. The six relational operators supported by Berry are given in Table 1.3.

Operator	Description	Example
<	Less than	`expr < expr`
<=	Less than or equal to	`expr <= expr`
==	equal	`expr == expr`
!=	not equal to	`expr != expr`
>=	greater or equal to	`expr >= expr`
>	more than the	`-expr`

Relational operator

By comparing the magnitude relationship of the operands or judging whether the operands are equal, evaluating the relational expression will produce a Boolean result. When the relationship is satisfied, the value of the relationship expression is `true`, otherwise it is `false`. Relational operators == and `!=` can use any type of operand, and allow the left and right operands to have different types. Other relational operators allow the use of the following combinations of operands:

integer relop **integer**

real relop **real**

integer relop **real**

real relop **integer**

string relop **string**

In relational operations, the equal sign == and inequality sign `!=` satisfy the commutative law. If the left and right operands are of the same type or are both numeric types (integer and real number), the operands are judged to be equal according to the value of the operands, otherwise the operands are considered unequal. Equality and inequality are reciprocal operations: if `a==b` is true, then `a!=b` is false, and vice versa. Other relational operators do not satisfy the commutative law, but have the following properties: < and >= are reciprocal operations, and > and <= are reciprocal operations. Relational operations require that the operands must be of the same type, otherwise it is an incorrect expression.

Instances can overload operators as methods. If the relational operator is overloaded, the program needs to ensure the above properties.

Among the relational operators, == and `!=` operators have more relaxed requirements than <, <=, > and >=, which only allow comparisons between the same types. In actual program development, the judgment of equality or inequality is usually simpler than the judgment of size. Some operation objects may not be able to judge the size but can only judge the equality or inequality. This is the case with the Boolean type.

logical operators

Logical operators are divided into three types: logical AND, logical OR and logical NOT. As shown in Table 1.4.

Operator	Description	Example
&&	Logical AND	`expr && expr`
||	Logical OR	`expr || expr`
!	Logical negation	`!expr`

Logical Operators

For the logical AND operator, when the values of both operands are `true`, the value of the logical expression is `true`, otherwise it is `false`.

For the logical OR operator, when the values of both operands are `false`, the value of the logical expression is `false`, otherwise it is `true`.

The role of the logical negation operator is to flip the logical state of the operand. When the operand value is `true`, the logical expression value is `false`, otherwise the value is `true`.

Logical operators require that the operand is of Boolean type, and if the operand is not of Boolean type, it will be converted. See section [section::type_bool] for conversion rules.

Logic operations use an evaluation strategy called **Short-circuit evaluation** (short-circuit evaluation). This evaluation strategy is: for the logical AND operator, the second operand will be evaluated if and only if the left operand is true; for the logical OR operator, if and only if the left operand is false Will evaluate the right operand. The nature of short-circuit evaluation causes the code in the logical expression to not all run.

Bitwise Operator

Bit operators can implement some binary bit operations, and bit operations can only be used on integer types. The detailed information of bit operators is shown in Table 1.5. Bit operation refers to the operation of binary bits directly on integers. Logical operations can be extended to bit operations. Taking logical AND as an example, we can perform this operation on each binary bit to achieve bitwise AND, such as 110_b AND $011_b = 010_b$. Bit operations also support shift operations, which move numbers on a binary basis.

Operator		Example
~	Bit flip	`~expr`
&	Bitwise and	`expr & expr`
|	Bitwise or	`expr | expr`
^	Bitwise exclusive or	`expr ^ expr`
<<	Shift left	`expr << expr`
>>	Shift right	`expr >> expr`

Bitwise operator

Although it can only be used for integers, bit operations are still versatile. Bit operations can implement many optimization techniques. In many algorithms, using bit operations can save a lot of code. For example, to determine whether a number `n` is a power of 2, we can judge whether the result of `n & (n - 1)` is `0`. In some languages with high execution efficiency, shift operations can also be used to optimize multiplication and division (usually there is no obvious effect in scripting languages).

The bitwise AND operator "`&`" is a binary operator, which performs the binary AND operation of two integer operands: only when the binary bits corresponding to the operands are all `1`, the result It was `1`. For example, 1110_b & 0111_b =0110_b.

The bitwise OR operator "`|`" is a binary operator, which performs a binary-bit OR operation on two integer operands: only when the binary bits corresponding to the operands are both `0`, the bit of the result It was `0`. For example, 1000_b | 0001_b=1001_b.

The bitwise exclusive OR operator "`^`" is a binary operator, which performs binary exclusive OR operation on two integer operands: when the binary bits corresponding to the operands are different, the bit value of the result is `1`. For example, 1100_b ^ 0101_b = 1001_b.

The left shift operator "<<" is a binary operator, which moves the left operand to the left by the number of bits specified by the right operand on a binary basis. For example 00001010_b <<3=01010000_b.The right shift operator ">>" is a binary operator, which shifts the left operand to the right by the number of bits specified by the right operand on a binary basis. For example, 10100000_b >>3=00010100_b.

The bitwise flip operator "~" is a unary operator, and the result of the expression is to flip the value of each binary bit of the operand. For example, 10100011_b=01011100_b.

The following are some examples of using bit operations. Usually we don't use binary directly. The results in the examples have been converted into common bases.

```
1 << 1 # 2
168 >> 4 # 10
456 & 127 # 72
456 | 127 # 511
0xA5 ^ 0x5A # 255
~2 # -3
```

Assignment operator

The assignment operator only appears in the assignment expression, and the operand of the operator must be a writable object. The assignment expression has no result, so continuous assignment operations cannot be used.

Simple assignment operator

The simple assignment operator = can be used for variable assignment. If the left operand variable is not defined, the variable will be defined. The assignment operator is used to bind the value of the right operand with the left operand. This process is also called "assignment". Therefore, the left operand cannot be a constant, nor can it be any object that cannot be written. These are some legal assignment expressions:

```
a = 45 b ='string' c = 0
```

And the following assignment expression is wrong:

```
1 = 5 # Trying to assign a constant 1
a = b = 0 # Continuous assignment
```

When assigning `nil`, integer, real and Boolean types to variables, the value of the object will be passed to the left operand, but for other types, the assignment operation just passes the reference of the object to the left operand. Since strings, functions, and class types are read-only, all passing references will not have side effects, but you must be extra careful with instance types.

Compound Assignment Operator

Compound assignment operators are operators that combine binary operators and assignment operators. They are practical extensions to simple assignment operators. Compound assignment operators can simplify the writing of some expressions. Table 1.6 lists all the compound assignment operators

Operator	Description
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Preliminary assignment
%=	Remainder assignment
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise XOR assignment
<<=	Left shift assignment
>>=	Right shift assignment

Bit operator

The compound assignment expression performs the binary operation corresponding to the compound assignment operator on the left operand and the right operand, and then assigns the result to the left operand. Taking += as an example, the expression `a += b` is equivalent to `a = a + b`. The compound assignment operator is also an assignment operator, so it has a lower priority. The binary operator corresponding to the compound assignment operator is always evaluated after the right operand, so an expression like `a *= 1 + 2` should be equivalent to `a = a * (1 + 2)`.

Unlike the simple assignment operator, the left operand of the compound assignment operator must participate in the evaluation, so the compound assignment expression does not have the function of defining variables. The assignment

operator itself cannot be overloaded in the class. Users can only overload the binary operator corresponding to the compound assignment operator. This also ensures that the compound assignment operator will always conform to the basic characteristics of assignment operations.

domain operator and subscript operator

Domain operator `.` is used to access an attribute or member of an object. You can use domain operators for both types of modules and instances:

```
l = list[]
l.push('item 0')
s = l.item(0) #'item 0'
```

The subscript operator `[]` is used to access the elements of an object, for example

```
l[2] = 10 # Read by index
n = l[2] # Write by index
```

Classes that support subscript reading must implement the `item` method, and classes that support subscript writing must implement the `setitem` method. The map and list in the standard container implement these two methods, so they support reading and writing using the subscript operator. The string supports subscript reading, but does not support subscript writing (strings are read-only values):

```
'string'[2] #'r'
'string'[2] ='a' # error: value'string' does not support index assignment
```

Currently, strings support integer subscripts, and the range of subscripts cannot exceed the length of the string.

Conditional Operator

The conditional operator (`? :`) is similar to the **if else** statement, but the former can be used in expressions. The usage form of the conditional operator is:

```
cond ? expr1 : expr2
```

cond is the expression used to judge the condition. The evaluation process of the conditional operator is: first find the value of **cond**, if the condition is true, evaluate **expr1** and return the value, otherwise, the value of **expr2**] Evaluate and return the value. **expr1** and **expr2** can have different types, so the following is correct:

```
result = scope <6?'bad': scope
```

This expression first determines whether `scope` is less than `6`, if it is, it returns `bad`, otherwise it returns the value of `scope`. Regardless of the condition of the conditional expression, only one of **expr1** or **expr2** will be executed, similar to the short-circuit characteristic of logical AND and logical OR operations.

Nested Condition Operators

One conditional operator can be nested in another conditional operator, that is, the conditional expression can be used as **cond** or **expr** of another conditional expression. For example, use conditional expressions to divide scores into three levels: excellent, good, and bad:

```
result = scope >= 9?'excellent': scope >= 6?'good':'bad'
```

The first condition checks whether the score is not lower than 9 points. If it is, execute the branch after `?` and return `'excellent'`; otherwise, execute the branch after `:`, which is also a conditional expression. The condition checks whether the score is not lower than 6, if it is, it returns `'good'`, otherwise it returns `'bad'`.

The conditional operator satisfies the right associativity, so the value of the branch expression must be evaluated first to get the value of the conditional expression. Therefore, in a nested conditional expression, the nested conditional expression is evaluated first, and then the outer conditional expression is evaluated.

Priority of conditional operators

Since the precedence of conditional expressions is very low (second only to assignment operators), it is often necessary to add parentheses outside the conditional expressions. For example, when a conditional expression is used as an operand of an arithmetic expression, parentheses will have different effects on the result:

```
result = 10 * (sign <0? -1: 1) # the result is -10 or 10
result = 10 * sign <0? -1: 1 # the result is -1 or 1
```

The result of the first expression is correct, and the second expression takes `10 * sign < 0` as a condition to judge, which does not meet the expectation of the conditional expression as the right operand of the multiplication.

Concatenation Operator

+ operator

When the left and right operands are both strings, the + operator is used to connect the two strings, and the new string obtained is the value of the expression. Therefore, this operator is often used for string concatenation:

```
result ='abc' + '123' # the result is'abc123'
```

+ Operators can also be used to connect two list instances:

```
result = [1, 2] + [3, 4] # the result is [1, 2, 3, 4]
```

Unlike the `list.push` method, the + operator merges two lists into a larger list object, with the elements of the left operand at the head of the result list, and the elements of the right operand at the end of the result list.

`..` operator

`..` is a special operator. If the left operand is a string, the behavior of the expression is to concatenate the left and right operands into a new string (automatic conversion if the right operand is not a string):

```
result ='abc' .. 123 # the result is'abc123'
```

The `..` operator is often used when concatenating a string and a non-string value.If the left operand is a list instance, the `..` operator will append the right operand to the end of the list, and then use this list as the value of the expression:

```
result = [1, 2] .. 3 # the result is [1, 2, 3]
```

This process will directly modify the left operand, which is very similar to the `push` method of `list` (except for strings which are immutable objects). The join operation of list can be executed in chain:

```
result = [1, 2] .. 3 .. 4 # the result is [1, 2, 3, 4]
```

All values in this process will be appended to the leftmost list object.

If the left and right operands are both integers, use the `..` operator to get an integer range object:

```
result = 1 .. 10 # the result is (1..10)
```

This object is used to represent a closed interval of integers, where the left operand is the lower limit and the right operand is the upper limit. Such integer range objects are often used for iteration.

4. Statement

Berry is an imperative programming language. This paradigm assumes that programs are executed step by step. Normally, Berry statements are executed sequentially, and this program structure is called sequential structure. Although the sequence structure is very basic, branch structures and loop structures are usually used in actual programs. Berry provides several control statements to realize this complex flow structure, such as conditional statements and iteration statements.

Except for line comments, carriage returns or line feeds ("`\r`" and "`\n`") are only used as blank characters, so statements can be written across lines. In addition, you can write multiple statements on the same line.

You can add a semicolon at the end of the statement to indicate the end of the statement, but the interpreter can usually split the statement automatically without using a semicolon. You can use semicolons to tell the interpreter how to parse the code for the code that will be ambiguous. However, it is better not to write ambiguous code.

4.1 Simple sentence

4.1.1 Expression statement

Expression statements are mainly statements composed of assignment expressions or function call expressions. Other expressions can also form sentences, but they have no meaning. For example, expression `1+2` is a sentence written alone, but it has no effect. The following routines give examples of expression statements and function statements:

```
a = 1 # Assignment statement
print(a) # Call statement
```

Line 2 is a simple assignment statement that assigns the literal value `i` to the variable `a`. The statement in line 2 is a function call statement, which prints the value of variable `a` by calling the `print` function.

Cross-line expressions are written in the same way as single-line expressions, and no special line continuation symbols are required. E.g:

```
a = 1 +
    func() # Wrap line
```

You can also write multiple expression statements on one line, and various types of statements can be written on one line. This example puts two expression statements on the same line:

```
b = 1 c = 2 # Multiple statements
```

Sometimes the programmer wants to write two statements, but the interpreter may mistakenly think it is one statement. This problem is caused by the ambiguity in the process of grammatical analysis. Take this code as an example:

```
a = c
(b) = 1 # Be regarded as a function call
```

Suppose the 4th and 5th lines are intended to be two expression sentences: `a = c` and `(b) = 1`, but the interpreter will interpret them as a sentence: `a = c(b) = 1`. The cause of this problem is that the interpreter incorrectly parses `c` and `(b)` into function calls. To avoid ambiguity, we can add a semicolon at the end of the statement to clearly separate the statement:

```
a = c; (b) = 1;
```

A better way is not to use parentheses on the left side of the assignment number. Obviously, there is no reason to use parentheses here. Under normal circumstances, complex expressions should not appear on the left side of the assignment operator, but only simple expressions composed of variable names, domain operation expressions, and subscript operation expressions:

```
a = c b = 1
```

Using simple expressions only on the left side of the assignment sign will not cause ambiguity in sentence segmentation. Therefore, in most cases, there is no need to use semicolons to separate expressions, and we do not recommend this way of writing.

Block

A **Block** is a collection of several sentences. A block is a scope, so the variables defined in the block can only be accessed inside the block and its sub-blocks. There are many places where blocks are used, such as `if` statements, `while` statements, function declarations, etc. These statements will contain a block through a pair of keywords. For example, the block used in the `if` statement:

```
if isOpen
    close()
    print('the device was closed')
end
```

The statements in lines 2 to 3 constitute a block, which is sandwiched between the pair of keywords `if` and `end` (the conditional expression of the statement in `if` is not in the block). The block does not need to contain any statements, which constitutes an empty block, or it can be said to be a block containing an empty statement. Broadly speaking, any number of consecutive sentences can be called a block, but we prefer to expand the scope of the block as much as

possible, which can ensure that the area of the block is consistent with the scope of the scope. In the above example, we tend to think that rows 2 to 3 are a whole block, which is the largest range between `if` keywords and `end` keywords.

`do` Statement

Sometimes we just want to open up a new scope, but don't want to use any control statements. In this case, we can use the `do` statement to encapsulate the block. `do` The statement has no control function. `do` The sentence has the form

`do` **block** `end`

Among them **block** is the block we need. This statement uses a pair of `do` and `end` keywords to contain blocks. `do` The statement has no control function, nor does it generate any runtime instructions.

conditional statement

Berry provides `if` statements to realize the function of conditional control execution. This kind of program structure is generally called branch structure. `if` The statement will determine the branch of execution based on the true (`true`) or false (`false`) conditional expression. In some languages, there are other options for implementing conditional control. For example, languages such as C and C++ provide `switch` statements, but in order to simplify the design, Berry does not support `switch` statements.

`if` Statement

`if` statement is used to implement the branch structure, which selects the branch of the program according to the true or false of a certain judgment condition. The statement can also include `else` branch or `elif` branch. The simple `if` statement form without branches is

`if` **condition** **block** `end`

condition is a conditional expression. When the value of **condition* is ``true``,**block** in the second line will be executed, otherwise the **block** will be skipped and the statement following `end` will be executed. In the case of **block** being executed, after the last statement in the block is executed, it will leave the `if` statement and start executing the statement following `end`.

Here is an example to illustrate the usage of the `if` statement:

```
if 8 % 2 == 0
    print('this number is even')
end
```

This code is used to judge whether the number `8` is even, and if it is, it will output `this number is even`. Although this example is very simple, it is enough to illustrate the basic usage of `if` sentences.

If you want to have a corresponding branch for execution when the condition is met and not met, use the `if` statement with the `else` branch. `if else` The form of the sentence is

`if` **condition** **block**
`else`
block
`end`

Different from the simple `if` statement, the `if else` statement will execute **block** under the `else` branch when the value of **condition** is `false`. No matter which branch is executed under **block**, after the last statement in the block is executed, the `if else` statement will pop out, that is, the statement after `end` will be executed. In other words, no matter whether the value of **condition** is `true` or `false`, one **block** will be executed.

Continue to use the judgment of parity as an example, this time change the demand to output corresponding information according to the parity of the input number. The code to achieve this requirement is:

```
if x % 2 == 0
    print('this number is even')
else
    print('this number is odd')
end
```

Before running this code, we must first assign an integer value to the variable `x`, which is the number we want to check for parity. If `x` is an even number, the program will output `this number is even`, otherwise it will output `this number is odd`.Sometimes we need to nest `if` statements. One way is to nest a `if` statement under the `else` branch. This is a very common requirement because many conditions need to be judged consecutively. For this kind of demand, use the `if else` statement to write:

```
if expr
    block
else
    if expr
        block
    end
end
```

Obviously, this way of writing will increase the indentation level of the code, and it is more cumbersome to use multiple `end` at the end. As an improvement, Berry provides the `elif` branch to optimize the above writing. Using the `elif` branch is equivalent to the above code, in the form

`if` **condition**
block
`elif` **condition**
block
`else`
block
`end`

`elif` The branch must be used after the `if` branch and before the branch, and the `elif` branch can be used multiple times in succession. If the **condition** corresponding to the `elif` branch is satisfied, the **block** under the branch will be executed. `elif` Branching is suitable for situations that require multiple conditions to be judged in sequence.

We use a piece of code that judges positive, negative, and 0 to demonstrate the `elif` branch:

```
if x> 0
    print('positive')
elif x == 0
    print('zero')
else
```

(continues on next page)

(continued from previous page)

```
    print('negative')
end
```

Here too, the variable `x` must be assigned first. This code is very simple and will not be explained.

Some languages have a problem called dangling "`else`", which refers to when a `if` sentence is nested inside another `if` sentence, where does the `else` branch belong? Problem with the sentence `if`. When using C/C++, we must consider the problem of dangling `else`. In order to avoid ambiguity on the problem of `if else`, C/C++ programmers often use curly braces to make a branch into a block. In Berry, the branch of the `if` statement must be a block, which also determines that Berry does not have the problem of overhanging `else`.

Iteration Statement

Iterative statements are also called loop statements, which are used to repeat certain operations until the termination condition is met. Berry provides `while` statement and `for` two iteration statements. Many languages also provide these two statements for iteration. Berry's `while` statement is similar to the `while` statement in C/C++, but Berry's `for` statement is only used to traverse the elements in the container, similar to the `foreach` statement provided by some languages and the one introduced by C++11 New `for` sentence style. The C-style `for` statement is not supported.

`while` Statement

`while` statement is a basic iterative statement. `while` statement uses a judgment condition. When the condition is true, the loop body is executed repeatedly, otherwise the loop is ended. The pattern of the statement is

`while` **condition** **block** `end`

When the program runs to the `while` statement, it will check whether the expression **condition** is true or false. If it is true, execute the loop body **block**, otherwise end the loop. After executing the last statement in **block**, the program will jump to the beginning of the statement `while` and start the next round of detection. If the **condition** expression is false when it is first evaluated, the loop body **block** will not be executed at all (same as the **condition** expression of the `if` statement is false).Generally speaking, the value of **condition** expression should be able to change during the loop, rather than a constant or a variable modified outside the loop, which will cause the loop to not execute or fail to terminate. A loop that never ends is called an endless loop. Usually we usually expect the loop to execute a specified number of times and then terminate. For example, when using the `while` loop to access all elements in the array, we hope that the number of loop executions is the length of the array, for example:

```
i = 0
l = ['a','b','c']
while i < l.size()
    print(l[i])
    i = i + 1
end
```

This loop gets the elements from the array `l` and prints them. We use a variable `i` as the loop counter and array index. We let the value of `i` reach the length of the array `l` to end the loop. In the last line of the loop body, we add `1` to the value of `i` to ensure that the next element of the array is accessed in the next loop, and the `while` loop ends when the number of loops reaches the length of the array.

for Statement

Berry's for statement is used to traverse the elements in the container, and its form is

for variable : expression
block end

expression The value of the expression must be an iterable container or function, such as the range class. for The statement obtains an iterator from the container, and obtains an element in the container every time through the call to the iterator.

variable is called an iteration variable, which is always defined in the statement for. Therefore **variable** must be a variable name and not an expression. The container element obtained from the iterator in each loop will be assigned to the iteration variable. This process occurs before the first statement in **block**.

The for statement will check whether there are any unvisited elements in the iterator for iteration. If there are, the next iteration will start, otherwise it will end the for statement and execute the statement following end. Currently, Berry only provides read-only iterators, which means that the elements in the container cannot be modified through the iteration variables in the for statement.

The scope of the iteration variable **variable** is limited to the loop body **block**, and the variable will not have any relationship with the variable with the same name outside the scope. To illustrate this point, let's use an example to illustrate. In this example, we use the for statement to access all the elements in the rang instance and print them out. Of course, we also use this example to demonstrate the scope of loop variables.

```
i = "Hi, I'm fine." # Outer variable
for i: 0 .. 2
    print(i) # Iteration variable
end
print(i)
```

In this example, relative to the iteration variable i defined in line 2, the variable i defined in line 1 is an external variable. Running this example will get the following result

```
0
1
2
Hi, I'm fine
```

It can be seen that the iteration variable i and the external variable i are two different variables. They just have the same name but different scopes.

for Principle of Statement

Unlike the traditional iterative statement while, the for statement uses iterators to traverse the container. If you need to use the for statement to traverse a custom class, you need to understand its implementation mechanism. When using the for statement, the interpreter hides a lot of implementation details. In fact, for such code:

```
for i: 0 .. 2
    print(i)
end
```

Will be translated into the following equivalent code by the interpreter:

```
var it = __iterator__(0 .. 2)
try
    while true
        var i = it()
        print(i)
    end
except 'stop_iteration'
    # do nothing
end
```

To some extent, the `for` statement is just a syntactic sugar, it is essentially just a simple way of writing a piece of complex code. In this equivalent code, an intermediate variable `it` is used. The value of the variable is an iterator. In this example, it is an iterator of the `range` container `0..2`. When processing the `for` statement, the interpreter hides the intermediate variable of the iterator, so it cannot be accessed in the code.

The parameter of function `__iterator__` is a container, and the function returns an iterator of parameters. This function gets the iterator by calling the parameter method. Therefore, if the return value of the `iter` method is an instance (`instance`) type, this instance must have a `next` method and a `hasnext` method.

The parameter of function `__hasnext__` is an iterator, which checks whether the iterator has the next element by calling the `hasnext` method of the iterator. `hasnext` The return value of the method is of type `boolean`. The parameter of function `__next__` is also an iterator, which gets the next element in the iterator by calling the `next` method of the iterator.

So far, the `__iterator__`, `__hasnext__` and `__next__` functions simply call some methods of the container or iterator and then return the return value of these methods. Therefore, the equivalent writing of the `for` statement can also be simplified into this form:

```
do
    var it = (0 .. 2).iter()
    while (it.hasnext())
        var i = it.next()
        print(i)
    end
end
```

This code is easier to read. It can be seen from the effective code that the scope of the iterator variable `it` is the entire `for` statement, but it is not visible outside the `for` statement, while the scope of the iteration variable `i` is in the loop body, so every time Iterations will define new iteration variables.

Jump Statement

The jump statement provided by Berry is used to realize the jump of the program flow in the loop process. Jump statements are divided into `break` statements and `continue` statements. These two statements must be used inside iterative statements and can only be used inside functions to jump. Some languages provide `goto` statements to realize arbitrary jumps within functions, which Berry does not provide, but the effects of `goto` statements can be replaced by conditional statements and iteration statements.

`break` Statement

`break` Used to terminate the iteration statement and jump out. After the execution of the `break` statement, the nearest level of the iteration statement will be terminated immediately and execution will continue from the position of the first statement after the iteration statement. In order to illustrate the execution flow of the `break` statement, we use an example to demonstrate:

```
while true
    print('before break')
    break
    print('after break')
end
print('out of the loop')
```

In this code, the `break` statement is in a `while` loop. Before and after the `break` statement and after the `while` statement, we have placed a print statement to test the execution flow of the program. The result of this code is:

```
before break
out of the loop
```

This shows that the `while` statement ends the loop at the `break` statement position on the 3rd line and the program continues to execute from the 6th line.

`continue` Statement

`continue` The statement is also used inside an iteration statement. Its function is to end an iteration and immediately start the next round. Therefore, after the execution of the `continue` statement, the remaining code in the iteration statement of the nearest layer will no longer be executed, but a new round of iteration will start. Here we use a `for` statement to demonstrate the function of the `continue` statement:

```
for i: 0 .. 5
    if i >= 2
        continue
    end
    print('i =', i)
end
print('out of the loop')
```

Here, the `for` statement will iterate 6 times. When the iteration variable `i` is greater than or equal to `2`, the `continue` statement on line 3 will be executed, and the print statement on line 5 will not be executed thereafter. In other words, line 5 will only be executed in the first two iterations (at this time `i<2`). The running result of this routine is:

```
i = 0
i = 1
out of the loop
```

It can be seen that the value of the variable `i` is only printed twice, which is in line with expectations. Readers can try to print the value of the variable `i` before the `continue` statement. You will find that the `for` statement does iterate 6 times, indicating that the `continue` statement does not terminate the iteration.

`import` Statement

Berry has some predefined modules, such as the `math` module for mathematical calculations. These modules cannot be used directly, but must be imported with the `import` statement. There are two ways to import a module:

`import` **name**

`import` **name** `as` **variable**

name For the name of the module to be imported, when using the first writing method to import the module, the imported module can be called directly by using the module name. The second way of writing is to import a module named **name** and modify the module name when calling it to **variable**. For example, a module named `math`, we use the first method to import and use:

```
import math
math.sin(0)
```

Here directly use `math` to call the module. If the name of a module is relatively long and it is not convenient to write, you can use the `import as` statement. Here, assume a module named `hardware`. We want to call the function `setled` of the module, we can import the module `hardware` into the variable named `hw` and use:

```
import hardware as hw
hw.setled(true)
```

To find modules, all paths in `sys.path()` are explored sequentially. If you want to add a specific path before the import (like reading from SD card) you can use the following helper function:

```
def push_path(p)
  import sys
  var path = sys.path()
  if path.find(p) == nil  # append only if it's not already there
    path.push(p)
  end
end
```

Exception Handling

The mechanism allows the program to capture and handle exceptions that occur during runtime. Berry supports an exception capture mechanism, which allows the exception capture and handling process to be separated. That is, part of the program is used to detect and collect exceptions, and the other part of the program is used to handle exceptions.

First of all, the problematic program needs to throw an exception first. When these programs are in an exception handling block, a specific program will catch and handle the exception.

Raise an exception

Using the `raise` statement raises an exception. `raise` The statement will pass a value to indicate the type of exception so that it can be identified by a specific exception handler. Here is how to use the `raise` statement:

`raise` **exception**

`raise` **exception , message**

The value of the expression **exception** is the thrown **Outliers**; the optional **message** expression is usually a string describing the exception information, and this expression is called **Abnormal parameter**. Berry allows any value to be used as an abnormal value, for example, a string can be used as an abnormal value:

```
raise 'my_error','an example of raise'
```

After the program executes to the `raise` statement, it will not continue to execute the statements following it, but will jump to the nearest exception handling block. If the most recent exception handling block is in other functions, the functions along the call chain will exit early. If there is no exception handling block, **Abnormal exit** will occur, and the interpreter will print the exception error message and the call stack of the error location.When the `raise` statement is in the `try` statement block, the exception will be caught by the latter. The caught exception will be handled by the `except` block associated with the `try` block. If the thrown exception can be handled by the `except` block, the execution of this block will continue from the statement after the last `except` block. If all `except` statements cannot handle the exception, the exception will be rethrown until it can be handled or the exception exits.

Outliers

In Berry, you can use any value as an outlier, but we usually use short strings. Berry may also throw some exceptions internally. We call these exceptions **Standard exception**. All standard exception values are of string type.

Outliers	Description	Parameter Description
`assert_failed`	Assertion failed	Specific exception information
`index_error`	(usually out of bounds)	Specific exception information
`io_error`	IO Malfunction	Specific exception information
`key_error`	Key error	Specific exception information
`runtime_error`	VM runtime exception	Specific exception information
`stop_iteration`	End of iterator	no
`syntax_error`	Syntax error	
by the compiler		
`unrealized_error`	Unrealized function	Specific exception information
`type_error`	Type error	Specific exception information

Standard exception list

Catch exceptions

Use the `except` statement to catch exceptions. It must be paired with the `try` statement, that is, a `try` statement block must be followed by one or more `except` statement blocks. `try-except` The basic form of the sentence is

`try` **block**
`except ...` **block** `end`

The `except` branch can have the following forms

`except` `..` `except` **exceptions**
`except` **exceptions**`as` **variable**
`except` **exceptions**`as` **variable** `,` **message**
`except` `..` `as` **variable**
`except` `..` `as` **variable** `,` **message**

The most basic `except` statement does not use parameters, this `except` branch will catch all exceptions; **Catch exception list exceptions** is a list of outliers that can be matched by the corresponding `except` branch, used between multiple values in the list Separate by commas; **variable** is **Abnormal variable**, if the branch catches an exception, the outlier will be bound to the variable; **message** is **Abnormal parameter variable**, if the branch catches an exception, the abnormal parameter value will be bound To the variable.

When an exception is caught in the `try` statement block, the interpreter will check the `except` branch one by one. If the exception value exists in the capture list of a branch, the code block under the branch will be called to handle the exception, and the entire `try-except` statement will exit after the code block is executed. If all the `except` branches do not match, the exception will be re-thrown and caught and handled by the outer exception handler.

5. Function

function is a "subroutine" that can be called by external code. As a part of the program, the function itself is also a piece of code. A function can have 0 or more parameters and will return a result, which is called the function's **return value**.

In Berry, the function is **first class value**. Therefore, in addition to calling functions, you can also pass functions as values, for example, bind functions to variables, use functions as return values, and so on.

5.1 Basic Information

The use of functions includes two parts: function definition and call. The function definition statement uses the `def` keyword as the beginning. The function definition is the process of packaging and naming the code of the function body. This process only generates the function structure and does not execute the function. The execution function must use **call operator**, which is a pair of parentheses. The call operator acts on an expression whose result is a function type. The parameters passed to the function are written in parentheses, and multiple parameters are separated by commas. The result of the calling expression is the return value of the function.

5.1.1 Function Definition

Named Function

named function is a function given a name when it is defined. Its definition statement consists of the following parts: `def` keywords, function names, lists consisting of 0 to multiple parameters, and function bodies (function body), multiple parameters in the parameter list are separated by commas, and all parameters are written in a pair of parentheses. We call the parameter when the function is defined as **Formal parameters**, and the parameter when calling the function as **Arguments**. The general form of the function definition is:

```
'def' name ´(´ arguments ´)´
  block
´end'
```

The function name **name** is an identifier; **arguments** is the formal parameter list; **block** is the function body. If the function body is an empty statement, the function is called an "empty function". The function return value statement is contained in the function body. If there is no return statement in **block**, the function will return `nil` by default. The function name is actually the variable name of the bound function object. If the name already exists in the current scope, defining the function is equivalent to binding the function object to this variable.

The following example defines a function named `add`. The function of this function is to sum two numbers and return.

```
def add(a, b)
    return a + b
end
```

`add` The function has two parameters `a` and `b`, and the two addends are passed into the function through these parameters for calculation. `return` The statement returns the result of the calculation.

A function as a class attribute is called a method. This part of the content will be explained in the object-oriented chapter.

Anonymous Function

Unlike named functions, **anonymous function** has no name, and its definition expression has the form:

```
´def' ´(´ arguments ´)´
  block
´end'
```

It can be seen that compared with named functions, there is no function name in the definition of anonymous functions **name**. The definition of an anonymous function is essentially an expression, which is called **Function literal**. In order to use anonymous functions, we can bind the function literal value to a variable:

```
add = def (a, b)
    return a + b
end
```

The function of this code is exactly the same as that of the function `add` in the previous section. An anonymous function can be used to conveniently pass the function value as a literal value. Like other types of literals, function literals are also the smallest unit of expressions. Therefore, between `def` keywords and `end` are an indivisible whole.

Call function

Take the `add` function as an example. To call this function, you need to provide two values, and you can get the sum of the two numbers by calling the function:

```
res = add(5, 3)
print(res) # 8
```

We call the called function (the `add` function in the example) as **Called function**, and the function that calls the called function (the `main` function in the example) as **Key function**. The function call process is as follows: First, the interpreter will (implicitly) initialize the formal parameter list of the called function with the argument list, and at the

same time suspend the calling function and save its state, then create an environment for the called function and execute the called function. function.

The function will end its execution when it encounters the `return` statement and pass the return value to the calling function. The interpreter will destroy the environment of the called function after the called function returns, then restore the environment of the calling function and continue to execute the calling function. The return value of the function is also the result of the function call expression. The following example defines a function `square` and binds this function to a variable `f`, and then calls the function `square` through the variable `f`. This usage is similar to function pointers in C language.

```
def square(n)
    return n * n
end
f = square
print(f(5)) # 25
```

It should be noted that the function object is only bound to these variables (refer to section [section::assign_operator]) and cannot be modified, so reassigning the variable corresponding to the function name will not make the function lose:

```
f = square
square = nil
print(f(5)) # 25
```

It can be seen that the function can still be called normally after reassigning `square`. Only after the function object is no longer bound to any variable will it be lost, and the resources occupied by this type of function object will be recycled by the system.

Forward call

The call of the function must be in the scope of the function variable, so it usually cannot be called before the function is defined. In order to solve this problem, you can use this method to compromise:

```
var func1
def func2(x)
    return func1(x)
end
def func1(x)
    return x * x
end
print(func2(4)) # 16
```

In this example, `func2` calls `func1`, but the function `func1` is defined after `func2`. After executing this code, the program will output the correct result 16. This routine uses the mechanism that the function will not be called when the function is defined. Define the variable `func1` before defining `func2` to ensure that the symbol `func1` will not be found during compilation. Then we define the function `func1` after `func2` so that the function will be used to overwrite the value of the variable `func1`. When the function `func2` is called in the last line `print(func2(4))`, the variable `func1` is already the function we need, so the correct result will be output.

Recursive call

recursive function refers to functions that call themselves directly or indirectly. Recursion refers to a strategy that divides the problem into similar sub-problems and then solves them. Taking factorial as an example, the recursive definition of factorial is 0!=1,n!=n(n1)!, we can write the recursive function for calculating factorial according to the definition:

```
def fact(n)
    if n == 0
        return 1
    end
    return n * fact(n-1)
end
```

Take the factorial of 5 as an example, the process of manually calculating the factorial of 5 is: 5!=5×4×3×2×1=120 The result of calling the `fact` function is also 120:

```
print(fact(5)) # 120
```

In order to ensure that the depth of the recursive call is limited (too deep recursion level will exhaust the stack space), the recursive function must have an end condition. `fact` The `if` statement in the second line of the function definition is used to detect the end condition, and the recursive process ends when `n` is calculated as `0`. The above factorial formula does not apply to non-integer parameters. Executing an expression like `fact(5.1)` will cause a stack overflow error due to the inability to end the recursion.

There is another situation `Indirect recursion`, that is, the function is not called by itself but by another function (directly or indirectly) called by it. Indirect recursion usually requires the use of forward function call techniques. Take the functions `is_odd` and `is_even` for calculating odd and even numbers as examples:

```
var is_odd
def is_even(n)
    if n == 0
        return true
    end
    return is_odd(n-1)
end
def is_odd(n)
    if n == 0
        return false
    end
    return is_even(n-1)
end
```

These two functions call each other. In order to ensure that this name is in the scope when calling the function `is_odd` on line 6, the variable `is_odd` is defined on line 1.

Anonymous function call

If an anonymous function will only be called once, the easiest way is to call it when it is defined, for example:

```
res = def (a, b) return a + b end (1, 2) # 3
```

In this routine, we use the call expression directly after the function literal to call the function. This usage is very suitable for functions that will only be called in one place.

You can also bind an anonymous function to a variable and call it:

```
add = def (a, b) return a + b end
res = add(1, 2) # 3
```

This usage is similar to the call of a named function, essentially calling the variable bound to the function value. It should be noted that it is more difficult to make recursive calls to anonymous functions, unless you use forward call techniques.

Formal and actual parameters

The function uses actual parameters to initialize the formal parameters when it is called. Under normal circumstances, the actual parameter and the shape parameter are equal and the positions correspond to each other, but Berry also allows the actual parameter to be unequal to the formal parameter: if the actual parameter is more than the formal parameter, the extra actual parameter will be discarded. Less than the formal parameters will initialize the remaining formal parameters to `nil`.

The process of parameter passing is similar to assignment operation. For `nil`, `boolean` and numeric types, parameter passing is by value, while other types are by reference. For the writable pass-by-reference type such as instance, modifying them in the called function will also modify the object in the calling function. The following example demonstrates this feature:

```
var l = [], i = 0
def func(a, b)
    a.push(1)
    b ='string'
end
func(l, i)
print(l, i) # [1] 0
```

It can be seen that the value of variable `l` has changed after calling function `func`, but the value of variable `i` has not changed.

Function with variable number of arguments (vararg)

You can define a function to take any arbitrary number of arguments and iterate on them. For example `print()` takes any number of arguments and prints each of them separated by spaces. You need to define the last argument as a capture-all-arguments using * before its name.

All arguments following the formal arguments are grouped at runtime in a `list` instance. If no arguments are captured, the list is empty.

Example:

```
def f(a, b, *c) return size(c) end
f(1,2) # returns 0, c is []
f(1,2,3) # returns 1, c is [3]
f(1,2,3,4) # returns 2, c is [3,4]
```

Calling a function with dynamic number of arguments

Berry syntax allows only to call with a fixed number of arguments. Use the `call(f, [args])` function to pass any arbitrary number or arguments.

You can statically add any number of arguments to `call()`. If the last argument is a `list`, it is automatically expanded to discrete arguments.

Example:

```
def f(a,b) return nil end

call(f,1)        # calls f(1)
call(f,1,2)      # calls f(1,2)
call(f,1,2,3)    # calls f(1,2,3), last arg is ignored by f
call(f,1,[2,3])  # calls f(1,2,3), last arg is ignored by f
call(f,[1,2])    # calls f(1,2)
call(f,[])       # calls f()
```

You can combine `call` and vararg. For example let's create a function that acts like `print()` but converts all arguments to uppercase.

Full example:

```
def print_upper(*a)  # take arbitrary number of arguments, args is a list
    import string
    for i:0..size(a)-1
        if type(a[i]) == 'string'
            a[i] = string.toupper(a[i])
        end
    end
    call(print, a)   # call print with all arguments
end

print_upper("a",1,"Foo","Bar")  # prints: A 1 FOO BAR
```

Functions and local variables

The function body itself is a scope, so the variables defined in the function are all local variables. Unlike directly nested blocks, every time a function is called, space is allocated for local variables. The space for local variables is allocated on the stack, and the allocation information is determined at compile time, so this process is very fast. When multiple levels of scope are nested in a function, the interpreter allocates stack space for the scope nesting chain with the most local variables, rather than the total number of local variables in the function.

`return` Statement

`return` The statement is used to return the result of a function, that is, the return value of the function. All functions in Berry have a return value, but you can not use any `return` statement in the function body. At this time, the interpreter will generate a default `return` statement to ensure that the function returns. `return` There are two ways to write sentences:

```
´return'
´return' expression
```

The first way of writing is to write only the `return` keyword and not the expression to be returned. In this case, the default `nil` value is returned. The second way of writing is to follow the expression **expression** after the `return` keyword, and the value of the expression will be used as the return value of the function. When the program executes to the `return` statement, the currently running function will end execution and return to the code that called the function to continue running.

When using a separate keyword `return` as the return statement of a function, it is easy to cause ambiguity. At this time, it is recommended to add a semicolon after `return` to prevent errors:

```
def func()
    return;
    x = 1
end
```

In this example, the `x = 1` statement after the `return` statement will not be executed, so it is redundant. If this kind of redundant code is avoided, the `return` statement is usually followed by keywords such as `end`, `else` or `elif`. In this case, even if a separate `return` statement is used, there is no need to worry about ambiguity.

closure

Basic Concepts

As mentioned earlier, functions are the first type of value in Berry. You can define functions anywhere, and you can also pass functions as parameters or return values. When another function is defined in a function, the nested function can access the local variables of any outer function. We call the "local variables of the outer function" used in the function the function **Free variable**. The generalized free variables also include global variables, but there is no such rule in Berry.**Closure** is a technique that binds functions to **environments**. The environment is a mapping that associates each free variable of a function with a value. In terms of implementation, closures associate the function prototype with its own variables. Function prototypes are generated at compile time, and environment is a runtime concept, so closures are also dynamically generated at runtime. Each closure binds the function prototype to the environment when it is generated, for example, in the following example:

```
def func(i) # The outer function
    def foo() # The inner function (closure)
        print(i)
    end
    foo()
end
```

The inner function `foo` is a closure, which has a free variable `i`, which is a parameter of the outer function `func`. When the closure `foo` is generated, its function prototype is bound to the environment containing the free variable `i`. When the variable `foo` leaves the scope, the closure will be destroyed. Usually, the inner function will be the return value of the outer function, for example:

```
def func(i) # The outer function
    return def () # Return a closure (anonymous function)
        print(i)
        i = i + 1
    end
end
```

The closure returned here is an anonymous function. When the closure is returned by the outer function, the local variables of the outer function will be destroyed, and the closure will not be able to directly access the variables in the original outer function. The system will copy the value of the free variable to the environment when the free variable is destroyed. The life cycle of these free variables is the same as the closure, and can only be accessed by the closure. The returned function or closure will not be executed automatically, so we need to call the closure returned by the function `func`:

```
f = func(0)
f()
```

This code will output `0`. If we continue to call the closure `f`, we will get the output `1`, `2`, `3`... This may not be well understood: variable [2.198] Is destroyed after the function `func` returns, and as a free variable of the closure `f`, `i` will be stored in the closure environment, so every time `f` is called, the value of `i` will be added to 1 (`func` function definition line 4).

Use of closures

Closures have many uses. Here are a few common uses:

Lazy evaluation

The closure does not do anything until it is called.

Function private communication

You can let some closures share free variables, which are only visible to these closures, and communicate between functions by changing the values of these free variables. This can avoid the use of external variables.

Generate multiple functions

Sometimes we may need to use multiple functions, these functions may only have different values of some variables. We can implement a function and then use these different variables as function parameters. A better way is to return the closure through a factory function, and use these possibly different variables as free variables of the closure, so that you don't always have to write those parameters when calling the function, and any number of similar functions can be generated.

Simulate private members

Some languages support the use of private members in objects, but Berry's class does not support private members. We can use the free variables of closures to simulate private members. This use is not the original intention of designing closures, but nowadays, this "misuse" of closures is very common.

Cache result

If there is a function that is very time-consuming to run, it will take a lot of time to call it every time. We can cache the result of this function, look it up in the cache before calling the function, and return the cached value if found, otherwise call the function and update the cached value. We can use closures to save the cached value so that it will not be exposed to the outer scope, and the cached result will be retained (until the closure is destroyed).

Binding free variables

If multiple closures bind the same free variable, all closures will always share this free variable. E.g:

```
def func(i) # The outer function
    return [# Return a closure list
        def () # The closure #1
            print("closure 1 log:", i)
            i = i + 1
        end,
        def () # The closure #2
            print("closure 2 log:", i)
            i = i + 1
        end
    ]
end
```

The function `func` in this example returns two closures through a list, and these two closures share free variables `i`. If we call these closures:

```
f = func(0)
f[0]() # closure 1 log: 0
f[1]() # closure 2 log: 1
```

As you can see, we updated the free variable `i` when we called the closure `f[0]`, and this change affected the result of calling the closure `f[1]`. This is because if a free variable is used by multiple closures, there is only one copy of the free variable, and all closures have a reference to the free variable entity. Therefore, any modification to the free variable is visible to all closures that use the free variable.

Similarly, before the local variables of the outer function are destroyed, modifying the value of the free variable will also affect the closure:

```
def func()
    i = 0
    def foo()
        print(i)
    end
    i = 1
```

(continues on next page)

(continued from previous page)

```
    return foo
end
```

In this example, we change the value of the variable `i` (which is the free variable of the closure `foo`) from `0` to `1` before the outer function `func` returns, then we call the closure afterwards The value of the free variable `i` when the package `foo` is also `1`:

```
func()() # 1
```

Create closure in loop

When constructing a closure in the loop body, you may not want the free variables of the closure to change with the loop variables. Let's first look at an example of creating a closure in a loop `while`:

```
def func()
    l = [] i = 0
    while i <= 2
        l.push(def () print(i) end)
        i = i + 1
    end
    return l
end
```

In this example, we construct a closure in a loop and put this closure in a `list`. Obviously, when the loop ends, the value of the variable `i` will be `3`, and all the closures in the list `l` are also references using this variable. If we execute the closure returned by `func` we will get the same result:

```
res = func()
res[0]() # 3
res[1]() # 3
res[2]() # 3
```

If we want each closure to refer to different free variables, we can define another layer of functions, and then bind the current loop variables with the function parameters:

```
def func()
    l = [] i = 0
    while i <= 2
        l.push(def (n)
            return def () print(n) end
        end (i))
        i = i + 1
    end
    return l
end
```

To help understand this seemingly incomprehensible code, we focus on the code from lines 4 to 6:

```
def (n)
    return def ()
        print(n)
```

(continues on next page)

(continued from previous page)

```
    end
end (i)
```

Here actually defines an anonymous function and calls it immediately. The function of this temporary anonymous function is to bind the value of the loop variable `i` to its parameter `n`, and the variable `n` is also what we need to close The free variables of the package, so that the free variables bound to the closure constructed during each loop are different. Now we will get the desired output:

```
res = func()
res[0]() # 0
res[1]() # 1
res[2]() # 2
```

There are some ways to solve the problem of loop variables as free variables. A slightly simpler way is to define a temporary variable in the loop body:

```
def func()
    l = [] i = 0
    while i <= 2
        temp = i
        l.push(def () print(temp) end)
        i = i + 1
    end
    return l
end
```

Here `temp` is a temporary variable. The scope of this variable is in the loop body, so it will be redefined every time it loops. We can also use the `for` statement to solve the problem:

```
def func()
    l = []
    for i: 0 .. 2
        l.push(def () print(i) end)
    end
    return l
end
```

This may be the simplest way. `for` The iteration variable of the statement will be created in each loop. The principle is similar to the previous method.

Lambda expression

Lambda expression is a special anonymous function. Lambda expression is composed of parameter list and function body, but the form is different from general function:

```
´/´ args ´->´ expr ´end'
```

args is the parameter list, the number of parameters can be zero or more, and multiple parameters are separated by commas or spaces (cannot be mixed at the same time); **expr** is the return expression, the lambda expression will return the expression value. Lambda expressions are suitable for implementing functions with very simple functions. For example, the lambda expression for judging the size of two numbers is:

```
/ a b -> a < b
```

This is easier than writing a function of the same function. In some general sorting algorithms, this type of size comparison function may need to be used extensively. Using lambda expressions can simplify the code and improve readability.

Like general functions, lambda expressions can form closures. Lambda expressions are called in the same way as ordinary functions. If you use the immediate calling method similar to anonymous functions:

```
lambda = / a b -> a < b
result = lambda(1, 2) # normal calling
result = (/ a b -> a < b)(1, 2) # direct calling
```

Since the function call operator has a higher priority, a pair of parentheses should be added to the lambda expression when making a direct call, so that it will be called as a whole.

6. Object Oriented Function

For optimization considerations, Berry did not consider simple types as objects. These simple types include `nil` types, numeric types, boolean types, and string types. But Berry provides classes to implement the object mechanism. Among Berry's basic data types, `list`, `map` and `range` are class objects. An object is a collection containing data and methods, where data is composed of some variables, and methods are functions. The type of an object is called a class, and the entity of an object is called an instance.

6.1 Class and instance

6.1.1 Class declaration

To use a class, you must first declare it. The declaration of a class starts with the keyword `class`. The member variables and methods of the class must be specified in the declaration. This is an example of declaring a class:

```
class person
    static var majority = 18
    var name, age
    def init(name, age)
        self.name = name
        self.age = age
    end
    def tostring()
        return 'name: '+ str(self.name) + ', age:' + str(self.age)
    end
    def isadult()
        return self.age >= self.majority
    end
end
```

Class member variables are declared with keyword `var`, while member methods are declared with keyword `def`. Currently, Berry does not support initializing member variables at the time of definition, so the initialization of member variables should be done by the constructor. The properties of the class cannot be modified after the declaration is completed, so the class is a read-only object.

This design is to ensure that the class can be statically constructed in the C language when the interpreter is implemented and the `const` property can be used Modified to save RAM

Berry's class does not support access restrictions, and all properties of the class are visible to the outside. In native classes, you can use some tricks to make properties invisible to Berry code (usually let the member name start with dot "."). You can use some conventions to restrict access to the members of the class, such as the convention that the attributes starting with an underscore are private attributes. This convention does not have any use at the grammatical level, but is conducive to the logical structure of the code.

Instantiate

The class itself is just an abstract description. Taking cars as an example, I know the concept of cars, and when we really want to use cars, we need real cars. The use of classes is similar. We will not only use this abstract description, but need to produce a concrete object based on this description. This process is called **Instantiation of the class**, abbreviated as instantiation, and the concrete object produced by instantiation is called **Instance**. The class itself does not have data, and instantiation produces an instance based on the information described by the class and gives the instance specific data.

Method and `self` Parameters

Class methods are essentially functions. Unlike ordinary functions, methods implicitly pass in a `self` parameter, and is always the first parameter, which stores a reference to the current instance. Due to the existence of `self` parameters, the number of parameters of the method will be one more than the number of parameters defined in the declaration. Here we use a simple example to demonstrate:

```
class Test
    def method()
        return self
    end
end
object = Test()
print(object)
print(object.method())
```

This example defines a `Test` class, which has a `method` method, which returns its `self` parameter. The last two lines in the routine print the value of the instance `object` of the `Test` class and the return value of the method `method` respectively. The running result of this example is:

```
<instance: Test()>
<instance: Test()>
```

It can be seen that the `self` parameter of the method and the name of the use instance (`object` in the example) both represent the same object, and they are both instance references. Use `self` to access the members or attributes of the instance in the method.

Synthetic methods

You can declare synthetic dynamic members and methods using the **Virtual members** as described in Chapter 8.2.

Class Variables `static`

Variables or functions can be declared `static`. Static variables have the same value for all instances of the same class. They are declare as `static a = 1` or `static var a = 1`. Static variables are initialized right after the creation of the class.

Class Methods `static`

Methods can be declared `static` which means that they act like regular function and do not take `self` as first argument. Within static methods, there is no implicit `self` variable declared. Static methods can be called via the class or via an instance.

```
class static_demo
    static def increment_static(i)
        return i + 1
    end
    def increment_instance(i)
        return i + 1
    end
end
a = static_demo()
static_demo.increment_static(1)    # call via class
```

2

```
a.increment_static(1)              # call via instance
static_demo.increment_instance(1)
```

type_error: unsupported operand type(s) for +: 'nil' and 'int'
stack traceback:
stdin:6: in function *increment_instance*
stdin:1: in function *main*

```
a.increment_instance(1)
```

2

Constructor and Destructor

Constructor

The constructor of the class is the `init` method. The constructor is called when the class is instantiated. Therefore, the constructor is generally used for member initialization, for example:

```
class Test
    var a
    def init()
```

(continues on next page)

(continued from previous page)

```
        self.a ='this is a test'
    end
end
```

The constructor in this example initializes the `a` member of the `Test` class to the string `'this is a test'`. If we instantiate the class, we can get the value of member `a`:

```
print(Test().a) # this is a test
```

Destructor

The destructor of the class is the `deinit` method. The destructor is called when the instance is destroyed. The destructor is generally used to complete some cleanup work. Because the garbage collection mechanism automatically releases the memory of useless objects, there is no need to release the memory in the destructor (and there is no way to release the memory in the destructor). In most cases, there is no need to use a destructor, unless a certain class requires certain processing when it is destroyed. A typical example is that a file object must close the file when it is destroyed.

Class inheritance

Berry only supports single inheritance, that is, a class can only have one base class, and the base class uses the operator `:` to declare:

```
class Test: Base
    ...
end
```

Here the `Test` class inherits from the `Base` class. The subclass will inherit all the methods and properties of the base class, and you can override them in the subclass. This mechanism is called **Overload**. Under normal circumstances, we will only overload methods, not properties.

The inheritance mechanism of the Berry class is relatively simple. Subclasses will contain references to the base class, and instance objects are similar. When instantiating a class with a base class, multiple objects are actually generated. These objects will be chained together according to the inheritance relationship, and finally we will get the instance object at the end of the inheritance chain.

Method Overload

Overload means that the subclass and the base class use the same name method, and the subclass method will override the mechanism of the base class method. To be precise, member variables can also be overloaded, but this overloading has no meaning. Method overloading is divided into ordinary method overloading and operator overloading.

Common method overload

Operator Overloading

You can use the operator overloading of the class to make the instance support the operation of the built-in operator. For example, for a class overloaded with the addition operator, we can use the addition operator to perform operations on the instance. An overloaded operator is a method with a special name, and the overloaded function form of a binary operator is

```
´def' operator ´(´ other ´)´
    block
´end'
```

operator is an overloaded binary operator. The left operand of the binary operator is the `self` object, and the right operand is the value of the parameter **other**. The overloaded function form of the unary operator is

```
´def' operator ´()´
    block
´end'
```

operator is an overloaded unary operator. To distinguish it from the subtraction operator, the unary minus sign is written as `-*` when overloaded. Operator overloaded functions should have a return value, because the default `nil` return value is usually not the expected result. Let's take an integer class as an example to illustrate the use of operator overloading. First define the `integer` class:

```
class integer
    var value
    def init(v)
        self.value = v
    end
    def +(other)
        return integer(self.value + other.value)
    end
    def *(other)
        return integer(self.value * other.value)
    end
    def -*()
        return integer(-self.value)
    end
    def tostring(other)
        return str(self.value)
    end
end
```

The `integer` class overloads the plus, multiplication, and symbolic operators, and the `tostring` method is to make the instance use the `print` function to output the result. We can use a simple line of code to test the operator overloading function of the class:

```
integer(1) + integer(2) * -integer(3) # -5
```

The result of this line of code is an instance of `integer`. The value of the `value` member of this instance is `-5`, which is the same as the result of the same four arithmetic operations on integers.

Logical operators cannot be overloaded directly. If you need an instance to support logical operations, you must implement the `tobool` method. The method has no parameters and the return value must be of Boolean type. The logic

operation of the instance is actually realized by converting the instance into a Boolean value, so the logic operation of the instance is completely in line with the nature of the general logic operation. The subscript operator is not directly overloaded, but is implemented by the methods `item` and `setitem`. `item` The method is used for subscript reading, its first parameter is the subscript value, and the return value is the result of the subscript operation; `setitem` is used for subscript writing, and its first parameter is the subscript Value, the second parameter is the value to be written, this method does not use the return value.

The overloaded operator can be assigned any meaning, even not satisfying the usual properties of operators. Considering the versatility of the code and the difficulty of understanding, it is not recommended that users give overloaded operators a function far from the general meaning.

Overload of compound assignment operator

The compound assignment operator cannot be directly overloaded, but we can achieve the purpose of "overloading" the compound assignment operator by overloading the binary operator corresponding to the compound assignment operator. For example, after overloading the "+" operator, you can use the "+=" operator for instances of related classes. It is worth noting that the use of compound assignment operations on the instance will cause the variables of the bound instance to lose their reference to the instance.

```
class integer
    var value
    def init(x)
        self.value = x
    end
    def +(other)
        return integer(self.value + other.value)
    end
end
a = integer(4) # a: <instance: 0x55edff400a78>
a += integer(5) # a: <instance: 0x55edff4011b8>
print(a.value) # 9
```

After the 11th line of code is executed, the instance bound in the variable `a` has actually changed. This line of code is equivalent to `a = integer(4) + integer(5)`. If the binary operator of the class overload does not modify the state of the instance, then the corresponding compound assignment operator will not modify any instance (it may generate new instances).

Instance

Instance is an object generated after class instantiation. A class can be instantiated multiple times to generate different instances. Berry instances are referenced by the class they belong to and the corresponding data fields. All instances of a class will refer to this class, but the data fields of these instances are independent of each other.

Access base class object

The built-in function `super` is used to access super class objects. It can be used on classes or instances.

Magic happens when you call a method from the superclass so that it behaves like you intuitively think it would. For example, the common pattern for `init()` is as follows:

```
def init(<args>)
    # do stuff before super init
    super(self).init(<args>)
    # do stuff after super init
end
```

Note that classes always contains an implicit `init()` methods that does nothing, so you can always call init from super class even if no `init()` method was declared.

Full example:

```
class A
    var val
    def init(val)
        # super(self).init(val)    # this would be valid but useless
        self.val = val
    end
    def tostring()
        return "val=" + str(self.val)
    end
end

class B: A
    var magic    # true if value is 42
    def init(val)
        super(self).init(val)     # call super init
        self.magic = (val == 42)
    end
    def tostring()
        if self.magic
            return "magic!"
        else
            return super(self).tostring()
        end
    end
end

####### Example of usage

> b1 = B(1)
```

(continues on next page)

(continued from previous page)

```
> b1
val=1
> b42 = B(42)
> b42
magic!
```

Advanced features When calling `super(self).<method>(<args>)` some magic happens. When the super-method is called, the `self` arguments refers to the lowest more specific class. However the `<method>` if searched not from the class of `self` (which is always the lowest), but from the super class of the class containing the method currently running.

Example:

```
> class A
      def init()
          print("In A::init, self is of type", classname(self))
      end
  end
> class B:A
      def init()
          print("In B::init, self is of type", classname(self))
          super(self).init()
      end
  end
> class C:B
      def init()
          print("In C::init, self is of type", classname(self))
          super(self).init()
      end
  end
> c = C()
In C::init, self is of type C
In B::init, self is of type C
In A::init, self is of type C
>
```

Explanation:

- calling `C:init()` on `instance<C>`
- in `C:init()` `self` is `instance<C>`, `super(self).init()` refers to the super class of `C` (current method) i.e. `B`, so `B:init()` is called with `instance<C>` argument
- in `B:init()` `self` is `instance<C>`, `super(self).init()` refers to the super class of `B` (current method) i.e. `A`, so `A:init()` is called with `instance<C>` argument
- in `A:init()` `self` is `instance<C>`, print and return

Note: for backwards compatibility, super can take a second argument `super(instance, class)` to specify the class where to resolve the method. This feature should not be used anymore as it is error-prone.

7. Libraries and Modules

7.1 Basic library

There are some functions and classes that can be used directly in the standard library. They provide basic services for Berry programs, so they are also called basic libraries. The functions and classes in the basic library are visible in the global scope (belonging to the built-in scope), so they can be used anywhere. Do not define variables with the same name as the functions or classes in the base library. Doing so will make it impossible to reference the functions and classes in the base library.

7.1.1 Built-in function

`print` function

Example

```
print(...)
```

Description

This function prints the input parameters to the standard output device. The function can accept any type and any number of parameters. All types will print their value directly, and for an instance, this function will check whether the instance has a `tostring()` method, and if there is, print the return value of the instance calling the `tostring()` method, otherwise it will print the address of the instance.

```
print('Hello World!') # Hello World!
print([1, 2, '3']) # [1, 2, '3']
print(print) # <function: 0x561092293780>
```

`input` function

Example

```
input()
input(prompt)
```

Description

`input` The function is used to input a line of character string from the standard input device. This function can use the `prompt` parameter as an input prompt, and the `prompt` parameter must be of string type. After calling the `input` function, characters will be read from the keyboard buffer until a newline character is encountered.

```
input('please enter a string:') # please enter a string:
```

`input` The function does not return until the "Enter" key is pressed, so the program "stuck" is not an error.

`type` function

Example

```
type(value)
```

- *value*: Input parameter (expect to get its type).
- *return value*: A string describing the parameter type.

Description

This function receives a parameter of any type and returns the type of the parameter. The return value is a string describing the type of the parameter. Table below shows the return values corresponding to the main parameter types:

Parameter Type	return value	Parameter Type	return value
Nil	`'nil'`	Integer	`'int'`
Real	`'real'`	Boolean	`'bool'`
Function	`'function'`	Class	`'class'`
String	`'string'`	Instance	`'instance'`
native pointer	`'ptr'`		

```
type(0) #'int'
type(0.5) #'real'
type('hello') #'string'
type(print) #'function'
```

`classname` function

Example

```
classname(object)
```

Description

This function returns the class name (string) of the parameter. Therefore the parameter must be a class or instance, and other types of parameters will return `nil`.

```
classname(list) #'list'
classname(list()) #'list'
classname({}) #'map'
classname(0) # nil
```

`classof` function

Example

```
classof(object)
```

Description

Returns the class of an instance object. The parameter `object` must be an instance. If the function is successfully called, it will return the class to which the instance belongs, otherwise it will return `nil`.

```
classof(list) # nil
classof(list()) # <class: list>
classof({}) # <class: map>
classof(0) # nil
```

str function

Example

```
str(value)
```

Description

This function converts the parameters into strings and returns. `str` Functions can accept any type of parameters and convert them. When the parameter type is an instance, it will check whether the instance has a `tostring()` method, if there is, the return value of the method will be used, otherwise the address of the instance will be converted into a string.

```
str(0) # '0'
str(nil) #'nil'
str(list) #'list'
str([0, 1, 2]) #'[0, 1, 2]'
```

number function

```
number(value)
```

Description

This function converts the input string or number into a numeric type and returns. If the input parameter is an integer or real number, it returns directly. If it is a character string, try to convert the character string to a numeric value in decimal format. The integer or real number will be automatically judged during the conversion. Other types return `nil`.

Example

```
number(5) # 5
number('45.6') # 45.6
number('50') # 50
number(list) # nil
```

int function

```
int(value)
```

Description

This function converts the input string or number into an integer and returns it. If the input parameter is an integer, return directly, if it is a real number, discard the decimal part. If it is a string, try to convert the string to an integer in decimal. Other types return `nil`. When the parameter type is an instance, it will check whether the instance has a `toint()` method, if there is, the return value of the method will be used.

Example

```
int(5) # 5
int(45.6) # 45
int('50') # 50
int('0x10) # 16 - hex literal are valid
int(list) # nil
```

real function

```
real(value)
```

Description

This function converts the input string or number into a real number and returns. If the input parameter is a real number, it will return directly, if it is an integer, it will be converted to a real number. If it is a string, try to convert the string to a real number in decimal. Other types return `nil`.

Example

```
real(5) # 5, type(real(5)) →'real'
real(45.6) # 45.6
real('50.5') # 50.5
real(list) # nil
```

bool function

```
bool(value)
```

Description

This function converts the input string or number into a boolean and returns it.

The conversion follows the following rules:

- `nil`: converted to `false`.
- **Integer**: when the value is `0`, it is converted to `false`, otherwise it is converted to `true`.
- **Real number**: when the value is `0.0`, it is converted to `false`, otherwise it is converted to `true`.
- **String**: when the value is "" (empty string) it is concerted to `false` otherwise it is converted to `true`.
- **Comobj** and **Comptr**: when the internal pointer is `NULL` it is converted to `false`, otherwise it is converted to `true`.
- **Instance**: if the instance contains a method `tobool()`, the return value of the method will be used, otherwise it will be converted to `true`.
- All other types: convert to `true`.

Example

```
bool() # false
bool(nil) # false
bool(false) # false
```

(continues on next page)

(continued from previous page)

```
bool(true) # true
bool(0) # false
bool(1) # true
bool("") # false
bool("a") # true
bool(3.5) # true
bool(list) # true
bool([]) # true
bool({}) # true
# advanced
import introspect
bool(introspect.toptr(0)) # false
bool(introspect.toptr(0x1000)) # true
```

size function

```
size(value)
```

Description

This function returns the size of the input string. If the input parameter is not a string, 0 is returned. The length of the string is calculated in bytes. This function also works for `list` and `map` instances and returns the number of elements.

Example

```
size(10) # 0
size('s') # 1
size('string') # 6
size([1,2]) # 2
size({"a":1}) # 1
```

super function

```
super(object)
```

Description

This function returns the parent object of the instance. When you instantiate a derived class, it will also instantiate its base class. The `super` function is required to access the instance of the base class (that is, the parent object).

Please look at chapter 6 about magic behavior of `super(object)` when calling a super method.

Example

```
class mylist: list end
l = mylist() # classname(l) -->'mylist'
sl = super(l) # classname(sl) -->'list'
```

`assert` function

```
assert(expression)
assert(expression, message)
```

Description

This function is used to implement the assertion function. `assert` The function accepts a parameter. When the value of the parameter is `false` or `nil`, the function will trigger an assertion error, otherwise the function will not have any effect. It should be noted that even if the parameter is a value equivalent to `false` in logical operations (for example, `0`), it will not trigger an assertion error. The parameter `message` is optional and must be a string. If this parameter is used, the text information given in `message` will be output when an assertion error occurs, otherwise the default "`Assert Failed`" message will be output.

Example

```
assert(false) # assert failed!
assert(nil) # assert failed!
assert() # assert failed!
assert(0) # assert failed!
assert(false,'user assert message.') # user assert message.
assert(true) # pass
```

`compile` function

```
compile(string)
compile(string, 'string')
compile(filename, 'file')
```

Description

This function compiles the Berry source code into a function. The source code can be a string or a text file. `compile` The first parameter of the function is a string, and the second parameter is a string `'string'` or `'file'`. When the second parameter is `'string'` or there is no second parameter, the `compile` function will compile the first parameter as the source code. When the second parameter is `'file'`, the `compile` function will compile the file corresponding to the first parameter. If the compilation is successful, `compile` will return the compiled function, otherwise it will return `nil`.

Example

```
compile('print(\'Hello World!\')')() # Hello World!
compile('test.be','file')
```

`list` Class

`list` is a built-in type, which is a sequential storage container that supports subscript reading and writing. `list` Similar to arrays in other programming languages. Obtaining an instance of the `list` class can be constructed using a pair of square brackets: `[]` will generate an empty `list` instance, and `[expr, expr, ...]` will generate a `list` instance with several elements. It can also be instantiated by calling the `list` class: executing `list()` will get an empty `list` instance, and `list(expr, expr, ...)` will return an instance with several elements.

`list` method (Constructor)

Initialize the `list` container. This method can accept 0 to multiple parameters. The `list` instance generated when multiple parameters are passed will have these parameters as elements, and the arrangement order of the elements is consistent with the arrangement order of the parameters.

`tostring` method

Serialize the `list` instance to a string and return it. For example, the result of executing `[1, [], 1.5].tostring()` is `'[1, [], 1.5]'`. If the `list` container refers to itself, the corresponding position will use an ellipsis instead of the specific value:

```
l = [1, 2]
l[0] = l
print(l) # [[...], 2]
```

`concat` method

Converts each element of the list to strings, and concatenate using the provided string.

```
l = [1, 2, 3]
l.concat()  # '123'
l.concat(", ")  # '1, 2, 3'
```

`push` method

Append an element to the end of the `list` container. The prototype of this method is `push(value)`, the parameter `value` is the value to be appended, and the appended value is stored at the end of the `list` container. The append operation increases the number of elements in the `list` container by 1. You can append any type of value to the `list` instance.

`insert` method

Insert an element at the specified position of the `list` container. The prototype of this method is `insert(index, value)`, the parameter `index` is the position to be inserted, and `value` is the value to be inserted. After inserting an element at the position `index`, all the elements that originally started from this position will move backward by one element. The insert operation increases the number of elements in the `list` container by 1. You can insert any type of value into the `list` container.

Suppose that the value of a `list` instance `l` is `[0, 1, 2]`, and we insert a string `'string'` at position 1, and we need to call `l.insert(1, 'string')`. Finally, the new `list` value is `[0, 'string', 1, 2]`.

If the number of elements in a `list` container is S, the value range of the insertion position is $\{i:Si<S\}$. When the insertion position is positive, index backward from the head of the `list` container, otherwise index forward from the end of the `list` container.

`remove` method

Remove an element from the container. The prototype of this method is `remove(index)`, and the parameter `index` is the position of the element to be removed. After the element is removed, the element behind the removed element will move forward by one element, and the number of elements in the container will be reduced by 1. Like the `insert` method, the `remove` method can also use positive or negative indexes.

`item` method

Get an element in the `list` container. The prototype of this method is `item(index)`, the parameter `index` is the index of the element to be obtained, and the return value of the method is the element at the index position. `list` The container supports multiple indexing methods:

- Integer index: The index value can be a positive integer or a negative integer. If the index is negative, it is relative to the end of the list; i.e. `-1` indicates the last element in the list. The return value of `item` is the element at the index position. If the index position exceeds the number of elements in the container or is before the 0th element, the `item` method will return `nil`.
- `list` Index: Using a list of integers as an index, `item` returns a `list`, and each element in the return value `list` is an element corresponding to each integer index in the parameter `list`. The value of the expression `[3, 2, 1].item([0, 2])` is `[3, 1]`. If an element type in the parameter `list` is not an integer, then the value at that position in the return value `list` is `nil`.
- `range` Index: Using an integer range as an index, `item` returns a `list`. The returned value stores the indexed elements from `list` from the lower limit to the upper limit of the parameter `range`. If the index exceeds the index range of the indexed `list`, the return value `list` will use `nil` to fill the position beyond the index.

`setitem` method

Set the value of the specified position in the container. The prototype of this method is `setitem(index, value)`, `index` is the position of the element to be written, and `value` is the value to be written. `index` is the integer index value of the writing position. Index positions outside the index range of the container will cause `setitem` to fail to execute.

`size` method

Returns the number of elements in the container, which is the length of the container. The prototype of this method is `size()`.

`resize` method

Reset `list` the length of the container. The prototype of this method is `resize(count)`, and the parameter `count` is the new length of the container. When using `resize` to increase the length of the container, the new element will be initialized to `nil`. Using `resize` to reduce the length of the container will discard some elements at the end of the container. E.g:

```
l = [1, 2, 3]
l.resize(5) # Expansion, l == [1, 2, 3, nil, nil]
l.resize(2) # Reduce, l == [1, 2]
```

`iter` method

Returns an iterator for traversing the current `list` container.

`find` method

Similar to `item` or `list[idx]`. The only difference is if the index is out of range, `find` return `nil` instead or raising an exception.

`reverse` method

Changes the list in-place and reverses the order of elements. Also returns the resulting list.

`map` Class

`map` Class is a built-in class type used to provide an unordered container of key-value pairs. Inside the Berry interpreter, `map` uses the Hash table to implement. You can use curly brace pairs to construct a `map` container. Using an empty curly brace pair `{}` will generate an empty `map` instance. If you need to construct a non-empty `map` instance, use a colon to separate the key and value, and use a semicolon to separate multiple key-value pairs. For example, `{0: 1, 2: 3}` has two key-value pairs (0,1) and (2,3). You can also get an empty `map` instance by calling the `map` class.

`map` method (Constructor)

Initialize the `map` container, this method does not accept parameters. Executing `map()` will get an empty `map` instance.

`tostring` method

Serialize `map` as a string and return. The serialized string is similar to literal writing. For example, the result of executing `'str': 1, 0: 2` is `"'str': 1, 0: 2"`. If the `map` container refers to itself, the corresponding position will use an ellipsis instead of the specific value:

```
m = {'map': nil,'text':'hello'}
m['map'] = m
print(m) # {'text':'hello','map': {...}}
```

`insert` method

Insert a key-value pair in the `map` container. The prototype of this method is `insert(key, value)`, the parameter `key` is the key to be inserted, and `value` is the value to be inserted. If the key `map` to be inserted exists in the container, the original key-value pair will be updated.

`remove` method

Remove a key-value pair from the `map` container. The prototype of this method is `remove(key)`, and the parameter `key` is the key of the key-value pair to be deleted.

`item` method

Get a value in the `map` container. The prototype of this method is `item(key)`, the parameter `key` is the key of the value to be obtained, and the return value of the method is the value corresponding to the key.

`setitem` method

Set the value corresponding to the specified key in the container. The prototype of this method is `setitem(key, value)`, `key` is the key of the key-value pair to be written, and `value` is the value to be written. If there is no key-value pair with the key `key` in the container, the `setitem` method will fail.

`size` method

Return the number of key-value pairs of the `map` container, which is the length of the container. The prototype of this method is `size()`.

`contains` method

Returns boolean `true` if a matching key-value pair is found in the `map` container, otherwise `false`. The prototype of this method is `contains(key)`.

`find` method

Returns the value corresponding to the specified key in the container. The prototype of this method is `find(key)` or `find(key, defaultvalue)`, `key` is the key of the key-value pair to be accessed, and `defaultvalue` is the default value returned if the key is not found. If no default value is specified, `nil` is returned instead.

`range` Class

`range` The class is used to represent an integer closed interval. Use the binary operator `..` to construct an instance of `range`. The left and right operands of the operator are required to be integers. For example, `0..10` means the integer interval [0,10].

If you don't specify the high range, it is set to `MAXINT`. Example: `print(0..) # (0..9223372036854775807)`

There are typically two ways to traverse a list:

```
l = [1,2,3,4]
for e:l print(e) end # 1/2/3/4
for i:0..size(l)-1 print(l[i]) end # 1/2/3/4
```

bytes Class

bytes object are represented as arrays of Hex bytes. bytes constructor takes a string of Hex and builds the in-memory buffer.

Example:

```
b = bytes()
print(b)   # bytes('')
b = bytes("1155AA")  # sequence of bytes 0x11 0x55 0xAA
size(b)  # 3 = 3 bytes
b[0]     # 17 (0x11)
b[0] = 16  # assign first byte
print(b)   # bytes('1055AA')
```

bytes method (Constructor)

Initialize a bytes array. There are several options.

Option 1: empty value

bytes() creates a new empty bytes array. size(bytes()) == 0.

There is no limit in the size of a bytes array, except the available memory. An internal buffer is allocated and reallocated in case the previous one was too small. The initial buffer is 36 bytes, but you can pre-allocated a larger (or smaller) buffer if you know in advance the size needed.

Similarly the buffer is automatically shrunk if it is used less than its needed size.

```
b = bytes(4096)   # pre-allocated 4096 bytes
```

Option 2: initial value

If first argument is a string it is parsed as a list of Hex values. You can add an optional second argument to pre-allocate a bigger buffer.

```
b = bytes("BEEF0000")
print(b)   # bytes('beef0000')
b = bytes("112233", 128)   # pre-allocate 128 bytes internally
print(b)   # bytes('112233')
```

Option 3: fixed size

If the size provided is negative, the array size is fixed and cannot be lowered nor raised.

```
b = bytes(-8)
print(b)   # bytes('0000000000000000')

b = bytes("AA", -4)
print(b)   # bytes('AA000000')

b = bytes("1122334455", -4)
attribute_error: bytes object size if fixed and cannot be resized
```

Option 4: memory mapping

Caution, use with great care

In this mode, the bytes array is mapped to a specific region in memory. You must provide the base address as `comptr` and the size. Size is always fixed whether it is positive or negative. This feature is **dangerous** since you can access any memory location, causing a crash if the location is protected or invalid. Use with care.

In this case `b.ismapped()` returns `true` indicating a mapped memory buffer. In all other cases `b.ismapped()` returns `false`. This is typically used to know if Berry allocated the buffer or not, and if sub-elements need to be explicitly deallocated.

Example:

```
import introspect
def f() return 0 end

addr = introspect.toptr(f)
print(addr)    # <ptr: 0x3ffeaf88>

b = bytes(addr, 8)
print(b)    # bytes('F8EAFE3F24000000')
# this example shows the first 8 bytes of the function object in memory
```

`size` method

Returns the number of bytes in the bytes array

```
b = bytes("1122334455")
b.size()    # 5
size(b)     # 5
```

`tostring` method

Shows a human readable form of the bytes array in hex. By default, it shows only the first 32 characters. You can request more characters by adding an int argument with the maximum number of bytes you want to convert. `tostring` is internally used when you print an object. `print(b)` is equivalent to `print(b.tostring())`. It is different from `asstring` which turns a bytes array to the equivalent low-level string object without any encoding.

```
b = bytes("1122334455")
b.tostring()    # 'bytes(\'1122334455\')'

b = bytes()
b.resize(64)    # resize to 64 bytes
b.tostring()    # 'bytes(\
↪'0000000000000000000000000000000000000000000000000000000000000000...\')'
b.tostring(500) # 'bytes(\
↪'0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪')'
```

`tohex` method

Converts the bytes array in an hex string, similar to the one returned by `tostring()` but without decorators.

```
b = bytes("1122334455")
b.tohex()   # '1122334455'
```

`fromhex` method

Updates the content of the bytes array from a new hex string. This allows to load a new hex string without allocating a new bytes object.

```
b = bytes("1122334455")
b.fromhex("AABBCC")  # bytes('AABBCC')
```

`clear` method

Sets back the bytes array to empty

```
b = bytes("1122")
b.clear()
print(b)   # bytes()
```

`resize` method

Shrink or expand the bytes array to match the specified size. If expanded, NULL (0x00) bytes are added at the end of the buffer.

```
b = bytes("11223344")
b.resize(6)
print(b)   # bytes('112233440000')
b.resize(2)
print(b)   # bytes('1122')
```

Concatenation + and `..` methods

You can use + to concatenate two `bytes` list, creating a new `bytes` object. `..` changes the list in place and can be used to add an `int` (1 bytes) or a `bytes` object

```
b = bytes("1122")
c = bytes("3344")
d = b + c           # b and c are unchanged
print(d)            # bytes('11223344')
print(b)            # bytes('1122')
print(c)            # bytes('3344')

e = b..c            # now b is changed
print(e)            # bytes('11223344')
```

(continues on next page)

(continued from previous page)

```
print(b)            # bytes('11223344')
print(c)            # bytes('3344')
```

bytes access [] method

You can access individual bytes as integers, to read and write. Values not in the range 0..255 are silently choped.

```
b = bytes("010203")
print(b[0])         # 1

# negative indices count from the end
print(b[-1])        # 3

# out of bounds generate an exception
print(b[5])         # index_error: bytes index out of range

b[0] = -1
print(b)            # bytes('FF0203')

b[1] = 256
print(b)            # bytes('FF0003')
```

range access [] method

You can use the `[]` accessor with a range to get an sub-list of `bytes`. If an index is negative, it is taken from the end of the array.

This construct cannot be used a an *lvalue*, i.e. you can't splice like `b[1..2] = bytes("0011") # not allowed`.

```
b = bytes("001122334455")
print(b[1..2])      # bytes('1122')

# remove first 2 bytes
print(b[2..-1])     # bytes('22334455')

# remove last 2 bytes
print(b[0..-3])     # bytes('00112233')

# overshooting is allowed
print(b[4..10])     # bytes('4455')

# inversed indices return an empty array
print(b[5..4])      # bytes('')
```

The standard `item` and `setitem` methods are implemented, and transparently mapped to `[]` operator.

copy method

Creates a fresh new copy of the `bytes` object. A new memory buffer is allocated and data is duplicated.

```
b = bytes("1122")
print(b)          # bytes('1122')

c = b.copy()
print(c)          # bytes('1122')

b.clear()
print(b)          # bytes('')
print(c)          # bytes('1122')bytes('1122')
```

get, geti methods

Read a 1/2/4 bytes value from any offset in the bytes array. The standard mode is little endian, if you specify a negative size it enables big endian. `get` returns unsigned values, while `geti` returns signed values.

```
b.get(<offset>, <size>) -> bytes object
```

If the offset is out of range, `0` is returned (no exception raised).

Example:

```
b = bytes("010203040506")
print(b.get(2,2))         # 1027 - 0x0403 read 2 bytes little endian
print(b.get(2,-2))        #  772 - 0x0304 read 2 bytes big endian

print(b.get(2,4))         # 100992003 - 0x06050403 - little endian
print(b.get(2,-4))        #  50595078 - 0x03040506 - big endian

b = bytes("FEFF")
print(b.get(0, 2))        # 65534 - 0xFFFE
print(b.geti(0, 2))       # -2 - 0xFFFE
```

set, seti methods

Similar to `get` and `geti`, allows to set a 1/2/4 bytes value at any offset. `seti` uses signed integers, `set` unsigned (actually it does not make a difference).

If the offset is out of range, no change is done (no exception raised).

```
bytes.set(<offset>, <value>, <size>)
```

add method

This methods adds value of 1/2/4 bytes (little endian or big endian) at the end of the buffer. If size is negative, the value is treated as big endian.

```
b.add(<value>, <size>)
```

Example:

```
b = bytes("0011")
b.add(0x22, 1)
print(b)              # bytes('001122')
b.add(0x2233, 2)
print(b)              # bytes('0011223322')
b.add(0x22334455, 4)
print(b)              # bytes('001122332255443322')
b.add(0x00)
print(b)              # bytes('0011223322554433200')
b.clear()
b.add(0x0102, -2)
print(b)              # bytes('0102')
b.add(0x01020304, -4)
print(b)              # bytes('010201020304')
```

asstring method

Converts a bytes buffer to a string. The buffer is converted as-is without any encoding considerations. If the buffer contains NULL characters, the string will be truncated.

```
b=bytes("3344")
print(b.asstring())   # '3D'
```

fromstring method

Converts a bytes buffer to a string. The buffer is converted as-is without any encoding considerations. If the buffer contains NULL characters, the string will be truncated.

```
b=bytes().fromstring("Hello")
print(b)              # bytes('48656C6C6F')
```

bits manipulation setbits, getbits methods

You can read and write at sub-byte level, specifying from which bit to which bit. The offset is in bits, not bytes. Add the number of bytes * 8.

```
b.setbits(<offset_bits>, <len_bits>, <value>)
b.getbits(<offset_bits>, <len_bits>)
```

base64 encode `tob64` method

Converts a bytes array to a base64 string.

```
b = bytes('deadbeef0011')
s = b.tob64()
print(s)                # 3q2+7wAR
```

base64 decode `fromb64` method

Converts a base64 string into a bytes array.

```
s = '3q2+7wAR'
b = bytes().fromb64(s)
print(b)                # bytes('DEADBEEF0011')
```

`getfloat` and `setfloat` methods

Similar to `get/set`, allows to read or write a 32 bits float value.

```
b.getfloat(<offset>)
b.getfloat(<offset>, <number>)
```

```
b = bytes("00000000")
b.getfloat(0)     # 0
b.setfloat(0, -1.5)
print(b)  # bytes('0000C0BF')
b.getfloat(0)  # -1.5
```

`_buffer` method

Advanced feature: returns the address of the buffer in memory, to be used with C code.

```
b = bytes('1122')
b._buffer()  # <ptr: 0x600000c283c0>
```

`_change_buffer` method

Advanced feature: works only for mapped buffers (i.e. `b.ismapped() == true`), allows to remap the buffer to a new memory address. This allows to reuse the bytes() object without reallocating a new instance.

```
# this example uses pointer allocation, use with great care
b1 = bytes("11223344")
b2 = bytes("AABBCCDD")
b1._buffer()  # <ptr: 0x600000c2c390>
b2._buffer()  # <ptr: 0x600000c24270>

# now we create c as a mapped buffer of 4 bytes to the address of b1
```

(continues on next page)

(continued from previous page)

```
c = bytes(b1._buffer(), 4)
print(c)  # bytes('11223344') -- mapped to b1
c._buffer()   # <ptr: 0x600000c2c390>

# let's change a byte to prove it
c[0] = 254
print(c)  # bytes('FE223344')
print(b1) # bytes('FE223344') -- b1 was changed

# reallocate c to map b2
c._change_buffer(b2._buffer())
print(c)  # bytes('AABBCCDD')
c._buffer()  # <ptr: 0x600000c24270>
```

Expansion Modules

JSON Module

JSON is a lightweight data exchange format. It is a subset of JavaScript. It uses a text format that is completely independent of the programming language to represent data. Berry provides a JSON module to provide support for JSON data. The JSON module only contains two functions `load` and `dump`, which are used to parse JSON strings and multiply Berry objects and serialize a Berry object into JSON text.

`load` function

```
load(text)
```

Description

This function is used to convert the input JSON text into a Berry object and return it. The conversion rules are shown in Table 1.1. If there is a syntax error in the JSON text, the function will return `nil`.

JSON type	Berry type
`null`	`nil`
`number`	`integer` or `real`
`string`	`string`
`array`	`list`
`object`	`map`

JSON type to Berry type conversion rules

Example

```
import json
json.load('0') # 0
json.load('[{"name": "liu", "age": 13}, 10.0]') # [{'name':'liu','age': 13}, 10]
```

dump function

```
dump(object, ['format'])
```

Description

This function is used to serialize the Berry object into JSON text. The conversion rules for serialization are shown in Table 1.2.

Berry type	JSON type
nil	null
integer	number
real	number
list	array
map	object
mapKey of	string
other	string

Berry type to JSON type conversion rules

Example

```
import json
json.dump('string') #'"string"'
json.dump('string') #'"string"'
json.dump({0:'item 0','list': [0, 1, 2]}) #'{"0":"item 0","list":[0,1,2]}'
json.dump({0:'item 0','list': [0, 1, 2],'func': print},'format')
#-
{
    "0": "item 0",
    "list": [
        0,
        1,
        2
    ],
    "func": "<function: 00410310>"
}
-#
```

Math Module

This module is used to provide support for mathematical functions, such as commonly used trigonometric functions and square root functions. To use the math module, first use the `import math` statement to import. All examples in this section assume that the module has been imported correctly.

`pi` constant

The approximate value of Pi , a real number type, approximately equal to 3.141592654.

Example

```
math.pi # 3.14159
```

`abs` function

```
abs(value)
```

Description

This function returns the absolute value of the parameter, which can be an integer or a real number. If there are no parameters, the function returns `0`, if there are multiple parameters, only the first parameter is processed. `abs` The return type of the function is a real number.

Example

```
math.abs(-1) # 1
math.abs(1.5) # 1.5
```

`ceil` function

```
ceil(value)
```

Description

This function returns the rounded up value of the parameter, that is, the smallest integer value greater than or equal to the parameter. The parameter can be an integer or a real number. If there are no parameters, the function returns `0`, if there are multiple parameters, only the first parameter is processed. `ceil` The return type of the function is a real number.

Example

```
math.ceil(-1.2) # -1
math.ceil(1.5) # 2
```

`floor` function

```
floor(value)
```

Description

This function returns the rounded down value of the parameter, which is not greater than the maximum integer value of the parameter. The parameter can be an integer or a real number. If there are no parameters, the function returns `0`, if there are multiple parameters, only the first parameter is processed. `floor` The return type of the function is a real number.

Example

```
math.floor(-1.2) # -2
math.floor(1.5) # 1
```

`sin` function

```
sin(value)
```

Description

This function returns the sine function value of the parameter. The parameter can be an integer or a real number, and the unit is radians. If there are no parameters, the function returns `0`, if there are multiple parameters, only the first parameter is processed. `sin` The return type of the function is a real number.

Example

```
math.sin(1) # 0.841471
math.sin(math.pi * 0.5) # 1
```

`cos` function

```
cos(value)
```

Description

This function returns the value of the cosine function of the parameter. The parameter can be an integer or a real number in radians. If there are no parameters, the function returns `0`, if there are multiple parameters, only the first parameter is processed. `cos` The return type of the function is a real number.

Example

```
math.cos(1) # 0.540302
math.cos(math.pi) # -1
```

tan function

```
tan(value)
```

Description

This function returns the value of the tangent function of the parameter. The parameter can be an integer or a real number, in radians. If there are no parameters, the function returns `0`, if there are multiple parameters, only the first parameter is processed. `tan` The return type of the function is a real number.

Example

```
math.tan(1) # 1.55741
math.tan(math.pi / 4) # 1
```

asin function

```
asin(value)
```

Description

This function returns the arc sine function value of the parameter. The parameter can be an integer or a real number. The value range is [1,1]. If there are no parameters, the function returns `0`, if there are multiple parameters, only the first parameter is processed. `asin` The return type of the function is a real number and the unit is radians.

Example

```
math.asin(1) # 1.5708
math.asin(0.5) * 180 / math.pi # 30
```

acos function

```
acos(value)
```

Description

This function returns the arc cosine function value of the parameter. The parameter can be an integer or a real number. The value range is [1,1]. If there are no parameters, the function returns `0`, if there are multiple parameters, only the first parameter is processed. `acos` The return type of the function is a real number and the unit is radians.

Example

```
math.acos(1) # 0
math.acos(0) # 1.5708
```

atan function

```
atan(value)
```

Description

This function returns the arctangent function value of the parameter. The parameter can be an integer or a real number. The value range is [∞,+∞]. If there are no parameters, the function returns 0, if there are multiple parameters, only the first parameter is processed. atan The return type of the function is a real number and the unit is radians.

Example

```
math.atan(1) * 180 / math.pi # 45
```

sinh function

```
sinh(value)
```

Description

This function returns the hyperbolic sine function value of the parameter. If there are no parameters, the function returns 0, if there are multiple parameters, only the first parameter is processed. sinh The return type of the function is a real number.

Example

```
math.sinh(1) # 1.1752
```

cosh function

```
cosh(value)
```

Description

This function returns the hyperbolic cosine function value of the parameter. If there are no parameters, the function returns 0, if there are multiple parameters, only the first parameter is processed. cosh The return type of the function is a real number.

Example

```
math.cosh(1) # 1.54308
```

tanh function

```
tanh(value)
```

Description

This function returns the hyperbolic tangent function value of the parameter. If there are no parameters, the function returns 0, if there are multiple parameters, only the first parameter is processed. tanh The return type of the function is a real number.

Example

```
math.tanh(1) # 0.761594
```

sqrt function

```
sqrt(value)
```

Description

This function returns the square root of the argument. The parameter of this function cannot be negative. If there are no parameters, the function returns `0`, if there are multiple parameters, only the first parameter is processed. `sqrt` The return type of the function is a real number.

Example

```
math.sqrt(2) # 1.41421
```

exp function

```
exp(value)
```

Description

This function returns the value of the parameter's exponential function based on the natural constant *e*. If there are no parameters, the function returns `0`, if there are multiple parameters, only the first parameter is processed. `exp` The return type of the function is a real number.

Example

```
math.exp(1) # 2.71828
```

log function

```
log(value)
```

Description

This function returns the natural logarithm of the argument. The parameter must be a positive number. If there are no parameters, the function returns `0`, if there are multiple parameters, only the first parameter is processed. `log` The return type of the function is a real number.

Example

```
math.log(2.718282) # 1
```

log10 function

```
log10(value)
```

Description

This function returns the logarithm of the parameter to the base 10. The parameter must be a positive number. If there are no parameters, the function returns 0, if there are multiple parameters, only the first parameter is processed. log10 The return type of the function is a real number.

Example

```
math.log10(10) # 1
```

deg function

```
deg(value)
```

Description

This function is used to convert radians to angles. The unit of the parameter is radians. If there are no parameters, the function returns 0, if there are multiple parameters, only the first parameter is processed. deg The return type of the function is a real number and the unit is an angle.

Example

```
math.deg(math.pi) # 180
```

rad function

```
rad(value)
```

Description

This function is used to convert angles to radians. The unit of the parameter is angle. If there are no parameters, the function returns 0, if there are multiple parameters, only the first parameter is processed. rad The return type of the function is a real number and the unit is radians.

Example

```
math.rad(180) # 3.14159
```

pow function

```
pow(x, y)
```

Description

The return value of this function is the result of the expression xy, which is the parameter x to the y power. If the parameters are not complete, the function returns 0, if there are extra parameters, only the first two parameters are processed. pow The return type of the function is a real number.

Example

```
math.pow(2, 3) # 8
```

srand function

```
srand(value)
```

Description

This function is used to set the seed of the random number generator. The type of the parameter should be an integer.

Example

```
math.srand(2)
```

rand function

```
rand()
```

Description

This function is used to get a random integer.

Example

```
math.rand()
```

Time Module

This module is used to provide time-related functions.

time function

```
time()
```

Description

Returns the current timestamp. The timestamp is the time elapsed since Unix Epoch (1st January 1970 00:00:00 UTC), in seconds.

dump function

```
dump(ts)
```

Description

The input timestamp `ts` is converted into a time `map`, and the key-value correspondence is shown in Table below:

key	value	key	value	key	value
``'year'``	Year (from 1900)	`' month'`	Month (1-12)	``'day'``	Day (1-31)
``'hour'``	Hour (0-23)	``'min'``	Points (0-59)	``'sec'``	Seconds (0-59)
`'we ekday'`	Week (1-7)				

`time.dump` The key-value relationship of the function return value

`clock` function

```
clock()
```

Description

This function returns the elapsed time from the start of execution of the interpreter to when the function is called in seconds. The return value of this function is of type `real`, and its timing accuracy is determined by the specific platform.

String Module

The String module provides string processing functions.

To use the string module, first use the `import string` statement to import. All examples in this section assume that the module has been imported correctly.

`count` function

```
string.count(s, sub[, begin[, end]])
```

Count the number of occurrences of the sub string in the string s. Search from the position between `begin` and `end` of `s` (default is 0 and size(s)).

`split` function

```
string.split(s, pos)
```

Split the string s into two substrings at position `pos`, and returns the list of those strings.

```
string.split(s, sep[, num])
```

Splits the string s into substrings wherever sep occurs, and returns the list of those strings. Split at most num times (default is string.count(s, sep)).

find function

```
string.find(s, sub[, begin[, end]])
```

Check whether the string s contains the substring sub. If the begin and end (default is 0 and size(s)) are specified, they will be searched in this range.

hex function

```
hex(number)
```

Convert number to hexadecimal string.

byte function

```
byte(s)
```

Get the code value of the first byte of the string s.

char function

```
char(number)
```

Convert the number used as the code to a character.

format function

```
string.format(fmt[, args])
```

Returns a formatted string. The pattern starting with '%' in the formatting template fmt will be replaced by the value of [args]: %[flags][fieldwidth][.precision]type

Type	Description
%d	Decimal integer
%o	Octal integer
%x	Hexadecimal integer lowercase
%X	Hexadecimal integer uppercase
%x	Octal integer lowercase
%X	Octal integer uppercase
%f	Floating-point in the form [-]nnnn.nnnn
%e %E	Floating-point in exp. form [-]n.nnnn e [+|-]nnn, uppercase if %E
%g %G	Floating-point as %f if 4 < exp. precision, else as %e; uppercase if %G
%c	Character having the code passed as integer
%s	String with no embedded zeros
%q	String between double quotes, with special characters escaped
%%	The '%' character (escaped)

Type	Description
•	Left-justifies, default is right-justify
•	Prepends sign (applies to numbers)
(space)	Prepends sign if negative, else space
#	Adds "0x" before %x, force decimal point; for %e, %f, leaves trailing zeros for %g

Field width and precision	Description
n	Puts at least n characters, pad with blanks
0n	Puts at least n characters, left-pad with zeros
.n	Use at least n digits for integers, rounds to n decimals for floating-point or no more than n chars. for strings

Module os

The OS module provides system-related functions, such as file and path-related functions. These functions are platform-related. Currently, Windows VC and POSIX style codes are implemented in the Berry interpreter. If it runs on other platforms, the functions in the OS module are not guaranteed to be provided.

TODO

Module global

Module `global` provides a way to access global variables via a module. The Berry compiler checks that a global exists when compiling code. However there are cases when globals are created dynamically by code and are not yet known at compile time. Using the module `global` gives complete freedom to access statically or dynamically global variables.

Accessing a global is simplay made with `global.<name>` for reading and writing. You can also use the special syntax `global.(name)` if `name` is a variable containing the name of the global as string.

Example:

```
> import global
> a = 1
> global.a
1
>
> b
syntax_error: stdin:1: 'b' undeclared (first use in this function)
> global.b = 2
> b
2
> global.b
2
> var name = "b"
> global.(name)
2
```

Calling `global()` returns the list of all global names currently defined (builtins are not included).

```
> import global
> a = 1
> global.b = 2
> global()
['_argv', 'b', 'global', 'a']
```

`global.contains(<name>) -> bool` provides an easy way to know if a global name is already defined.

```
> import global
> global.contains("g")
false
> g = 1
> global.contains("g")
true
```

Module `introspect`

Module `introspect` provides primitives to dynamically access variables or modules. Use with `import introspect`.

`introspect.members(object: class or module or instance or nil) -> list` returns the list of names of members for the `class`, `instance` or `module`. Keep in mind that it does not include potential virtual members created via `member` and `setmember`.

`introspect.members()` returns the list of global variables (not including builtins) and is equivalent to `global()`

`introspect.get(object: class or instance or module, name:string) -> any` and `introspect.set(object: class or instance or module, name:string, value:any) -> nil` allows to read and write any member by name.

`introspect.get(o, "a")` is equivalent ot `o.a`, `introspect.set(o, "a", 1)` is equivalent to `o.a = 1`. There is also an alternative syntax: `o.("a")` is equivalent to `o.a` and `o.("a) = 1` is equivalent to `o.a = 1`.

`introspect.module(name:string) -> any` is equivalent to `import name` except that it does not create the global or local variable, but returns the module. This is the only way to load a module with a dynamic name, `import name` only takes a static name.

`introspect.toptr(addr:int) -> comptr` converts an integer to a comptr pointer. `introspect.fromptr(addr:comptr) -> int` does the reverse and converts a pointer to an int. Warning: use with care. On platforms where `int` and `void*` don't have the same size, these functions will most certainly give unusable results.

`introspect.ismethod(f:function) -> bool` checks if the provided function is a method of an instance (taking self as first argument), or a plain function. This is mainly use to prevent a common mistake of passing an instance method as callbakc, where you should use a closure capturing the instance like `/ -> self.do()`.

Module `solidify`

This module allows to solidify Berry bytecode into flash. This allows to save RAM since the code is in Flash, makes it a good alternative to native C functions.

See 8.4 Solidification

8. Advanced features

8.1 `strict` mode

Berry allows full freedom from the developer. But after some experience in coding with Berry, you will find that there are common mistakes that are hard to find and that the compiler could help you catch. The `strict` mode does additional checks **at compile time** about some common mistakes.

This mode is enabled with `import strict` or when running Berry with `-s` option: `berry -s`

Mandatory `var` for local variables

This is the most common mistake, a variable assigned without `var` is either global if a global already exists or local otherwise. Strict mode rejects the assignment if there is no global with the same name.

No more allowed:

```
def f()
  i = 0    # this is a local variable
  var j = 0
end
```

syntax_error: stdin:2: strict: no global 'i', did you mean 'var i'?

But still works for globals:

```
g_i = 0
def f()
  g_i = 1
end
```

No overriding of builtins

Berry allows to override a builtin. This is however generally not desirable and a source of hard to find bugs.

```
map = 1
syntax_error: stdin:1: strict: redefinition of builtin 'map'
```

Multiple `var` with same name not allowed in same scope

Berry tolerated multiple declaration of a local variable with the same name. This is now considered as an error (even without strict mode).

```
def f()
  var a
  var a   # redefinition of a
end
syntax_error: stdin:3: redefinition of 'a'
```

No hiding of local variable from outer scope

In Berry you can declare local variables with the same name in inner scope. The variable in the inner scope hides the variable from outer scope for the duration of the scope.

The only exception is that variables starting with dot '.' can mask from outer scope. This is the case with hidden local variable `.it` when multiple `for` are embedded.

```
def f()
  var a    # variable in outer scope
  if a
    var a  # redefinition of a in inner scope
  end
end
syntax_error: stdin:4: strict: redefinition of 'a' from outer scope
```

8.2 Virtual members

Virtual members allows you to dynamically and programmatically add members and methods to classes and modules. You are no more limited to the members declared at creation time.

This feature is inspired from Python's `__getattr__()` / `__setattr__()`. The motivation comes from LVGL integration to Berry in Tasmota. The integration needs hundreds of constants in a module and thousands of methods mapped to C functions. Statically creation of attributes and methods does work but consumes a significant amount of code space.

This features allows to create two methods:

Berry method	Description
`member`	`(name:string) -> any`Should return the value of the specified `name`
`setmember`	`(name:st ring, value:any) -> nil`Should store the `value` to the virtual member with the specified `name`

module undefined

The `member()` function must be able to distinguish between a member with a `nil` value and the member not existing. To avoid any ambiguity, the `member()` function can indicate that the member does not exist in two ways:

- either raise an exception
- or `import undefined` and return the `undefined` module. This is used as a marker for the VM to know that the attribute does not exist, while benefitting from consistent exceptions

Example of a dynamic object to which you can add members, but would return an error if the member was not previously added.

```
class dyn
    var _attr
    def init()
        self._attr = {}
    end
    def setmember(name, value)
        self._attr[name] = value
    end
    def member(name)
        if self._attr.contains(name)
            return self._attr[name]
        else
            import undefined
            return undefined
        end
    end
end
```

Exemple of usage:

```
a = dyn()
a.a
```

attribute_error: the 'dyn' object has no attribute 'a' stack traceback: stdin:1: in function *main*

```
a.a = 1
a.a
```

1

```
a.a = nil
a.a
```

implicit call of `member()`

When the following code `a.b` is executed, the Berry VM does the following:

- Get the object named `a` (local or global), raise an exception if it doesn't exist
- Check if the object `a` is of type `module`, `instance` or `class`. Raise an exception otherwise
- Check if object `a` has a member called `b`. If yes, return its value, if no proceed below
- If object `a` is of type `class`, raise an exception because virtual members do not work for static (class) methods
- Check if object `a` has a member called `member` and it is a `function`. If yes call it with parameter `"b"` as string. If no, raise an exception
- Check the return value. If it is the module `undefined` raise an exception indicating that the member does not exist

implicit call of `setmember()`

When the following code `a.b = 0` (mutator) is executed, the Berry VM does the following:

- Get the object named `a` (local or global), raise an exception if it doesn't exist
- Check if the object `a` is of type `module`, `instance` or `class`. Raise an exception otherwise
  - If `a` is of type `class`, check if member `b` exists. If yes, change its value. If no, raise an exception. (virtual members don't work for classes or static methods)
  - If `a` is of type `instance`, check if member `b` exists. If yes, change its value. If no, proceed below
    - Check if `a` has a member called `setmember`. If yes call it, if no raise an exception
  - If `a` is of type `module`. If the module is not read-only, create of change the value (`setmember` is never called for a writable module). If the module is read-only, then `setmember` is called if it exists.

Exception handling

To indicate that a member does not exist, `member()` shall return `undefined` after `import undefined`.

You can also raise an exception in `member()` but be aware that Berry might try to call methods like `tostring()` that will land on your `member()` method if they don't exist as static methods.

To indicate that a member is invalid, `setmember()` should raise an exception or return `undefined`. Returning anything else like `nil` inficates that the assignment was succesful.

Be aware that you may receive member names that are not valid Berry identifiers. The syntax `a.("<->")` will call `a.member("<->")` with a virtual member name that is not lexically valid, i.e. cannot be called in regular code, except by using indirect ways like `introspect` or `member()`.

Specificities for classes

Access to members of class object do not trigger virtual members. Hence it is not possible to have virtual static methods.

Specificities for modules

Modules do support reading static members with `member()`.

When writing to a member, the behavior depends whether the module is writable (in memory) or read-only (in firmware).

If the module is writable, the new members are added directly to the module and `setmember()` is never called.

If the module is read-only, then `setmember()` is called whenever you try to change or create a member. It is then your responsibility to store the values in a separate object like a global.

Example

Example:

```
class T
    var a
    def init()
        self.a = 'a'
    end

    def member(name)
        return "member "+name
    end

    def setmember(name, value)
        print("Set '"+name+"': "+str(value))
    end
end
t=T()
```

Now let's try it:

```
t.a
```

'a'

```
t.b
```

'member b'

```
t.foo
```

'member foo'

```
t.bar = 2
```

Set 'bar': 2

This works for modules too:

```
m = module()
m.a = 1
m.member = def (name)
    return "member "+name
end
m.setmember(name, value)
    print("Set '"+name+"': "+str(value))
end
```

Trying:

```
m.a
```

1

```
m.b
```

'member b'

```
m.c = 3   # the allocation is valid so `setmember()` is not called
m.c
```

3

More advanced example:

```
class A
    var i

    def member(n)
      if n == 'ii' return self.i end
      return nil      # we make it explicit here, but this line is optional
    end

    def setmember(n, v)
      if n == 'ii' self.i = v end
    end
  end
a=A()

a.i       # returns nil
a.ii      # implicitly calls `a.member("ii")`
```

attribute_error: the 'A' object has no attribute 'ii'
stack traceback:
stdin:1: in function *main*

```
# returns an exception since the member is nil (considered is non-existant)
```

(continues on next page)

(continued from previous page)

```
a.ii = 42    # implicitly calls `a.setmember("ii", 42)`
a.ii         # implicitly calls `a.member("ii")` and returns `42`
```

42

```
a.i          # the concrete variable was changed too
```

42

8.3 How-to package a module

This guide drives you through the different options of packaging code for reuse using Berry's `import` directive.

Behavior of `import`

When you use `import <module> [as <name>]`, the following steps happen:

- There is a global cache of all modules already imported. If `<module>` was already imported, `import` returns the value in cache already returned by the first call to `import`. No other actions are taken.
- `import` searches for a module of name `<module>` in the following order:

1. in native modules embedded in the firmware at compile time
2. in file system, starting with current directory, then iterating in all directories from `sys.path`: look for file `<name>`, then `<name>.bec` (compiled bytecode), then `<name>.be`. If `BE_USE_SHARED_LIB` is enabled, it also looks for shared libraries like `<name>.so` or `<name>.dll` although this optional is generally not available on MCUs.

- The code loaded is executed. The code should finish with a `return` statement. The object returned is stored in the global cache and made available to caller (in local or global scope).
- If the returned object is a `module` and if the module as a `init` member, then an extra step is taken. The function `<module>.init(m)` is called passing as argument the module object itself. The value returned by `init()` replaces the value in the global cache. Note that the `init()` is called at most once during the first `import`.

Note: an implicit `init(m)` function is always present in all modules, even if none was declared. This implicit function has no effect.

Packaging a module

Here is a simple example of a module:

File `demo_module.be`:

```
# simple module
# use `import demo_module`

demo_module = module("demo_module")

demo_module.foo = "bar"

demo_module.say_hello = def ()
```

(continues on next page)

(continued from previous page)

```
    print("Hello Berry!")
end

return demo_module      # return the module as the output of import
```

Example of use:

```
import demo_module

demo_module
<module: demo_module>

demo_module.say_hello()
```

Hello Berry!

```
demo_module.foo
```

'bar'

```
demo_module.foo = "baz"     # the module is writable, although this is highly discouraged
demo_module.foo
```

'baz'

Package a singleton (monad)

The problem of using modules is that they don't have instance variables to keep track of data. They are essentially designed for state-less libraries.

Below you will find an elegant way of packaging a class singleton returned as an `import statement`.

To do this, we use different tricks. First we declare the class for the singleton as an inner class of a function, this prevents from polluting the global namespace with this class. I.e. the class will not be accessible by other code.

Second we declare a module `init()` function that creates the class, creates the instance and returns it.

By this scheme, `import <module>` actually returns an instance of a hidden class.

Example of `demo_monad.be`:

```
# simple monad
# use `import demo_monad`

demo_monad = module("demo_monad")

# the module has a single member `init()` and delegates everything to the inner class
demo_monad.init = def (m)

    # inncer class
    class my_monad
        var i

        def init()
```

(continues on next page)

(continued from previous page)

```
            self.i = 0
        end

        def say_hello()
            print("Hello Berry!")
        end
    end

    # return a single instance for this class
    return my_monad()
end

return demo_monad      # return the module as the output of import, which is eventually␣
↪replaced by the return value of 'init()'
```

Example:

```
import demo_monad
demo_monad
<instance: my_monad()>     # it's an instance not a module

demo_monad.say_hello()
```

Hello Berry!

```
demo_monad.i = 42        # you can use it like any instance
demo_monad.i
```

42

```
demo_monad.j = 0         # there is strong member checking compared to modules
```

attribute_error: class 'my_monad' cannot assign to attribute 'j' stack traceback: stdin:1: in function *main*

8.4 Solidification

Solidification is the process of capturing compiled Berry structures and code (classes, modules, maps, lists…) and storing them into firmware. It reduces dramatically the use of memory, but has some limitations.

`solidify` module

Solidification is handle by `solidify` module. This module is not compiled by default because of its size (~10kB). You need to compile with `#define BE_USE_SOLIDIFY_MODULE 1` directive.

The module has a single member `dump(x)` that takes a single argument (the object to solidify) and output to `stdout` the solidified code.

By default, solidify adds all string constants to the global pool. You can generate weak strings instead (eligible to pruning by the linker) by setting the second argument to `true`.

By default `solidify.dump` outputs the solidified code to standard output. You can specify a file as third argument. The file needs to be open in writeable mode, and is not closed so that you can concatenate multiple objects.

```
solidify.dump(object:any, [, strings_weak:bool, file_out:file]) -> nil
```

Solidification of functions

You can solidify a single function.

Example:

```
> def f() return "hello" end
> import solidify
> solidify.dump(f)
```

```
/********************************************************************
** Solidified function: f
********************************************************************/
be_local_closure(f,   /* name */
  be_nested_proto(
    0,                          /* nstack */
    0,                          /* argc */
    0,                          /* varg */
    0,                          /* has upvals */
    NULL,                       /* no upvals */
    0,                          /* has sup protos */
    NULL,                       /* no sub protos */
    1,                          /* has constants */
    ( &(const bvalue[ 1]) {     /* constants */
    /* K0   */  be_nested_str(hello),
    }),
    &be_const_str_f,
    &be_const_str_solidified,
    ( &(const binstruction[ 1]) {  /* code */
      0x80060000,  //  0000  RET    1   K0
    })
  )
);
/********************************************************************/
```

To compile using weak strings (i.e. strings that can be eliminated by the linker if the object is not included in the target executable), use `solidify.dump(f, true)`:

```
/********************************************************************
** Solidified function: f
********************************************************************/
be_local_closure(f,   /* name */
  be_nested_proto(
    0,                          /* nstack */
    0,                          /* argc */
    0,                          /* varg */
    0,                          /* has upvals */
    NULL,                       /* no upvals */
    0,                          /* has sup protos */
    NULL,                       /* no sub protos */
    1,                          /* has constants */
    ( &(const bvalue[ 1]) {     /* constants */
    /* K0   */  be_nested_str_weak(hello),
```

(continues on next page)

(continued from previous page)

```
    }),
    be_str_weak(f),
    &be_const_str_solidified,
    ( &(const binstruction[ 1]) {  /* code */
      0x80060000,  //  0000  RET    1   K0
    })
  )
);
/********************************************************************/
```

Solidification of classes

When you solidify a class, it embeds all the sub-elements. An `C` stub is also added to create the class and add it to the global scope.

```
>  class demo
      var i
      static foo = "bar"

      def init()
          self.i = 0
      end

      def say_hello()
          print("Hello Berry!")
      end
  end
> import solidify
> solidify.dump(demo)
```

```
/********************************************************************
** Solidified function: init
********************************************************************/
be_local_closure(demo_init,   /* name */
  be_nested_proto(
    1,                          /* nstack */
    1,                          /* argc */
    2,                          /* varg */
    0,                          /* has upvals */
    NULL,                       /* no upvals */
    0,                          /* has sup protos */
    NULL,                       /* no sub protos */
    1,                          /* has constants */
    ( &(const bvalue[ 2]) {     /* constants */
    /* K0   */  be_nested_str(i),
    /* K1   */  be_const_int(0),
    }),
    &be_const_str_init,
    &be_const_str_solidified,
    ( &(const binstruction[ 2]) {  /* code */
```

(continues on next page)

(continued from previous page)

```
      0x90020101,  //  0000  SETMBR R0  K0  K1
      0x80000000,  //  0001  RET    0
    })
  )
);
/*********************************************************************/


/*********************************************************************
** Solidified function: say_hello
*********************************************************************/
be_local_closure(demo_say_hello,   /* name */
  be_nested_proto(
    3,                          /* nstack */
    1,                          /* argc */
    2,                          /* varg */
    0,                          /* has upvals */
    NULL,                       /* no upvals */
    0,                          /* has sup protos */
    NULL,                       /* no sub protos */
    1,                          /* has constants */
    ( &(const bvalue[ 1]) {     /* constants */
    /* K0   */  be_nested_str(Hello_X20Berry_X21),
    }),
    &be_const_str_say_hello,
    &be_const_str_solidified,
    ( &(const binstruction[ 4]) {  /* code */
      0x60040001,  //  0000  GETGBL R1  G1
      0x58080000,  //  0001  LDCONST    R2  K0
      0x7C040200,  //  0002  CALL   R1  1
      0x80000000,  //  0003  RET    0
    })
  )
);
/*********************************************************************/


/*********************************************************************
** Solidified class: demo
*********************************************************************/
be_local_class(demo,
    1,
    NULL,
    be_nested_map(4,
    ( (struct bmapnode*) &(const bmapnode[]) {
        { be_const_key(i, -1), be_const_var(0) },
        { be_const_key(say_hello, 2), be_const_closure(demo_say_hello_closure) },
        { be_const_key(init, -1), be_const_closure(demo_init_closure) },
        { be_const_key(foo, 1), be_nested_str(bar) },
    })),
    (bstring*) &be_const_str_demo
);
```

(continues on next page)

(continued from previous page)

```
/********************************************************************/

void be_load_demo_class(bvm *vm) {
    be_pushntvclass(vm, &be_class_demo);
    be_setglobal(vm, "demo");
    be_pop(vm, 1);
}
```

Sub-classes are also supported.

```
> class demo_sub : demo
      var j

      def init()
          super(self).init()
          self.j = 1
      end
  end
> solidify.dump(demo_sub)
```

```
/********************************************************************
** Solidified function: init
********************************************************************/
be_local_closure(demo_sub_init,   /* name */
  be_nested_proto(
    3,                          /* nstack */
    1,                          /* argc */
    0,                          /* varg */
    0,                          /* has upvals */
    NULL,                       /* no upvals */
    0,                          /* has sup protos */
    NULL,                       /* no sub protos */
    1,                          /* has constants */
    ( &(const bvalue[ 3]) {     /* constants */
    /* K0   */  be_nested_str(init),
    /* K1   */  be_nested_str(j),
    /* K2   */  be_const_int(1),
    }),
    &be_const_str_init,
    &be_const_str_solidified,
    ( &(const binstruction[ 7]) {  /* code */
      0x60040003,  //  0000  GETGBL R1  G3
      0x5C080000,  //  0001  MOVE   R2  R0
      0x7C040200,  //  0002  CALL   R1  1
      0x8C040300,  //  0003  GETMET R1  R1  K0
      0x7C040200,  //  0004  CALL   R1  1
      0x90020302,  //  0005  SETMBR R0  K1  K2
      0x80000000,  //  0006  RET    0
    })
  )
);
/********************************************************************/
```

(continues on next page)

(continued from previous page)

```
/********************************************************************
** Solidified class: demo_sub
********************************************************************/
extern const bclass be_class_demo;
be_local_class(demo_sub,
    1,
    &be_class_demo,
    be_nested_map(2,
    ( (struct bmapnode*) &(const bmapnode[]) {
        { be_const_key(init, -1), be_const_closure(demo_sub_init_closure) },
        { be_const_key(j, 0), be_const_var(0) },
    })),
    be_str_literal("demo_sub")
);
/********************************************************************/

void be_load_demo_sub_class(bvm *vm) {
    be_pushntvclass(vm, &be_class_demo_sub);
    be_setglobal(vm, "demo_sub");
    be_pop(vm, 1);
}
```

Solidification of modules

When you solidify a module, it embeds all the sub-elements. It also works with embedded lists or maps.

```
> def say_hello() print("Hello Berry!") end
> m = module("demo_module")
> m.i = 0
> m.s = "foo"
> m.f = say_hello
> m.l = [0,1,"a"]
> m.m = {"a":"b", "2":3}
> import solidify
> solidify.dump(m)
```

```
/********************************************************************
** Solidified function: say_hello
********************************************************************/
be_local_closure(demo_module_say_hello,   /* name */
  be_nested_proto(
    2,                          /* nstack */
    0,                          /* argc */
    0,                          /* varg */
    0,                          /* has upvals */
    NULL,                       /* no upvals */
    0,                          /* has sup protos */
    NULL,                       /* no sub protos */
```

(continues on next page)

(continued from previous page)

```
    1,                          /* has constants */
    ( &(const bvalue[ 1]) {     /* constants */
    /* K0   */  be_nested_str(Hello_X20Berry_X21),
    }),
    &be_const_str_say_hello,
    &be_const_str_solidified,
    ( &(const binstruction[ 4]) {  /* code */
      0x60000001,  //  0000  GETGBL R0  G1
      0x58040000,  //  0001  LDCONST    R1  K0
      0x7C000200,  //  0002  CALL   R0  1
      0x80000000,  //  0003  RET    0
    })
  )
);
/*********************************************************************/


/*********************************************************************
** Solidified module: demo_module
*********************************************************************/
be_local_module(demo_module,
    "demo_module",
    be_nested_map(5,
    ( (struct bmapnode*) &(const bmapnode[]) {
        { be_const_key(l, -1), be_const_simple_instance(be_nested_simple_instance(&be_
↪class_list, {
        be_const_list( *     be_nested_list(3,
    ( (struct bvalue*) &(const bvalue[]) {
        be_const_int(0),
        be_const_int(1),
        be_nested_str(a),
    }))    ) } )) },
        { be_const_key(m, 3), be_const_simple_instance(be_nested_simple_instance(&be_
↪class_map, {
        be_const_map( *     be_nested_map(2,
    ( (struct bmapnode*) &(const bmapnode[]) {
        { be_const_key(a, -1), be_nested_str(b) },
        { be_const_key(2, -1), be_const_int(3) },
    }))    ) } )) },
        { be_const_key(i, 4), be_const_int(0) },
        { be_const_key(f, -1), be_const_closure(demo_module_say_hello_closure) },
        { be_const_key(s, -1), be_nested_str(foo) },
    }))
);
BE_EXPORT_VARIABLE be_define_const_native_module(demo_module);
/*********************************************************************/
```

Limitations of solidification

Solidification works for many objects: `class`, `module`, `functions` and embedded constants or objects like `int`, `real`, `string`, `list` and `map`.

Limitations:

- Upvals are not supported. You cannot solidify a closure that captures upvals from outer scope
- Capturing global variables requires to compile with `-g` "named globals" option (enabled by default on Tasmota)
- String constants are limited to 255 bytes, long strings (above 255 characters are not supported - because nobody ever had a need for)
- Solidified objects are read-only, this has some consequences on classes. You can solidify a class with its static members when it is created, but you cannot solidify a function that creates a class deriving from another class or with static members. The core reason is that setting the superclass or assigning the static members is implemented using mutating code on the new class - which cannot work on a read-only non-mutating class.
- Solidified code may be dependent on the size of `int` and `real` and may not be ported across MCUs with different sizes of types. You may need to re-solidify for each target.

9. FFI

Foreign Function Interface (FFI) is an interface for interaction between different languages. Berry provides a set of FFI to realize the interaction with C language, this set of interfaces is also very easy to use in C++. Most of the FFI interfaces are functions, and their declarations are placed in the *berry.h* file. In order to reduce the amount of RAM used, FFI also provides a mechanism for generating a fixed hash table during C compilation. This mechanism must use external tools to generate C code.

9.1 Basics

The most important interactive function in FFI should be the function of calling Berry code and C function mutually. In order to realize that two languages call each other's functions, we must first understand the parameter passing mechanism of the Berry function.

9.1.1 Virtual Machine

Unlike compiled languages, Berry language cannot run directly on a physical machine, but in a specific software environment, which is **Virtual Machine** (VM). Similar to a real computer, the source code in text form cannot be executed in a virtual machine, but must be converted into "bytecode" by a compiler. The Berry virtual machine is defined as a C structure `bvm`, the content of this structure is invisible to FFI. Through some FFI functions, we can create and initialize a virtual machine. We introduce the use of virtual machines through a simple example:

```
void berry_test(void)
{
    bvm *vm = be_vm_new(); // Construct a VM
    be_loadstring(vm, "print('Hello Berry')"); // Compile test code
    be_pcall(vm, 0); // Call function
    be_vm_delete(vm); // Destroy the VM
}
```

This code gives a complete example of using a virtual machine. First, call the function `be_vm_new` to construct a new virtual machine, and then all operations are completed in this virtual machine object. `be_vm_new` The function will

automatically link the standard library when creating a virtual machine. The function of lines 4 to 5 is to compile the source code in a string into a Berry function and then call it. Finally, call the `be_vm_delete` function on line 6 to destroy the virtual machine. Executing this function will get a line of output in the terminal:

```
Hello Berry
```

In all scenarios, the virtual machine construction, library loading and destruction process are the same as the 3rd, 4th and 6th lines in the above example. If necessary, the way to compile or load the source code may be different. For example, for the source code in the form of a file, it can be compiled through the `be_loadfile` function. The source code will be compiled into a Berry function, and the function will be stored on the top of the stack. The Berry function can be executed by calling the FFI function `be_pcall` or `be_call`. You can also use the REPL through the `be_repl` function. The interface of the REPL will be described in the relevant chapters.

9.1.2 Virtual Stack

Berry uses a virtual stack and native functions written in C to pass values. Each element in the stack is a Berry value. When Berry code calls a native function, it always creates a new stack and pushes all the parameters onto the stack. This virtual stack can also be used in C code to store data, and the value stored in the stack will not be reclaimed by the garbage collector.

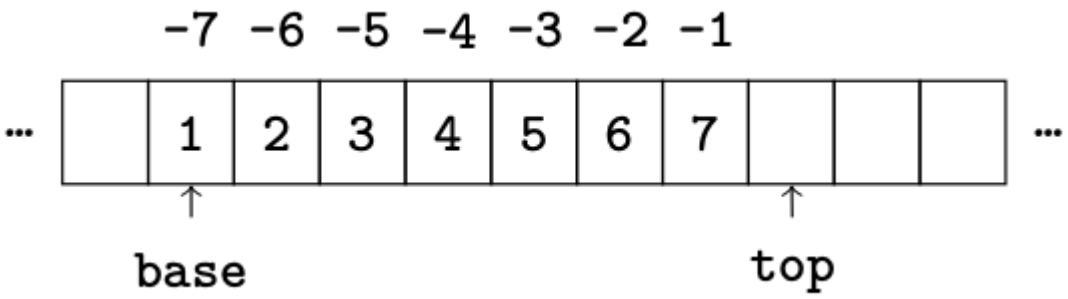

Fig. 1: Virtual_Stack

The virtual stack used by Berry is shown in Figure above.

The virtual stack grows from left to right. When Berry code calls a native function, it will get an initial stack. The position of the first value of the stack is called **base**, and the last position is called **top**, the native function Only the value from the bottom of the stack to the position before the top of the stack can be accessed. The position of the bottom of the stack is fixed, while the position of the top of the stack can be moved, and the top of the stack is always empty. The reason for this property is that after the new value is pushed into the virtual stack, the original position of the top of the stack will be written The new value, the top pointer of the stack will move forward to the next position; conversely, if the value at the top of the virtual stack is popped, the top pointer of the stack will decrease 1. At this time, although the position of the top pointer of the stack is objectively Value, but this value is invalid and may be cleared at any time, so the pointer position on the top of the stack is still empty. When the virtual stack is empty, the bottom pointer `base` is equal to the top pointer `top`. The virtual stack does not strictly follow the operating rules of the stack: in addition to push and pop, the virtual stack can also be accessed by index, and even insert or delete values at any position. There are two ways to index elements in the stack: one is based on the bottom of the stack **Absolute index**, the absolute index value is a positive integer starting from 1; the other is based on the top of the stack **Relative index**, The relative index value is a negative integer starting from 1. Take Figure above as an example, the index value 1,2…8 is an absolute index, and the absolute index of an element is the distance from the element to the bottom of the stack. The index value 1,2…8 is a relative index, and the relative index value of an element is the negative number of the distance from the element to the top of the stack. If an index value *index* is valid, then the element it refers to should be between the bottom of the stack and the top of the stack, which means that the expression

`1abs(*index*)*top* *base*+1` is satisfied.

For convenience, we stipulate that the stack bottom pointer `base` is used as a reference, and its absolute index 1, and the previous value of `base` is not considered (usually, `base` is not the bottom position of the entire stack). For example, when a native function returns, the location where the return value is stored is just before `base`, and these locations are usually not accessible by the native function.

Operate Virtual Stack

Index and stack size

As mentioned earlier, two indexing methods can be used to access the virtual stack, and the index value must be valid. At the same time, in many cases it is also necessary to push new values onto the stack. In this case, the programmer must ensure that the stack will not overflow. By default, Berry guarantees `BE_STACK_FREE_MIN` space for native functions to use. This value can be modified in the file *berry.h*. Its default value is usually 10, which should be sufficient in most cases. If you really need to expand the stack, you can call the FFI function `be_stack_require`. The prototype of this function is:

```
void be_stack_require(bvm *vm, int count);
```

The parameter `count` is the amount of space needed. When the remaining space in the virtual stack is insufficient, the stack capacity will be expanded, otherwise this function will do nothing.

caveat: If a stack overflow occurs, or if an invalid index is used to access the stack, the program will crash. You can turn on the debugging switch `BE_DEBUG` (section [section::BE_DEBUG]), which will turn on the assertion function, and you can get some debugging information at runtime to catch errors such as stack overflow or invalid index.

Get value from stack

There is a set of functions in FFI to get values from the virtual stack. These functions usually convert the values in the stack into simple values supported by the C language and then return. The following are commonly used FFIs to get values from the stack:

```
bint be_toint(bvm *vm, int index);
breal be_toreal(bvm *vm, int index);
int be_tobool(bvm *vm, int index);
const char* be_tostring(bvm *vm, int index);
void* be_tocomptr(bvm *vm, int index);
```

The parameter form of these functions is the same, but the return value is different. The first four functions are easy to understand. Just like their names, the function of `be_toint` is to convert the values in the virtual stack to C integer values (`bint` is usually an alias of type `int`) and return. The function of the last function `be_tocomptr` is to take out a pointer value of a general type from the virtual stack. The specific meaning of this pointer is explained by the C program itself.

These functions use the same way to interpret the parameters: the parameter `vm` is the pointer of the virtual machine instance; `index` is the index of the element to be retrieved, which can be a relative index or an absolute index. You cannot use FFI to remove Berry's complex data types from the virtual stack, so you cannot remove a `map` type or `class` type from the stack. One of the benefits of this design is that there is no need to consider garbage collection in native functions.

Native function

Native function It is implemented by C language and can be called by Berry code. The native function can be an ordinary function. In this case, calling the native function will not generate any dynamically allocated space, just like a normal C function call. Native functions can also be closures, and space needs to be allocated for free variables when creating native closures. Under normal circumstances, simple native functions are sufficient to meet the needs. They save resources than native closures and are easier to use.

Define native function

The native function itself is a C function, but they all have a specific form. The definition of the native function is:

```
int a_native_function(bvm *vm)
{
    // do something ...
}
```

The native function must be a C function whose parameter is a pointer to `bvm` and the return value is `int`. Berry's functions must return a value, and native functions are no exception. Unlike the return value of the C language, the return value of the native function is not the value carried by the C `return` statement. You can use these FFIs to return the value of the native function, and they also make the C function return:

```
be_return(bvm *vm);
be_return_nil(bvm *vm);
```

These FFIs are actually two macros, and there is no need to use the C `return` statement when using them. `be_return` Will put the top of the virtual stack

Use native function

After the native function is defined, it must be added to the interpreter in some way before it can be called in Berry code. One of the easiest ways is to add native functions to Berry's built-in object table. The process of setting native objects as Berry built-in objects is called **registered**. The FFI of the registered native function is:

```
void be_regfunc(bvm *vm, const char *name, bntvfunc f);
```

`vm` is the current virtual machine instance, `name` is the name of the native function, and `f` is the pointer of the native function. The specific behavior of this function is related to the value of the `BE_USE_PRECOMPILED_OBJECT` macro (although the FFI is still available when using the compile-time construction technique, it cannot dynamically register the built-in variables. In this case, please refer to the method of registering the built-in objects. 1.3). The definition of native function type `bntvfunc` is:

```
typedef int (*bntvfunc)(bvm*);
```

In fact, the `bntvfunc` type is the function pointer type with the parameter `bvm` and the return value type `int`. `be_regfunc` The function should be called before parsing the Berry source code.

You can also push the native function into the virtual stack, and then use an FFI function `be_call` to call it. A more common usage is to use the native function object in the virtual stack as the return value.

Complete example

We end this section with a simple example. Here, we are going to implement a `add` function that adds two numbers and returns the result of the calculation. First, we define a native function to implement this function:

```
static int l_add(bvm *vm)
{
    int top = be_top(vm); // Get the number of arguments
    /* Verify the number and type of arguments */
    if (top == 2 && be_isnumber(vm, 1) && be_isnumber(vm, 2)) {
        breal x = be_toreal(vm, 1); // Get the first argument
        breal y = be_toreal(vm, 2); // Get the second argument
        be_pushreal(vm, x + y); // Push the result onto the stack
        be_return(vm); // Return the value at the top of the stack
    }
    be_return_nil(vm); // Return nil when something goes wrong
}
```

Native functions usually do not need to be used outside the C file, so they are generally declared as `static` types. Use the `be_top` function to get the absolute index of the top of the virtual stack (`top` value), which is the capacity of the stack. We can call `be_top` before the native function performs the virtual stack operation, at this time the virtual stack capacity is equal to the real parameter amount. For the `add` function, we need two parameters to participate in the operation, so check whether the number of parameters is 2 in the fourth line (`top == 2`). And we need to check whether the two parameters are both numeric types, so we need to call the `be_isnumber` function to check. If everything is correct, the parameters will be taken out of the virtual stack, then the calculation result will be pushed onto the stack, and finally returned using `be_return`. If the parameter verification fails, `be_return_nil` will be called to return the value of `nil`.

Next, register this native function to the built-in object table. For simplicity, we register it after loading the library:

```
bvm *vm = be_vm_new(); // Construct a VM
be_regfunc(vm, "myadd", l_add); // Register the native function "myadd"
```

The second line is where the native function is registered, and we named it `myadd`. At this point, the definition and registration of the native function is complete. As a verification, you can compile the interpreter, then enter the REPL and run some tests. You should get results like this:

```
> myadd
<function: 0x562a210f0f90>
> myadd(1.0, 2.5)
3.5
> myadd(2.5, 2)
4.5
> myadd(1, 2)
3
```

Types and Functions

Type

This section will introduce some types that need to be used in FFI. These types are generally used by FFI functions. Generally, the types and declarations in FFI can be found in the *berry.h* file. Unless otherwise specified in this section, the definition or declaration is provided in *berry.h* by default.

`bvm` Type is used to store the state information of the Berry virtual machine. Details of this type are not visible to external programs. Therefore, this definition can only be found in the *berry.h* file:

```
typedef struct bvm bvm;
```

Most FFI functions use the `bvm` type as the first parameter, because they all operate on the virtual machine internally. Hiding the internal implementation of `bvm` helps reduce the coupling between the FFI standard and the VM. Outside the interpreter, usually only `bvm` pointers are used. To create a new `bvm` object, use the `be_vm_new` function, and destroy the `bvm` object using the `be_vm_delete` function.

Native function type. The definition of this type is:

```
typedef int (*bntvfunc)(bvm*);
```

This type is a native function pointer, and some FFIs that register or add native functions to the virtual machine use parameters of this type. Variables or parameters of this type need to be initialized with a function name whose parameter is of type `bvm` and whose return value is of type `int`.

This type is used when registering native functions in batches or building native classes. It is defined as:

```
typedef struct {
    const char *name; // The name of the function or object
    bntvfunc function; // The function pointer
} bnfuncinfo;
```

The `name` member of `bnfuncinfo` represents the name of a function or object, and the `function` member is a native function pointer.

This type is a built-in integer type of Berry. It is defined in the *berry.h* document. By default, `bint` is implemented using the `long long` type, and the implementation of `bint` can be modified by modifying the configuration file.

This is Berry's built-in real number type, which is actually the floating point type in C language. `breal` is defined as:

```
#if BE_SINGLE_FLOAT != 0
    typedef float breal;
#else
    typedef double breal;
#endif
```

You can use the macro `BE_SINGLE_FLOAT` to control the specific implementation of `breal`: when the value of `BE_SINGLE_FLOAT` is `0`, the `double` type implementation `breal` will be used, otherwise the `float` type implementation `breal` will be used.

[section::errorcode]

This enumeration type is used in some FFI return values. The definition of this type is:

```
enum berrorcode {
    BE_OK = 0,
```

(continues on next page)

(continued from previous page)

```
    BE_IO_ERROR,
    BE_SYNTAX_ERROR,
    BE_EXEC_ERROR,
    BE_MALLOC_FAIL,
    BE_EXIT
};
```

The meaning of these enumeration values are:

- `BE_OK`: There is no error, the function is executed successfully.
- `BE_IO_ERROR`: A file reading error occurred when the interpreter was reading the source file. The error is generally caused by the absence of the file.
- `BE_SYNTAX_ERROR`: A syntax error occurred when the interpreter was compiling the source code. After this error occurs, the interpreter will not generate bytecode, so it cannot continue to execute bytecode.
- `BE_EXEC_ERROR`: Runtime error. When this error occurs, execution of Berry code is stopped and the environment is restored to the most recent recovery point.
- `BE_MALLOC_FAIL`: Memory allocation failed. This error is caused by insufficient heap space.
- `BE_EXIT`: Indicates that the program exits and the enumeration value is not an error. Executing Berry's `exit` function causes the interpreter to return this value.

It should be noted that when a `BE_MALLOC_FAIL` error occurs, dynamic memory allocation can no longer be performed, which means that string objects can no longer be allocated, so the function that returns this error usually does not give more detailed error information.

Functions and Macros

This function is used to create a new virtual machine instance. The function prototype is:

```
bvm* be_vm_new(void);
```

The return value of the function is a pointer to the virtual machine instance. `be_vm_new` The number is the first function called when the Berry interpreter is created. This function will do a lot of work: apply for memory for the virtual machine, initialize the state and attributes of the virtual machine, initialize the GC (garbage collector), and The standard library is loaded into the virtual machine instance, etc.

This function is used to destroy a virtual machine instance, the function prototype is:

```
void be_vm_delete(bvm *vm);
```

The parameter `vm` is the pointer of the virtual machine object to be destroyed. Destroying the virtual machine will release all the objects in the virtual machine, including the values in the stack and the objects managed by the GC. The virtual machine pointer after destruction will be an invalid value, and it can no longer be referenced.

This function is used to load a piece of source code from the buffer and compile it into bytecode. The prototype of the function is:

```
int be_loadbuffer(bvm *vm, const char *name, const char *buffer, size_t length);
```

The parameter `vm` is the virtual machine pointer. `name` is a string, which is usually used to mark the source of the source code. For example, the source code input from the standard input device can pass the string `"stdin"` to this parameter, and the source code input from the file can be The file name is passed to this parameter. `buffer` The parameter is the buffer for storing the source code. The organization of this buffer is very similar to the string of C. It is a continuous

sequence of characters, but the buffer pointed to by `buffer` does not require `'\0'` characters as Terminator. `length` The parameter indicates the length of the buffer. This length refers to the number of bytes of source code text in the buffer.

To give a simple example, if we want to use the `be_loadbuffer` function to compile a string, the general usage is:

```
const char *str = "print('Hello Berry')";
be_loadbuffer(vm, "string", str, strlen(str));
```

Here we use the string `"string"` to represent the source code, you can also modify it to any value. Note that the C standard library function `strlen` function is used here to get the length of the string buffer (actually the number of bytes in the string).

If the compilation is successful, `be_loadbuffer` will compile the source code into a Berry function and place it on the top of the virtual stack. If the compilation encounters an error, `be_loadbuffer` will return an error value of type `berrorcode` (Section errorcode, and if possible, will store the specific error message string at the top of the virtual stack.

`be_loadstring` is a macro defined as:

```
#define be_loadstring(vm, str) be_loadbuffer((vm), "string", (str), strlen(str))
```

This macro is just a simple wrapper for the `be_loadbuffer` function. `vm` The parameter is a pointer to the virtual machine instance, and the `str` parameter is a pointer to the source code string. It is very convenient to use `be_loadstring` to compile strings, for example:

```
be_loadstring(vm, "print('Hello Berry')");
```

This way of writing is more concise than using `be_loadbuffer`, but you must ensure that the string ends with a `'\0'` character.

This function is used to compile a source code file. The function prototype is:

```
int be_loadfile(bvm *vm, const char *name);
```

The function of this function is similar to the `be_loadbuffer` function, except that the function will be compiled by reading the source code file. The parameter `vm` is the pointer of the virtual machine instance, and the parameter `name` is the file name of the source file. This function will call the file interface, and by default it will use functions such as `fopen` in the C standard library to manipulate files.

If you use the file interface of the C standard library, you can use relative path or absolute path file names. If the file does not exist, `be_loadfile` will return a `BE_IO_ERROR` error (section errorcode) and push the error message onto the top of the stack. Other error messages are the same as the `be_loadbuffer` function. It is recommended to use the `be_loadfile` function to compile the source file, instead of reading all the source files into a buffer, and then call the `be_loadbuffer` function to compile the source code. The former will read the source file in segments and only create a small read buffer in the memory, thus saving more memory.

This function returns the absolute index value of the top element in the virtual stack. This value is also the number of elements in the virtual stack (the capacity of the virtual stack). Call this function before adding or subtracting elements in the virtual stack to get the number of parameters of the native function. The prototype of this function is:

```
int be_top(bvm *vm);
```

This function is usually used to obtain the number of parameters of a native function. When used for this purpose, it is recommended to call `be_top` at the top of the native function body. E.g:

```
static int native_function_example(bvm *vm)
{
    int argc = be_top(vm); // Get the number of arguments
    // ...
}
```

This function converts the type of the Berry object into a string and returns it. For example, it returns "`int`" for an integer object, and "`function`" for a function object. The prototype of this function is:

```
const char* be_typename(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance, and `index` is the index of the object to be operated. The `type` function in the Berry standard library is implemented by calling `be_typename`. Please refer to section baselib_type for the return string corresponding to the parameter type.

This function is used to get the class name of an object or class. The function prototype is:

```
const char* be_classname(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance, and `index` is the index of the object to be operated. If the value at `index` is an instance, the `be_classname` function will return the class name string to which the instance belongs, and if the value at `index` is a class, it will directly return the class name string. In other cases `be_classname` will return `NULL`.

This function returns the length of the specified Berry string. The function prototype is:

```
int be_strlen(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance, and `index` is the index of the object to be operated. This function returns the number of bytes in the string at `index` (the `'\0'` characters at the end of the Berry string are not counted). If the value of the `index` position is not a string, the `be_strlen` function will return `0`.

Although the `Berry` string is compatible with the C string format, it is not recommended to use the `strlen` function of the C standard library to measure the length of the Berry string. For Berry strings, `be_strlen` is faster than `strlen` and has better compatibility.

This function is used to splice two Berry strings. The function prototype is:

```
void be_strconcat(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance. This function will concatenate the string at the parameter position of `index` with the string at the top position of the stack, and then put the resulting string into the position indexed by `index`.

This function pops the value at the top of the stack. The function prototype is:

```
void be_pop(bvm *vm, int n);
```

The parameter `vm` is the pointer of the virtual machine instance, and the parameter `n` is the number of values to be popped from the stack. Note that the value of `n` cannot exceed the capacity of the stack.

This function will remove a value from the stack. The function prototype is:

```
void be_remove(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance, and the parameter `index` is the index of the object to be removed. After the value at `index` is moved out, the following values will be filled forward, and the stack capacity will be reduced by one. The value of `index` cannot exceed the capacity of the stack.

This function returns the absolute index value of a given index value, and its function prototype is:

```
int be_absindex(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance, and the parameter `index` is the input index value. If `index` is positive, the return value of `be_absindex` is the value of `index`. If `index` is negative, the return value of texttttbe_absindex is the absolute index value corresponding to `index`. When `index` is a negative value (relative index), its index position cannot be lower than the bottom of the stack.

This function creates a new `list` value, and its function prototype is:

```
void be_newlist(bvm *vm);
```

The parameter `vm` is the pointer of the virtual machine instance. After this function is successfully called, the new `list` value will be pushed onto the top of the stack. `list` value is an internal representation of a list, not to be confused with an instance of the `list` class.

This function creates a new `map` value, and its function prototype is:

```
void be_newmap(bvm *vm);
```

The parameter `vm` is the pointer of the virtual machine instance. After this function is successfully called, the new `map` value will be pushed onto the top of the stack. `map` value is an internal representation of a list, not to be confused with an instance of the `map` class.

This function pushes the global variable with the specified name onto the stack. Its function prototype is:

```
void be_getglobal(bvm *vm, const char *name);
```

The parameter `vm` is the pointer of the virtual machine instance, and the parameter `name` is the name of the global variable. After this function is called, the global variable named `name` will be pushed onto the top of the virtual stack.

This function is used to set the value of the member variable of the instance object class. The function prototype is:

```
void be_setmember(bvm *vm, int index, const char *k);
```

The parameter `vm` is the pointer of the virtual machine instance, the parameter `index` is the index of the instance object, and the parameter `k` is the name of the member. This function will copy the value at the top of the stack to the member `k` of the index position instance. Note that the top element of the stack will not pop up automatically.

This function is used to get the value of the member variable of the instance object class. The function prototype is:

```
void be_getmember(bvm *vm, int index, const char *k);
```

The parameter `vm` is the pointer of the virtual machine instance, the parameter `index` is the index of the instance object, and the parameter `k` is the name of the member. This function pushes the value of the member of the index position instance `k` onto the top of the virtual stack.

This function is used to get the value of `list` or `map`. The function prototype is:

```
void be_getindex(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance, and the parameter `index` is the index of the object to be operated. This function is used to get an element from the `map` or `list` container (internal values, not instances of `map` or `list` classes), and the index of the element is stored at the top of the stack (relative index is 1). After calling this function, the value obtained from the container will be pushed onto the top of the stack. If there is no subscript pointed to by the container, the value of `nil` will be pushed onto the top of the stack. For example, if the element with index 1 in the virtual stack is a `list`, and we want to extract the element with index 0 from it, then we can use the following code:

```
be_pushint(vm, 0); // Push the index value 0 onto the virtual-stack
be_getindex(vm, 1); // Get an element from the list container
```

We first push the integer value `0` onto the stack, and this value will be used as the index to get the element from the `list` container. The second line of code implements to get elements from the `list` container. The index value of the `list` container in the example is 1 in the virtual stack. The retrieved element is stored at the top of the stack, and we can use the relative index 1 to access it.

This function is used to set a value in `list` or `map`. The function prototype is:

```
void be_setindex(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance, and the parameter `index` is the subscript of the object to be operated. This function is used to write an element of the `map` or `list` container. The index of the value to be written in the virtual stack is 1, and the index of the subscript of the write position in the virtual stack is 2. If the element with the specified subscript does not exist in the container, the write operation will fail.

Assuming that the position with index 1 in the virtual stack has a value of `map`, and it has an element with a subscript of `"test"`, an example of setting the element at the subscript of `"test"` to `100` is:

```
be_pushstring(vm, "test"); // Push the index "index"
be_pushint(vm, 100); // Push the value 100
be_setindex(vm, 1); // Set the key-value pair to map["test"] = 100
```

We must first push the subscript and the value to be written on the stack in order. For `map`, it is a key-value pair. In the example, the first two lines of code complete these tasks. The third line calls the `be_setindex` function to write the value into the `map` object.

This function is used to read an Up Value of the native closure. The function prototype is:

```
void be_getupval(bvm *vm, int index, int pos);
```

The parameter `vm` is the pointer of the virtual machine instance; `index` is the native closure index value of the Up Value to be read; `pos` is the position of the Up Value in the native closure Up Value table (numbering starts from 0). The read Up Value will be pushed onto the top of the virtual stack.

This function is used to set an Up Value of the native closure. The function prototype is:

```
void be_setupval(bvm *vm, int index, int pos);
```

The parameter `vm` is the pointer of the virtual machine instance; `index` is the native closure index value to be written into the Up Value; `pos` is the position of the Up Value in the native closure Up Value table (numbering starts from 0). This function obtains a value from the top of the virtual stack and writes it to the target Up Value. After the operation is completed, the top value of the stack will not be popped from the stack.

This function is used to get the parent object of the base class or instance of the class. The function prototype is:

```
void be_getsuper(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance; `index` is the class or object to be operated. If the value at `index` is a class with a base class, the function will push its base class onto the top of the stack; if the value at `index` is an object with a parent object, the function will take its parent The object is pushed onto the top of the stack; otherwise, a value of `nil` is pushed onto the top of the stack.

This function is used to get the number of elements contained in the container. The function prototype is:

```
int be_data_size(bvm *vm, int index);
```

The parameter vm is the pointer of the virtual machine instance; index is the index of the container object to be operated. If the value at index is a Map value or List value, the function returns the number of elements contained in the container, otherwise it returns -1.

This function is used to append a new element to the end of the container. The function prototype is:

```
void be_data_push(bvm *vm, int index);
```

The parameter vm is the pointer of the virtual machine instance; index is the index of the container object to be operated. The object at index must be a List value. This function gets a value from the top of the stack and appends it to the end of the container. After the operation is completed, the value at the top of the stack will not be popped from the stack.

This function is used to insert a pair of elements into the container. The function prototype is:

```
void be_data_insert(bvm *vm, int index);
```

The parameter vm is the pointer of the virtual machine instance; index is the index of the container object to be operated. The object at index must be a List value or a Map value. The inserted element forms a pair of key-value pairs. The value is stored at the top of the stack, and the key is stored at the previous index on the top of the stack. It should be noted that the key inserted into the Map container cannot be a nil value, and the key inserted into the List container must be an integer value. If the operation is successful, the function will return bture, otherwise it will return bfalse.

This function is used to remove an element in the container. The function prototype is:

```
void be_data_remove(bvm *vm, int index);
```

The parameter vm is the pointer of the virtual machine instance; index is the index of the container object to be operated. The object at index must be a List value or Map value. For the Map container, the key to delete the element is stored on the top of the virtual stack (need to be pressed before the function call); for the List container, the index of the element to be deleted is stored on the top of the virtual stack (need to be before the function call) push into). If the operation is successful, the function will return bture, otherwise it will return bfalse.

This function is used to reset the capacity of the container. The function prototype is:

```
void be_data_resize(bvm *vm, int index);
```

The parameter vm is the pointer of the virtual machine instance; index is the index of the container object to be operated. This function is only available for List containers, and the new capacity is stored on the top of the virtual stack (must be an integer).

This function is used to get the next element of the iterator. The function prototype is:

```
int be_iter_next(bvm *vm, int index);
```

The parameter vm is the pointer of the virtual machine instance; index is the index of the iterator to be operated. The iterator object may be an iterator of a List container or a Map container. For the List iterator, this function pushes the iteration result value onto the top of the stack, while for the Map iterator, it pushes the key value into the previous position and the top of the stack respectively. Calling this function will update the iterator. If the function returns 0, the call fails, returning 1 to indicate that the current iterator is a List iterator, and returning 2 to indicate that the current iterator is a Map iterator.

This function is used to test whether there is another element in the iterator. The function prototype is:

```
int map_hasnext(bvm *vm, int index)
```

The parameter `vm` is the pointer of the virtual machine instance; `index` is the index of the iterator to be operated. The iterator object may be an iterator of a List container or a Map container. If there are more iterable elements in the iterator, return `1`, otherwise return `0`.

This function is used to test whether there is a reference to the specified object in the reference stack. It must be used in conjunction with `be_refpush` and `be_refpop`. This API can avoid recursion when traversing objects that have their own references. The function prototype is:

```
int be_refcontains(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance; `index` is the index of the object to be operated. This function is used for the value of an instance type. If there is a reference to the object in the reference stack, it returns `1`, otherwise it returns `0`.

Push the reference of the specified object onto the reference stack. The function prototype is:

```
int be_refpush(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance; `index` is the index of the object to be operated. This function is used for the value of an instance type.

Pop the object at the top of the reference stack. The function prototype is:

```
int be_refpop(bvm *vm);
```

The parameter `vm` is the pointer of the virtual machine instance. This function is used in pairs with `be_refpush`. The following is the use of the reference stack API to avoid the problem of infinite recursive traversal when the object itself is referenced:

```
int list_traversal(bvm *vm)
{
    // ...
    if (be_refcontains(vm, 1)) {
        be_return(vm);
    }
    be_refpush(vm, 1);
    // Traversing the container, may call list_traversal recursively.
    be_refpop(vm);
    be_return(vm);
}
```

This is a simplified traversal process of the List container. For the complete code, please refer to the source code of the function `m_tostring` in *be_listlib.c*. We assume that the index of the List object is `1`. First, we check whether the List already exists in the reference stack (line 4), and if the reference already exists, return directly, otherwise proceed with subsequent processing. To make `be_refcontains` usable, we need to use `be_refpush` and `be_refpop` to process the reference stack before and after the actual traversal operation (lines 7 and 9).

This function tests the amount of free space on the stack and expands the stack space if it is insufficient. The function prototype is:

```
void be_stack_require(bvm *vm, int count);
```

The parameter `vm` is the pointer of the virtual machine instance; `count` is the required free stack capacity. If the free capacity of the virtual stack allocated by the VM to the native function is lower than this value, an expansion operation will be performed.

This function returns whether the value indexed by the parameter `index` in the virtual stack is `nil`, if it is, it returns `1`, otherwise it returns `0`. The prototype of this function is:

```
int be_isnil(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance, and `index` is the index of the value to be measured.

This function returns whether the value indexed by the parameter `index` in the virtual stack is of type `bool`, if it is, the function returns `1`, otherwise it returns `0`. The prototype of this function is:

```
int be_isbool(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance, and `index` is the index of the value to be measured.

This function returns whether the value indexed by the parameter `index` in the virtual stack is an integer type, if it is, it returns `1`, otherwise it returns `0`. The prototype of this function is:

```
int be_isint(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance, and `index` is the index of the value to be measured.

This function returns whether the value indexed by the parameter `index` in the virtual stack is a real number type, if it is, it returns `1`, otherwise it returns `0`. The prototype of this function is:

```
int be_isreal(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance, and `index` is the index of the value to be measured.

This function returns whether the value indexed by the parameter `index` in the virtual stack is an integer or a real number type, if it is, it returns `1`, otherwise it returns `0`. The prototype of this function is:

```
int be_isnumber(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance, and `index` is the index of the value to be measured.

This function returns whether the value indexed by the parameter `index` in the virtual stack is a string type, if it is, it returns `1`, otherwise it returns `0`. The prototype of this function is:

```
int be_isstring(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance, and `index` is the index of the value to be measured.

This function returns whether the value indexed by the parameter `index` in the virtual stack is a closure type, if it is, it returns `1`, otherwise it returns `0`. The prototype of this function is:

```
int be_isclosure(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance, and `index` is the index of the value to be measured.

This function returns whether the value indexed by the parameter `index` in the virtual stack is a primitive closure type, if it is, it returns `1`, otherwise it returns `0`. The prototype of this function is:

```
int be_isntvclos(bvm *vm, int index);
```

The parameter `vm` is the pointer of the virtual machine instance, and `index` is the index of the value to be measured.

This function returns whether the value indexed by the parameter `index` in the virtual stack is a function type, if it is, it returns `1`, otherwise it returns `0`. The prototype of this function is:

```
int be_isfunction(bvm *vm, int index);
```

The parameter vm is the pointer of the virtual machine instance, and index is the index of the value to be measured. There are three types of functions: closure, native function and native closure.

This function returns whether the value indexed by the parameter index in the virtual stack is of type proto, if it is, it returns 1, otherwise it returns 0. The prototype of this function is:

```
int be_isproto(bvm *vm, int index);
```

The parameter vm is the pointer of the virtual machine instance, and index is the index of the value to be measured. proto The type is the function prototype of the Berry closure.

This function returns whether the value indexed by the parameter index in the virtual stack is of type class, if it is, it returns 1, otherwise it returns 0. The prototype of this function is:

```
int be_isclass(bvm *vm, int index);
```

The parameter vm is the pointer of the virtual machine instance, and index is the index of the value to be measured.

This function returns whether the value indexed by the parameter index in the virtual stack is of type instance, if it is, it returns 1, otherwise it returns 0. The prototype of this function is:

```
int be_isinstance(bvm *vm, int index);
```

The parameter vm is the pointer of the virtual machine instance, and index is the index of the value to be measured.

This function returns whether the value indexed by the parameter index in the virtual stack is an instance or sub-instance of class bytes, if it is, it returns 1, otherwise it returns 0. The prototype of this function is:

```
int be_isbytes(bvm *vm, int index);
```

This function returns whether the value indexed by the parameter index in the virtual stack is of type list, if it is, it returns 1, otherwise it returns 0. The prototype of this function is:

```
int be_islist(bvm *vm, int index);
```

The parameter vm is the pointer of the virtual machine instance, and index is the index of the value to be measured.

This function returns whether the value indexed by the parameter index in the virtual stack is of type map, if it is, it returns 1, otherwise it returns 0. The prototype of this function is:

```
int be_ismap(bvm *vm, int index);
```

The parameter vm is the pointer of the virtual machine instance, and index is the index of the value to be measured.

This function returns whether the value indexed by the parameter index in the virtual stack is a universal pointer type, if it is, it returns 1, otherwise it returns 0. The prototype of this function is:

```
int be_iscomptr(bvm *vm, int index);
```

The parameter vm is the pointer of the virtual machine instance, and index is the index of the value to be measured.

```
bint be_toint(bvm *vm, int index);
```

Get the value of the index position of index from the virtual stack and return it as an integer type. This function does not check the correctness of the type. If the value is an instance, the method toint() is called if it exists.

```
breal be_toreal(bvm *vm, int index);
```

Get the value of the index position of `index` from the virtual stack and return it as a floating-point number type. This function does not check the correctness of the type.

```
bint be_toindex(bvm *vm, int index);
```

Get the value of the index position of `index` from the virtual stack and return it as an integer type. This function does not check the correctness of the type. Unlike `be_toint`, the return value type of `be_toindex` is `int`, while the return value of the former is `bint`.

```
bbool be_tobool(bvm *vm, int index);
```

Get the value of the index position of `index` from the virtual stack and return it as a Boolean type. If the indexed value is not of Boolean type, it will be converted according to the rules in section type_bool, and the conversion process will not cause the indexed value to change. If the value is an instance, the method `tobool()` is called if it exists.

```
const char* be_tostring(bvm *vm, int index);
```

Get the value of the index position of `index` from the virtual stack and return it as a string type. If the indexed value is not a string type, the indexed value will be converted to a string, and the conversion process will replace the value at the indexed position in the virtual stack with the converted string. The string returned by this function always ends with `'\0'` characters. If the value is an instance, the method `tostring()` is called if it exists.

```
void* be_tocomptr(bvm* vm, int index);
```

Get the value of the index position of `index` from the virtual stack and return it as a general pointer type. This function does not check the correctness of the type.

```
const void* be_tobytes(bvm *vm, int index, size_t *len);
```

Get the value of the index position of `index` from the virtual stack and return it as a bytes buffer. The pointer of the buffer is returned, and the size is stored in `*len` (unless `len` is NULL). This function works only for instances of the `bytes` class, or returns `NULL`.

```
void be_pushnil(bvm *vm);
```

Push a `nil` value onto the virtual stack.

```
void be_pushbool(bvm *vm, int b);
```

Push a Boolean value onto the virtual stack. The parameter `b` is the boolean value to be pushed onto the stack. When the value is `0`, it means false, otherwise it is true.

```
void be_pushint(bvm *vm, bint i);
```

Push an integer value `i` onto the virtual stack.

```
void be_pushreal(bvm *vm, breal r);
```

Push a floating point value `r` onto the virtual stack.

```
void be_pushstring(bvm *vm, const char *str)
```

Push the string `str` onto the virtual stack. The parameter `str` must point to a C string that ends with a null character `'\0'`, and a null pointer cannot be passed in.

```
void be_pushnstring(bvm *vm, const char *str, size_t n);
```

Push the string `str` of length `n` onto the virtual stack. The length of the string is subject to the parameter `n`, and the null character is not used as the end mark of the string.

```
const char* be_pushfstring(bvm *vm, const char *format, ...);
```

Push the formatted string into the virtual stack. The parameter `format` is a formatted string, which contains the text to be pushed onto the stack, and the `format` parameter contains a label, which can be replaced by the value specified by the subsequent additional parameter and formatted as required. According to the label of the `format` string, a series of additional parameters may be required, and each additional parameter will replace the corresponding `%` label in the `format` parameter.

specifier	Description	
d	Format as decimal signed integer (positive numbers do not output sign)	
f	Single or double precision floating point number formatted as a decimal	
g	Single or double precision floating point number formatted as exponential	
s	Format as string	
c	Format as a single character	
p	Format as pointer address	
%	Escaped as % Character (no parameter)	

`format` Label parameter description

`be_pushfstring` The function is similar to the standard function of C `printf`, but the function of formatting strings is relatively basic and does not support operations such as customizing the width and decimal places. A typical example is:

```
be_pushfstring(vm, "%s: %d", "hello", 12); // Good, it works!
be_pushfstring(vm, "%s: %.5d", "hello", 12); // Error, the specified width is not␣
↪supported.
```

This means that `be_pushfstring` can only perform simple formatting operations. If the requirements cannot be met, it is recommended to use `sprintf` formatted strings for operations.

```
void be_pushvalue(bvm *vm, int index);
```

Push the value with index `index` onto the top of the virtual stack.

```
void be_pushntvclosure(bvm *vm, bntvfunc f, int nupvals);
```

Push a native closure onto the top of the virtual stack. The parameter `f` is the C function pointer of the native closure, and `nupvals` is the upvalue number of the closure.

```
void be_pushntvfunction(bvm *vm, bntvfunc f);
```

Push a native function onto the top of the virtual stack, and the parameter `f` is the native function pointer.

```
void be_pushclass(bvm *vm, const char *name, const bnfuncinfo *lib);
```

Push a native class onto the top of the virtual stack. The parameter `name` is the name of the native class, and the parameter `lib` is the attribute description of the native class.

```
void be_pushcomptr(bvm *vm, void *ptr);
```

Push a general pointer onto the top of the virtual stack. The general pointer `ptr` points to a certain C data area. Since the content pointed to by this pointer is not maintained by Berry's garbage collector, users have to maintain the life cycle of the data themselves.

```
void* be_pushbytes(bvm *vm, const void *buf, size_t len);
```

Push a `bytes()` buffer starting at position `buf` and of size `len`. The buffer is copied into Berry allocated memory, you don't need to keep the buffer valid after this call.

```
bbool be_pushiter(bvm *vm, int index);
```

Push an iterator onto the top of the virtual stack.

Push an error message onto the top of the stack. After executing the FFI, the interpreter will directly return to the position that can handle the error, and the code immediately following will not be executed. The function prototype is:

```
void be_pusherror(bvm *vm, const char *msg);
```

The parameter `vm` is the pointer of the virtual machine instance; `msg` is the string containing the error information.

Move the value at the `from` index to the `to` index position. This function does not delete the value of `from` index position, only modifies the value of `to` index position.

Compile-time construction technology

The compile-time construction technology is mainly implemented by *coc* which is located in the *coc/coc* path of the interpreter source code directory. *coc* Tool is used to generate constant strings, and constant objects as C code, and will be compiled into constants when the interpreter is compiled. In principle, the *coc* tool will generate code from the declaration information of the constant object (in accordance with a specific format). The process will automatically calculate the Hash value and generate the Hash table.

Makefile in the root directory of the interpreter project will automatically compile this tool and run the tool before compiling the interpreter source code. The content of *Makefile* ensures that when using the `make` command, the code for constructing the object at compile time will always be updated through the tool (if it needs to be updated). The code for constructing objects at compile time can be manually generated through the `make prebuild` command, which is stored in the *generate* folder.

The compile-time construction can be turned on or off by modifying the `BE_USE_PRECOMPILED_OBJECT` macro. In any case, the tool *coc* is called to generate constant object codes (the codes are not used when compile-time construction is turned off).

Use command

`coc` Tool is used to generate code for constant objects. The format of the command is

```
tools/coc/coc -o <dst_path> <src_path(s)> -c <include_path>
```

The output path *dst_path* is used to store the generated code, and the source path *src_path* is a list of paths that need to be scanned for the source code (use spaces to separate multiple paths). `include_path` contains a C header file scanned to detect compilation directives. `coc` tries to compile only the necessary constants. Since *generate* is used as the generated code path in the source code of the interpreter, *dst_path* must be *generate*. Taking the standard interpreter project as an example, the command to use the tool in `map_build` should be

```
tools/coc/coc -o generate default src -c default/berry_conf.h
```

The meaning of this command is: the output path is *generate*, and the source path is *src* and *default*.

output path

Strictly speaking, the *generate* folder used as the output path cannot be placed anywhere, it must be stored in a parent directory containing the path. The include path refers to the path where the header file will be searched in the project. Taking the standard interpreter source code as an example, the include path is *src* and *default*. Therefore, in the standard interpreter project, the *generate* folder is stored in the root directory of the interpreter source code (the parent directory of *src* and *default*).

The reason for the above rules is that the following codes are used in the interpreter source code to refer to constant objects:

```
#include "../generate/be_fixed_xxx.h"
```

If readers want to define constant objects themselves, they also need to use such code to include the corresponding header files. This section will introduce how to use these tools to define and use constant objects.

Compile-time string table

The compile-time string table is used to store constant strings. Constant strings are objects that are transparent to the script. They are not created or destroyed when the interpreter is running, but are always stored as constants in the data segment of the interpreter program. If you need to use a string as a constant string, you can add the prefix `be_const_str_` in front of the string in the interpreter source code, and the declaration can be placed anywhere in the source file (including comments). For example, to create a constant string with the content `"string"`, you need to declare the symbol `be_const_str_string` in the source file, and this symbol is also the variable name that refers to the constant string in the C code.

All keywords will create constant strings. If you modify the keyword-related code in the Berry interpreter, the corresponding code in *coc* must also be modified.

If the string contains special symbols, they are automatically escaped as `_XHH` where `HH` is the hex representation (uppercase) of the character. For example `"` is represented by `_X3A`. This representation is bihective so it's easy to convert to and from the original string.

Use constant string

Normally, there is no need to manually declare constant strings, nor to use them manually. If you really need to call the constant string manually, include the header file *be_constobj.h* to use all constant string variables (this header file has declarations for all constant strings). The typical use of constant strings is to construct objects at compile time. The declaration and definition of constant strings in this process are automatically handled by the tool.

In any case, the FFI function `be_pushstring` should be used directly to create a string. When a string has a constant string, it will not repeatedly create a new string object, but directly use the corresponding constant string.

By default, all strings used are referenced in a global `m_const_string_table` hashtable. However, some projects may have many compilation variants for which some sets of string are not needed. If all string constants are stored in all variants, this creates a waste of flash size. For this reason, some strings can be declared as `weak` strings in the sense of having a `weak` reference. In such case the string constant is declared in C code, but not included in the global map object. This means that the linker can choose to not include the string constants if it is not referenced by any code. The

con is that if you dynamically create a string object with the same value, a new object is created in memory (while it would not for a regular string constant). To indicate weak strings, use the `strings: weak` modifier (see below).

Construct object at compile time

Objects constructed at compile time are also called constant objects. The data structure of these objects is constructed when the interpreter is compiled and cannot be modified at runtime. `map_build` A set of declaration rules is defined in the tool to generate C code for constant objects. The declaration information of the constant object is directly stored in the source file (*.c). In order to distinguish it from other content, a complete declaration information should be included in the following boot code:

```
@const_object_info_begin
@const_object_info_end
```

The constant object declaration information does not conform to the C language syntax, so they should be placed in a multi-line comment (included with `/* */`). All constant objects have the same declaration form. The declaration structure of a constant object is called "object declaration block", which is composed of

```
type object_name (attributes) {
    member_fields
}
```

`type` is the type of constant object, it can be `map`, `class`, `module` or `vartab`. `object_name` is the variable name of the constant object in C language. `attributes` is the attribute list of constant objects. An attribute is composed of attribute name and attribute value. The attribute name and attribute value are separated by semicolons, and multiple attributes are separated by commas. For example, the attribute list `scope: global, name: map` means that the `scope` attribute of a constant object is `global`, and the `name` attribute is `map`. Also `strings: weak` indicate to generate weak string constants for the names of member fields or any string constant. `member_fields` is the list of member domains of constant objects. A member is composed of name and value, separated by commas. Each line can declare one member, and multiple members must be declared on multiple lines.

The **coc** tool uses regular expressions to parse the object declaration block. In the parsing process, the entire object declaration block will be matched first, and the information `type` and `object_name` will be matched. For the information of `attributes` and `member_fields`, further Analysis. In order to facilitate the implementation, the tool does not have strict requirements on the syntax of the object declaration block, and lacks a complete error handling mechanism, so you should ensure that the syntax is correct when writing the object declaration block.

In order to facilitate understanding, we illustrate with a simple constant class:

```
/* @const_object_info_begin
class be_class_map (scope: global, name: map) {
    .data, var
    init, func(m_init)
    tostring, func(m_tostring)
}
@const_object_info_end */
#include "../generate/be_fixed_be_class_map.h"
```

In this example, the declaration information of the entire constant class is in the C language comment, so it will not affect the compilation of the C code. The object declaration block is placed between `@const_object_info_begin` and `@const_object_info_end` to ensure that the **coc** tool detects the object declaration block.

Since it is a constant class declaration, the value of *type* in the object declaration block is `class`, and `be_class_map` is the variable name of the constant object in the C code. Two attributes are declared in the attribute list of the object (the part enclosed in parentheses), and the meaning of these attributes will be introduced in the "Compile-Time Construction

Class" section of this section. Three members are defined in the member list surrounded by curly braces, multiple members are separated by newlines, and the name of the member and the value of the member are separated by a comma.There are several legal formats for member names:

- Berry variable name format: start with a letter or underscore, followed by several letters, underscores or numbers.
- Use "`.`" as the first character, followed by letters, underscores or numbers.
- Overloadable operators, such as "+", "-" and "<<" etc.

The value of a member can be of the following types:

- `var`: This symbol will be compiled into an integer object (`be_const_var`), and the value of the integer object is automatically incremented from `0`. `var` It is designed for the declaration of member variables in the class, and its automatic numbering feature is used to realize the serial number of member variables.
- `func(symbol)`: Declare native member functions or methods of constant objects. The symbol will be compiled into a native function value (`be_const_func`), `symbol` is the native function pointer corresponding to the member. `m_init` and `m_tostring` in the example are two native functions.
- `closure(symbol)`: Declare pre-compiled bytecode member functions or methods of constant objects. The symbol will be compiled into a native function value (`be_const_closure`), `symbol` is the name of the solidified function. See `module solidify` to know how to solidify objects.
- `nil()`: This symbol will be compiled into an nil value (`be_const_nil`).
- `int(value)`: This symbol will be compiled into an integer object (`be_const_int`), the value of the integer object is `value`.
- `real(value)`: This symbol will be compiled into a real number object (`be_const_real`), the value of the real number object is `value`.
- `comptr(value)`: This symbol will be compiled into a pointer object (`be_const_comptr`), the value of the pointer is `value` and can be used to pass the address of a C global structure.
- `class(symbol)`: This symbol will be compiled into a class object (`be_const_class`). `symbol` is a pointer to this type of object, and the pointer needs to point to a constant type object.
- `module(symbol)`: This symbol will be compiled into a module object (`be_const_module`). `symbol` is a pointer to the module object, and the pointer needs to point to a constant module object.
- `ctype_func(symbol)`: This symbol will be compiled into a native function (`be_const_ctype_func`). `symbol` is a pointer to the C mapping definition. This feature is used by berry_mapping

In order to use the `be_class_map` object, we need to include the corresponding header file in the C code to ensure that the object will be compiled. The usual practice is to include the corresponding header file near the object declaration block. In the example, line 8 contains it. The corresponding header file can be used to construct `be_class_map` objects at compile time.

After processing by the **coc** tool , each object declaration block will be compiled into a header file named *be_fixed_be_xxx.h*, and *xxx* is the C variable name of the object. In order to compile constant objects in C code, we need to include the corresponding header files. It is usually recommended to include the corresponding header files near the object declaration block. The 8th line in the example contains *be_fixed_be_class_map.h* to construct the `be_class_map` object at compile time.

Construct Map at Compile Time

Maps constructed at compile-time are also constant `map` objects. They are generally not declared directly using object declaration blocks, but are declared in other compile-time construction structures. When constructing the constant `map`, the constant object type information should be `map`, which supports a `scope` attribute. When the `scope` attribute value is `local`, the constant object is `static`, the attribute When it is `global`, it is `extern`, and the value of this attribute is `local` by default. The constant `map` object's `member_fields` supports common member name/member value specifications, and member values are only stored as data without special interpretation. The following is an example of using the object declaration block to directly declare a constant `map` object:

```
map map_name (scope: local/global) {
    init, func(m_init)
}
```

Compile-time construction class

To construct a class at compile time, use the object declaration block to declare, and the type information of the object is `class`. The declared properties of this object are `scope` and `name`. `scope` The scope of the C variable of the attribute declaration object, when the value is `local` (default), the scope is `static`, when it is `global`, the scope is `extern`; `name` The value of the attribute is that class Name, anonymous class can omit this parameter. Since the attribute list of a class only stores methods and member variable indexes, the `member_fields` of the class constructed at compile time can only use the values `var` and `func()`. A simple compile-time construction class declaration block is:

```
class be_class_map (scope: global, name: map) {
    .data, var
    init, func(m_init)
    tostring, func(m_tostring)
}
```

Building Module at Compile Time

The type information of the building block declaration block at compile time is `module`.

```
module math (scope: global) {
    sin, func(m_sin)
    cos, func(m_cos)
    pi, real(M_PI)
}
```

Construct Built-in Domain at Compile Time

```
vartab m_builtin (scope: local) {
    assert, func(l_assert)
    print, func(l_print)
    list, class(be_class_list)
}
```

Grammar Definition

This chapter will give some grammar definitions related to Berry. We use **Extended Backus Normal Form** (EBNF) to define or express grammar. We did not use strict EBNF grammar to define, but made a lot of simplifications, but these simplifications will not affect readers' understanding of the grammar.

The EBNF definition of Berry language grammar is as follows:

```
(* program define *)
program = block;

(* block define *)
block = {statement};

(* statement define *)
statement = class_stmt | func_stmt | var_stmt | if_stmt | while_stmt |
         for_stmt | break_stmt | return_stmt | expr_stmt | import_stmt |
         try_stmt | throw_stmt | ';';
if_stmt = 'if' expr block {'elif' expr block} ['else' block] 'end';
while_stmt = 'while' expr block 'end';
for_stmt = 'for' ID ':' expr block 'end';
break_stmt = 'break' | 'continue';
return_stmt = 'return' [expr];

(* function define statement *)
func_stmt = 'def' ID func_body;
func_body = '(' [arg_field {',' arg_field}] ')' block 'end';
arg_field = ['*'] ID;

(* class define statement *)
class_stmt = 'class' ID [':' ID] class_block 'end';
class_block = {'var' ID {',' ID} | 'static' ['var'] ID ['=' expr] {',' ID ['=' expr] } |
↪'static' func_stmt | func_stmt};
import_stmt = 'import' (ID (['as' ID] | {',' ID}) | STRING 'as' ID);

(* exceptional handling statement *)
try_stmt = 'try' block except_block {except_block} 'end';
except_block = except_stmt block;
except_stmt = 'except' (expr {',' expr} | '..') ['as' ID [',' ID]];
throw_stmt = 'raise' expr [',' expr];

(* variable define statement *)
var_stmt = 'var' ID ['=' expr] {',' ID ['=' expr]};

(* expression define *)
expr_stmt = expr [assign_op expr];
expr = suffix_expr | unop expr | expr binop expr | range_expr | cond_expr;
cond_expr = expr '?' expr ':' expr; (* conditional expression *)
assign_op = '=' | '+=' | '-=' | '*=' | '/=' |
            '%=' | '&=' | '|=' | '^=' | '<<=' | '>>=';
binop = '<' | '<=' | '==' | '!=' | '>' | '>=' | '||' | '&&' |
        '<<' | '>>' | '&' | '|' | '^' | '+' | '-' | '*' | '/' | '%';
range_expr = expr '..' [expr]
```

(continues on next page)

(continued from previous page)

```
unop = '-' | '!' | '~';
suffix_expr = primary_expr {call_expr | ('.' ID) | '[' expr ']'};
primary_expr = '(' expr ')' | simple_expr | list_expr | map_expr | anon_func | lambda_
↪expr;
simple_expr =  INTEGER | REAL | STRING | ID | 'true' | 'false' | 'nil';
call_expr = '(' [expr {',' expr}] ')';
list_expr = '[' {expr ','} [expr] ']';
map_expr = '{' {expr ':' expr ','} [expr ':' expr] '}';
anon_func = 'def' func_body;

(* anonymous function *)
lambda_expr = '/' [arg_field {',' arg_field}] | {arg_field}] '->' expr;
```

The standard EBNF format can be found in related materials. Here is an explanation of the details that need attention when reading the above grammar. The symbols that have appeared to the left of the equal sign are non-terminal symbols, and the others are terminal symbols. The terminator enclosed in quotation marks ' is a fixed string, which is usually a language keyword or operator. There are several terminators that are inconvenient to describe directly in EBNF: `INTEGER` represents the integer face value; `REAL` represents the real number face value; `STRING` represents the string literal value; `ID` represents the identifier. These terminators can be defined using regular expressions:

- `INTEGER`: `0x[a-fA-F0-9]+|\d+`.
- `REAL`: `(\d+\.?|\.\d)\d*([eE][+-]?\d+)?`.
- `STRING`: `"(\\.|[^"])*"|'(\\.|[^'])*'`.
- `ID`: `[_a-zA-Z]\w*`

The symbols that appear sequentially in the standard EBNF are separated by commas. For intuitiveness, I use spaces to implement the comma function. The vertical bar symbol "|" is pronounced as "or", it means that the left and right patterns can only match one of them, or has the lowest priority. For example, the grammar $a0a1|a2$ means either the matching formula $a0a1$ or the matching $a2$. The square brackets indicate that the sub-expression inside the parentheses are matched 0 or 1 times, the curly braces indicate that the internal sub-expression is matched 0 or more times, and the parentheses only have the function of taking the internal sub-expression as a whole.

The following is the JSON grammar definition supported by the JSON module in the Berry standard library. The usage of EBNF still complies with the above conventions:

```
json = value;
value = object | array |
        string | number | 'true' | 'false' | 'null';
object = '{' [ string ':' value ] { ',' string ':' value } '}';
array = '[' [json] { ',' json } ']';
```

Non-terminal symbols `string` and `number` can also be defined using regular expressions. http://www.json.org gives the standard grammar of JSON, which also includes the definitions of `string` and `number`. The Berry JSON library's support for numbers is different from the standard. The standard JSON numbers must start with "-" or the number "0-9", while the Berry JSON library also accepts numbers starting with a decimal point.

Compiler Interpreter

1. Overview

The source code of the Berry interpreter is written using the ISO C99 standard, and the core code does not rely on third-party libraries, so it has strong versatility. Take the Ubuntu system as an example, execute the following command in the terminal to install the Berry interpreter:

```
apt install git gcc g++ make libreadline-dev
git clone https://github.com/berry-lang/berry
cd berry
make
make install
```

The Makefile provided in the GitHub repository is built using the GCC compiler. Other compilers can also compile the Berry interpreter correctly. The currently tested and available compilers include GCC, Clang, MSVC, ARMCC and ICCARM. The compiler that compiles the Berry interpreter should have the following characteristics:

- C compiler that supports the C99 standard
- C++ compiler supporting C++11 standard (only for native compilation)
- 32 or 64 bit target platform

The C++ compiler is only used to compile *map_build* tools, so there is no need to provide a C++ cross compiler for the Berry interpreter when cross-compiling, but the user should prepare a native C++ compiler (unless the user can obtain the *map_build* tool executable file).

2. Porting

The following is how to port the Berry interpreter to the user's project:

1. Add all source files in the *src* directory to the user project, and the directory should be added to the include path
2. Users need to implement by themselves *default* files other than *berry.c* in the directory. If conditions permit, they don't need to modify them
3. Use *map_build* tool to generate constant object code and then compile

3. Platform Support

Currently Berry interpreter has been tested on some platforms. Windows, Linux and MacOS operating systems running on X86 CPUs can run normally. Embedded platforms that have been tested include Cortex M3/M0/M4/M7. The Berry interpreter should be able to run well on the basis of the necessary C runtime library. At present, when only Berry language core is compiled, the interpreter code generated by the ARMCC compiler is only about 40KiB, and the interpreter can run on a device with only 8KiB of RAM.

Porting Guide

Configuration File

The configuration header file of the Berry interpreter is *berry_conf.h*. This file includes a batch of macros for configuration and defines some platform-related content.

berry_conf.h File

Configuration Macro Switch

The configuration macros introduced in this section are usually used for compiling switches of some source codes. For the convenience of description, we call this macro "macro switch". For the macro switch, "on" refers to setting the macro switch to a non-zero value, and "off" refers to setting the value of the macro switch to `0`.

Some macro switches have multiple states, not just "on" or "off". These macro switches are generally used for configurations with multiple options. There are also some configuration macros that are not macro switches. No matter what the value of these macros is, there will be no code and therefore will not participate in compilation. These macros are generally used to configure the quantity value.

[section::BE_DEBUG]

This macro switch is used to turn on or off the debugging function of the interpreter itself. When the value of `BE_DEBUG` is `0`, debugging is turned off, otherwise it will be turned on. The debugging function mentioned here refers to the debugging of the interpreter, not the debugging function of the Berry program. The default value of `BE_DEBUG` is `0`. If you use *Makefile* that comes with the interpreter project to compile, this macro switch will be turned on automatically when you use the `make debug` command.

When this macro is opened, some assertions will be turned on, and an error message will be output when the interpreter encounters an error that the assertion can catch. You can open `BE_DEBUG` when debugging the interpreter, and close it when compiling the release.

This macro switch configures the floating point type used by the `breal` type. When the value of the macro is `0`, the `double` type will be used to implement `breal`, otherwise the `float` type will be used to implement `breal`. This macro switch can be turned on in some environments with low performance or memory configuration. In the default implementation, this macro switch is turned off.

This macro configures the implementation of the `bint` type. When the value of the macro is `0`, the `int` type will be used to implement `bint`, when the value is `1`, the `long` type will be used to implement `bint`, and when the value is `2`, 64 will be used] Bit signed integer type (`__int64` under Windows, `long long` on other platforms) implements `bint`. The default value of this macro is `2`. If you want to reduce memory usage, you can set this macro to `0` or `1` to enable 32-bit integer type.

This macro is used to configure runtime debugging information of Berry code. It has 3 available values: set to `0` to turn off the output of the file name and line number of the runtime debugging information, and set to `1` to output the file name and line number in the runtime debugging information, set to `2` When using `uint16_t` (16-bit integer) type to store the row number information. Its default value is `1`.

Setting this macro to `0` will not store the file name and line number information, so the memory consumption is minimal. When set to `2`, it consumes less memory, but if the program is too long, `uint16_t` will overflow.

This macro switch configures the function of constructing objects at compile time. Turning on this macro means that constructing objects at compile time is enabled. This macro is turned on by default. When this macro is turned on, the native objects in the standard library will be generated at compile time, and when this macro is turned off, the objects in the standard library will be built at runtime.

be_regfunc and be_regclass functions will be affected by this macro. The built-in object table cannot be modified when using compile-time object construction. At this time, these two functions cannot register objects in the built-in scope, but register objects in the global scope.

The objects constructed during the compile time are stored together with the code and will not occupy RAM (or the readable and writable area in the memory) resources. The construction technology during the compile time can also reduce the startup time of the interpreter, so it is recommended to open this macro. Please refer to section [section::precompiled_build] for more details on compile-time construction techniques.

This macro defines the maximum Berry stack capacity, which refers to the number of Berry objects. When the Berry code uses more than this amount of stack, it will stop executing the program and return an error message. The default value of this macro is 2000, which can be modified according to the memory capacity of the system.

This value does not affect the memory usage of the Berry stack, because the capacity of the Berry stack is dynamically adjusted, so no matter how much it is set to, it cannot help reduce memory usage. Its main function is to terminate execution when the Berry program consumes too much stack. These programs are very likely to be incorrect, for example, recursive function calls without return conditions will continue to consume the stack.

This macro defines the minimum available space in the Berry stack, and its default value is 10. The native function may push a value into the Berry stack. At this time, the stack will not automatically grow, so make sure that there is enough space in the stack for the native function to use. It is not recommended to modify this value, but to use the be_stack_require function where more stack space is really needed.

In order to detect stack overflow errors when debugging the interpreter, you can open the BE_DEBUG macro (section [section::BE_DEBUG]).

When this macro is opened, the short string object will save the hash value of the string to improve the running speed, but the size of each string object will increase by 4 bytes. This macro is turned off by default, and the current tests have not found that opening this macro will bring significant improvement.

This macro switch is used to enable or disable the string module, which is turned on by default.

This macro switch is used to enable or disable the json module, which is turned on by default.

This macro switch is used to enable or disable the math module, which is turned on by default.

This macro switch is used to enable or disable the time module, which is turned on by default.

This macro switch is used to enable or disable the os module, which is turned on by default.

This macro determines the abort function used internally by the Berry interpreter. By default or when the macro is not defined, the abort function in the C standard library will be used. This macro is defined as abort by default. If the user needs to explicitly specify the abort function used by the interpreter, then replace the macro definition with the function required by the user. This function should be in the same form as the abort function declaration in the standard library.

This macro determines the exit function used internally by the Berry interpreter. By default or when the macro is not defined, the exit function in the C standard library will be used. This macro is defined as exit by default. If the user needs to explicitly specify the exit function used by the interpreter, then replace the macro definition with the function required by the user. This function should be in the same form as the exit function declaration in the standard library.

This macro determines the malloc function used internally by the Berry interpreter. By default or when the macro is not defined, the malloc function in the C standard library will be used. This macro is defined as malloc by default. If the user needs to explicitly specify the function malloc used by the interpreter, then replace the macro definition with the function required by the user. This function should be in the same form as the malloc function declaration in the standard library.

This macro determines the free function used internally by the Berry interpreter. By default or when the macro is not defined, the free function in the C standard library will be used. This macro is defined as free by default. If the user needs to explicitly specify the free function used by the interpreter, then replace the macro definition with the function required by the user. This function should be in the same form as the free function declaration in the standard library.

This macro determines the `realloc` function used internally by the Berry interpreter. By default or when the macro is not defined, the `realloc` function in the C standard library will be used. This macro is defined as `realloc` by default. If the user needs to explicitly specify the `realloc` function used by the interpreter, then replace the macro definition with the function required by the user. This function should be in the same form as the `realloc` function declaration in the standard library.

This macro is used to define the implementation of the assertion function. By default, the `assert` function in the C standard library is used to implement the assertion. If the target system is inconvenient to use the `assert()` function in the standard library to make an assertion, you can modify the definition of the `be_assert` macro. A correct assertion function should use the following declaration:

```
void assert(int condition);
```

Among them, `condition` is the assertion condition. If the condition is not met, an error message will be output and the program will be terminated. Of course, the "assert" function is usually implemented using a macro.

berry_port.c File

This file implements the low-level IO functions of the Berry interpreter, including standard input and output and file system support. The *berry_port.c* file in the *default* directory contains a set of portable IO support. File operations and standard input and output are implemented using APIs in the C standard library. Path and folder operations support both Windows and POSIX standard APIs. This file also implements a set of FatFs-based IO operation functions for users to use directly. If you need to use the Berry interpreter in other environments, then these functions must be implemented separately (may only need to be implemented partially).

This section will introduce the functions of the functions implemented in the *berry_port.c* file and guide users to implement their own version.

```
void be_writebuffer(const char *buffer, size_t length);
```

Output a piece of data to the standard output device, the parameter `buffer` is the first address of the output data block, and `length` is the length of the output data block. This function outputs to the `stdout` file by default. Inside the interpreter, this function is usually used as a character stream output, not a binary stream.

`be_writebuffer` Functions are very versatile and must be implemented.

```
char* be_readstring(char *buffer, size_t size);
```

Input a piece of data from the standard input device, and read at most one row of data each time this function is called. The parameter `buffer` is the data buffer passed in by the caller, and the capacity of the buffer is `size`. This function will stop reading and return when the buffer capacity is used up, otherwise it will return when a newline character or end of file character is read. If the function executes successfully, it will directly use the `buffer` parameter as the return value, otherwise it will return `NULL`.

This function will add the read line breaks to the read data, and each time the `be_readstring` function is called, it will continue to read from the current position. This function is only called in the implementation of the native function `input`, and the `be_readstring` function may not be implemented when it is not necessary.

```
void* be_fopen(const char *filename, const char *modes);
```

To open a file, `filename` is the name of the file to be opened, and `modes` is the opening method. The function will return a file handle or a pointer to the file operation structure. The usage of this function is similar to the `fopen` function in the C standard library. The file name is a C-style string (ending with a `\0` character), and the pattern should at least support the following conditions:

- `r`, `rt`: To open a text file in read-only mode, the file must exist.

- r+, rt+: Open a text file in read-write mode, and create a new file if the file does not exist.
- rb: Open a binary file in read-only mode, the file must exist.
- rb+: Open a binary file in read-write mode, and create a new file if the file does not exist.
- w, wt: Create and open a text file in write-only mode, and the existing file will be deleted.
- w+, wt+: Create and open a text file in read-write mode, and the existing file will be deleted.
- wb: Create and open a binary file in write-only mode, and the existing file will be deleted.
- wb+: Create and open a binary file in read-write mode, and the existing file will be deleted.

By default, the fopen function in the C standard library is used to implement be_fopen. If you use other methods to achieve, you should ensure that the above operating modes can be achieved. If no file operations are required, this function can be left blank. The file operations here include all scenarios such as using the open function in the script, loading the script from a file (using the be_loadfile function), etc.

```
int be_fclose(void *hfile);
```

Close a file, hfile is the closed file handle. The function of this function is similar to the function fclose in the C standard library.

```
size_t be_fwrite(void *hfile, const void *buffer, size_t length);
```

Write a piece of data to the specified file. Parameter hfile is the file handle to be written, buffer is the pointer of the data to be written, length is the number of data to be written (in bytes).

```
size_t be_fread(void *hfile, void *buffer, size_t length);
```

Read a piece of data from the specified file. The parameter hfile is the file handle to be read, buffer is the pointer to the read buffer, and length is the number of bytes to be read.

```
char* be_fgets(void *hfile, void *buffer, int size);
```

Read a line from the file, similar to the fgets function in the C standard library. Parameter hfile is the file handle to be read, buffer is the pointer of the read buffer, and size is the capacity of the read buffer. This function will return when size - 1 bytes, newline characters and end of file characters are read, and the return value is buffer.

```
int be_fseek(void *hfile, long offset);
```

Set the position of the file read and write pointer. The parameter hfile is the file handle to be operated, and offset is the value to be set.

```
long int be_ftell(void *hfile);
```

Get the current read and write pointer of the file, the parameter hfile is the handle of the file to be operated, and the return value of this function is the read and write pointer of the file.

```
long int be_fflush(void *hfile);
```

Write the data in the file buffer to the file. Parameter hfile is the file to be operated.

```
size_t be_fsize(void *hfile);
```

Get the size of the file. Parameter hfile is the file to be operated.

1.1.2 Making a Native Function

Berry's C FFI (Foreign Function Interface) operates on a virtual stack to interact with the VM. If we need to make an add function to add two numbers and use it in Berry in this way:

```
result = add(1, 2)
```

We need to know how the C code gets the arguments from the Berry function call and how to return the value.

The function arguments are stored in a stack, and the first argument to the last argument of the function is stored from the bottom of the stack to the top of the stack. If you want to use C to fetch elements from the stack, use the following set of FFIs:

```
int be_toint(bvm *vm, int index);
breal be_toreal(bvm *vm, int index);
int be_tobool(bvm *vm, int index);
const char* be_tostring(bvm *vm, int index);
void* be_tocomptr(bvm *vm, int index);
```

If you want to test if a value in the stack is a specified type, use the following set of FFIs:

```
int be_isnil(bvm *vm, int index);
int be_isbool(bvm *vm, int index);
int be_isint(bvm *vm, int index);
int be_isreal(bvm *vm, int index);
int be_isnumber(bvm *vm, int index);
int be_isstring(bvm *vm, int index);
int be_isclosure(bvm *vm, int index);
int be_isntvclos(bvm *vm, int index);
int be_isfunction(bvm *vm, int index);
int be_isproto(bvm *vm, int index);
int be_isclass(bvm *vm, int index);
int be_isinstance(bvm *vm, int index);
int be_islist(bvm *vm, int index);
int be_ismap(bvm *vm, int index);
int be_iscomptr(bvm *vm, int index);
```

If you need to push values onto the stack, use these FFIs:

```
void be_pushnil(bvm *vm);
void be_pushbool(bvm *vm, int b);
void be_pushint(bvm *vm, bint i);
void be_pushreal(bvm *vm, breal r);
void be_pushstring(bvm *vm, const char *str);
void be_pushnstring(bvm *vm, const char *str, size_t n);
const char* be_pushfstring(bvm *vm, const char *format, ...);
void be_pushvalue(bvm *vm, int index);
void be_pushntvclosure(bvm *vm, bntvfunc f, int nupvals);
void be_pushntvfunction(bvm *vm, bntvfunc f);
void be_pushclass(bvm *vm, const char *name, const bnfuncinfo *lib);
void be_pushcomptr(bvm *vm, void *ptr);
```

`index` is the position of the element on the stack, the positive value is the offset from the bottom of the stack to the top of the stack, and the negative value is the offset from the top of the stack to the bottom of the stack.

The return value uses two FFIs:

```
be_return(vm)
be_return_nil(vm)
```

These FFIs are actually macros. `be_return` returns the object at the top of the stack, and be_return_nil returns `nil`.

These FFIs are defined in berry.h.

Now let's implement the `add` function:

```
int my_add_func(bvm *vm)
{
    /* check the arguments are all integers */
    if (be_isint(vm, 1) && be_isint(vm, 2)) {
        bint a = be_toint(vm, 1); /* get the first argument */
        bint b = be_toint(vm, 2); /* get the second argument */
        be_pushint(vm, a + b); /* push the result to the stack */
    } else if (be_isnumber(vm, 1) && be_isnumber(vm, 2)) { /* check the arguments are␣
↪all numbers */
        breal a = be_toreal(vm, 1); /* get the first argument */
        breal b = be_toreal(vm, 1); /* get the second argument */
        be_pushreal(vm, a + b); /* push the result to the stack */
    } else { /* unacceptable parameters */
        be_pushnil(vm); /* push the nil to the stack */
    }
    be_return(vm); /* return calculation result */
}
```

Then register it in the appropriate place:

```
be_regcfunc(vm, "add", my_add_func);
```

1.1.3 Instantiate a `list` object in a native function

Generating instantiated native classes in C can be cumbersome compared to simple types. This section will guide the reader to instantiate the `list` class.

The `list` class is a wrapper around the list structure, which has a `.data` property for the list structure. Therefore, we first need to construct a list structure:

```
be_newlist(vm);
```

The `be_newlist` function constructs a value of type `BE_LIST`. Then we can operate on the data:

```
be_pushint(vm, 100);
be_data_append(vm, -2);
be_pop(vm, 1); /* popping the integer 100 */
```

The first two lines of code are used to append the integer `100` to the list, and the third line to the integer `100` is popped to facilitate subsequent operations.

Since the `BE_LIST` type cannot be used directly in Berry, but is used by the `list` class, we have to build the `list` class for it:

```
be_getglobal(vm, "list");
be_pushvalue(vm, -2); /* push the list data to top */
be_call(vm, 1); /* call constructor */
```

The constructor of the `list` class allows the use of the `BE_LIST` type parameter, which takes the argument as list data.

The complete code is as follows:

```
int m_listtest(bvm *vm)
{
    be_getglobal(vm, "list");
    be_newlist(vm);
    be_pushint(vm, 100);
    be_data_append(vm, -2);
    be_pop(vm, 1);
    be_call(vm, 1);
    be_pop(vm, 1); /* pop the arguments */
    be_return(vm);
}
```

Register the native function in the appropriate place:

```
be_regcfunc(vm, "listtest", m_listtest);
```

1.1.4 Roadmap

TODO

- REPL multi-line support.
- Runtime debug information.
- API stack protection.
- File operation support.
- Fixed (ROM based) hash table.
- Destructor support.
- Native module support (use `import xxx` to import a module).
- Conditional expression support.
- Anonymous function.
- Bitwise operation.
- Compound assignment statement.
- Built-in scope.
- Variable arguments function.
- Native function: `classof(obj)`.
- [STRIKEOUT:Native function: `copy(obj)`].
- Auto release call stack.
- Regular expression.

- Stack usage optimized GC (no recursion).
- Simplify stack overflow error messages.
- Lambda expression.
- Exceptional handling.
- Bytecode file support.
- Optimized iterator and `for` statement.
- Connection operator (redefine the range operator `..`).
  - String connection, e.g., `'string' .. expr`.
  - List connection, e.g., `[] .. expr`.
  - List serialization method (for high performance string connection).
- Complete module support.
  - Script file module export / import.
  - Bytecode file module build / import.
  - Shared native module (like *.so,*.dll) load.
- Universal Equality Check API (like == operator but use in C).
  - Complete map key matching.
  - Complete exception value matching.
- Debugger support.
- Complete class and object mechanism.
- Support error message for *map_build* tool.
- Overloaded call `()` operator.

Release version

V0.2.0

- Complete Chinese documentation.
- Exceptional handling.
- Complete module support.

V0.1.0

- Portable file system interface.
- Precompiled constant object support.
- Complete precompiled constant module.
- [STRIKEOUT:More complete documentation (Chinese)].
- [STRIKEOUT:Porting guide].

1.1.5 Memory Requirements

RAM required

- extreme: 4 KB
- low: 8 KB
- recommended: 16 KB

ROM/Flash required

- extreme: 64 KB
- low: 128 KB
- recommended: 256 KB

Explanation

- **extreme**: The memory capacity required in the minimum configuration. Lower than this value means that it is almost impossible to run Berry.
- **low**: The minimum memory required for the complete Berry interpreter includes some user and third-party code.
- **recommended**: In addition to the complete Berry interpreter, there are many users and third-party code.

1.2 Bienvenido a la wiki de Berry!

1.2.1 Manual de referencia del lenguaje Berry Script

Prefacio

Hace algunos años, traté de portar el lenguaje de script Lua al microcontrolador STM32F4 y luego experimenté un firmware basado en Lua en ESP8266: NodeMCU. Estas experiencias me hicieron experimentar la conveniencia del desarrollo de guiones. Más tarde, entré en contacto con algunos lenguajes de script, como Python, JavaScript, Basic y MATLAB. En la actualidad, solo unos pocos lenguajes son adecuados para portar a la plataforma de microcontroladores. Solía prestar más atención a Lua porque es un lenguaje de script integrado muy compacto y su objetivo de diseño es estar integrado en el programa host. Sin embargo, para el microcontrolador, el intérprete de Lua puede no ser lo suficientemente pequeño y no puede ser ejecutado en un microcontrolador de 32 bits con una memoria relativamente pequeña. Con este fin, comencé a leer el código Lua y desarrollé mi propio lenguaje de script, Berry, sobre esta base.

Este es un lenguaje de script integrado ultraligero, también es un lenguaje dinámico multiparadigma. Admite estilos orientados a objetos, orientados a procesos y funcionales (menos soportado). Muchos aspectos de este lenguaje se refieren a Lua, pero su sintaxis también se inspira en el diseño de otros lenguajes. Si el lector ya conoce un lenguaje de alto nivel, la gramática de Berry debería ser muy fácil de entender: el lenguaje tiene solo algunas reglas simples y un diseño de alcance muy natural.

El principal escenario de aplicación que considero son los sistemas integrados con bajo rendimiento. La memoria de estos sistemas puede ser muy pequeña, por lo que es muy difícil ejecutar un lenguaje de script con todas las funciones. Esto significa que es posible que tengamos que hacer una elección. Es posible que Berry solo proporcione las funciones principales básicas y más utilizadas, mientras que otras características innecesarias solo se usan como módulos opcionales. Esto conducirá inevitablemente a que la biblioteca estándar del idioma sea demasiado pequeña, incluso el lenguaje. También habrá diseños inciertos (como la implementación de números de punto flotante y enteros, etc.). Los

beneficios de estas compensaciones son más espacio para la optimización, mientras que la desventaja es obviamente la inconsistencia de los estándares del idioma. El intérprete de Berry se refiere a la implementación del intérprete de Lua, que se divide principalmente en dos partes: compilador y máquina virtual. El compilador de Berry es un compilador de un solo paso para generar bytecode. Esta solución no genera un árbol de sintaxis abstracto, por lo que ahorra memoria. La máquina virtual es de tipo registro. En términos generales, la máquina virtual de tipo registro es más eficiente que la máquina virtual de tipo pila. Además de implementar funciones de lenguaje, también esperamos optimizar el uso de la memoria y la eficiencia operativa del intérprete. En la actualidad, los indicadores de rendimiento del intérprete de Berry no son comparables con los de los lenguajes principales, pero la huella de memoria es muy pequeña.

No fue hasta más tarde que me enteré del proyecto MicroPython: un intérprete de Python simplificado diseñado para microcontroladores. Hoy en día, Python está muy de moda, y este intérprete de Python diseñado para microcontroladores también es muy popular. En comparación con la tecnología madura actual, Berry es un lenguaje nuevo sin una base de usuarios suficiente. Su ventaja es que es fácil de dominar la gramática y puede tener ventajas en términos de consumo de recursos y rendimiento.

Si necesita migrar el intérprete de Berry, debe asegurarse de que el compilador que usa brinde soporte para el estándar C99 (anteriormente cumplia completamente con C89, y algunos trabajos de optimización más tarde me hicieron renunciar a esta decisión). En la actualidad, la mayoría de los compiladores brindan soporte para C99, y los compiladores comunes como ARMCC (KEIL MDK), ICC (IAR) y GCC en el desarrollo de procesadores ARM también admiten C99. Este documento presenta las reglas gramaticales de Berry, la biblioteca estándar y otras instalaciones, y finalmente guía a los lectores para trasplantar y ampliar Berry. Este documento no explica el mecanismo de implementación del intérprete, y puede ser explicado en otros documentos, si tenemos tiempo.

El nivel del autor es limitado y la escritura apresurada. Si hay omisiones o errores en el artículo, espero que los lectores no duden en corregirlos.

Guan Wenliang

abril 2019

Información básica

1.1 Introducción

Berry es un lenguaje de script integrado de tipo dinámico ultraligero. El lenguaje admite principalmente programación procedural, así como programación orientada a objetos y programación funcional. Un objetivo de diseño importante de Berry es poder ejecutarse en dispositivos integrados con muy poca memoria, por lo que el lenguaje está muy optimizado. Sin embargo, Berry sigue siendo un lenguaje de secuencias de comandos rico en funciones.

1.2 Empezar a usar

Obtener el intérprete

Los lectores pueden ir a la página de GitHub del proyecto: https://github.com/berry-lang/berry para obtener el código fuente del intérprete de Berry. Los lectores necesitan compilar el intérprete de Berry por sí mismos. El método de compilación específico se puede encontrar en el documento README.md en el directorio raíz del código fuente, que también se puede ver en la página de GitHub del proyecto.

Primero, debe instalar un software como GCC, git y make. Si no usa el control de versiones, puede descargar el código fuente directamente en GitHub sin instalar git. Los lectores pueden usar motores de búsqueda para recuperar información sobre este software. Los lectores que utilizan sistemas Linux y macOS también deben instalar la biblioteca GNU Readline[1]. Use el comando `git` [2] Clone el código fuente del intérprete del almacén remoto al local:

```
git clone https://github.com/berry-lang/berry
```

Ingrese al directorio berry y use el comando `make` para compilar el intérprete:

```
cd berry
make
```

Ahora debería poder encontrar el archivo ejecutable del intérprete en el directorio *berry* (en los sistemas Windows, el nombre del archivo del programa intérprete es "*berry.exe*", mientras que en los sistemas Linux y macOS el nombre del archivo es "*berry*"), puede ejecutar el archivo ejecutable directamente. Para iniciar el intérprete: En Linux o macOS, puede usar el comando `sudo make install` para instalar el intérprete, y luego puede iniciar el intérprete con el comando `berry` en la terminal.

Entorno REPL

REPL (Leer Evaluar Imprimir Bucle) generalmente se traduce como un intérprete interactivo, y este elemnto también se ha convertido en el modo interactivo del intérprete. Este modo consta de cuatro elementos: **Leer**, lee el código fuente ingresado por el usuario desde el dispositivo de entrada; **Evaluar**, compila y ejecuta el código fuente ingresado por el usuario; **Imprimir**, muestra el resultado del proceso de evaluación; **Bucle**, realiza un ciclo de las operaciones anteriores.

Inicie el intérprete directamente (ingrese `berry` en la terminal o en la ventana de comandos sin parámetros, o haga doble clic en berry.exe en Windows) para ingresar al modo REPL y verá la siguiente interfaz:

```
Berry 1.0.0 (build in Feb 1 2022, 13:14:04)
[GCC 8.1.0] on Windows (default)
>
```

Las dos primeras líneas de la interfaz muestran la versión, el tiempo de compilación, el compilador y el sistema operativo del intérprete de Berry. El símbolo ">" en la tercera línea se llama indicador y el cursor se muestra detrás del indicador. Al usar el modo REPL, después de ingresar el código fuente, al presionar la tecla "Enter" se ejecutará inmediatamente el código y se generará el resultado. Presione la combinación de teclas `Ctrl + C` para salir de REPL. En el caso de usar la biblioteca Readline, use la combinación de teclas `Ctrl + D` para salir de REPL cuando la entrada esté vacía. Dado que no hay necesidad de editar archivos de secuencias de comandos, el modo REPL se puede utilizar para un desarrollo rápido o verificación de ideas.

Programa Hola Mundo

Tomemos como ejemplo el clásico programa "Hola Mundo". Introduzca `print('Hola Mundo')` en el REPL y ejecútelo. Los resultados son los siguientes:

```
Berry 1.0.0 (build in Feb 1 2022, 13:14:04)
[GCC 8.1.0] on Windows (default)
> print('Hola Mundo')
Hola Mundo
>
```

El intérprete emite el texto "`Hola Mundo`". Esta línea de código realiza la salida de la cadena `'Hola Mundo'` llamando a la función `print`. En REPL, si el valor de retorno de la expresión no es `nil`, se mostrará el valor de retorno. Por ejemplo, ingresar la expresión `1 + 2` mostrará el resultado del cálculo `3`:

```
> 1 + 2
3
```

Por lo tanto, el programa "Hola Mundo" más simple bajo REPL es ingresar directamente la cadena `'Hola Mundo'` y ejecutar:

```
> 'Hola Mundo'
Hola Mundo
```

Más uso de REPL

También puede utilizar el modo interactivo del intérprete de Berry como calculadora científica. Sin embargo, algunas funciones matemáticas no se pueden utilizar directamente. En su lugar, use la declaración `import math` para importar la biblioteca matemática y luego use las funciones en la biblioteca. Utilice "`math.`" como prefijo, por ejemplo, la función `sin` debe escribirse como `math.sin`:

```
> import math
> math.pi
3.14159
> math.sin(math.pi / 2)
1
> math.sqrt(2)
1.41421
```

Archivo de comandos

El archivo de secuencia de comandos de Berry es un archivo que almacena el código de Berry, y un intérprete puede ejecutar el archivo de script. Normalmente, el archivo de script es un archivo de texto con la extensión ".be". El comando para ejecutar el script usando el intérprete es:

```
berry script_file
```

`script_file` es el nombre de archivo del archivo de script. El uso de este comando ejecutará el intérprete para ejecutar el código Berry en el archivo de script `script_file` y el intérprete se cerrará después de la ejecución.

Si desea que el intérprete ingrese al modo REPL después de ejecutar el archivo de script, puede agregar el parámetro `-i` al comando para llamar al intérprete:

```
berry script_file
```

Este comando primero ejecutará el código en el archivo `script_file` y luego ingresará al modo REPL.

1.3 Palabras (Words)

Antes de presentar la sintaxis de Berry, echemos un vistazo a un código simple (puede ejecutar este código en modo REPL):

```
def func(x) # una función ejemplo
    return x + 1.5
end
print('func(10) =', func(10))
```

Este código define una función `func` y la llama más tarde. Antes de entender cómo funciona este código, primero presentaremos los elementos de sintaxis del lenguaje Berry.

En el código anterior, la clasificación específica de los elementos gramaticales es: `def`, `return` y `end`. Estas son palabras clave del lenguaje Berry; y "`# una función ejemplo`" en la primera línea se llama comentario; `print`, `func` y `x` son algunos identificadores, generalmente se usan para representar una variable; `1.5` y `10` estos números se llaman literales numéricos, son equivalentes a los números usados en la vida diaria; `'func(10) ='` Es un literal de cadena, se usan en lugares donde necesitas representar texto; + es un operador de suma, aquí el operador de suma se puede usar para sumar la variable `x` y el valor `1.5`.

La clasificación anterior se realiza en realidad desde la perspectiva de un analizador lexicográfico. El análisis lexicográfico es el primer paso en el análisis del código fuente de Berry. Para escribir el código fuente correcto, comenzamos con la introducción lexicográfica más básica.

Comentario

Los comentarios son textos que no genera ningún código. Se utilizan para hacer comentarios en el código fuente y ser leídos por personas, mientras que el compilador no interpretará su contenido. Berry admite comentarios de una sola línea y comentarios de bloque de líneas cruzadas. Los comentarios de una sola línea comienzan con el carácter "#' hasta el final del carácter de nueva línea. La nota rápida comienza con el texto "#-" y termina con el texto "-#". El siguiente es un ejemplo del uso de anotaciones:

```
#  Este es un comentario de línea
#- Este es un
   bloque de comentario
-#
```

Similar al lenguaje C, los comentarios rápidos no admiten el anidamiento. El siguiente código terminará el análisis de los comentarios en el primer texto "-#":

```
#- - Algunos comentarios -# ... -#
```

Valor literal

El valor literal es un valor fijo escrito directamente en el código fuente durante la programación. Los literales de Berry son números enteros, números reales, booleanos, cadenas y nil. Por ejemplo, el valor `34` es una denominación entera.

Valor literal numérico

Los literales numéricos incluyen literales **Integer** (entero) y literales **Número real** (real).

```
40 #  Literal entero
0x80 # Literal hexadecimal (entero)
3.14 #  Literal real
1.1e-6 #  Literal real
```

Los literales numéricos se escriben de manera similar a la escritura cotidiana. Berry admite denominaciones enteras hexadecimales. Los literales hexadecimales comienzan con el prefijo `0x` o `0X`, seguido de un número hexadecimal.

Valor literal booleano

Los valores booleanos (booleanos) se utilizan para representar verdadero y falso en el estado lógico. Puede utilizar las palabras clave `true` y `false` para representar literales booleanos.

Literal de cadena

Una cadena es un fragmento de texto, y su escritura literal consiste en usar un par de ' o " para rodear el texto de la cadena:

```
'esto es una cadena'
"esto es una cadena"
```

Los literales de cadena proporcionan algunas secuencias de escape para representar caracteres que no se pueden ingresar directamente. La secuencia de escape comienza con el carácter `'\'` y luego sigue una secuencia específica de caracteres para lograr el escape. Las secuencias de escape especificadas por Berry son

Caracter de Escape	significado	Caracter de Escape	significado
\a	Suena la campana	\b	Retroceso
\f	Alimentación de formulario.	\n	Nueva línea
\r	Retorno de carro	\t	Tabulación Horizontal
\v	Tabulación Vertical	\\	Barra invertida
\'	Apóstrofe	\"	Comillas dobles
\?	Signo de interrogación	\0	Caracter Null

Tabla 1: Secuencia de Caracter de Escape

Las secuencias de escape se pueden usar en cadenas, por ejemplo

```
print('caracter de escape LF\n\tnueva línea')
```

El resultado de la operación es:

```
caracter de escape LF
        nueva línea
```

También puede usar secuencias de escape generalizadas, en forma de \x seguido de 2 dígitos hexadecimales, o \ 3 dígitos octales, usando esta secuencia de escape puede representar cualquier carácter. Estos son algunos ejemplos del uso del conjunto de caracteres ASCII:

```
'\115' #-'M' -#'\x34' #- '4' -#'\064' #- '4' -#
```

Valor literal nulo

Nil representa un valor nulo, y su valor literal está representado por la palabra clave `nil`.

Identificador

El identificador es un nombre definido por el usuario, que comienza con un guión bajo o una letra, y luego consiste en una combinación de varios guiones bajos, letras o números. Al igual que la mayoría de los lenguajes, Berry distingue entre mayúsculas y minúsculas, por lo que los identificadores `A` y los identificadores `a` se resolverán en dos identificadores diferentes.

```
a
TestVariable
Test_Var
_init
baseCass
_
```

Palabras clave

Berry reserva los siguientes tokens como palabras clave del lenguaje:

```
if elif else while for def
end class break continue return true
false nil var do import as static
```

El uso específico de palabras clave se presentará en los capítulos correspondientes. Tenga en cuenta que las palabras clave no se pueden utilizar como identificadores. Debido a que Berry distingue entre mayúsculas y minúsculas, `If` puede usarse para identificadores.

[1] Para GNU Readline, el comando de instalación para la serie Debian de distribuciones de Linux es `sudo apt install libreadline-dev`, y el comando de instalación para la serie RedHat de distribuciones de Linux es `yum install readline-devel`, bajo macOS el comando de instalación es `brew install readline`. Además, es fácil encontrar documentación de GNU Readline y materiales relacionados en los motores de búsqueda.

[2] Los comandos deben usarse en la "interfaz de línea de comandos" después de completar el trabajo de preparación. El entorno de la línea de comandos en los sistemas Windows suele ser una ventana del símbolo del sistema (CMD), mientras que el entorno de la línea de comandos en los sistemas similares a Unix suele llamarse "Terminal". Esto no es muy preciso, pero no se ampliará aquí.

[3] En Windows, puede hacer doble clic directamente para ejecutar el archivo ejecutable. En Linux o macOS, use la terminal para ejecutarlo. También puede ejecutar el intérprete en la ventana del símbolo del sistema de Windows. Consulte el archivo README.md para un uso específico.

2. Tipos y Variables

Tipo es un atributo de datos, que define el significado de los datos y las operaciones que se pueden realizar en los datos. Los tipos se pueden dividir en tipos integrados y tipos definidos por el usuario. Los tipos integrados se refieren a algunos tipos básicos integrados en el lenguaje Berry, entre los cuales los tipos que no se basan en definiciones de clase se denominan **Tipo simple**. Los tipos basados en definiciones de clase se denominan **Tipo de clase**, algunos de los tipos integrados son tipos de clase y los tipos definidos por el usuario también son tipos de clase.

2.1 Tipo incorporado

2.1.1 Tipo simple

`nil`

El tipo Nil es el tipo nulo, lo que significa que el objeto tiene un valor no válido, o se puede decir que el objeto no tiene un valor significativo. Este es un tipo muy especial. Aunque podríamos decir que una variable es `nil`, de hecho el tipo nil no tiene valor, entonces de lo que estamos hablando aquí es que el tipo de la variable es nil (no un valor).

El valor predeterminado de una variable antes de la asignación es `nil`. Este tipo se puede utilizar en operaciones lógicas. En este caso 'nil' es equivalente a 'falso'.

Tipo entero

El tipo entero (integer) representa un entero con signo, denominado entero. El número de bits del entero representado por este tipo depende de la implementación específica y, por lo general, consta de un entero de 32 bits con signo en una plataforma de 32 bits. Integer es un tipo aritmético y admite todas las operaciones aritméticas. Preste atención al rango de valores de los enteros cuando use este tipo. El rango de valores típico de los enteros con signo de 32 bits está entre 2147483648 y 2147483647.

Cualquier valor se puede convertir a `int` usando la función `int()`; sin embargo `int(nil) == nil`. Si el argumento es una instancia, y si contiene un miembro `toint()`, se nos llamará y el valor devuelto se convertirá en `int`.

Tipo de número real

El tipo real (real), para ser precisos, es un tipo de coma flotante. Los tipos de números reales generalmente se implementan como números de punto flotante de precisión simple o números de punto flotante de precisión doble. El tipo de número real también es un tipo aritmético. En comparación con el tipo de número entero, el tipo de número real tiene mayor precisión y un mayor rango de valores, por lo que este tipo es más adecuado para cálculos matemáticos. Cabe señalar que el tipo de número real es en realidad un número de coma flotante, por lo que aun existen problemas de precisión. Por ejemplo, no se recomienda comparar dos valores de tipo `real` para la igualdad.

Cuando los enteros y los números reales participan en operaciones al mismo tiempo, los enteros generalmente se convierten en números reales.

Tipo booleano

El tipo booleano (boolean) se utiliza para operaciones lógicas. Tiene dos valores `true` y `false`, que representan los dos valores verdaderos (verdadero y falso) en lógica y álgebra booleana. El tipo booleano se utiliza principalmente para el juicio condicional. Los operandos y los valores devueltos de las expresiones lógicas y las expresiones relacionales son todos de tipo booleano, y las sentencias como `if` y `while` utilizan tipos booleanos como comprobaciones condicionales.

En muchos casos, los valores no booleanos también se pueden usar como tipos booleanos. Esto se debe a que el intérprete convertirá implícitamente los parámetros. Esta es también la razón por la que las expresiones de verificación condicional, como las declaraciones `if`, pueden usar cualquier tipo de parámetros. Las reglas para convertir varios tipos a tipos booleanos son:

- `nil`: convertido a `falso`.
- **Entero**: cuando el valor es `0`, se convierte en `falso`, de lo contrario, se convierte en `verdadero`.
- **Número real**: cuando el valor es `0.0`, se convierte en `falso`, de lo contrario, se convierte en `verdadero`.
- **Cadena**: cuando el valor es "" (cadena vacía) se convierte en `falso` de lo contrario, se convierte en `verdadero`.
- **Comobj** y **Comptr**: cuando el puntero interno es `NULL` es convertido a `falso`, de lo contrario se convierte a `verdadero`.
- **Instancia**: si la instancia contiene un método `tobool()`, se utilizará el valor de retorno del método, de lo contrario, se convertirá en `verdadero`.
- Todos los demás tipos: converten a `verdadero`.

Cualquier valor se puede convertir a `bool` usando la función `bool()`.

Cadena

Una cadena es una secuencia de caracteres. En términos de almacenamiento, Berry divide las cadenas en cadenas largas y cadenas cortas. Solo hay una instancia de la misma cadena de caracteres cortos en la memoria, y todas las cadenas de caracteres cortos están vinculadas en una tabla hash. Este diseño ayuda a mejorar el rendimiento de la comparación de igualdad de cadenas y puede reducir el uso de memoria. Dado que la frecuencia de uso de cadenas largas es baja y la sobrecarga de la operación hash es bastante alta, no están vinculadas a la tabla hash, por lo que puede haber varias instancias idénticas en la memoria. La cadena es de solo lectura después de que se crea. Por lo tanto, "modificar" la cadena generará una nueva cadena y la cadena original no se modificará.

Berry no se preocupa por el formato o la codificación de los caracteres. Por ejemplo, la cadena `'abc'` es en realidad el código ASCII de los caracteres `'a'`, `'b'` y `'c'`. Por lo tanto, si hay caracteres anchos en la cadena (la longitud de los caracteres es superior a 1 byte), el número de caracteres de la cadena no se puede contar directamente. De hecho, usar la función `size()` solo puede obtener el número de bytes en la cadena. Además, para facilitar la interacción con el lenguaje C, la cadena de Berry siempre termina con los caracteres `'\0'`. Esta función es transparente para el programa Berry.

El tipo de cadena se puede comparar en tamaño, por lo que se puede usar en operaciones relacionales.

Función

Una función es una pieza de código que está encapsulada y disponible para llamadas, generalmente utilizada para implementar una función específica. La función es en realidad una categoría grande, que incluye varios subtipos, como cierres, funciones nativas y cierres nativos. Para el código Berry, todos los subtipos de funciones tienen el mismo comportamiento. Las funciones pertenecen al primer tipo de valor en Berry, por lo que se pueden pasar como valores. Además, se puede usar directamente en expresiones a través de la forma "literal" de "funciones anónimas".

Una función es un objeto de solo lectura y no se puede modificar una vez definida. Puede comparar si dos funciones son iguales (si son la misma función), pero no se puede comparar el tipo de función. **Función nativa** y **Cierre nativo** se refieren a funciones y cierres implementados en lenguaje C. Uno de los propósitos principales de las funciones nativas y los cierres nativos es proporcionar funciones que el lenguaje Berry no proporciona, como operaciones de E/S y operaciones de bajo nivel. Si un fragmento de código se usa con frecuencia y tiene requisitos de rendimiento, se recomienda reescribirlo como una función nativa o un cierre nativo.

Clase

En la programación orientada a objetos, una clase es una plantilla de código de programa extensible. Las clases se utilizan para crear objetos de instancia, por lo que se puede decir que la clase es el "tipo" de la instancia. Todos los objetos de instancia son del tipo `instancia` y todos tienen una clase correspondiente, que se llama instancia **Tipo de clase**. En pocas palabras, una clase es un valor que representa el tipo de un objeto de instancia y una clase es una abstracción de las características de una instancia. Una clase también es un objeto de solo lectura, una vez definida, no se puede modificar.

Las clases solo pueden comparar iguales y desiguales, pero no pueden comparar tamaños.

Ejemplos

Una instancia es un objeto materializado generado por una clase, y el proceso de generar una instancia a partir de una clase se llama `Instanciación`. En la programación orientada a objetos, "instancia" suele ser sinónimo de "objeto". Sin embargo, para distinguir los objetos que no son de instancia, no usamos el término "objeto" solo, sino que usamos "instancia" u "objeto de instancia". Las instancias de Berry siempre se asignan dinámicamente y deben usarse con un recolector de basura. Además de la asignación de memoria, el proceso de creación de instancias también necesita inicializar la instancia, este proceso lo completa el `Constructor`. Además, puedes completar la destrucción del objeto a través del `Destructor` antes de recuperar la memoria del objeto.

En la implementación interna, la instancia contendrá una referencia a la clase, y la instancia en sí misma solo almacena variables miembro y no métodos.

2.1.2 Tipo de clase

Algunos de los tipos incorporados son tipos de clase, son `list`, `map` y `range`. A diferencia de los tipos personalizados, los tipos de clase integrados se pueden construir usando literales, por ejemplo, `[1, 2, 3]` es un literal de tipo `list`.

Lista

La clase List es un contenedor que proporciona soporte para tipos de datos de lista. La lista de Berry es una colección ordenada de elementos, y cada elemento de la lista tiene un índice entero único, y se puede acceder a cada elemento directamente según el índice. List admite la inserción o eliminación de elementos en cualquier posición, y el elemento puede ser de cualquier tipo. Además de usar índices, también puede usar iteradores para acceder a los elementos de la lista.

La implementación de List es una matriz dinámica y esta estructura de datos tiene un buen rendimiento de acceso aleatorio. La eficiencia de agregar y eliminar elementos al final de la lista es muy alta, pero la eficiencia de agregar y eliminar elementos en el medio de la lista es baja.

El método de inicialización literal del contenedor List es usar una lista de objetos entre corchetes y múltiples objetos separados por comas, por ejemplo:

```
[]
['string']
[0, 1, 2,'list']
```

Operaciones: ver capítulo 7.

Mapa

El mapa también es un tipo de contenedor, el mapa es una colección de pares clave-valor, y cada clave posible aparece como máximo una vez en la colección. El contenedor Mapa proporciona las siguientes operaciones básicas:

- Agregar pares clave-valor a la colección
- Eliminar pares clave-valor de la colección
- Modificar el valor correspondiente a una clave existente
- Encuentra el valor correspondiente por clave

El mapa se implementa utilizando una tabla hash y tiene una alta eficiencia de búsqueda. La operación de agregar y eliminar pares clave-valor consumirá más tiempo si se produce un "rehashing".

El contenedor Map también se puede inicializar con valores literales, escritos entre llaves para encerrar una lista de pares clave-valor, separados por dos puntos entre claves y valores, y separados por comas entre pares clave-valor. P.ej:

```
{}
{'str':'hola'}
{'str':'hola','int': 45, 78: nil}
```

Operaciones: ver capítulo 7.

Rango

El contenedor Range representa un rango de enteros, que generalmente se usa para iterar en un rango de enteros. Este tipo tiene un miembro `__lower__` y un miembro `__upper__`, que representan los límites inferior y superior del rango, respectivamente. El valor literal de Range es un par de enteros conectados mediante el operador `..`:

```
0 .. 10
-5 .. 5
```

Cuando la clase Range se usa para la iteración, los elementos de la iteración son todos valores enteros desde el límite inferior hasta el límite superior, incluidos los valores límite. Por ejemplo, el resultado de la iteración de `0..5` es:

```
0 1 2 3 4 5
```

Por tanto, cabe señalar que para un rango de *x* .. (*x*+*n*), el número de iteraciones es *n* + 1. Una construcción común para iterar a través de los elementos de una lista por elemento es:

```
for i: 0..size(l)-1
```

Rango abierto: si omite el último rango, se reemplaza implícitamente con MAXINT.

```
> r = 10..
> r
(10..9223372036854775807)
```

Bytes

El objeto Bytes denota un búfer de bytes que se puede usar para manipular búferes de bytes o para leer/escribir algunas áreas o estructuras de memoria C.

Consulte el Capítulo 7.

2.2 Variables

Una variable es un espacio de almacenamiento con un nombre, y los datos o la información almacenados en el espacio de almacenamiento se denominan valor de la variable. Los nombres de variables se utilizan para hacer referencia a las variables en el código fuente. En diferentes ámbitos, un nombre de variable puede vincular varias variables independientes, pero las variables no tienen alias. El valor de la variable se puede acceder o cambiar en cualquier momento durante la ejecución del programa. Berry es un lenguaje de tipo dinámico, por lo que el tipo de valor de la variable se determina en tiempo de ejecución y la variable puede almacenar cualquier tipo de valor.

2.2.1 Definir variables

La primera forma de definir una variable es usar una declaración de asignación para asignar un valor a un nuevo nombre de variable:

```
'var' =  expresión
```

var es el nombre de la variable, y el nombre de la variable es un identificador (consulte la sección identificador). **expresión** es la expresión para inicializar la variable.

```
a = 1
b ='str'
```

Sin embargo, este método de definición de variables tiene algunas limitaciones. Tome el siguiente código como ejemplo:

```
i = 0
do
    i = 1
```

(continues on next page)

(continued from previous page)

```
    print(i) # 1
end
print(i) # 1
```

La instrucción do en la rutina constituye el alcance interno. Modificamos el valor de la variable i en la línea 3, y el valor de i sigue siendo 1 después de dejar el alcance interno en la línea 6. Si queremos que la variable i del ámbito interno sea una variable independiente, el método de definir la variable mediante la asignación directa al nuevo nombre de variable no puede cumplir el requisito, porque el identificador i ya existe en el ámbito externo. En este caso, la variable se puede definir mediante la palabra clave var:

```
'var' variable
'var' variable = expresión
```

Hay dos formas de usar var para definir una variable: la primera es seguir el nombre de la variable **variable** después de la palabra clave var, en este caso la variable se inicializará a nil, y la otra se escribe en la variable y se inicializa al mismo tiempo que se define la variable. En este caso, se requiere una expresión de valor inicial **expresión**. Usar var para definir una variable tiene dos posibles resultados: si el alcance actual no define la variable de **variable**, definir e inicializar la variable, de lo contrario, es equivalente a reinicializar la variable. Por lo tanto, la variable definida con var protegerá a la variable con el mismo nombre en el ámbito externo.

Ahora cambiamos el ejemplo anterior para usar la palabra clave var para definir variables:

```
i = 0
do
    var i = 1
    print(i) # 1
end
print(i) # 0
```

A partir de la rutina modificada, se puede encontrar que el valor de la variable i en el ámbito interno es 1, y su valor en el ámbito externo es 0. Esto prueba que después de usar la palabra clave var, se define una nueva variable i en el ámbito interno y se bloquea la variable con el mismo nombre en el ámbito externo. Una vez que finaliza el ámbito interno, el identificador i vuelve a vincularse a la variable i en el ámbito externo.

Al usar la palabra clave var para definir una variable, también puede usar una lista de múltiples nombres de variables, separados por comas. También puede inicializar una o más variables al definir variables:

```
var a = 0, b, c ='test'
```

2.2.2 Alcance y Ciclo de Vida

Como se mencionó anteriormente, los nombres de las variables se pueden vincular a varias entidades de variables (espacios de almacenamiento) y los nombres de las variables se vinculan a una sola entidad en cada posición. La entidad vinculada por el nombre de la variable debe determinarse de acuerdo con la posición en la que aparece el nombre de la variable.

Ámbito se refiere al área de código donde el nombre y la entidad están vinculados de forma única. Fuera del alcance, el nombre puede estar vinculado a otras entidades, o no estar vinculado a ninguna entidad. La entidad solo es visible en el alcance asociado al nombre, es decir, la variable solo es válida en su alcance. Un bloque de código (ver bloque) es un alcance. Una variable solo está disponible dentro del bloque, y los nombres en diferentes bloques pueden vincular diferentes entidades variables. El siguiente ejemplo demuestra el alcance de las variables:

```
var i = 0
do
    var j ='str'
    print(i, j) # 0 str
end
#  La variable j no está disponible aquí
print(i) # 0
```

Los nombres `i` y `j` se definen en esta rutina. El nombre `i` se define fuera de la oración `do`, y el nombre definido en el bloque más externo tiene **Alcance global**. El nombre con alcance global está disponible en todo el programa después de la personalización. El nombre `j` se define en el bloque en la oración `do`, y el nombre de este tipo de definición en el bloque no exterior tiene **Ámbito local**. No se puede acceder a un nombre con un ámbito local fuera del ámbito.

Berry tiene algunos objetos integrados, que están todos en el ámbito global. Sin embargo, los objetos integrados y las variables globales definidas en los scripts no están en el mismo ámbito global. Los objetos integrados en realidad pertenecen al **Alcance integrado**. El alcance es visible globalmente como el alcance global ordinario, pero puede estar cubierto por el alcance global ordinario. Los objetos incorporados incluyen funciones y clases en la biblioteca estándar. Estos objetos incluyen funciones de `impresión`, funciones de `tipo` y clases de `mapa`. A diferencia de otros ámbitos, las variables en el ámbito integrado son de solo lectura, por lo que la "asignación" a las variables en el ámbito integrado en realidad define una variable con el mismo nombre en el ámbito global, que anula los símbolos en el ámbito en el alcance incorporado.

Alcance anidado

Ámbito anidado significa que el ámbito contiene otro ámbito. Llamamos al ámbito contenido **Ámbito interno** y al ámbito que contiene el ámbito interno **Ámbito externo**. Se puede acceder al nombre definido en el ámbito externo en todos los ámbitos internos. El ámbito interno también puede volver a vincular el nombre ya definido en el ámbito externo. El ejemplo anterior usando `var` para definir variables describe este escenario.

Ciclo de vida variable

No existe el concepto de nombres de variables cuando el programa se está ejecutando, y las variables existen en forma de entidades en este momento. El "período de validez" de una variable durante la ejecución del programa es el **ciclo de vida** de la variable. Las variables en tiempo de ejecución solo son válidas dentro del alcance. Después de salir del alcance, las variables se destruirán para recuperar recursos.

Las variables definidas en el ámbito global se denominan **Variable global** y tienen **Ciclo de vida estático**. Se puede acceder a dichas variables durante todo el programa en ejecución y no se destruirán. Las variables definidas en el ámbito local se denominan **Variable local** y tienen **Ciclo de vida dinámico**. No se puede acceder a dichas variables después de abandonar el alcance y se destruirán.

Debido a los diferentes ciclos de vida, las variables locales y las variables globales usan diferentes formas de asignar el espacio de almacenamiento. Las variables locales se asignan en una estructura llamada **Pila** (stack), y los objetos asignados en función de la pila se pueden reclamar rápidamente al final del alcance. Las variables globales se asignan en **Tabla global** (tabla global). Los objetos de la tabla global no se reciclarán una vez creados y se puede acceder a la tabla desde cualquier parte del programa.

2.2.3 Tipo de variable

Berry determina el tipo de variable en tiempo de ejecución. En otras palabras, la variable puede almacenar cualquier tipo de valor. Por lo tanto, Berry es un lenguaje de **escritura dinámica**. El intérprete no deduce el tipo de la variable en tiempo de compilación, lo que puede provocar que se expongan algunos errores en tiempo de ejecución. Por ejemplo, el error generado al ejecutar la expresión `'1' + 1` es un error de tiempo de ejecución y no un error del compilador. La ventaja de usar tipos dinámicos es que se pueden simplificar muchos diseños y el programa será más flexible, sin mencionar la necesidad de diseñar un sistema de inferencia de tipos complejo.

Debido a que el intérprete no verifica el tipo, es posible que el código de usuario deba determinar el tipo de valor por sí mismo, y esta característica también se puede usar para implementar algunas operaciones especiales. Esta característica también hace que las funciones sobrecargadas sean innecesarias. Por ejemplo, la función nativa `type` acepta cualquier tipo de parámetro y devuelve una cadena que describe el tipo de parámetro.

3. Expresiones

3.1 Conceptos básicos

Una expresión (Declaración) se compone de uno a más operandos y operadores, y se puede obtener un resultado evaluando la expresión. Este resultado se llama el valor de la expresión. El operando puede ser un valor literal, una variable, una llamada de función o una subexpresión, etc. Las expresiones y operadores simples también se pueden combinar en expresiones más complejas. Similar a las cuatro operaciones aritméticas, la precedencia de los operadores afecta el orden de evaluación de las expresiones. Cuanto mayor sea la precedencia del operador, antes se evaluará la expresión.

Operadores y expresiones

Berry proporciona algunos operadores unarios y operadores binarios. Por ejemplo, el operador AND lógico `&&` es un operador binario, y el operador de negación lógica `!` es un operador unario. Algunos operadores pueden ser operadores unarios u operadores binarios. El significado específico de tales operadores depende del contexto. Por ejemplo, el operador `-` es un símbolo unario en la expresión `-1`, pero es un signo menos binario en la expresión `1-2`.

Expresión de combinación de operadores

Tanto el lado izquierdo como el derecho de un operador binario pueden ser subexpresiones, por lo que puede usar operadores binarios para combinar expresiones. Una expresión más compleja suele tener varios operadores y operandos. En este momento, el orden de evaluación de cada subexpresión en la expresión puede afectar el valor de la expresión. La precedencia y asociatividad de los operadores garantizan la unicidad del orden de evaluación de la expresión. Por ejemplo, una expresión combinada:

```
1 + 10/2 * 3
```

Las cuatro operaciones aritméticas de uso diario calcularán primero la expresión de división `10/2`, luego la expresión de multiplicación y finalmente la expresión de suma. Esto se debe a que la multiplicación y la división tienen mayor prioridad que la suma.

Tipo de operando

En la operación de expresiones, los operandos pueden tener tipos que no coincidan con los operadores. Además, los operadores binarios normalmente requieren que los operandos izquierdo y derecho sean del mismo tipo. La expresión `'10'+10` es incorrecta. No puede agregar una cadena a un número entero. El problema con la expresión `-'b'` es que no puedes tomar un valor negativo en una cadena. A veces, un operador binario tiene diferentes tipos de operandos pero puede realizar operaciones. Por ejemplo, al agregar un número entero a un número real, el objeto entero se convertirá en un número real y se agregará a otro objeto de número real. Los operadores lógicos AND y OR lógicos permiten que los operandos a ambos lados del operador sean de cualquier tipo. En expresiones lógicas, siempre se convertirán al tipo `booleano` de acuerdo con ciertas reglas.

Otra situación es que los operadores pueden sobrecargarse al usar clases personalizadas. En esencia, puede interpretar este operador arbitrariamente y depende de usted decidir qué tipo de operando debe ser.

3.1.1 Prioridad y asociatividad

En una expresión compuesta de varios operadores, la precedencia y la asociatividad de los operadores determinan el orden de evaluación de las expresiones. La precedencia y asociatividad de cada operador se dan en la Tabla 2 .

La precedencia especifica el orden de evaluación entre diferentes operadores, y las expresiones con operadores de mayor precedencia se evaluarán primero. Por ejemplo, el proceso de evaluar la expresión `1+2*3` calculará primero el resultado de `2*3` y luego el resultado de la expresión de suma. El uso de paréntesis puede mejorar el orden de evaluación de las expresiones de baja prioridad. Por ejemplo, en la evaluación de la expresión `(1+2)*3`, primero se calcula el resultado de la expresión `1+2` entre paréntesis y luego se calcula la expresión de multiplicación fuera de los paréntesis.

La asociatividad se refiere al orden de evaluación de los operandos en ambos lados del operador, donde los operandos pueden ser subexpresiones. Por ejemplo, en la expresión de suma `expr1 + expr2`, el valor de `expr1` se calcula primero y luego el valor de `expr2`, porque el operador de suma es asociativo por la izquierda.

prioridad	Operador	Descripción	Asociatividad
1	`()`	Símbolo de agrupación	•
2	`() [] .`	Operación de campo	izquierda
3	`-! ~`	Signo negativo, negación lógica, cambio de bit	izquierda
4	`* / %`	M ultiplicación, división y resto	izquierda
5	`+ -`	Suma, resta	izquierda
6	`<< >>`	Mover a la izquierda, mover a la derecha	izquierda
7	`&`	Bit a bit Y	izquierda
8	`^`	XOR bit a bit	izquierda
9	`|`	Bit a bit O	izquierda
10	`..`	Operador de concatenación	izquierda
11	`< <= > >=`	Mayor que, mayor que o igual a	izquierda
12	`== !=`	Igual a, no igual a	izquierda
13	`&&`	Y lógico	izquierda
14	`||`	O lógico	izquierda
15	`? :`	Operador condicional	derecha
16	`&= | = ^= <<= >>=`	Asignación	derecha

Tabla 2: Lista de operadores

Utilice corchetes para aumentar la prioridad

Los paréntesis se pueden usar cuando necesitamos que los operadores con menor precedencia se evalúen primero. Durante la evaluación de expresiones, primero se calcula el valor de la expresión entre paréntesis. En otras palabras, para toda la expresión, la expresión entre paréntesis es equivalente a un operando, independientemente de la composición de la expresión entre paréntesis.

3.2 Operador

3.2.1 Operadores aritméticos

Los operadores aritméticos se utilizan para implementar operaciones aritméticas. Estos operadores son similares a los símbolos matemáticos que solemos usar. Los operadores aritméticos provistos por Berry se muestran en la Tabla 3 .

Operador	Descripción	Ejemplo
-	menos unario	`- expr`
+	Concatenación más/cadena	`expr + expr`
-	signo menos	`expr-expr`
*	Signo de multiplicación	`expr * expr`
/	Signo de división	`expr / expr`
%	Toma el resto	`expr%expr`

Tabla 3: Operador aritmético

Operador binario + Además de ser un signo más, también es una concatenación de cadenas. Cuando el operando de este operador es una cadena, la concatenación de cadenas se realizará para concatenar dos cadenas en una cadena más larga. Para ser precisos, + como concatenación de cadenas ya no está en la categoría de operadores aritméticos.

El operador binario `%` es el símbolo de resto. Sus operandos deben ser números enteros. El resultado de la operación de resto es el resto después de dividir el operando izquierdo por el operando derecho. Por ejemplo, el resultado de `11%4` es 3. El tipo de número real no puede ser divisible, por lo que no se admite el resto.

En general, los operadores aritméticos no cumplen la ley conmutativa. Por ejemplo, los valores de las expresiones `2/4` y `4/2` no son iguales.

Todos los operadores aritméticos se pueden sobrecargar en la clase. Los operadores sobrecargados no están necesariamente limitados a su diseño funcional original, sino que son determinados por el programador.

3.2.2 Operadores relacionales

Los operadores relacionales se utilizan para comparar la magnitud de los operandos. Los seis operadores relacionales soportados por Berry se dan en la Tabla 4 .

Operador	Descripción	Ejemplo
<	Menor que	`expr < expr`
<=	Menor o igual que	`expr <= expr`
==	Igual	`expr == expr`
!=	No es igual a	`expr != expr`
>=	Mayor o igual a	`expr >= expr`
>	Mayor qu e	`-expr`

Tabla 4: Operador relacional

Al comparar la relación de magnitud de los operandos o juzgar si los operandos son iguales, la evaluación de la expresión relacional producirá un resultado booleano. Cuando se cumple la relación, el valor de la expresión de la relación es "verdadero", de lo contrario, es "falso". Los operadores relacionales == y `!=` pueden usar cualquier tipo de operando y permiten que los operandos izquierdo y derecho tengan diferentes tipos. Otros operadores relacionales permiten el uso de las siguientes combinaciones de operandos:

integer relop **integer**

real relop **real**

integer relop **real**

real relop **integer**

string relop **string**

En operaciones relacionales, el signo igual == y el signo de desigualdad `!=` satisfacen la ley conmutativa. Si los operandos izquierdo y derecho son del mismo tipo o ambos son de tipo numérico (número entero y número real), los operandos se consideran iguales según el valor de los operandos; de lo contrario, los operandos se consideran desiguales. La igualdad y la desigualdad son operaciones recíprocas: si `a==b` es verdadero, entonces `a!=b` es falso, y viceversa. Otros operadores relacionales no satisfacen la ley conmutativa, pero tienen las siguientes propiedades: < y >= son operaciones recíprocas, y > y <= son operaciones recíprocas. Las operaciones relacionales requieren que los operandos sean del mismo tipo, de lo contrario es una expresión incorrecta.

Las instancias pueden sobrecargar a los operadores como métodos. Si el operador relacional está sobrecargado, el programa debe garantizar las propiedades anteriores.

Entre los operadores relacionales, los operadores == y `!=` tienen requisitos más relajados que <, <=, > y >=, que solo permiten comparaciones entre los mismos tipos. En el desarrollo de un programa real, el juicio de igualdad o desigualdad suele ser más simple que el juicio de tamaño. Es posible que algunos objetos de operación no puedan juzgar el tamaño, pero solo pueden juzgar la igualdad o desigualdad. Este es el caso del tipo booleano.

Operadores logicos

Los operadores lógicos se dividen en tres tipos: AND lógico, OR lógico y NOT lógico. Como se muestra en la Tabla 5 .

Operador	Descripción	Ejemplo
&&	Y lógico	`expr && expr`
||	O lógico	`expr|| expr`
!	Negación lógica	`!expr`

Tabla 5: Operadores logicos

Para el operador lógico AND, cuando los valores de ambos operandos son "verdaderos", el valor de la expresión lógica es "verdadero", de lo contrario, es "falso".

Para el operador lógico OR, cuando los valores de ambos operandos son `falso`, el valor de la expresión lógica es `falso`, de lo contrario es `verdadero`.

El papel del operador de negación lógica es cambiar el estado lógico del operando. Cuando el valor del operando es "verdadero", el valor de la expresión lógica es "falso", de lo contrario, el valor es "verdadero".

Los operadores lógicos requieren que el operando sea de tipo booleano, y si el operando no es de tipo booleano, se convertirá. Consulte la sección [Capitulo-2: Tipo booleano] para conocer las reglas de conversión.

Las operaciones lógicas utilizan una estrategia de evaluación llamada **Evaluación de cortocircuito**. Esta estrategia de evaluación es: para el operador lógico AND, el segundo operando se evaluará si y solo si el operando de la izquierda es verdadero; para el operador lógico OR, si y solo si el operando izquierdo es falso evaluará el operando derecho. La naturaleza de la evaluación de cortocircuito hace que no se ejecute todo el código de la expresión lógica.

Operador bit a bit

Los operadores de bits pueden implementar algunas operaciones de bits binarios, y las operaciones de bits solo se pueden usar en tipos enteros. La información detallada de los operadores de bit se muestra en la Tabla 6 . La operación de bits se refiere a la operación de bits binarios directamente en números enteros. Las operaciones lógicas se pueden extender a operaciones de bits. Tomando AND lógico como ejemplo, podemos realizar esta operación en cada bit binario para lograr AND bit a bit, como 110_b AND 011_b = 010_b. Las operaciones de bits también admiten operaciones de cambio, que mueven números de forma binaria.

Operador		Ejemplo
~	Negar	`~expr`
&	Bit a bit y	`expr & expr`
|	Bit a bit o	`expr| expr`
^	O exclusivo bit a bit	`expr^expr`
<<	Desplazar a la izquierda	`expr << expr`
>>	Desplazar a la derecha	`expr >> expr`

Tabla 6: Operador bit a bit

Aunque solo se puede usar para números enteros, las operaciones con bits siguen siendo versátiles. Las operaciones de bits pueden implementar muchas técnicas de optimización. En muchos algoritmos, el uso de operaciones de bits puede ahorrar mucho código. Por ejemplo, para determinar si un número `n` es una potencia de 2, podemos juzgar si el resultado de `n & (n - 1)` es `0`. En algunos lenguajes con alta eficiencia de ejecución, las operaciones de cambio también se pueden usar para optimizar la multiplicación y la división (por lo general, no hay un efecto obvio en los lenguajes de secuencias de comandos).

El operador AND bit a bit "`&`" es un operador binario, que realiza la operación AND binaria de dos operandos enteros: solo cuando los bits binarios correspondientes a los operandos son todos `1`, el resultado es `1`. Por ejemplo, 1110_b & 0111_b =0110_b.

El operador OR bit a bit "`|`" es un operador binario, que realiza una operación OR de bits binarios en dos operandos enteros: solo cuando los bits binarios correspondientes a los operandos son ambos `0`, el bit del resultado es `0`. Por ejemplo, 1000_b | 0001_b=1001_b.

El operador OR exclusivo bit a bit "`^`" es un operador binario, que realiza una operación OR exclusiva binaria en dos operandos enteros: cuando los bits binarios correspondientes a los operandos son diferentes, el valor de bit del resultado es 1. Por ejemplo, 1100_b ^ 0101_b = 1001_b.

El operador de desplazamiento a la izquierda "<<" es un operador binario, que mueve el operando izquierdo hacia la izquierda el número de bits especificado por el operando derecho sobre una base binaria. Por ejemplo, 00001010_b <<3=01010000_b. El operador de desplazamiento a la derecha ">>" es un operador binario, que desplaza el operando izquierdo hacia la derecha el número de bits especificado por el operando derecho en un binario. base. Por ejemplo, 10100000_b >>3=00010100_b.

El operador de inversión bit a bit "~" es un operador unario, y el resultado de la expresión es invertir el valor de cada bit binario del operando. Por ejemplo, 10100011_b=01011100_b.

Los siguientes son algunos ejemplos del uso de operaciones con bits. Por lo general, no usamos binario directamente. Los resultados de los ejemplos se han convertido en bases comunes.

```
1 << 1 # 2
168 >> 4 # 10
456 & 127 # 72
456 | 127 # 511
0xA5 ^ 0x5A # 255
~2 # -3
```

Operador de asignación

El operador de asignación solo aparece en la expresión de asignación y el operando del operador debe ser un objeto de escritura. La expresión de asignación no tiene resultado, por lo que no se pueden utilizar operaciones de asignación continua.

Operador de asignación simple

El operador de asignación simple = se puede utilizar para la asignación de variables. Si la variable de operando de la izquierda no está definida, se definirá la variable. El operador de asignación se utiliza para vincular el valor del operando derecho con el operando izquierdo. Este proceso también se llama "asignación". Por lo tanto, el operando izquierdo no puede ser una constante, ni puede ser ningún objeto que no se pueda escribir. Estas son algunas expresiones legales de asignación:

```
a = 45 b ='string' c = 0
```

Y la siguiente expresión de asignación es incorrecta:

```
1 = 5 # Tratando de asignar una constante 1
a = b = 0 #  Asignación continua
```

Al asignar tipos `nil`, enteros, reales y booleanos a variables, el valor del objeto se pasará al operando izquierdo, pero para otros tipos, la operación de asignación simplemente pasa la referencia del objeto al operando izquierdo. Dado que las cadenas, las funciones y los tipos de clase son de solo lectura, todas las referencias que pasan no tendrán efectos secundarios, pero debe tener mucho cuidado con los tipos de instancia.

Operador de asignación compuesto

Los operadores de asignación compuestos son operadores que combinan operadores binarios y operadores de asignación. Son extensiones prácticas de operadores de asignación simples. Los operadores de asignación compuestos pueden simplificar la escritura de algunas expresiones. La Tabla 7 enumera todos los operadores de asignación compuestos

Operador	Descripción
+=	Asignación de adición
-=	Asignación de resta
*=	Asignación de multiplicación
/=	Asignación de división
%=	Asignación de resto
&=	Asignación AND bit a bit
|=	Asignación OR bit a bit
^=	Asignación XOR bit a bit
<<=	Asignación de desplazamiento a la izquierda
>>=	Asignación de desplazamiento a la derecha

Tabla 7: Operador de bits

La expresión de asignación compuesta realiza la operación binaria correspondiente al operador de asignación compuesta en el operando izquierdo y el operando derecho, y luego asigna el resultado al operando izquierdo. Tomando += como ejemplo, la expresión `a += b` es equivalente a `a = a + b`. El operador de asignación compuesto también es un operador de asignación, por lo que tiene una prioridad más baja. El operador binario correspondiente al operador de asignación compuesto siempre se evalúa después del operando derecho, por lo que una expresión como `a *= 1 + 2` debería ser equivalente a `a = a * (1 + 2)`.

A diferencia del operador de asignación simple, el operando izquierdo del operador de asignación compuesto debe participar en la evaluación, por lo que la expresión de asignación compuesta no tiene la función de definir variables. El operador de asignación en sí no se puede sobrecargar en la clase. Los usuarios solo pueden sobrecargar el operador binario correspondiente al operador de asignación compuesto. Esto también asegura que el operador de asignación compuesto siempre se ajustará a las características básicas de las operaciones de asignación.

Operador de dominio y operador de subíndice

El operador de dominio `.` se utiliza para acceder a un atributo o miembro de un objeto. Puede utilizar operadores de dominio para ambos tipos de módulos e instancias:

```
l = list[]
l.push('item 0')
s = l.item(0) # 'item 0'
```

El operador de subíndice `[]` se utiliza para acceder a los elementos de un objeto, por ejemplo

```
l[2] = 10 # Read by index
n = l[2] # Write by index
```

Las clases que admiten la lectura de subíndices deben implementar el método `item` y las clases que admiten la escritura de subíndices deben implementar el método `setitem`. El mapa y la lista en el contenedor estándar implementan estos dos métodos, por lo que admiten la lectura y escritura mediante el operador de subíndice. La cadena admite la lectura de subíndices, pero no admite la escritura de subíndices (las cadenas son valores de solo lectura):

```
'string'[2] #'r'
'string'[2] ='a' # error:  valor 'string' no admite asignación de índice
```

Actualmente, las cadenas admiten subíndices enteros y el rango de subíndices no puede exceder la longitud de la cadena.

Operador condicional

El operador condicional (`? :`) es similar a la declaración **if else**, pero la primera puede usarse en expresiones. La forma de uso del operador condicional es:

```
cond ? expr1 : expr2
```

cond es la expresión utilizada para juzgar la condición. El proceso de evaluación del operador condicional es: primero encuentra el valor de **cond**, si la condición es verdadera, evalúa **expr1** y devuelve el valor, de lo contrario, el valor de **expr2**] Evalúa y devuelve el valor. **expr1** y **expr2** pueden tener diferentes tipos, por lo que lo siguiente es correcto:

```
resultado = alcance < 6 ? 'malo' : alcance
```

Esta expresión primero determina si `alcance` es menor que `6`, si lo es, devuelve `malo`, de lo contrario, devuelve el valor de `alcance`. Independientemente de la condición de la expresión condicional, solo se ejecutará uno de **expr1** o **expr2**, similar a la característica de cortocircuito de las operaciones lógicas AND y lógicas OR.

Operadores de condiciones anidadas

Un operador condicional se puede anidar en otro operador condicional, es decir, la expresión condicional se puede usar como **cond** o **expr** de otra expresión condicional. Por ejemplo, use expresiones condicionales para dividir puntajes en tres niveles: excelente, bueno y malo:

```
resultado = alcance >= 9 ? 'excelente' : alcance >= 6 ? 'bueno' : 'malo'
```

La primera condición comprueba si la puntuación no es inferior a `9` puntos. Si es así, ejecute la rama después de `?` y devuelva `'excelente'`; de lo contrario, ejecute la rama después de `:`, que también es una expresión condicional. La condición comprueba si la puntuación no es inferior a `6`, si lo es, devuelve `'bueno'`, de lo contrario, devuelve `'malo'`.

El operador condicional satisface la asociatividad correcta, por lo que el valor de la expresión de bifurcación debe evaluarse primero para obtener el valor de la expresión condicional. Por lo tanto, en una expresión condicional anidada, la expresión condicional anidada se evalúa primero y luego se evalúa la expresión condicional externa.

Prioridad de los operadores condicionales

Dado que la precedencia de las expresiones condicionales es muy baja (sólo superada por los operadores de asignación), a menudo es necesario agregar paréntesis fuera de las expresiones condicionales. Por ejemplo, cuando se usa una expresión condicional como operando de una expresión aritmética, los paréntesis tendrán diferentes efectos en el resultado:

```
resultado = 10 * (signo < 0 ? -1 : 1) # el resultado es -10 ó 10
resultado = 10 * signo < 0 ? -1 : 1 # el resultado es -1 ó 1
```

El resultado de la primera expresión es correcto, y la segunda expresión toma `10 * signo < 0` como condición a juzgar, lo que no cumple con la expectativa de la expresión condicional como el operando derecho de la multiplicación.

Operador de concatenación

Operador +

Cuando los operandos izquierdo y derecho son cadenas, el operador + se usa para conectar las dos cadenas, y la nueva cadena obtenida es el valor de la expresión. Por lo tanto, este operador se usa a menudo para la concatenación de cadenas:

```
resultado ='abc' + '123' # el resultado es 'abc123'
```

Los operadores + también se pueden usar para conectar dos instancias de lista:

```
resultado = [1, 2] + [3, 4] # el resultado es [1, 2, 3, 4]
```

A diferencia del método `list.push`, el operador + fusiona dos listas en un objeto de lista más grande, con los elementos del operando izquierdo al principio de la lista de resultados y los elementos del operando derecho al final de la lista de resultados

Operador `..`

`..` es un operador especial. Si el operando izquierdo es una cadena, el comportamiento de la expresión es concatenar los operandos izquierdo y derecho en una nueva cadena (conversión automática si el operando derecho no es una cadena):

```
resultado ='abc' .. 123 # el resultado es 'abc123'
```

El operador `..` se usa a menudo cuando se concatena una cadena y un valor que no es una cadena. Si el operando izquierdo es una instancia de lista, el operador `..` agregará el operando derecho al final de la lista y luego usará esta lista como el valor de la expresión:

```
resultado = [1, 2] .. 3 # el resultado es [1, 2, 3]
```

Este proceso modificará directamente el operando izquierdo, que es muy similar al método `push` de `list` (excepto las cadenas que son objetos inmutables). La operación de unión de la lista se puede ejecutar en cadena:

```
resultado = [1, 2] .. 3 .. 4 # el resultado es [1, 2, 3, 4]
```

Todos los valores en este proceso se agregarán al objeto de lista más a la izquierda.

Si los operandos izquierdo y derecho son enteros, utilice el operador `..` para obtener un objeto de rango de enteros:

```
resultado = 1 .. 10 # el resultado es (1..10)
```

Este objeto se utiliza para representar un intervalo cerrado de enteros, donde el operando izquierdo es el límite inferior y el operando derecho es el límite superior. Dichos objetos de rango de enteros se utilizan a menudo para la iteración.

4. Declaración

Berry es un lenguaje de programación imperativo. Este paradigma asume que los programas se ejecutan paso a paso. Normalmente, las declaraciones de Berry se ejecutan secuencialmente, y esta estructura de programa se denomina estructura secuencial. Aunque la estructura de la secuencia es muy básica, las estructuras de rama y las estructuras de bucle se utilizan normalmente en los programas reales. Berry proporciona varias declaraciones de control para realizar esta compleja estructura de flujo, como declaraciones condicionales y declaraciones de iteración.

A excepción de los comentarios de línea, los retornos de carro o los saltos de línea ("\r" y "\n") solo se usan como caracteres en blanco, por lo que las declaraciones se pueden escribir en líneas. Además, puede escribir varias declaraciones en la misma línea.

Puede agregar un punto y coma al final de la declaración para indicar el final de la declaración, pero el intérprete generalmente puede dividir la declaración automáticamente sin usar un punto y coma. Puede usar punto y coma para decirle al intérprete cómo analizar el código para el código que será ambiguo. Sin embargo, es mejor no escribir código ambiguo.

4.1 Oración simple

4.1.1 Declaración de expresión

Las declaraciones de expresión son principalmente declaraciones compuestas de expresiones de asignación o expresiones de llamada de función. Otras expresiones también pueden formar oraciones, pero no tienen significado. Por ejemplo, la expresión 1+2 es una oración escrita sola, pero no tiene ningún efecto. Las siguientes rutinas dan ejemplos de sentencias de expresión y sentencias de función:

```
a = 1 #  Declaración de asignación
print(a) #  Declaración de llamada
```

La línea 1 es una declaración de asignación simple que asigna el valor literal `i` a la variable `a`. La declaración en la línea 2 es una declaración de llamada de función, que imprime el valor de la variable 'a' llamando a la función 'imprimir'.

Las expresiones de líneas cruzadas se escriben de la misma manera que las expresiones de una sola línea y no se requieren símbolos especiales de continuación de línea. P.ej:

```
a = 1 +
    func() # Ajustar línea
```

También puede escribir varias declaraciones de expresión en una línea y varios tipos de declaraciones se pueden escribir en una línea. Este ejemplo pone dos declaraciones de expresión en la misma línea:

```
b = 1 c = 2 #  sentencias múltiples
```

A veces el programador quiere escribir dos declaraciones, pero el intérprete puede pensar erróneamente que es una declaración. Este problema es causado por la ambigüedad en el proceso de análisis gramatical. Tome este código como ejemplo:

```
a = c
(b) = 1 #  Considérese como una llamada de función
```

Supongamos que las líneas 1 y 2 están destinadas a ser dos oraciones de expresión: `a = c` y `(b) = 1`, pero el intérprete las interpretará como una oración: `a = c(b) = 1`. La causa de este problema es que el intérprete analiza incorrectamente `c` y `(b)` en llamadas de función. Para evitar ambigüedades, podemos agregar un punto y coma al final de la declaración para separar claramente la declaración:

```
a = c; (b) = 1;
```

Una mejor manera es no usar paréntesis en el lado izquierdo del número de tarea. Obviamente, no hay razón para usar paréntesis aquí. En circunstancias normales, las expresiones complejas no deberían aparecer en el lado izquierdo del operador de asignación, sino solo expresiones simples compuestas de nombres de variables, expresiones de operación de dominio y expresiones de operación de subíndice:

```
a = c b = 1
```

Usar expresiones simples solo en el lado izquierdo del signo de asignación no causará ambigüedad en la segmentación de oraciones. Por lo tanto, en la mayoría de los casos, no es necesario usar punto y coma para separar expresiones y no recomendamos esta forma de escritura.

Bloque

Un **Bloque** es una colección de varias oraciones. Un bloque es un alcance, por lo que solo se puede acceder a las variables definidas en el bloque dentro del bloque y sus sub-bloques. Hay muchos lugares donde se utilizan bloques, como declaraciones `if`, declaraciones `while`, declaraciones de funciones, etc. Estas declaraciones contendrán un bloque a través de un par de palabras clave. Por ejemplo, el bloque utilizado en la sentencia `if`:

```
if isOpen
    close()
    print('el dispositivo fue cerrado')
end
```

Las sentencias en las líneas 2 a 3 constituyen un bloque, que se intercala entre el par de palabras clave `if` y `end` (la expresión condicional de la sentencia en `if` no está en el bloque). No es necesario que el bloque contenga declaraciones, lo que constituye un bloque vacío, o se puede decir que es un bloque que contiene una declaración vacía. En términos generales, cualquier cantidad de oraciones consecutivas puede llamarse bloque, pero preferimos expandir el alcance del bloque tanto como sea posible, lo que puede garantizar que el área del bloque sea consistente con el alcance del alcance. En el ejemplo anterior, tendemos a pensar que las filas 2 a 3 son un bloque completo, que es el rango más grande entre las palabras clave `if` y `end`.

Declaración do

A veces solo queremos abrir un nuevo ámbito, pero no queremos usar ninguna declaración de control. En este caso, podemos usar la instrucción `do` para encapsular el bloque, entonces la sentencia no tiene función de control. La oración tiene la forma:

`do` **bloque** `end`

Entre ellos **bloque** está el bloque que necesitamos. Esta instrucción utiliza un par de palabras clave `do` y `end` para contener bloques. La declaración no tiene función de control, ni genera ninguna instrucción de tiempo de ejecución.

Sentencia condicional

Berry proporciona sentencias `if` para realizar la función de ejecución de control condicional. Este tipo de estructura de programa generalmente se denomina estructura de rama `if`. La declaración determinará la rama de ejecución basada en la expresión condicional verdadera (`true`) o falsa (`false`). En algunos lenguajes, existen otras opciones para implementar el control condicional. Por ejemplo, los lenguajes como C y C++ proporcionan sentencias `switch`, pero para simplificar el diseño, Berry no admite sentencias `switch`.

Declaración `if`

La instrucción `if` se utiliza para implementar la estructura de rama, que selecciona la rama del programa de acuerdo con el verdadero o falso de una determinada condición de juicio. La sentencia también puede incluir la rama `else` o la rama `elif`. La forma simple de declaración `if` sin ramas es

`if` **condición** **bloque** `end`

condición es una expresión condicional. Cuando el valor de **condición* es ``verdadero``, se ejecutará**bloque** en la segunda línea; de lo contrario, se omitirá el **bloque** y se ejecutará la instrucción que sigue a `end`. En el caso de que se ejecute **bloque**, después de que se ejecute la última declaración en el bloque, dejará la declaración `if` y comenzará a ejecutar la declaración que sigue a `end`.

Aquí hay un ejemplo para ilustrar el uso de la sentencia `if`:

```
if 8 % 2 == 0
    print('este número es par')
end
```

Este código se usa para juzgar si el número '8' es par y, si lo es, generará 'este número es par'. Aunque este ejemplo es muy simple, es suficiente para ilustrar el uso básico de las oraciones `if`.

Si desea tener una rama correspondiente para la ejecución cuando la condición se cumple y no se cumple, use la instrucción `if` con la rama `else`. La forma de la oración es:

`if` **condición** **bloque** `else` **bloque** `end`

A diferencia de la simple instrucción `if`, la declaración `if else` ejecutará **bloque** en la rama `else` cuando el valor de **condición** sea `falso`. No importa qué rama se ejecute bajo **bloque**, después de que se ejecute la última declaración en el bloque, aparecerá la declaración `if else`, es decir, se ejecutará la declaración después de `end`. En otras palabras, no importa si el valor de **condición** es `verdadero` o `falso`, se ejecutará un **bloque**.

Continúe usando el juicio de paridad como ejemplo, esta vez cambie la demanda para generar la información correspondiente de acuerdo con la paridad del número de entrada. El código para lograr este requisito es:

```
if x % 2 == 0
    print('este número es par')
else
    print('este número es impar')
end
```

Antes de ejecutar este código, primero debemos asignar un valor entero a la variable `x`, que es el número cuya paridad queremos comprobar. Si 'x' es un número par, el programa generará 'este número es par'; de lo contrario, generará 'este número es impar'. A veces necesitamos anidar declaraciones `if`. Una forma es anidar una instrucción `if` debajo de la rama `else`. Este es un requisito muy común porque muchas condiciones deben juzgarse consecutivamente. Para este tipo de demanda, use la instrucción `if else` para escribir:

```
if expr
    bloque
else
    if expr
        bloque
    end
end
```

Obviamente, esta forma de escribir aumentará el nivel de sangría del código, y es más engorroso usar múltiples `end` al final. Como mejora, Berry proporciona la rama `elif` para optimizar la escritura anterior. Usar la rama `elif` es equivalente al código anterior, en la forma

`if` **condición bloque** `elif` **condición bloque** `else` **bloque** `end`

La rama debe usarse después de la rama `if` y antes de la rama, y la rama `elif` se puede usar varias veces seguidas. Si se cumple la **condición** correspondiente a la rama `elif`, se ejecutará el **bloque** debajo de la rama. La ramificación `elif` es adecuada para situaciones que requieren que se juzguen múltiples condiciones en secuencia.

Usamos un fragmento de código que juzga positivo, negativo y 0 para demostrar la rama `elif`:

```
if x> 0
    print('positivo')
elif x == 0
    print('cero')
else
    print('negativo')
end
```

Aquí también, la variable `x` debe asignarse primero. Este código es muy simple y no será explicado.

Algunos lenguajes tienen un problema llamado "`else`" colgante, que se refiere a cuando una oración `if` está anidada dentro de otra oración `if`, ¿a dónde pertenece la rama `else`? Es un problema con la sentencia `if`. Cuando usamos C/C++, debemos considerar el problema de colgar `else`. Para evitar la ambigüedad en el problema de `if else`, los programadores de C/C++ a menudo usan llaves para convertir una rama en un bloque. En Berry, la rama de la instrucción `if` debe ser un bloque, lo que también determina que Berry no tiene el problema de sobresalir por `else`.

Declaración de iteración

Las declaraciones iterativas también se denominan declaraciones de bucle, que se utilizan para repetir ciertas operaciones hasta que se cumple la condición de terminación. Berry proporciona las declaraciones `while` y `for`, dos declaraciones de iteración. Muchos lenguajes también proporcionan estas dos declaraciones para la iteración. La declaración `while` de Berry es similar a la declaración `while` en C/C++, pero la declaración `for` de Berry solo se usa para recorrer los elementos en el contenedor, similar a la declaración `foreach` proporcionada por algunos lenguajes y la que se introdujo por el nuevo estilo en C++11 de `for`. No se admite la instrucción `for` de estilo C.

Sentencia `while`

La declaración `while` es una declaración iterativa básica. La instrucción `while` utiliza una condición de juicio. Cuando la condición es verdadera, el cuerpo del ciclo se ejecuta repetidamente; de lo contrario, el ciclo finaliza. El patrón de la declaración es

`while` **condición** **bloque** `end`

Cuando el programa ejecuta la sentencia `while`, comprobará si la expresión **condición** es verdadera o falsa. Si es cierto, ejecuta el **bloque** del cuerpo del ciclo; de lo contrario, finaliza el ciclo. Después de ejecutar la última declaración en **bloque**, el programa saltará al comienzo de la declaración `while` y comenzará la siguiente ronda de detección. Si la expresión de **condición** es falsa cuando se evalúa por primera vez, el **bloque** del cuerpo del bucle no se ejecutará en absoluto (al igual que la expresión de **condición** de la declaración `if` es falsa). En términos generales, el valor de la expresión **condición** debería poder cambiar durante el ciclo, en lugar de ser una constante o una variable modificada fuera del ciclo, lo que hará que el ciclo no se ejecute o no termine. Un bucle que nunca termina se llama bucle sin fin. Por lo general, esperamos que el ciclo se ejecute un número específico de veces y luego termine. Por ejemplo, cuando usamos el bucle `while` para acceder a todos los elementos de la matriz, esperamos que el número de ejecuciones del bucle sea igual a la longitud de la matriz, por ejemplo:

```
i = 0
l = ['a','b','c']
while i < l.size()
    print(l[i])
    i = i + 1
end
```

Este bucle obtiene los elementos del arreglo `l` y los imprime. Usamos una variable `i` como contador de bucles e índice de matriz. Dejamos que el valor de `i` alcance la longitud de la matriz `l` para finalizar el bucle. En la última línea del cuerpo del bucle, añadimos `1` al valor de `i` para asegurar que se acceda al siguiente elemento de la matriz en el siguiente bucle, y el bucle `while` finaliza cuando el número de bucles alcanza la longitud de la matriz.

Sentencia `for`

La instrucción `for` de Berry se usa para recorrer los elementos en el contenedor, y su forma es

`for` **variable : expresión bloque** `end`

expresión El valor de la expresión debe ser un contenedor iterable o una función, como la clase `range`. La declaración obtiene un iterador del contenedor y obtiene un elemento en el contenedor cada vez que se llama al iterador.

variable se denomina variable de iteración, que siempre se define en la instrucción `for`. Por lo tanto, **variable** debe ser un nombre de variable y no una expresión. El elemento contenedor obtenido del iterador en cada bucle se asignará a la variable de iteración. Este proceso ocurre antes de la primera declaración en **bloque**.

La declaración `for` verificará si hay elementos no visitados en el iterador para la iteración. Si los hay, comenzará la siguiente iteración; de lo contrario, finalizará la declaración `for` y ejecutará la declaración que sigue a `end`. Actualmente, Berry solo proporciona iteradores de solo lectura, lo que significa que los elementos del contenedor no se pueden modificar a través de las variables de iteración en la instrucción `for`.

El alcance de la variable de iteración **variable** se limita al **bloque** del cuerpo del ciclo, y la variable no tendrá ninguna relación con la variable con el mismo nombre fuera del alcance. Para ilustrar este punto, usemos un ejemplo para ilustrar. En este ejemplo, usamos la instrucción `for` para acceder a todos los elementos en la instancia `range` e imprimirlos. Por supuesto, también usamos este ejemplo para demostrar el alcance de las variables de bucle.

```
i = "Hola, estoy bien". # Variable exterior
for i: 0 .. 2
```

(continues on next page)

(continued from previous page)

```
    print(i) #  variable de iteración
end
print(i)
```

En este ejemplo, en relación con la variable de iteración `i` definida en la línea 2, la variable `i` definida en la línea 1 es una variable externa. Al ejecutar este ejemplo obtendrá el siguiente resultado

```
0 1 2 Hola, estoy bien
```

Se puede ver que la variable de iteración `i` y la variable externa `i` son dos variables diferentes. Solo tienen el mismo nombre pero diferentes alcances.

Principio de enunciado `for`

A diferencia de la sentencia iterativa tradicional `while`, la sentencia `for` utiliza iteradores para atravesar el contenedor. Si necesita usar la declaración `for` para atravesar una clase personalizada, debe comprender su mecanismo de implementación. Cuando se usa la instrucción `for`, el intérprete oculta muchos detalles de implementación. De hecho, para dicho código:

```
for i: 0 .. 2
    print(i)
end
```

Será traducido al siguiente código equivalente por el intérprete:

```
var it = __iterator__(0 .. 2)
try
    while true
        var i = it()
        print(i)
    end
except 'stop_iteration'
    #  no hacer nada
end
```

Hasta cierto punto, la declaración `for` es solo un azúcar sintáctico, y es esencialmente solo una forma simple de escribir una pieza de código complejo. En este código equivalente se usa una variable intermedia `it`. El valor de la variable es un iterador y, en este ejemplo, es un iterador del contenedor `range` `0..2`. Al procesar la instrucción `for`, el intérprete oculta la variable intermedia del iterador, por lo que no se puede acceder a ella en el código.

El parámetro de la función `__iterator__` es un contenedor y la función devuelve un iterador de parámetros. Esta función obtiene el iterador llamando al método de parámetro. Por lo tanto, si el valor de retorno del método `iter` es un tipo de instancia (`instance`), esta instancia debe tener un método `next` y un método `hasnext`.

El parámetro de la función `__hasnext__` es un iterador, que comprueba si el iterador tiene el siguiente elemento llamando al método `hasnext` del iterador `hasnext` El valor de retorno del método es de tipo `booleano`. El parámetro de la función `__next__` también es un iterador, que obtiene el siguiente elemento en el iterador llamando al método `next` del iterador.

Hasta ahora, las funciones `__iterator__`, `__hasnext__` y `__next__` simplemente llaman a algunos métodos del contenedor o iterador y luego devuelven el valor de retorno de estos métodos. Por lo tanto, la escritura equivalente de la instrucción `for` también se puede simplificar de esta forma:

```
do
    var it = (0 .. 2).iter()
    while (it.hasnext())
        var i = it.next()
        print(i)
    end
end
```

Este código es más fácil de leer. Se puede ver en el código que el alcance de la variable iteradora `it` es la declaración `for` completa, pero no es visible fuera de la declaración `for`, mientras que el alcance de la variable de iteración `i` está en el cuerpo del bucle, por lo que cada iteración definirá nuevas variables de iteración.

Declaración de salto

La declaración de salto proporcionada por Berry se usa para realizar el salto del flujo del programa en el proceso de bucle. Las sentencias de salto se dividen en sentencias de "ruptura" y sentencias de "continuación". Estas dos declaraciones deben usarse dentro de declaraciones iterativas y solo pueden usarse dentro de funciones para saltar. Algunos lenguajes proporcionan sentencias `goto` para realizar saltos arbitrarios dentro de las funciones, que Berry no proporciona, pero los efectos de las sentencias `goto` se pueden reemplazar por sentencias condicionales y sentencias de iteración.

Declaración `break`

`break` se usa para terminar la declaración de iteración y saltar. Después de la ejecución de la sentencia `break`, el nivel más cercano de la sentencia de iteración terminará inmediatamente y la ejecución continuará desde la posición de la primera sentencia después de la sentencia de iteración. Para ilustrar el flujo de ejecución de la declaración `break`, usamos un ejemplo para demostrarlo:

```
while true
    print('antes del break')
    break
    print('después del break')
end
print('fuera del bucle')
```

En este código, la sentencia `break` está en un bucle `while`. Antes y después de la declaración `break` y después de la declaración `while`, hemos colocado una declaración de impresión para probar el flujo de ejecución del programa. El resultado de este código es:

```
antes del break
fuera del bucle
```

Esto muestra que la sentencia `while` finaliza el bucle en la posición de la sentencia `break` en la tercera línea y el programa continúa ejecutándose desde la sexta línea.

Declaración `continue`

Esta declaración también se usa dentro de una declaración de iteración. Su función es finalizar una iteración e iniciar inmediatamente la siguiente ronda. Por lo tanto, después de la ejecución de la sentencia `continue`, el código restante en la sentencia de iteración de la capa más cercana ya no se ejecutará, pero comenzará una nueva ronda de iteración. Aquí usamos una sentencia `for` para demostrar la función de la sentencia `continue`:

```
for i: 0 .. 5
    if i >= 2
        continue
    end
    print('i =', i)
end
print('fuera del bucle')
```

Aquí, la instrucción `for` iterará 6 veces. Cuando la variable de iteración `i` es mayor o igual que `2`, se ejecutará la declaración `continue` en la línea 3, y la declaración de impresión en la línea 5 no se ejecutará a partir de entonces. En otras palabras, la línea 5 solo se ejecutará en las dos primeras iteraciones (en este momento `i < 2`). El resultado de ejecución de esta rutina es:

```
i = 0
i = 1
fuera del bucle
```

Se puede ver que el valor de la variable `i` solo se imprime dos veces, lo cual está en línea con las expectativas. Los lectores pueden intentar imprimir el valor de la variable `i` antes de la instrucción `continue`. Encontrará que la declaración `for` itera 6 veces, lo que indica que la declaración `continue` no finaliza la iteración.

Declaración `import`

Berry tiene algunos módulos predefinidos, como el módulo `math` para cálculos matemáticos. Estos módulos no se pueden usar directamente, sino que se deben importar con la instrucción `import`. Hay dos formas de importar un módulo:

`import` **nombre**

`import` **nombre** `as` **variable**

nombre Para importar el nombre del módulo, al usar el primer método de escritura para importar el módulo, el módulo importado se puede llamar directamente usando el nombre del módulo. La segunda forma de escribir es importar un módulo llamado **nombre** y modificar el nombre del módulo al llamarlo a **variable**. Por ejemplo, un módulo llamado `math`, usamos el primer método para importar y usar:

```
import math
math.sin(0)
```

Aquí usa directamente `math` para llamar al módulo. Si el nombre de un módulo es relativamente largo y no es conveniente escribirlo, puede usar la instrucción `import as`. Aquí, asuma un módulo llamado `hardware`. Queremos llamar a la función `setled` del módulo, podemos importar el `hardware` del módulo a la variable llamada `hw` y usar:

```
import hardware as hw
hw.setled(true)
```

Para encontrar módulos, todas las rutas en `sys.path()` se exploran secuencialmente. Si desea agregar una ruta específica antes de la importación (como leer desde la tarjeta SD), puede usar la siguiente función de ayuda:

```
def push_path(p)
  import sys
  var path = sys.path()
  if path.find(p) == nil  # agregar solo si aún no está allí
    path.push(p)
  end
end
```

Manejo de excepciones

El mecanismo permite que el programa capture y maneje las excepciones que ocurren durante el tiempo de ejecución. Berry admite un mecanismo de captura de excepciones que permite separar el proceso de captura y manejo de excepciones. Es decir, parte del programa se usa para detectar y recopilar excepciones, y la otra parte del programa se usa para manejar excepciones.

En primer lugar, el programa problemático necesita lanzar primero una excepción. Cuando estos programas están en un bloque de manejo de excepciones, un programa específico atrapará y manejará la excepción.

Generar una excepción

El uso de la instrucción `raise` genera una excepción `raise`. La declaración pasará un valor para indicar el tipo de excepción para que pueda ser identificada por un manejador de excepciones específico. A continuación se explica cómo utilizar la sentencia `raise`:

`raise` **excepción**

`raise` **excepción , mensaje**

El valor de la expresión **excepción** son los **valores atípicos** arrojados; la expresión de **mensaje** opcional suele ser una cadena que describe la información de la excepción, y esta expresión se denomina **parámetro anómalo**. Berry permite que cualquier valor se use como un valor anormal, por ejemplo, una cadena se puede usar como un valor anormal:

```
raise 'mi_error','un ejemplo de subida'
```

Después de que el programa se ejecute en la declaración `raise`, no continuará ejecutando las declaraciones que le siguen, sino que saltará al bloque de manejo de excepciones más cercano. Si el bloque de manejo de excepciones más reciente está en otras funciones, las funciones a lo largo de la cadena de llamadas se cerrarán antes. Si no hay un bloque de manejo de excepciones, se producirá una **salida anormal** y el intérprete imprimirá el mensaje de error de la excepción y la pila de llamadas de la ubicación del error. Cuando la instrucción `raise` está en el bloque de instrucciones `try`, la excepción será capturada por este último. La excepción capturada será manejada por el bloque `except` asociado con el bloque `try`. Si la excepción lanzada puede ser manejada por el bloque 'except', la ejecución de este bloque continuará desde la declaración después del último bloque `except`. Si ninguna de las sentencias `except` pueden manejar la excepción, la excepción se volverá a generar hasta que se pueda manejar o la excepción finalice.

Valores atípicos

En Berry, puede usar cualquier valor como valor atípico, pero generalmente usamos cadenas cortas. Berry también puede lanzar algunas excepciones internamente. Llamamos a estas excepciones **Excepción estándar**. Todos los valores de excepción estándar son de tipo cadena.

Valores atípicos	Descripción	Descripción del parámetro
`asser t_failed`	Afirmación fallida	Información sobre excepciones específicas
`ind ex_error`	(generalmente fuera de los límites)	Información sobre excepciones específicas
`` io_error``	Mal funcionamiento de E/S	Información sobre excepciones específicas
`k ey_error`	Error clave	Información sobre excepciones específicas
`runti me_error`	Excepción de tiempo de ejecución de máquina virtual	Información sobre excepciones específicas
`stop_i teration`	Fin del iterador	no
`synt ax_error`	Error de sintaxis	
por el compilador		
`unrealiz ed_error`	Función no realizada	Información sobre excepciones específicas
`ti pe_error`	Error de tipo	Información sobre excepciones específicas

Tabla 8: Lista de excepciones estándar

Capturar excepciones

Utilice la instrucción `except` para detectar excepciones. Debe estar emparejado con la sentencia `try`, es decir, un bloque de sentencia `try` debe ir seguido de uno o más bloques de sentencia `except`. La forma básica de la oración `try-except` es

`try` **bloque**
`except` **... bloque** `end`

La rama `except` puede tener las siguientes formas

`except ..` `except` **excepciones**
`except` **excepciones**`as` **variable**
`except` **excepciones**`as` **variable , mensaje**
`except .. as` **variable**
`except .. as` **variable , mensaje**

La instrucción `except` más básica no usa parámetros, esta rama `except` capturará todas las excepciones; **Lista de excepciones de captura**: **excepciones** es una lista de valores atípicos que pueden coincidir con la correspondiente rama `except`, que se utiliza entre varios valores de la lista Separados por comas; **variable** es **variable anormal**, si la

rama detecta una excepción, el valor atípico se vinculará a la variable; **mensaje** es **Variable de parámetro anómalo**, si la rama detecta una excepción, el valor del parámetro anómalo se vinculará a la variable.

Cuando se detecta una excepción en el bloque de instrucciones `try`, el intérprete verificará la rama `except` una por una. Si el valor de la excepción existe en la lista de captura de una rama, se llamará al bloque de código debajo de la rama para manejar la excepción, y la declaración `try-except` completa se cerrará después de que se ejecute el bloque de código. Si ninguna de las ramas `except` coinciden, el controlador de excepciones externo volverá a lanzar y capturar y manejar la excepción.

5. Función

Una **función** es una "subrutina" que puede ser llamada por un código externo. Como parte del programa, la función en sí también es una pieza de código. Una función puede tener 0 o más parámetros y devolverá un resultado, que se denomina **valor de retorno** de la función.

En Berry, la función es un **valor de primera clase**. Por lo tanto, además de llamar a funciones, también puede pasar funciones como valores, por ejemplo, vincular funciones a variables, usar funciones como valores devueltos, etc.

5.1 Información básica

El uso de funciones incluye dos partes: definición de función y llamada. La declaración de definición de función usa la palabra clave `def` como el comienzo. La definición de la función es el proceso de empaquetar y nombrar el código del cuerpo de la función. Este proceso solo genera la estructura de la función y no ejecuta la función. La función de ejecución debe usar un **operador de llamada**, que es un par de paréntesis. El operador de llamada actúa sobre una expresión cuyo resultado es un tipo de función. Los parámetros que se pasan a la función se escriben entre paréntesis y los parámetros múltiples se separan con comas. El resultado de la expresión de llamada es el valor de retorno de la función.

5.1.1 Definición de funciones

Función con nombre

Una **función con nombre** es una función a la que se le da un nombre cuando se define. Su declaración de definición consta de las siguientes partes: palabra clave `def`, nombre de funcion, lista que constan de 0 a múltiples parámetros y cuerpo de función, múltiples parámetros en la lista de parámetros están separados por comas, y todos los parámetros están escritos en un par de paréntesis. Llamamos al parámetro cuando la función se define como **Parámetros formales**, y al parámetro cuando llamamos a la función como **Argumentos**. La forma general de la definición de la función es:

```
'def' name ´(´ argumentos ´)´
  bloque
´end'
```

El nombre de función **nombre** es un identificador; **argumentos** es la lista de parámetros formales; **bloque** es el cuerpo de la función. Si el cuerpo de la función es una declaración vacía, la función se denomina "función vacía". La declaración del valor de retorno de la función está contenida en el cuerpo de la función. Si no hay declaración de devolución en **bloque**, la función devolverá `nil` por defecto. El nombre de la función es en realidad el nombre de la variable del objeto de la función vinculada. Si el nombre ya existe en el ámbito actual, definir la función equivale a vincular el objeto de función a esta variable.

El siguiente ejemplo define una función llamada `add`. La función de este ejemplo es sumar dos números y devolver el resultado.

```
def add(a, b)
    return a + b
end
```

La función add tiene dos parámetros a y b, y los dos sumandos se pasan a la función a través de estos parámetros para el cálculo. La instrucción return devuelve el resultado del cálculo.

Una función como atributo de clase se llama método. Esta parte del contenido se explicará en el capítulo orientado a objetos.

Función anónima

A diferencia de las funciones con nombre, la **función anónima** no tiene nombre y su expresión de definición tiene la forma:

```
´def' ´(´ argumentos ´)´
  bloque
´end'
```

Se puede ver que, en comparación con las funciones con nombre, no hay un **nombre** de función en su definición.. La definición de una función anónima es esencialmente una expresión, que se denomina **Función literal**. Para usar funciones anónimas podemos vincular el valor literal de la función a una variable:

```
add = def (a, b)
    return a + b
end
```

La función de este código es exactamente la misma que la función add en la sección anterior. Se puede usar una función anónima para pasar convenientemente el valor de la función como un valor literal. Al igual que otros tipos de literales, los literales de función también son la unidad de expresión más pequeña. Por lo tanto, lo que hay entre las palabras clave def y end es un todo indivisible.

Función de llamada

Tome la función add como ejemplo. Para llamar a esta función, debe proporcionar dos valores y puede obtener la suma de los dos números llamando a la función:

```
res = add(5, 3)
print(res) # 8
```

Llamamos a la función llamada (la función add en el ejemplo) como **Función llamada**, y la función que llama a la función llamada (la función principal en el ejemplo) como **Función clave**. El proceso de llamada de función es el siguiente: Primero, el intérprete (implícitamente) inicializará la lista de parámetros formales de la función llamada con la lista de argumentos y, al mismo tiempo, suspenderá la función de llamada y guardará su estado, luego creará un entorno para la función llamada y ejecutará la función llamada.

La función finalizará su ejecución cuando encuentre la instrucción return y pase el valor de retorno a la función que llama. El intérprete destruirá el entorno de la función llamada después de que regrese la función llamada, luego restaurará el entorno de la función que llama y continuará ejecutando la función que llama. El valor de retorno de la función también es el resultado de la expresión de la llamada a la función. El siguiente ejemplo define una función cuadrado y vincula esta función a una variable f, y luego llama a la función cuadrado a través de la variable f. Este uso es similar a los punteros de función en lenguaje C.

```
def cuadrado(n)
    return n * n
end
f = cuadrado
print(f(5)) # 25
```

Cabe señalar que el objeto de la función solo está vinculado a estas variables (consulte la sección Capitulo-3: Operador de asignación

```
f = cuadrado
cuadrado = nil
print(f(5)) # 25
```

Se puede ver que la función todavía se puede llamar normalmente después de reasignar `cuadrado`. Solo después de que el objeto de función ya no esté vinculado a ninguna variable, se perderá y el sistema reciclará los recursos ocupados por este tipo de objeto de función.

Desviar la llamada

La llamada de la función debe estar en el ámbito de la variable de función, por lo que normalmente no se puede llamar antes de que se defina la función. Para resolver este problema, puede utilizar este método para comprometer:

```
var func1
def func2(x)
    return func1(x)
end
def func1(x)
    return x * x
end
print(func2(4)) # 16
```

En este ejemplo, `func2` llama a `func1`, pero la función `func1` se define después de `func2`. Después de ejecutar este código, el programa generará el resultado correcto 16. Esta rutina utiliza el mecanismo de que no se llamará a la función cuando se defina. Defina la variable `func1` antes de definir `func2` para asegurarse de que el símbolo `func1` no se encontrará durante la compilación. Luego definimos la función `func1` después de `func2` para que la función se use para sobrescribir el valor de la variable `func1`. Cuando se llama a la función `func2` en la última línea `print(func2(4))`, la variable `func1` ya es la función que necesitamos, por lo que se mostrará el resultado correcto.

Llamada recursiva

Con **función recursiva** se refiere a funciones que se llaman a sí mismas directa o indirectamente. La recursividad se refiere a una estrategia que divide el problema en subproblemas similares y luego los resuelve. Tomando el factorial como ejemplo, la definición recursiva de factorial es $0! = 1$, $n! = n \ (n1)!$. Entonces podemos escribir la función recursiva para calcular el factorial según la definición:

```
def fact(n)
    if n == 0
        return 1
    end
    return n * fact(n-1)
end
```

Tome el factorial de 5 como ejemplo, el proceso de calcular manualmente el factorial de 5 es: ¡5! = 5 × 4 × 3 × 2 × 1 = 120. El resultado de llamar a la función `fact` también es 120:

```
print(fact(5)) # 120
```

Para garantizar que la profundidad de la llamada recursiva sea limitada (un nivel de recursividad demasiado profundo agotará el espacio de la pila), la función recursiva debe tener una condición de finalización. En `fact` la declaración `if` en la segunda línea de la definición de la función se usa para detectar la condición final, y el proceso recursivo finaliza cuando `n` se calcula como `0`. La fórmula factorial anterior no se aplica a parámetros no enteros. Ejecutar una expresión como `fact(5.1)` provocará un error de desbordamiento de pila debido a la imposibilidad de finalizar la recursividad.

Existe otra situación, la `Recurrencia indirecta`, es decir, la función no es llamada por sí misma sino por otra función (directa o indirectamente) llamada por ella. La recursividad indirecta generalmente requiere el uso de técnicas de llamada de función hacia adelante. Tome las funciones `es_impar` y `es_par` para calcular números pares e impares como ejemplos:

```
var es_impar
def es_par(n)
    if n == 0
        return true
    end
    return es_impar(n-1)
end
def es_impar(n)
    if n == 0
        return false
    end
    return es_par(n-1)
end
```

Estas dos funciones se llaman entre sí. Para garantizar que este nombre esté en el alcance cuando se llama a la función `es_impar` en la línea 6, la variable `es_impar` se define en la línea 1.

Llamada de función anónima

Si una función anónima solo se llamará una vez, la forma más fácil es llamarla cuando esté definida, por ejemplo:

```
res = def (a, b) return a + b end (1, 2) # 3
```

En esta rutina, usamos la expresión de llamada directamente después del literal de función para llamar a la función. Este uso es muy adecuado para funciones que solo se llamarán en un lugar.

También puede vincular una función anónima a una variable y llamarla:

```
add = def (a, b) return a + b end
res = add(1, 2) # 3
```

Este uso es similar a la llamada de una función con nombre, esencialmente llamando a la variable vinculada al valor de la función. Cabe señalar que es más difícil realizar llamadas recursivas a funciones anónimas, a menos que utilice técnicas de llamada de reenvío.

Parámetros formales y reales

La función utiliza parámetros reales para inicializar los parámetros formales cuando se llama. En circunstancias normales, el parámetro real y el parámetro de forma son iguales y las posiciones se corresponden entre sí, pero Berry también permite que el parámetro real sea diferente del parámetro formal: si el parámetro real es mayor que el parámetro formal, el parámetro real adicional al parámetro será descartado. De otra forma los parámetros formales restantes se inicializarán a `nil`.

El proceso de paso de parámetros es similar a la operación de asignación. Para los tipos `nil`, `boolean` y numéricos, el paso de parámetros es por valor, mientras que otros tipos son por referencia. Para el tipo de referencia de paso de escritura, como una instancia, modificarlos en la función llamada también modificará el objeto en la función de llamada. El siguiente ejemplo demuestra esta función:

```
var l = [], i = 0
def func(a, b)
    a.push(1)
    b ='cadena'
end
func(l, i)
print(l, i) # [1] 0
```

Se puede ver que el valor de la variable `l` ha cambiado después de llamar a la función `func`, pero el valor de la variable `i` no ha cambiado.

Función con número variable de argumentos (vararg)

Puede definir una función para tomar cualquier número arbitrario de argumentos e iterarlos. Por ejemplo, `print()` toma cualquier cantidad de argumentos e imprime cada uno de ellos separados por espacios. Debe definir el último argumento como una captura de todos los argumentos usando * antes de su nombre.

Todos los argumentos que siguen a los argumentos formales se agrupan en tiempo de ejecución en una instancia de `list`. Si no se capturan argumentos, la lista está vacía.

Ejemplo:

```
def f(a, b, *c) return size(c) end
f(1,2) # devuelve 0, c is []
f(1,2,3) # devuelve 1, c is [3]
f(1,2,3,4) # devuelve 2, c is [3,4]
```

Llamar a una función con un número dinámico de argumentos

La sintaxis de Berry solo permite llamar con un número fijo de argumentos. Utilice la función `call(f, [args])` para pasar cualquier número de argumentos arbitrario.

Puede agregar estáticamente cualquier número de argumentos a `call()`. Si el último argumento es una `lista`, se expande automáticamente a argumentos discretos.

Ejemplo:

```
def f(a,b) return nil end

call(f,1)        # llama a f(1)
```

(continues on next page)

(continued from previous page)

```
call(f,1,2)      # llama a f(1,2)
call(f,1,2,3)    # llama a f(1,2,3), el último argumento es ignorado por f
call(f,1,[2,3])  # llama a f(1,2,3), el último argumento es ignorado por f
call(f,[1,2])    # llama a f(1,2)
call(f,[])       # llama a f()
```

Puede combinar `call` y vararg. Por ejemplo, creemos una función que actúe como `print()` pero convierta todos los argumentos a mayúsculas.

Ejemplo completo:

```
def print_upper(*a) # toma un número arbitrario de argumentos, args es una lista
    import string
    for i:0..size(a)-1
        if type(a[i]) == 'string'
            a[i] = string.toupper(a[i])
        end
    end
    call(print, a) #  llama a print con todos los argumentos
end

print_upper("a",1,"Foo","Bar")  # imprime: A 1 FOO BAR
```

Funciones y variables locales

El cuerpo de la función en sí es un ámbito, por lo que las variables definidas en la función son todas variables locales. A diferencia de los bloques directamente anidados, cada vez que se llama a una función, se asigna espacio para las variables locales. El espacio para las variables locales se asigna en la pila y la información de asignación se determina en el momento de la compilación, por lo que este proceso es muy rápido. Cuando se anidan varios niveles de alcance en una función, el intérprete asigna espacio de pila para la cadena de anidamiento de alcance con la mayoría de las variables locales, en lugar del número total de variables locales en la función.

Declaración `return`

La declaración `return` se utiliza para devolver el resultado de una función, es decir, el valor de retorno de la función. Todas las funciones en Berry tienen un valor de retorno, pero no puede usar ninguna declaración `return` en el cuerpo de la función. En este momento, el intérprete generará una declaración `return` predeterminada para garantizar que la función regrese `return`. Hay dos formas de escribir oraciones:

```
´return'
´return' expresión
```

La primera forma de escribir es escribir solo la palabra clave `return` y no la expresión que se devolverá. En este caso, se devuelve el valor `nil` predeterminado. La segunda forma de escribir es seguir la expresión **expresión** después de la palabra clave `return`, y el valor de la expresión se usará como valor de retorno de la función. Cuando el programa ejecuta la declaración `return`, la función que se está ejecutando actualmente finalizará la ejecución y volverá al código que llamó a la función para continuar ejecutándose.

Cuando se usa una palabra clave separada `return` como declaración de retorno de una función, es fácil causar ambigüedad. En ese caso se recomienda agregar un punto y coma después de `return` para evitar errores:

```
def func()
    return;
    x = 1
end
```

En este ejemplo, la declaración `x = 1` después de la declaración `return` no se ejecutará, por lo que es redundante. Si se evita este tipo de código redundante, la instrucción `return` suele ir seguida de palabras clave como `end`, `else` o `elif`. En este caso, incluso si se usa una declaración `return` por separado, no hay necesidad de preocuparse por la ambigüedad.

Cierre (closure)

Conceptos básicos

Como se mencionó anteriormente, las funciones son el primer tipo de valor en Berry. Puede definir funciones en cualquier lugar y también puede pasar funciones como parámetros o devolver valores. Cuando se define otra función en una función, la función anidada puede acceder a las variables locales de cualquier función externa. Llamamos a las "variables locales de la función externa" utilizadas en la función la función como **Variables libres**. Las variables libres generalizadas también incluyen variables globales, pero no existe tal regla en Berry. El **Cierre** es una técnica que vincula funciones a **entornos**. El entorno es un mapeo que asocia cada variable libre de una función con un valor. En términos de implementación, los cierres asocian el prototipo de función con sus propias variables. Los prototipos de funciones se generan en tiempo de compilación y el entorno es un concepto de tiempo de ejecución, por lo que los cierres también se generan dinámicamente en tiempo de ejecución. Cada cierre vincula el prototipo de función al entorno cuando se genera, como se ve en el siguiente ejemplo:

```
def func(i) # La función externa
    def foo() # La función interna (closure)
        print(i)
    end
    foo()
end
```

La función interna `foo` es un cierre y tiene una variable libre `i`, que es un parámetro de la función externa `func`. Cuando se genera el cierre `foo`, su prototipo de función se vincula al entorno que contiene la variable libre `i`. Cuando la variable `foo` sale del alcance, el cierre se destruirá. Por lo general, la función interna será el valor de retorno de la función externa, por ejemplo:

```
def func(i) # La función externa
    return def () # Devuelve un cierre (función anónima)
        print(i)
        i = i + 1
    end
end
```

El cierre devuelto aquí es una función anónima. Cuando la función externa devuelve el cierre, las variables locales de la función externa se destruirán y el cierre no podrá acceder directamente a las variables en la función externa original. El sistema copiará el valor de la variable libre al entorno cuando se destruya la variable libre. El ciclo de vida de estas variables libres es el mismo que el del cierre, y solo el cierre puede acceder a ellas. La función o el cierre devuelto no se ejecutará automáticamente, por lo que debemos llamar al cierre devuelto por la función `func`:

```
f = func(0)
f()
```

Este código generará `0`. Si continuamos llamando al cierre `f`, obtendremos la salida `1`, `2`, `3`... Esto puede no entenderse bien: la variable [2.198] se destruye después de que la función `func` regresa , y como la variable libre del cierre `f`, `i` se almacenará en el entorno de cierre, por lo que cada vez que se llame a `f`, el valor de `i` se sumará a 1 (definición de la función `func` línea 4).

Uso de cierres

Los cierres tienen muchos usos. Aquí hay algunos usos comunes:

Evaluación perezosa

El cierre no hace nada hasta que se llama.

Función de comunicación privada

Puede permitir que algunos cierres compartan variables libres, que solo son visibles para estos cierres, y se comuniquen entre funciones cambiando los valores de estas variables libres. Esto puede evitar el uso de variables externas.

Generar múltiples funciones

A veces es posible que necesitemos usar múltiples funciones, estas funciones pueden tener solo diferentes valores de algunas variables. Podemos implementar una función y luego usar estas diferentes variables como parámetros de función. Una mejor manera es devolver el cierre a través de una función de fábrica y usar estas variables posiblemente diferentes como variables libres del cierre, de modo que no siempre tenga que escribir esos parámetros al llamar a la función, y cualquier número de funciones similares puede ser generado.

Simular miembros privados

Algunos lenguajes admiten el uso de miembros privados en objetos, pero la clase de Berry no lo admite. Podemos usar las variables libres de los cierres para simular miembros privados. Este uso no es la intención original de diseñar cierres, pero hoy en día, este "mal uso" de los cierres es muy común.

Resultado de caché

Si hay una función que requiere mucho tiempo para ejecutarse, llevará mucho tiempo llamarla cada vez. Podemos almacenar en caché el resultado de esta función, buscarlo en el caché antes de llamar a la función y devolver el valor almacenado en caché si lo encuentra; de lo contrario, se llama a la función y se actualiza el valor almacenado en caché. Podemos usar los cierres para guardar el valor almacenado en caché para que no quede expuesto al alcance externo, y el resultado almacenado en caché se conservará (hasta que se destruya el cierre).

Vinculación de variables libres

Si varios cierres vinculan la misma variable libre, todos los cierres siempre compartirán esta variable libre. Por ejemplo:

```
def func(i) # La función externa
    return [# Devuelve la lista de cierre
        def () # El cierre #1
            print("cierre 1 log:", i)
            i = i + 1
        end,
        def () # El cierre #2
            print("cierre 2 log:", i)
            i = i + 1
        end
    ]
end
```

La función `func`, en este ejemplo, devuelve dos cierres a través de una lista, y estos dos cierres comparten la variable libres `i`. Si llamamos a estos cierres:

```
f = func(0)
f[0]() # cierre 1 log: 0
f[1]() # cierre 2 log: 1
```

Como puede ver, actualizamos la variable libre `i` cuando llamamos al cierre `f[0]`, y este cambio afectó el resultado de llamar al cierre `f[1]`. Esto se debe a que si varios cierres utilizan una variable libre, solo hay una copia de la variable libre y todos los cierres tienen una referencia a la entidad de variable libre. Por lo tanto, cualquier modificación a la variable libre es visible para todos los cierres que usan dicha variable.

De manera similar, antes de que se destruyan las variables locales de la función externa, modificar el valor de la variable libre también afectará el cierre:

```
def func()
    i = 0
    def foo()
        print(i)
    end
    i = 1
    return foo
end
```

En este ejemplo cambiamos el valor de la variable `i` (que es la variable libre del cierre `foo`) de `0` a `1` antes de que regrese la función externa `func`, luego llamamos al cierre, y después el valor de la variable libre `i` cuando el paquete `foo` también es 1:

```
func()() # 1
```

Crear cierre en bucle

Al construir un cierre en el cuerpo del ciclo, es posible que no desee que las variables libres del cierre cambien con las variables del ciclo. Primero veamos un ejemplo de cómo crear un cierre en un bucle `while`:

```
def func()
    l = [] i = 0
    while i <= 2
        l.push(def () print(i) end)
        i = i + 1
    end
    return l
end
```

En este ejemplo, construimos un cierre en un ciclo y colocamos este cierre en una `lista`. Obviamente, cuando finalice el ciclo, el valor de la variable `i` será `3`, y todos los cierres de la lista `l` también son referencias usando esta variable. Si ejecutamos el cierre devuelto por `func` obtendremos el mismo resultado:

```
res = func()
res[0]() # 3
res[1]() # 3
res[2]() # 3
```

Si queremos que cada cierre se refiera a diferentes variables libres, podemos definir otra capa de funciones y luego vincular las variables del ciclo actual con los parámetros de la función:

```
def func()
    l = [] i = 0
    while i <= 2
        l.push(def (n)
            return def () print(n) end
        end (i))
        i = i + 1
    end
    return l
end
```

Para ayudar a entender este código aparentemente incomprensible, nos enfocaremos en el código de las líneas 4 a 6:

```
def (n)
    return def ()
        print(n)
    end
end (i)
```

Aquí realmente se define una función anónima y se llama inmediatamente. La función de esta función anónima temporal es vincular el valor de la variable de bucle `i` a su parámetro `n`, y la variable `n` también es lo que necesitamos para cerrar las variables libres del paquete, de modo que las las variables vinculadas al cierre construido durante cada ciclo son diferentes. Ahora obtendremos la salida deseada:

```
res = func()
res[0]() # 0
res[1]() # 1
res[2]() # 2
```

Hay algunas formas de resolver el problema de las variables de bucle como variables libres. Una forma un poco más simple es definir una variable temporal en el cuerpo del bucle:

```
def func()
    l = [] i = 0
    while i <= 2
        temp = i
        l.push(def () print(temp) end)
        i = i + 1
    end
    return l
end
```

Aquí `temp` es una variable temporal. El alcance de esta variable está en el cuerpo del ciclo, por lo que se redefinirá cada vez que se realice un ciclo. También podemos usar la instrucción `for` para resolver el problema:

```
def func()
    l = []
    for i: 0 .. 2
        l.push(def () print(i) end)
    end
    return l
end
```

Esta puede ser la forma más sencilla de`for`. La variable de iteración de la instrucción se creará en cada ciclo. El principio es similar al método anterior.

Expresión lambda

La **Expresión lambda** es una función anónima especial. La expresión lambda se compone de una lista de parámetros y un cuerpo de función, pero la forma es diferente de la función general:

```
´/´ args ´->´ expr ´end'
```

args es la lista de parámetros, la cantidad de parámetros puede ser cero o más, y los parámetros múltiples están separados por comas o espacios (no se pueden mezclar al mismo tiempo); **expr** es la expresión de retorno, la expresión lambda devolverá el valor de la expresión. Las expresiones lambda son adecuadas para implementar funciones muy simples. Por ejemplo, la expresión lambda para juzgar el tamaño de dos números es:

```
/ a b -> a < b
```

Esto es más fácil que escribir una función con la misma funcionalidad. En algunos algoritmos generales de clasificación, este tipo de función de comparación de tamaño puede necesitar un uso extensivo. El uso de expresiones lambda puede simplificar el código y mejorar la legibilidad.

Al igual que las funciones generales, las expresiones lambda pueden formar cierres. Las expresiones lambda se llaman de la misma manera que las funciones ordinarias. Si usa el método de llamada inmediata similar a las funciones anónimas:

```
lambda = / a b -> a < b
result = lambda(1, 2) #  llamada normal
result = (/ a b -> a < b)(1, 2) #  llamada directa
```

Dado que el operador de llamada de función tiene una prioridad más alta, se debe agregar un par de paréntesis a la expresión lambda cuando se realiza una llamada directa, para que se llame como un todo.

6. Función orientada a objetos

Por consideraciones de optimización, Berry no consideró los tipos simples como objetos. Estos tipos simples incluyen `nil`, numéricos, booleanos y cadena. Pero Berry proporciona clases para implementar el mecanismo de objetos. Entre los tipos de datos básicos de Berry, `list`, `map` y `range` son objetos de clase. Un objeto es una colección que contiene datos y métodos, donde los datos se componen de algunas variables y los métodos son funciones. El tipo de un objeto se denomina clase y la entidad de un objeto se denomina instancia.

6.1 Clase e instancia

6.1.1 Declaración de clase

Para usar una clase, primero debe declararla. La declaración de una clase comienza con la palabra clave `class`. Las variables miembro y los métodos de la clase deben especificarse en la declaración. Este es un ejemplo de declaración de una clase:

```
class persona
    static var mayor = 18
    var nombre, edad
    def init(nombre, edad)
        self.nombre = name
        self.edad = edad
    end
    def tostring()
        return 'nombre: '+ str(self.nombre) + ', edad:' + str(self.edad)
    end
    def es_adulto()
        return self.edad >= self.mayor
    end
end
```

Las variables miembro de clase se declaran con la palabra clave `var`, mientras que los métodos miembro se declaran con la palabra clave `def`. Actualmente, Berry no admite la inicialización de variables miembro en el momento de la definición, por lo que el constructor debe realizar la inicialización de las variables miembro. Las propiedades de la clase no se pueden modificar después de completar la declaración, por lo que la clase es un objeto de solo lectura.

Este diseño es para garantizar que la clase se pueda construir estáticamente en el lenguaje C cuando se implemente el intérprete y se pueda usar la propiedad `const` modificada para ahorrar RAM

La clase de Berry no admite restricciones de acceso y todas las propiedades de la clase son visibles desde el exterior. En las clases nativas, puede usar algunos trucos para hacer que las propiedades sean invisibles para el código Berry (por lo general, hacer que el nombre del miembro comience con un punto "."). Puede usar algunas convenciones para restringir el acceso a los miembros de la clase, como la convención de que los atributos que comienzan con un guión bajo son atributos privados. Esta convención no tiene ningún uso a nivel gramatical, pero favorece la estructura lógica del código.

Instanciar

La clase en sí es solo una descripción abstracta. Tomando los autos como ejemplo, conozco el concepto de autos, y cuando realmente queremos usar autos, necesitamos autos reales. El uso de las clases es similar. No solo usaremos esta descripción abstracta, sino que necesitaremos producir un objeto concreto basado en esta descripción. Este proceso se llama **Instanciación de la clase**, abreviado como instanciación, y el objeto concreto producido por la instanciación se llama **Instancia**. La clase en sí no tiene datos, y la creación de instancias produce una instancia basada en la información descrita por la clase y proporciona datos específicos a la instancia.

Método y parámetros `self`

Los métodos de clase son esencialmente funciones. A diferencia de las funciones ordinarias, los métodos pasan implícitamente un parámetro `self`, y siempre es el primer parámetro, que almacena una referencia a la instancia actual. Debido a la existencia de parámetros `self`, el número de parámetros del método será uno más que el número de parámetros definidos en la declaración. Aquí usamos un ejemplo simple para demostrar:

```
class Test
    def metodo()
        return self
    end
end
objecto = Test()
print(objecto)
print(objecto.metodo())
```

Este ejemplo define una clase `Test`, que tiene un método `metodo`, que devuelve su parámetro `self`. Las dos últimas líneas de la rutina imprimen el valor de la instancia 'objeto' de la clase `Test` y el valor de retorno del método 'metodo' respectivamente. El resultado de ejecución de este ejemplo es:

```
<instance: Test()>
<instance: Test()>
```

Se puede ver que el parámetro `self` del método y el nombre de la instancia de uso (`objecto` en el ejemplo) representan el mismo objeto y ambos son referencias de instancia. Use `self` para acceder a los miembros o atributos de la instancia en el método.

Métodos sintéticos

Puede declarar métodos y miembros dinámicos sintéticos usando **Miembros virtuales** como se describe en el Capítulo 8.2.

Variables de clase `static`

Las variables o funciones se pueden declarar `static`. Las variables estáticas tienen el mismo valor para todas las instancias de la misma clase. Se declaran como `static a = 1` o `static var a = 1`. Las variables estáticas se inicializan justo después de la creación de la clase.

Métodos de clase `static`

Los métodos se pueden declarar `static`, lo que significa que actúan como una función regular y no toman `self` como primer argumento. Dentro de los métodos estáticos, no se declara ninguna variable "auto" implícita. Los métodos estáticos se pueden llamar a través de la clase o a través de una instancia.

```
class static_demo
      static def incremento_static(i)
          return i + 1
      end
      def incremento_instancia(i)
          return i + 1
      end
  end
a = static_demo()
static_demo.incremento_static(1)    # llamada via clase
```

2

```
a.incremento_static(1)              # llamada via instancia
static_demo.incremento_instancia(1)
```

type_error: unsupported operand type(s) for +: 'nil' and 'int'
stack traceback:
stdin:6: in function *increment_instancia*
stdin:1: in function *main*

```
a.increment_instancia(1)
```

2

Constructor y Destructor

Constructor

El constructor de la clase es el método `init`. Se llama al constructor cuando se crea una instancia de la clase. Por lo tanto, el constructor generalmente se usa para la inicialización de miembros, por ejemplo:

```
class Test
    var a
    def init()
        self.a ='esto es una prueba'
    end
end
```

El constructor de este ejemplo inicializa el miembro `a` de la clase `Test` con la cadena `'esto es una prueba'`. Si instanciamos la clase, podemos obtener el valor del miembro `a`:

```
class Test
    var a
    def init()
        self.a ='esta es una prueba'
    end
end
```

Destructor

El destructor de la clase es el método `deinit`. Se llama al destructor cuando se destruye la instancia. El destructor se usa generalmente para completar algún trabajo de limpieza. Debido a que el mecanismo de recolección de basura libera automáticamente la memoria de los objetos inútiles, no hay necesidad de liberar la memoria en el destructor (y tampoco hay forma de hacerlo en el destructor). En la mayoría de los casos, no hay necesidad de usar un destructor, a menos que cierta clase requiera cierto procesamiento cuando se destruye. Un ejemplo típico es que un objeto de archivo debe cerrar el archivo cuando se destruye.

Herencia de clases

Berry solo admite herencia simple, es decir, una clase solo puede tener una clase base, y la clase base usa el operador `:` para declarar:

```
class Test: Base
    ...
end
```

Aquí la clase `Test` hereda de la clase `Base`. La subclase heredará todos los métodos y propiedades de la clase base y puede anularlos en la subclase. Este mecanismo se llama **Sobrecarga**. En circunstancias normales, solo sobrecargaremos métodos, no propiedades.

El mecanismo de herencia de la clase Berry es relativamente simple. Las subclases contendrán referencias a la clase base y los objetos de instancia son similares. Al instanciar una clase con una clase base, en realidad se generan múltiples objetos. Estos objetos se encadenarán de acuerdo con la relación de herencia y, finalmente, obtendremos el objeto de instancia al final de la cadena de herencia.

Sobrecarga de método

La **Sobrecarga** significa que la subclase y la clase base usan el mismo método de nombre, y el método de la subclase anulará el mecanismo del método de la clase base. Para ser precisos, las variables miembro también se pueden sobrecargar, pero esta sobrecarga no tiene sentido. La sobrecarga de métodos se divide en sobrecarga de métodos ordinarios y sobrecarga de operadores.

Sobrecarga de método común

Sobrecarga del operador

Puede usar la sobrecarga de operadores de la clase para hacer que la instancia admita la operación del operador integrado. Por ejemplo, para una clase sobrecargada con el operador de suma, podemos usar el operador de suma para realizar operaciones en la instancia. Un operador sobrecargado es un método con un nombre especial, y la forma de función sobrecargada de un operador binario es

```
´def' operador ´(´ otro ´)´
  bloque
´end'
```

operador es un operador binario sobrecargado. El operando izquierdo del operador binario es el objeto `self` y el operando derecho es el valor del parámetro **otro**. La forma de función sobrecargada del operador unario es

```
´def' operador ´()´
  bloque
´end'
```

operador es un operador unario sobrecargado. Para distinguirlo del operador de resta, el signo menos unario se escribe como `-*` cuando está sobrecargado. Las funciones sobrecargadas del operador deben tener un valor de retorno, porque el valor de retorno `nil` predeterminado no suele ser el resultado esperado. Tomemos una clase entera como ejemplo para ilustrar el uso de la sobrecarga de operadores. Primero defina la clase `integer`:

```
class integer
    var value
    def init(v)
        self.value = v
    end
    def +(other)
        return integer(self.value + other.value)
    end
    def *(other)
        return integer(self.value * other.value)
    end
    def -*()
        return integer(-self.value)
    end
    def tostring(other)
        return str(self.value)
    end
end
```

La clase `integer` sobrecarga los operadores suma, multiplicación y simbólicos, y el método `tostring` hace que la instancia use la función `print` para generar el resultado. Podemos usar una simple línea de código para probar la

función de sobrecarga de operadores de la clase:

```
integer(1) + integer(2) * -integer(3) # -5
```

El resultado de esta línea de código es una instancia de `integer`. El valor del miembro `value` de esta instancia es `-5`, que es el mismo resultado de las mismas cuatro operaciones aritméticas con números enteros.

Los operadores lógicos no se pueden sobrecargar directamente. Si necesita una instancia para admitir operaciones lógicas, debe implementar el método `tobool`. El método no tiene parámetros y el valor devuelto debe ser de tipo booleano. La operación lógica de la instancia en realidad se realiza convirtiendo la instancia en un valor booleano, por lo que la operación lógica de la instancia está completamente en línea con la naturaleza de la operación lógica general. El operador de subíndice no se sobrecarga directamente, pero se implementa mediante los métodos `item` y `setitem`. El método `item` se utiliza para la lectura de subíndices, su primer parámetro es el valor del subíndice y el valor de retorno es el resultado de la operación del subíndice; `setitem` se utiliza para la escritura de subíndices, y su primer parámetro es el valor del subíndice, el segundo parámetro es el valor que se va a escribir; este método no utiliza el valor de retorno.

Al operador sobrecargado se le puede asignar cualquier significado, incluso sin satisfacer las propiedades habituales de los operadores. Dada la versatilidad del código y la dificultad de comprensión, no se recomienda que los usuarios den a los operadores sobrecargados una función alejada del significado general.

Sobrecarga del operador de asignación compuesto

El operador de asignación compuesto no se puede sobrecargar directamente, pero podemos lograr el propósito de "sobrecargar" el operador de asignación compuesto sobrecargando el operador binario correspondiente al operador de asignación compuesto. Por ejemplo, después de sobrecargar el operador "+", puede usar el operador "+=" para instancias de clases relacionadas. Vale la pena señalar que el uso de operaciones de asignación compuestas en la instancia hará que las variables de la instancia vinculada pierdan su referencia a la instancia.

```
class integer
    var valor
    def init(x)
        self.valor = x
    end
    def +(other)
        return integer(self.valor + other.valor)
    end
end
a = integer(4) # a: <instance: 0x55edff400a78>
a += integer(5) # a: <instance: 0x55edff4011b8>
print(a.valor) # 9
```

Después de que se ejecuta la línea 11 de código, la instancia enlazada en la variable `a` realmente ha cambiado. Esta línea de código es equivalente a `a = integer(4) + integer(5)`. Si el operador binario de la sobrecarga de clase no modifica el estado de la instancia, entonces el operador de asignación compuesto correspondiente no modificará ninguna instancia (puede generar nuevas instancias).

Instancia

Una **Instancia** es un objeto generado después de la instanciación de la clase. Una clase se puede instanciar varias veces para generar diferentes instancias. Las instancias de Berry están referenciadas por la clase a la que pertenecen y los campos de datos correspondientes. Todas las instancias de una clase se referirán a esta clase, pero los campos de datos de estas instancias son independientes entre sí.

Objeto de clase base de acceso

La función integrada `super` se utiliza para acceder a objetos de clase superior. Se puede utilizar en clases o instancias.

La magia ocurre cuando llamas a un método de la superclase para que se comporte como intuitivamente crees que lo haría. Por ejemplo, el patrón común para `init()` es el siguiente:

```
def init(<args>)
    # hacer cosas antes de super init
    super(self).init(<args>)
    # hacer cosas después de super init
end
```

Tenga en cuenta que las clases siempre contienen métodos `init()` implícitos que no hacen nada, por lo que siempre puede llamar a init desde la superclase incluso si no se declaró ningún método `init()`.

Ejemplo completo:

```
class A
    var val
    def init(val)
        # super(self).init(val)    # esto sería válido pero inútil
        self.val = val
    end
    def tostring()
        return "val=" + str(self.val)
    end
end

class B: A
    var magia    # verdadero si el valor es 42
    def init(val)
        super(self).init(val)     # llamar a superinit
        self.magia = (val == 42)
    end
    def tostring()
        if self.magia
            return "magia!"
        else
            return super(self).tostring()
        end
    end
end

####### Ejemplo de uso
```

(continues on next page)

(continued from previous page)

```
> b1 = B(1)
> b1
val=1
> b42 = B(42)
> b42
magia!
```

Características avanzadas: Al llamar a `super(self).<method> (<args> )` ocurre algo de magia. Cuando se llama al supermétodo, los argumentos `self` se refieren a la clase específica más baja. Sin embargo, el `<method>` no se busca desde la clase de `self` (que siempre es la más baja), sino desde la superclase de la clase que contiene el método que se está ejecutando actualmente.

Ejemplo:

```
> class A
      def init()
          print("In A::init, self es de tipo", classname(self))
      end
  end
> class B:A
      def init()
          print("In B::init, self es de tipo", classname(self))
          super(self).init()
      end
  end
> class C:B
      def init()
          print("En C::init, self es de tipo", classname(self))
          super(self).init()
      end
  end
> c = C()
En C::init, self es de tipo C
In B::init, self es de tipo C
In A::init, self es de tipo C
>
```

Explicación:

- llamando a `C:init()` en `instancia<C>` - en `C:init()` `self` es `instancia<C>`, `super(self).init()` se refiere a la superclase de `C` (método actual), es decir, `B`, por lo que `B:init()` se llama con `instance<C>` argumento - en `B:init()` `self` es `instancia<C>`, `super(self).init()` se refiere a la superclase de `B` (método actual), es decir, `A`, por lo que `A:init()` se llama con `instance<C>` argumento - en `A:init()` `self` es `instancia<C>`, imprimir y devolver

Nota: por compatibilidad con versiones anteriores, super puede tomar un segundo argumento `super(instancia, clase)` para especificar la clase donde resolver el método. Esta función no debe usarse más, ya que es propensa a errores.

7. Bibliotecas y Módulos

7.1 Biblioteca básica

Hay algunas funciones y clases que se pueden usar directamente en la biblioteca estándar. Proporcionan servicios básicos para los programas de Berry, por lo que también se denominan bibliotecas básicas. Las funciones y clases de la biblioteca básica están visibles en el ámbito global (perteneciente al ámbito integrado), por lo que se pueden utilizar en cualquier lugar. No defina variables con el mismo nombre que las funciones o clases en la biblioteca base. Si lo hace, será imposible hacer referencia a las funciones y clases en la biblioteca base.

7.1.1 Función integrada

Función `print`

Ejemplo

```
print(...)
```

Descripción

Esta función imprime los parámetros de entrada en el dispositivo de salida estándar. La función puede aceptar cualquier tipo y cualquier número de parámetros. Todos los tipos imprimirán su valor directamente, y para una instancia, esta función verificará si la instancia tiene un método `tostring()`, y si lo hay, imprimirá el valor de retorno de la instancia llamando al método `tostring()`, de lo contrario, imprimirá la dirección de la instancia.

```
print('Hola mundo!') # Hola mundo!
print([1, 2, '3']) # [1, 2, '3']
print(print) # <function: 0x561092293780>
```

Función `input`

Ejemplo

```
input()
input(prompt)
```

Descripción

La función `input` se usa para ingresar una línea de cadena de caracteres desde el dispositivo de entrada estándar. Esta función puede usar el parámetro `prompt` como un indicador de entrada, y el parámetro `prompt` debe ser de tipo cadena. Después de llamar a la función `input`, los caracteres se leerán desde el búfer del teclado hasta que se encuentre un carácter de nueva línea.

```
input('por favor ingrese una cadena:') # por favor ingrese una cadena:
```

La función `input` no regresa hasta que se presiona la tecla “Enter”, por lo que el programa queda “atascado” y no es un error.

Función `type`

Ejemplo

```
type(valor)
```

- *valor*: parámetro de entrada (se espera obtener su tipo).
- *valor devuelto*: una cadena que describe el tipo de parámetro.

Descripción

Esta función recibe un parámetro de cualquier tipo y devuelve el tipo del parámetro. El valor devuelto es una cadena que describe el tipo del parámetro. La siguiente tabla muestra los valores de retorno correspondientes a los principales tipos de parámetros:

Tipo de parámetro	Valor devuelto	Tipo de parámetro	Valor devuelto
Nil	`'nil'`	Integer	`'int'`
Real	`'real'`	Boolean	`'bool'`
Function	`'function'`	Class	`'class'`
String	`'string'`	Instance	`'instance'`
puntero nativo	`'ptr'`		

```
type(0) # 'int'
type(0.5) # 'real'
type('hello') # 'string'
type(print) # 'función'
```

Función `classname`

Ejemplo

```
classname(objeto)
```

Descripción

Esta función devuelve el nombre de clase (cadena) del parámetro. Por lo tanto, el parámetro debe ser una clase o instancia, y otros tipos de parámetros devolverán `nil`.

```
classname(list) # 'list'
classname(list()) # 'list'
classname({}) # 'map'
classname(0) # nil
```

Función `classof`

Ejemplo

```
classof(objeto)
```

Descripción

Devuelve la clase de un objeto de instancia. El parámetro `objeto` debe ser una instancia. Si la función se llama con éxito, devolverá la clase a la que pertenece la instancia; de lo contrario, devolverá `nil`.

```
classof(list) # nil
classof(list()) # <class: list>
classof({}) # <class: map>
classof(0) # nil
```

Función `str`

Ejemplo

```
str(valor)
```

Descripción

Esta función convierte los parámetros en cadenas y los devuelve. Las funciones `str` pueden aceptar cualquier tipo de parámetros y convertirlos. Cuando el tipo de parámetro es una instancia, verificará si la instancia tiene un método `tostring()`, si lo hay, se usará el valor de retorno del método; de lo contrario, la dirección de la instancia se convertirá en una cadena.

```
str(0) # '0'
str(nil) # 'nil'
str(list) # 'list'
str([0, 1, 2]) # '[0, 1, 2]'
```

Función `number`

```
number(valor)
```

Descripción

Esta función convierte la cadena o el número de entrada en un tipo numérico y lo devuelve. Si el parámetro de entrada es un número entero o real, devuelve directamente. Si es una cadena de caracteres, intenta convertir la cadena de caracteres en un valor numérico en formato decimal. El número entero o real se juzgará automáticamente durante la conversión. Otros tipos devuelven `nil`.

Ejemplo

```
number(5) # 5
number('45.6') # 45.6
number('50') # 50
number(list) # nil
```

Función `int`

```
int(valor)
```

Descripción

Esta función convierte la cadena o el número de entrada en un número entero y lo devuelve. Si el parámetro de entrada es un número entero, regresa directamente, si es un número real, descarta la parte decimal. Si es una cadena, intenta convertir la cadena en un número entero en decimal. Otros tipos devuelven `nil`. Cuando el tipo de parámetro es una instancia, verificará si la instancia tiene un método `toint()`, si lo hay, se utilizará el valor de retorno del método.

Ejemplo

```
int(5) # 5
int(45.6) # 45
int('50') # 50
int('0x10) # 16 - literal hexadecimal es válido
int(list) # nil
```

Función `real`

```
real(valor)
```

Descripción

Esta función convierte la cadena o el número de entrada en un número real y lo devuelve. Si el parámetro de entrada es un número real, devolverá directamente, si es un número entero, se convertirá en un número real. Si es una cadena, intenta convertir la cadena en un número real en decimal. Otros tipos devuelven `nil`.

Ejemplo

```
real(5) # 5, type(real(5)) →'real'
real(45.6) # 45.6
real('50.5') # 50.5
real(list) # nil
```

Función `bool`

```
bool(valor)
```

Descripción

Esta función convierte la cadena o el número de entrada en un valor booleano y lo devuelve.

La conversión sigue las siguientes reglas:

- `nil`: convertido a `falso`.
- **Entero**: cuando el valor es `0`, se convierte en `falso`, de lo contrario, se convierte en `verdadero`.
- **Número real**: cuando el valor es `0.0`, se convierte en `falso`, de lo contrario, se convierte en `verdadero`.
- **Cadena**: cuando el valor es "" (cadena vacía) se convierte en `falso` de lo contrario, se convierte en `verdadero`.
- **Comobj** y **Comptr**: cuando el puntero interno es `NULL` es convertido a `falso`, de lo contrario se convierte a `verdadero`.

- **Instancia**: si la instancia contiene un método `tobool()`, se utilizará el valor de retorno del método, de lo contrario, se convertirá en `verdadero`.
- Todos los demás tipos: convierte a `verdadero`.

Ejemplo

```
bool() # false
bool(nil) # false
bool(false) # false
bool(true) # true
bool(0) # false
bool(1) # true
bool("") # false
bool("a") # true
bool(3.5) # true
bool(list) # true
bool([]) # true
bool({}) # true
# avanzado
import introspect
bool(introspect.toptr(0)) # false
bool(introspect.toptr(0x1000)) # true
```

Función `size`

```
size(valor)
```

Descripción

Esta función devuelve el tamaño de la cadena de entrada. Si el parámetro de entrada no es una cadena, se devuelve 0. La longitud de la cadena se calcula en bytes. Esta función también funciona para instancias de `list` y `map` y devuelve el número de elementos.

Ejemplo

```
size(10) # 0
size('s') # 1
size('string') # 6
size([1,2]) # 2
size({"a":1}) # 1
```

Función `super`

```
super(objeto)
```

Descripción

Esta función devuelve el objeto principal de la instancia. Cuando crea una instancia de una clase derivada, también creará una instancia de su clase base. Se requiere la función `super` para acceder a la instancia de la clase base (es decir, el objeto principal).

Consulte el capítulo 6 sobre el comportamiento mágico de `super(objeto)` al llamar a un supermétodo.

Ejemplo

```
class mi_lista: lista end
l = mi_lista() # classname(l) -->'mi_lista'
sl = super(l) # classname(sl) -->'lista'
```

Función `assert`

```
assert(expresión)
assert(expresión, mensaje)
```

Descripción

Esta función se utiliza para implementar la función de aserción. La función `assert` acepta un parámetro. Cuando el valor del parámetro es `false` o `nil`, la función activará un error de aserción; de lo contrario, la función no tendrá ningún efecto. Cabe señalar que incluso si el parámetro es un valor equivalente a `false` en operaciones lógicas (por ejemplo, `0`), no generará un error de aserción. El parámetro `mensaje` es opcional y debe ser una cadena. Si se utiliza este parámetro, la información de texto proporcionada en `message` se mostrará cuando se produzca un error de aserción; de lo contrario, se mostrará el mensaje predeterminado "`Assert Fail`".

Ejemplo

```
assert(false) # aserción fallida!
assert(nil) # aserción fallida!
assert() # aserción fallida!
assert(0) # aserción fallida!
assert(false,'mensaje de aserción del usuario.') # mensaje de aserción.
assert(true) # pasa
```

Función `compile`

```
compile(cadena)
compile(cadena, 'string')
compile(nombre_archivo, 'file')
```

Descripción

Esta función compila el código fuente de Berry en una función. El código fuente puede ser una cadena o un archivo de texto. El primer parámetro de la función `compile` es una cadena, y el segundo parámetro es una `'cadena'` o `'archivo'`. Cuando el segundo parámetro es `'cadena'` o no hay un segundo parámetro, la función `compile` compilará el primer parámetro como código fuente. Cuando el segundo parámetro es `'file'`, la función `compile` compilará el archivo correspondiente al primer parámetro. Si la compilación es exitosa, `compile` devolverá la función compilada; de lo contrario, devolverá `nil`.

Ejemplo

```
compile('print(\'Hola mundo!\')')() # Hola mundo!
compile('test.be','file')
```

Clase `list`

`list` es un tipo incorporado, y define un contenedor de almacenamiento secuencial que admite la lectura y escritura de subíndices. Es similar a las matrices en otros lenguajes de programación. La obtención de una instancia de la clase `list` se puede construir usando un par de corchetes: `[]` generará una instancia vacía de `list`, y `[expr, expr, ...]` generará una `list` ejemplo con varios elementos. También se puede instanciar llamando a la clase `list`: ejecutar `list()` obtendrá una instancia vacía de `list`, y `list(expr, expr, ...)` devolverá una instancia con varios elementos.

Método `list` (Constructor)

Inicializa el contenedor `list`. Este método puede aceptar de 0 a múltiples parámetros. La instancia `list` generada cuando se pasan múltiples parámetros tendrá estos parámetros como elementos, y el orden de disposición de los elementos es coherente con el orden de disposición de los parámetros.

Método `tostring`

Serializa la instancia de `list` en una cadena y la devuelve. Por ejemplo, el resultado de ejecutar `[1, [], 1.5].tostring()` es `'[1, [], 1.5]''. Si el contenedor`list` se refiere a sí mismo, la posición correspondiente utilizará puntos suspensivos en lugar del valor específico:

```
l = [1, 2]
l[0] = l
print(l) # [[...], 2]
```

Método `concat`

Convierte cada elemento de la lista en cadenas y la concatena usando la cadena proporcionada.

```
l = [1, 2, 3]
l.concat()  # '123'
l.concat(", ")  # '1, 2, 3'
```

Método `push`

Agrega un elemento al final del contenedor `list`. El prototipo de este método es `push(valor)`, el parámetro `valor` es el valor que se agregará, y el valor agregado se almacena al final del contenedor `list`. La operación de agregar aumenta el número de elementos en el contenedor `list` en 1. Puede agregar cualquier tipo de valor a la instancia de `lista`.

Método `insert`

Inserta un elemento en la posición especificada del contenedor `list`. El prototipo de este método es `insert(indice, valor)`, el parámetro `indice` es la posición a insertar, y `valor` es el valor a insertar. Después de insertar un elemento en la posición `indice`, todos los elementos que originalmente comenzaron desde esta posición se moverán hacia atrás un elemento. La operación de inserción aumenta el número de elementos en el contenedor `list` en 1. Puede insertar cualquier tipo de valor en el contenedor `list`.

Supongamos que el valor de una instancia de `lista l` es `[0, 1, 2]`, e insertamos una cadena `'cadena'` en la posición 1, y necesitamos llamar a `l.insert(1, 'cadena')`. Finalmente, el nuevo valor de `lista` es `[0, 'cadena', 1, 2]`.

Si el número de elementos en un contenedor `list` es S, el rango de valores de la posición de inserción es $\{i \ : \ S \ i < S\}$. Cuando la posición de inserción es positiva, indexa hacia atrás desde el principio del contenedor `list`; de lo contrario, indexa hacia adelante desde el final del contenedor `list`.

Método `remove`

Quita un elemento del contenedor. El prototipo de este método es `remove(indice)`, y el parámetro `indice` es la posición del elemento a eliminar. Después de eliminar el elemento, el elemento detrás del elemento eliminado avanzará un elemento y la cantidad de elementos en el contenedor se reducirá en 1. Al igual que el método `insert`, el método `remove` también puede usar índices positivos o negativos.

Método `item`

Obtiene un elemento en el contenedor `list`. El prototipo de este método es `item(indice)`, el parámetro `indice` es el índice del elemento a obtener, y el valor de retorno del método es el elemento en la posición del índice. `list` El contenedor admite múltiples métodos de indexación:

- Índice entero: El valor del índice puede ser un número entero positivo o un entero negativo. Si el índice es negativo, es relativo al final de la lista; es decir, `-1` indica el último elemento de la lista. El valor de retorno de `item` es el elemento en la posición del índice. Si la posición del índice excede el número de elementos en el contenedor o está antes del elemento 0, el método `item` devuelve `nil`.
- Índice `list`: Utilizando una lista de enteros como índice, `item` devuelve una `lista`, y cada elemento en el valor devuelto `lista` es un elemento correspondiente a cada índice entero en el parámetro `lista`. los el valor de la expresión `[3, 2, 1].item([0, 2])` es `[3, 1]`. Si una tipo de elemento en el parámetro `lista` no es un número entero, entonces el el valor en esa posición en el valor de retorno `lista` es `nil`.
- Índice `range`: Usando un rango de enteros como índice, `item` devuelve una `lista`. El valor devuelto almacena los elementos indexados de la `lista` desde el límite inferior hasta el límite superior del parámetro `range`. Si el índice excede el rango de índice de la 'lista' indexada, el retorno value `list` usará `nil` para llenar la posición más allá del índice.

Método `setitem`

Establece el valor de la posición especificada en el contenedor. El prototipo de este método es `setitem(indice, valor)`, `indice` es la posición del elemento a escribir y `valor` es el valor a escribir. `indice` es el valor de índice entero de la posición de escritura. Las posiciones de índice fuera del rango de índice del contenedor harán que `setitem` no se ejecute.

Método `size`

Devuelve el número de elementos en el contenedor, que es la longitud del contenedor. El prototipo de este método es `size()`.

Método `resize`

Restablece la `lista` a la longitud del contenedor. El prototipo de este método es `resize(count)`, y el parámetro `count` es la nueva longitud del contenedor. Al usar `resize` para aumentar la longitud del contenedor, el nuevo elemento se inicializará en `nil`. El uso de `resize` para reducir la longitud del contenedor descartará algunos elementos al final del contenedor. P.ej:

```
l = [1, 2, 3]
l.resize(5) # Expansion, l == [1, 2, 3, nil, nil]
l.resize(2) # Reduce, l == [1, 2]
```

Método `iter`

Devuelve un iterador para recorrer el contenedor `list` actual.

Método `find`

Similar a `item` o `list[idx]`. La única diferencia es que si el índice está fuera de rango, `find` devuelve `nil` en su lugar o genera una excepción.

Método `reverse`

Cambia la lista en el lugar e invierte el orden de los elementos. También devuelve la lista resultante.

Clase `map`

La clase `map` es un tipo de clase incorporado que se utiliza para proporcionar un contenedor desordenado de pares clave-valor. Dentro del intérprete de Berry, `map` usa la tabla Hash para su implementación. Puede utilizar pares de llaves para construir un contenedor `map`. El uso de un par de llaves vacías `{}` generará una instancia de `map` vacía. Si necesita construir una instancia de `map` que no esté vacía, use dos puntos para separar la clave y el valor, y use un punto y coma para separar varios pares clave-valor. Por ejemplo, `{0: 1, 2: 3}` tiene dos pares clave-valor (0,1) y (2,3). También puede obtener una instancia de `map` vacía llamando a la clase `map`.

Método `map` (Constructor)

Inicializa el contenedor `map`, este método no acepta parámetros. Ejecutar `map()` obtendrá una instancia de `map` vacía.

Método `tostring`

Serializa `map` como una cadena y regresa. La cadena serializada es similar a la escritura literal. Por ejemplo, el resultado de ejecutar `'str': 1, 0: 2` es `"'str': 1, 0: 2"`. Si el contenedor `map` se refiere a sí mismo, la posición correspondiente utilizará puntos suspensivos en lugar del valor específico:

```
m = {'map': nil,'texto':'hola'}
m['map'] = m
print(m) # {'texto':'hola','map': {...}}
```

Método `insert`

Inserta un par clave-valor en el contenedor `map`. El prototipo de este método es `insert(llave, valor)`, el parámetro `llave` es la clave a insertar, y `valor` es el valor a insertar. Si el `map` clave que se va a insertar existe en el contenedor, se actualizará el par clave-valor original.

Método `remove`

Elimina un par clave-valor del contenedor `map`. El prototipo de este método es `remove(llave)`, y el parámetro `llave` es la clave del par clave-valor que se eliminará.

Método `item`

Obtiene un valor en el contenedor `map`. El prototipo de este método es `item(llave)`, el parámetro `llave` es la clave del valor a obtener, y el valor de retorno del método es el valor correspondiente a la clave.

Método `setitem`

Establece el valor correspondiente a la clave especificada en el contenedor. El prototipo de este método es `setitem(clave, valor)`, `clave` es la clave del par clave-valor a escribir, y `valor` es el valor a escribir. Si no hay un par clave-valor con la clave `clave` en el contenedor, el método `setitem` fallará.

Método `size`

Devuelve el número de pares clave-valor del contenedor `map`, que es la longitud del contenedor. El prototipo de este método es `size()`.

Método contains

Devuelve `true` booleano si se encuentra un par clave-valor coincidente en el contenedor `map`; de lo contrario, `false`. El prototipo de este método es `contains(llave)`.

Método find

Devuelve el valor correspondiente a la clave especificada en el contenedor. El prototipo de este método es `find(llave)` o `find(llave, valor_defecto)`, `llave` es la clave del par clave-valor al que se accederá, y `valor_defecto` es el valor predeterminado devuelto si la clave no se encuentra. Si no se especifica ningún valor predeterminado, se devuelve `nil` en su lugar.

Clase range

La clase se usa para representar un intervalo cerrado entero. Utilice el operador binario `..` para construir una instancia de `range`. Los operandos izquierdo y derecho del operador deben ser números enteros. Por ejemplo, `0..10` significa el intervalo entero [0,10] .

Si no especifica el rango alto, se establece en `MAXINT`. Ejemplo: `imprimir(0..) # (0..9223372036854775807)`

Por lo general, hay dos formas de recorrer una lista:

```
l = [1,2,3,4]
for e:l print(e) end # 1/2/3/4
for i:0..size(l)-1 print(l[i]) end # 1/2/3/4
```

Clase bytes

Los objetos `bytes` se representan como matrices de bytes hexadecimales. El constructor `bytes` toma una cadena de Hex y construye el búfer en memoria.

Ejemplo:

```
b = bytes()
print(b)   # bytes('')
b = bytes("1155AA")  # secuencia de bytes 0x11 0x55 0xAA
size(b)  # 3 = 3 bytes
b[0]     # 17 (0x11)
b[0] = 16  #  asigna el primer byte
print(b)   # bytes('1055AA')
```

Método bytes (Constructor)

Inicializar una matriz de bytes. Hay varias opciones.

Opción 1: valor vacío

`bytes()` crea una nueva matriz de bytes vacía. `tamaño(bytes()) == 0`.

No hay límite en el tamaño de una matriz de bytes, excepto la memoria disponible. Se asigna un búfer interno y se reasigna en caso de que el anterior fuera demasiado pequeño. El búfer inicial es de 36 bytes, pero puede preasignar un búfer más grande (o más pequeño) si sabe de antemano el tamaño necesario.

De manera similar, el búfer se reduce automáticamente si se usa menos del tamaño necesario.

```
b = bytes(4096)   # 4096 bytes preasignados
```

Opción 2: valor inicial

Si el primer argumento es una "cadena", se analiza como una lista de valores hexadecimales. Puede agregar un segundo argumento opcional para preasignar un búfer más grande.

```
b = bytes("BEEF0000")
print(b)   # bytes('beef0000')
b = bytes("112233", 128)   #  preasignar 128 bytes internamente
print(b)   # bytes('112233')
```

Opción 3: tamaño fijo

Si el tamaño proporcionado es negativo, el tamaño de la matriz es fijo y no se puede reducir ni aumentar.

```
b = bytes(-8)
print(b)   # bytes('0000000000000000')

b = bytes("AA", -4)
print(b)   # bytes('AA000000')

b = bytes("1122334455", -4)
atributo_error: tamaño del objeto en bytes es fijo y no se puede cambiar el tamaño
```

Opción 4: asignación de memoria

Precaución, use con mucho cuidado

En este modo, la matriz de bytes se asigna a una región específica de la memoria. Debe proporcionar la dirección base como `comptr` y el tamaño. El tamaño siempre se fija, ya sea positivo o negativo. Esta función es **peligrosa** ya que puede acceder a cualquier ubicación de la memoria, lo que provoca un bloqueo si la ubicación está protegida o no es válida. Usar con cuidado.

En este caso, `b.ismapped()` devuelve `true` indicando un búfer de memoria mapeado. En todos los demás casos, `b.ismapped()` devuelve `falso`. Esto se usa típicamente para saber si Berry asignó el búfer o no, y si los subelementos deben desasignarse explícitamente.

Ejemplo:

```
import introspect
def f() return 0 end

addr = introspect.toptr(f)
print(addr)   # <ptr: 0x3ffeaf88>

b = bytes(addr, 8)
print(b)   # bytes('F8EAFE3F24000000')
#  este ejemplo muestra los primeros 8 bytes del objeto de función en la memoria
```

Método `size`

Devuelve el número de bytes en la matriz de bytes

```
b = bytes("1122334455")
b.size()    # 5
size(b)     # 5
```

Método `tostring`

Muestra una forma legible por humanos la matriz de bytes en hexadecimal. Por defecto, muestra solo los primeros 32 caracteres. Puede solicitar más caracteres agregando un argumento int con la cantidad máxima de bytes que desea convertir. `tostring` se usa internamente cuando se imprime un objeto. `print(b)` es equivalente a `print(b.tostring())`. Es diferente de `asstring`, que convierte una matriz de bytes en el objeto de cadena de bajo nivel equivalente sin ninguna codificación.

```
b = bytes("1122334455")
b.tostring()    # 'bytes(\'1122334455\')'

b = bytes()
b.resize(64)    # redimenciona a 64 bytes
b.tostring()    # 'bytes(\
↪'0000000000000000000000000000000000000000000000000000000000000000...\')'
b.tostring(500) # 'bytes(\
↪'00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
↪')'
```

Método `tohex`

Convierte la matriz de bytes en una cadena hexadecimal, similar a la devuelta por `tostring()` pero sin decoradores.

```
b = bytes("1122334455")
b.tohex()    # '1122334455'
```

Método `fromhex`

Actualiza el contenido de la matriz de bytes a partir de una nueva cadena hexadecimal. Esto permite cargar una nueva cadena hexadecimal sin asignar un nuevo objeto de bytes.

```
b = bytes("1122334455")
b.fromhex("AABBCC")   # bytes('AABBCC')
```

Método `clear`

Vuelve a poner la matriz de bytes en vacío

```
b = bytes("1122")
b.clear()
print(b)   # bytes()
```

Método `resize`

Reduce o expande la matriz de bytes para que coincida con el tamaño especificado. Si se expande, se agregan bytes NULL (0x00) al final del búfer.

```
b = bytes("11223344")
b.resize(6)
print(b)   # bytes('112233440000')
b.resize(2)
print(b)   # bytes('1122')
```

Métodos de concatenación + y `..`

Puede usar + para concatenar dos listas de `bytes`, creando un nuevo objeto `bytes`. `..` cambia la lista en su lugar y se puede usar para agregar un objeto `int` (1 byte) o `bytes`

```
b = bytes("1122")
c = bytes("3344")
d = b + c           # b y c no cambian
print(d)            # bytes('11223344')
print(b)            # bytes('1122')
print(c)            # bytes('3344')

e = b..c            # ahora b ha cambiado
print(e)            # bytes('11223344')
print(b)            # bytes('11223344')
print(c)            # bytes('3344')
```

Método de acceso a bytes `[]`

Puede acceder a bytes individuales como enteros, para leer y escribir. Los valores que no están en el rango de 0 a 255 se cortan silenciosamente.

```
b = bytes("010203")
print(b[0])         # 1

#  índices negativos cuentan desde el final
print(b[-1])        # 3

#  fuera de los límites genera una excepción
print(b[5])         # index_error: índice de bytes fuera de rango
```

(continues on next page)

(continued from previous page)

```
b[0] = -1
print(b)            # bytes('FF0203')

b[1] = 256
print(b)            # bytes('FF0003')
```

Método de acceso de rango []

Puede usar el descriptor de acceso [] con un rango para obtener una sublista de bytes. Si un índice es negativo, se toma del final de la matriz.

Esta construcción no se puede usar como un *lvalue*, es decir, no se puede empalmar como `b[1..2] = bytes("0011") # no permitido`.

```
b = bytes("001122334455")
print(b[1..2])      # bytes('1122')

# elimina los primeros 2 bytes
print(b[2..-1])     # bytes('22334455')

# eliminar los últimos 2 bytes
print(b[0..-3])     # bytes('00112233')

#  se permite el sobreimpulso
print(b[4..10])     # bytes('4455')

#  índices invertidos devuelven una matriz vacía
print(b[5..4])      # bytes('')
```

Los métodos estándar item y setitem se implementan y se asignan de forma transparente al operador [].

Método copy

Crea una nueva copia nueva del objeto bytes. Se asigna un nuevo búfer de memoria y se duplican los datos.

```
b = bytes("1122")
print(b)          # bytes('1122')

c = b.copy()
print(c)          # bytes('1122')

b.clear()
print(b)          # bytes('')
print(c)          # bytes('1122')bytes('1122')
```

Métodos `get, geti`

Lea un valor de 1/2/4 bytes de cualquier desplazamiento en la matriz de bytes. El modo estándar es little endian, si se especifica un tamaño negativo habilita big endian. `get` devuelve valores sin signo, mientras que `geti` devuelve valores con signo.

```
b.get(<offset>, <size>) ->  objeto de bytes
```

Si el desplazamiento está fuera de rango, se devuelve `0` (no se genera ninguna excepción).

Ejemplo:

```
b = bytes("010203040506")
print(b.get(2,2))          # 1027 - 0x0403 read 2 bytes little endian
print(b.get(2,-2))         #  772 - 0x0304 read 2 bytes big endian

print(b.get(2,4))          # 100992003 - 0x06050403 - little endian
print(b.get(2,-4))         #  50595078 - 0x03040506 - big endian

b = bytes("FEFF")
print(b.get(0, 2))         # 65534 - 0xFFFE
print(b.geti(0, 2))        # -2 - 0xFFFE
```

Métodos `set, seti`

Similar a `get` y `geti`, permite establecer un valor de 1/2/4 bytes en cualquier desplazamiento. `seti` usa números enteros con signo, `set` no tiene signo (en realidad, no hace la diferencia).

Si el desplazamiento está fuera de rango, no se realiza ningún cambio (no se genera ninguna excepción).

```
bytes.set(<offset>, <valor>, <tamaño>)
```

Método `add`

Este método agrega valor de 1/2/4 bytes (little endian o big endian) al final del búfer. Si el tamaño es negativo, el valor se trata como big endian.

```
b.add(<valor>, <tamaño>)
```

Ejemplo:

```
b = bytes("0011")
b.add(0x22, 1)
print(b)              # bytes('001122')
b.add(0x2233, 2)
print(b)              # bytes('0011223322')
b.add(0x22334455, 4)
print(b)              # bytes('001122332255443322')
b.add(0x00)
print(b)              # bytes('0011223322554433200')
b.clear()
b.add(0x0102, -2)
```

(continues on next page)

(continued from previous page)

```
print(b)              # bytes('0102')
b.add(0x01020304, -4)
print(b)              # bytes('010201020304')
```

Método asstring

Convierte un búfer de bytes en una cadena. El búfer se convierte tal cual sin ninguna consideración de codificación. Si el búfer contiene caracteres NULL, la cadena se truncará.

```
b=bytes("3344")
print(b.asstring())   # '3D'
```

Método fromstring

Convierte un búfer de bytes en una cadena. El búfer se convierte tal cual sin ninguna consideración de codificación. Si el búfer contiene caracteres NULL, la cadena se truncará.

```
b=bytes().fromstring("Hola")
print(b)              # bytes('48656C6C6F')
```

Métodos de manipulación de bits setbits, getbits

Puede leer y escribir a nivel de subbytes, especificando de qué bit a qué bit. El desplazamiento está en bits, no en bytes. Agregue el número de bytes * 8.

```
b.setbits(<offset_bits>, <len_bits>, <value>)
b.getbits(<offset_bits>, <len_bits>)
```

Codificación en base64, método tob64

Convierte una matriz de bytes en una cadena base64.

```
b = bytes('deadbeef0011')
s = b.tob64()
print(s)              # 3q2+7wAR
```

Decodificación en base64, método fromb64

Convierte una cadena base64 en una matriz de bytes.

```
s = '3q2+7wAR'
b = bytes().fromb64(s)
print(b)              # bytes('DEADBEEF0011')
```

Métodos `getfloat` y `setfloat`

Similar a `get/set`, permite leer o escribir un valor en coma flotante de 32 bits.

```
b.getfloat(<offset>)
b.getfloat(<offset>, <number>)
```

```
b = bytes("00000000")
b.getfloat(0)     # 0
b.setfloat(0, -1.5)
print(b)  # bytes('0000C0BF')
b.getfloat(0)  # -1.5
```

Método `_buffer`

Característica avanzada: devuelve la dirección del búfer en la memoria, para usar con código C.

```
b = bytes('1122')
b._buffer()  # <ptr: 0x600000c283c0>
```

Método `_change_buffer`

Característica avanzada: funciona solo para búferes mapeados (es decir, `b.ismapped() == true`), permite reasignar el búfer a una nueva dirección de memoria. Esto permite reutilizar el objeto bytes() sin reasignar una nueva instancia.

```
#  este ejemplo usa la asignación de punteros, use con mucho cuidado
b1 = bytes("11223344")
b2 = bytes("AABBCCDD")
b1._buffer()  # <ptr: 0x600000c2c390>
b2._buffer()  # <ptr: 0x600000c24270>

# ahora creamos c como un búfer asignado de 4 bytes a la dirección de b1
c = bytes(b1._buffer(), 4)
print(c)  # bytes('11223344') -- asignado a b1
c._buffer()   # <ptr: 0x600000c2c390>

#  cambiemos un byte para probarlo
c[0] = 254
print(c)  # bytes('FE223344')
print(b1) # bytes('FE223344') -- b1 ha cambiado

#  reasignar c al mapa b2
c._change_buffer(b2._buffer())
print(c)  # bytes('AABBCCDD')
c._buffer()  # <ptr: 0x600000c24270>
```

Módulos de expansión

Módulo JSON

JSON es un formato ligero de intercambio de datos. Es un subconjunto de JavaScript. Utiliza un formato de texto que es completamente independiente del lenguaje de programación para representar datos. Berry proporciona un módulo JSON para proporcionar soporte para datos JSON. El módulo JSON solo contiene dos funciones, "cargar" y "volcar", que se utilizan para analizar cadenas JSON y multiplicar objetos Berry y serializar un objeto Berry en texto JSON.

Función `load`

```
load(text)
```

Descripción

Esta función se usa para convertir el texto JSON de entrada en un objeto Berry y devolverlo. Las reglas de conversión se muestran en la Tabla 1.1 . Si hay un error de sintaxis en el texto JSON, la función devolverá `nil`.

Tipo JSON	Tipo Berry
nulo	nil
número	entero o real
cadena	cadena
matriz	lista
objeto	mapa

Tabla 9: Reglas de conversión de tipo JSON a tipo Berry

Ejemplo

```
import json
json.load('0') # 0
json.load('[{"nombre": "liu", "edad": 13}, 10.0]') # [{'nombre':'liu','edad': 13}, 10]
```

Función `dump`

```
dump(objectp, ['formato'])
```

Descripción

Esta función se usa para serializar el objeto Berry en texto JSON. Las reglas de conversión para la serialización se muestran en la Tabla 10 .

Tipo berry	Tipo JSON
cero	nulo
entero	número
verdadero	número
lista	matriz
mapa	objeto
mapa Clave de	cadena
otro	cadena

Tabla 10: Reglas de conversión de tipo Berry a tipo JSON

Ejemplo

```
import json
json.dump('string') #'"string"'
json.dump('string') #'"string"'
json.dump({0:'item 0','list': [0, 1, 2]}) #'{"0":"item 0","list":[0,1,2]}'
json.dump({0:'item 0','list': [0, 1, 2],'func': print},'format')
#-
{
    "0": "item 0",
    "list": [
        0,
        1,
        2
    ],
    "func": "<function: 00410310>"
}
-#
```

Módulo matemático

Este módulo se utiliza para proporcionar soporte para funciones matemáticas, como las funciones trigonométricas y las funciones de raíz cuadrada de uso común. Para usar el módulo matemático, primero use la instrucción `import math`. Todos los ejemplos de esta sección asumen que el módulo se ha importado correctamente.

Constante `pi`

El valor de Pi es un tipo de número real, aproximadamente igual a 3.141592654.

Ejemplo

```
math.pi # 3.14159
```

Función `abs`

```
abs(valor)
```

Descripción

Esta función devuelve el valor absoluto del parámetro, que puede ser un número entero o un número real. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primer parámetro. El tipo de retorno de la función `abs` es un número real.

Ejemplo

```
math.abs(-1) # 1
math.abs(1.5) # 1.5
```

Función `ceil`

```
ceil(valor)
```

Descripción

Esta función devuelve el valor redondeado hacia arriba del parámetro, es decir, el valor entero más pequeño mayor o igual que el parámetro. El parámetro puede ser un número entero o un número real. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primero. El tipo de retorno de la función es un número real.

Ejemplo

```
math.ceil(-1.2) # -1
math.ceil(1.5) # 2
```

Función `floor`

```
floor(valor)
```

Descripción

Esta función devuelve el valor redondeado hacia abajo del parámetro, que no es mayor que el valor entero máximo del parámetro. El parámetro puede ser un número entero o un número real. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primero. El tipo de retorno de la función es un número real.

Ejemplo

```
math.floor(-1.2) # -2
math.floor(1.5) # 1
```

Función `sin`

```
sin(valor)
```

Descripción

Esta función devuelve el valor de la función seno del parámetro. El parámetro puede ser un número entero o un número real, y la unidad son los radianes. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primer parámetro. El tipo de retorno de la función es un número real.

Ejemplo

```
math.sin(1) # 0.841471
math.sin(math.pi * 0.5) # 1
```

Función cos

```
cos(valor)
```

Descripción

Esta función devuelve el valor de la función coseno del parámetro. El parámetro puede ser un número entero o un número real en radianes. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primero. El tipo de retorno de la función es un número real.

Ejemplo

```
math.cos(1) # 0.540302
math.cos(math.pi) # -1
```

Función tan

```
tan(valor)
```

Descripción

Esta función devuelve el valor de la función tangente del parámetro. El parámetro puede ser un número entero o un número real, en radianes. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primero. El tipo de retorno de la función es un número real.

Ejemplo

```
math.tan(1) # 1.55741
math.tan(math.pi / 4) # 1
```

Función asin

```
asin(valor)
```

Descripción

Esta función devuelve el valor de la función arco seno del parámetro. El parámetro puede ser un número entero o un número real. El rango de valores es [1,1]. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primero. El tipo de retorno de la función es un número real y la unidad es radianes.

Ejemplo

```
math.asin(1) # 1.5708
math.asin(0.5) * 180 / math.pi # 30
```

Función acos

```
acos(valor)
```

Descripción

Esta función devuelve el valor de la función de arco coseno del parámetro. El parámetro puede ser un número entero o un número real. El rango de valores es [1,1]. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primero. El tipo de retorno de la función es un número real y la unidad es radianes.

Ejemplo

```
math.acos(1) # 0
math.acos(0) # 1.5708
```

Función atán

```
atan(valor)
```

Descripción

Esta función devuelve el valor de la función arco tangente del parámetro. El parámetro puede ser un número entero o un número real. El rango de valores es [∞,+∞]. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primero. El tipo de retorno de la función es un número real y la unidad es radianes.

Ejemplo

```
math.atan(1) * 180 / math.pi # 45
```

Función sinh

```
sinh(valor)
```

Descripción

Esta función devuelve el valor de función de seno hiperbólico del parámetro. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primer parámetro. El tipo de retorno de la función es un número real.

Ejemplo

```
math.sinh(1) # 1.1752
```

Función cosh

```
cosh(valor)
```

Descripción

Esta función devuelve el valor de la función coseno hiperbólico del parámetro. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primero. El tipo de retorno de la función es un número real.

Ejemplo

```
math.cosh(1) # 1.54308
```

Función tanh

```
tanh(valor)
```

Descripción

Esta función devuelve el valor de la función tangente hiperbólica del parámetro. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primero. El tipo de retorno de la función es un número real.

Ejemplo

```
math.tanh(1) # 0.761594
```

Función sqrt

```
sqrt(valor)
```

Descripción

Esta función devuelve la raíz cuadrada del argumento. El parámetro de esta función no puede ser negativo. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primero. El tipo de retorno de la función es un número real.

Ejemplo

```
math.sqrt(2) # 1.41421
```

Función exp

```
exp(valor)
```

Descripción

Esta función devuelve el valor de la función exponencial del parámetro en función de la constante natural *e*. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primero. El tipo de retorno de la función es un número real.

Ejemplo

```
math.exp(1) # 2.71828
```

Función `log`

```
log(valor)
```

Descripción

Esta función devuelve el logaritmo natural del argumento. El parámetro debe ser un número positivo. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primero. El tipo de retorno de la función es un número real.

Ejemplo

```
log(valor)
```

Función `log10`

```
log10(valor)
```

Descripción

Esta función devuelve el logaritmo del parámetro en base 10. El parámetro debe ser un número positivo. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primero. El tipo de retorno de la función es un número real.

Ejemplo

```
math.log10(10) # 1
```

Función `deg`

```
deg(valor)
```

Descripción

Esta función se utiliza para convertir radianes en ángulos. La unidad del parámetro es radianes. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primero. El tipo de retorno de la función es un número real y la unidad es un ángulo.

Ejemplo

```
math.deg(math.pi) # 180
```

Función `rad`

```
rad(valor)
```

Descripción

Esta función se utiliza para convertir ángulos a radianes. La unidad del parámetro es el ángulo. Si no hay parámetros, la función devuelve `0`, si hay varios parámetros, solo se procesa el primero. El tipo de retorno de la función es un número real y la unidad es radianes.

Ejemplo

```
math.rad(180) # 3.14159
```

Función pow

```
pow(x, y)
```

Descripción

El valor de retorno de esta función es el resultado de la expresión $x\ y$, que es el parámetro `x` elevado a `y`. Si los parámetros no están completos, la función devuelve `0`, si hay parámetros adicionales, solo se procesan los dos primeros parámetros. El tipo de retorno de la función es un número real.

Ejemplo

```
math.pow(2, 3) # 8
```

Función srand

```
srand(valor)
```

Descripción

Esta función se utiliza para establecer la semilla del generador de números aleatorios. El tipo del parámetro debe ser un número entero.

Ejemplo

```
math.srand(2)
```

Función rand

```
rand()
```

Descripción

Esta función se utiliza para obtener un número entero aleatorio.

Ejemplo

```
math.rand()
```

Módulo de tiempo

Este módulo se utiliza para proporcionar funciones relacionadas con el tiempo.

Función `time`

```
time()
```

Descripción

Devuelve la marca de tiempo actual. La marca de tiempo es el tiempo transcurrido desde Unix Epoch (1 de enero de 1970 00:00:00 UTC), en segundos.

Función `dump`

```
dump(ts)
```

Descripción

La marca de tiempo de entrada `ts` se convierte en un `map` de tiempo, y la correspondencia clave-valor se muestra en la siguiente tabla:

clave	valor	clave	valor	clave	valor
`` 'year'''	Año (desde 1900)	`' month'`	Mes (1-12)	`` 'day''	Día (1-31)
`` 'hour'''	Hora (0-23)	`` 'min''	Puntos (0-59)	`` 'sec''	Segundos (0-59)
`'we ekday'`	Semana (1-7)				

Tabla 11: La relación clave-valor del valor de retorno de la función ``time.dump``

Función `clock`

```
clock()
```

Descripción

Esta función devuelve el tiempo transcurrido desde el inicio de la ejecución del intérprete hasta que se llama a la función en segundos. El valor de retorno de esta función es del tipo "real" y su precisión de tiempo está determinada por la plataforma específica.

Módulo de cadena (string)

El módulo `cadena` proporciona funciones de procesamiento de cadenas.

Para usar el módulo de cadena, primero use la instrucción `import string`. Todos los ejemplos de esta sección asumen que el módulo se ha importado correctamente.

Función `count`

```
string.count(s, sub[, inicio[, fin]])
```

Cuenta el número de ocurrencias de la subcadena en la cadena s. Busque desde la posición entre `inicio` y `fin` de `s` (el valor predeterminado es 0 y tamaño(s)).

Función `split`

```
string.split(s, pos)
```

Divide la cadena s en dos subcadenas en la posición `pos` y devuelve la lista de esas cadenas.

```
string.split(s, sep[, num])
```

Divide la cadena s en subcadenas dondequiera que ocurra sep, y devuelve la lista de esas cadenas. Dividir como máximo un número de veces (el valor predeterminado es string.count(s, sep)).

Función `find`

```
string.find(s, sub[, inicio[, fin]])
```

Compruebe si la cadena s contiene la subcadena sub. Si se especifican el inicio y el final (el valor predeterminado es 0 y el tamaño(s)), se buscarán en este rango.

Función `hex`

```
hex(numero)
```

Convertir número a cadena hexadecimal.

Función `byte`

```
byte(s)
```

Obtiene el valor del código del primer byte de la cadena s.

Función `char`

```
char(numero)
```

Convierte el número usado como código en un carácter.

Función `format`

```
string.format(fmt[, args])
```

Devuelve una cadena formateada. El patrón que comienza con '%' en la plantilla de formato fmt será reemplazado por el valor de [args]: %[flags][fieldwidth][.precision]type

Ti po	Descripción
%d	Entero decimal
%o	Entero octal
%x	Entero hexadecimal en minúsculas
%X	Entero hexadecimal en mayúsculas
%x	Entero octal en minúsculas
%X	Entero octal en mayúsculas
%f	Punto flotante en la forma [-]nnnn.nnnn
%e %E	Punto flotante en exp. forma [-]n.nnnn e [+|-]nnn, mayúsculas si %E
%g %G	Punto flotante como %f si 4 < exp. precision, sino como %e; mayúsculas si %G
%c	Carácter que tiene el código pasado como entero
%s	Cadena sin ceros incrustados
%q	Cadena entre comillas dobles, con caracteres especiales escapados
%%	El carácter '%' (escapado)

Tipo	Descripción
•	Justificación a la izquierda, el valor predeterminado es justificación a la derecha
•	antepone el signo (se aplica a los números)
(espa cio)	Antepone signo si es negativo, de lo contrario espacio
#	Agrega "0x" antes de %x, fuerza el punto decimal; para %e, %f, deja ceros finales para %g

Ancho de campo y precisión	Descripción
n	Pone al menos n caracteres, rellena con espacios en blanco
0n	Pone al menos n caracteres, teclado izquierdo con ceros
.n	Usa al menos n dígitos para números enteros, redondee a n decimales para punto flotante o no más de n caracteres. para cadenas

Módulo os

El módulo OS proporciona funciones relacionadas con el sistema, como funciones relacionadas con archivos y rutas. Estas funciones están relacionadas con la plataforma. Actualmente, los códigos de estilo Windows VC y POSIX se implementan en el intérprete de Berry. Si se ejecuta en otras plataformas, no se garantiza que se proporcionen las funciones en el módulo del sistema operativo.

[COMPLETAR]

Módulo `global`

El módulo `global` proporciona una forma de acceder a las variables globales a través de un módulo. El compilador Berry comprueba que existe un global al compilar el código. Sin embargo, hay casos en los que los globales se crean dinámicamente por código y aún no se conocen en tiempo de compilación. El uso del módulo `global` da total libertad para acceder a variables globales estáticas o dinámicas.

Acceder a un global es simplay hecho con `global.<name>` para leer y escribir. También puede usar la sintaxis especial `global.(nombre)` si `nombre` es una variable que contiene el nombre del global como cadena.

Ejemplo:

```
> import global
> a = 1
> global.a
1
>
> b
syntax_error: stdin:1: 'b' no declarado (primer uso en esta función)
> global.b = 2
> b
2
> global.b
2
> var nombre = "b"
> global.(nombre)
2
```

Llamar a `global()` devuelve la lista de todos los nombres globales actualmente definidos (los componentes integrados no están incluidos).

```
> import global
> a = 1
> global.b = 2
> global()
['_argv', 'b', 'global', 'a']
```

`global.contains(<nombre>) -> bool` proporciona una manera fácil de saber si un nombre global ya está definido.

```
> import global
> global.contains("g")
false
> g = 1
> global.contains("g")
true
```

Módulo `introspect`

El módulo `introspect` proporciona primitivas para acceder dinámicamente a variables o módulos. Usar con `import introspect`.

`introspect.members(objeto: clase o módulo o instancia o nil) -> lista` devuelve la lista de nombres de miembros para la `clase`, `instancia` o `módulo`. Tenga en cuenta que no incluye miembros virtuales potenciales creados a través de `member` y `setmember`.

`introspect.members()` devuelve la lista de variables globales (sin incluir las incorporadas) y es equivalente a `global()`

`introspect.get(objeto: clase o instancia o módulo, nombre:cadena) -> cualquiera` e `introspect.set(objeto: clase o instancia o módulo, nombre:cadena, valor:cualquiera) -> nil` permite leer y escribir cualquier miembro por su nombre.

`introspect.get(o, "a")` es equivalente a `oa`, `introspect.set(o, "a", 1)` es equivalente a `oa = 1`. También hay una sintaxis alternativa: `o.("a")` es equivalente a `oa` y `o.("a) = 1` es equivalente a `oa = 1`.

`introspect.module(nombre:cadena) -> any` es equivalente a `import nombre` excepto que no crea la variable global o local, sino que devuelve el módulo. Esta es la única manera de cargar un módulo con un nombre dinámico, `import nombre` solo toma un nombre estático.

`introspect.toptr(addr:int) -> comptr` convierte un número entero en un puntero comptr. `introspect.fromptr(addr:comptr) -> int` hace lo contrario y convierte un puntero en un int. Advertencia: usar con cuidado. En plataformas donde `int` y `void*` no tienen el mismo tamaño, estas funciones seguramente darán resultados inutilizables.

`introspect.ismethod(f:function) -> bool` comprueba si la función proporcionada es un método de una instancia (tomando a sí mismo como primer argumento) o una función simple. Esto se usa principalmente para evitar un error común de pasar un método de instancia como callback, donde debe usar un cierre que capture la instancia como `/ -> self.do()`.

Módulo `solidify`

Este módulo permite solidificar el bytecode de Berry en flash. Esto permite ahorrar RAM ya que el código está en ROM. Esto lo convierte en una buena alternativa a las funciones nativas de C.

Ver 8.4 Solidificación

8. Características avanzadas

8.1 Modo `estricto`

Berry permite total libertad del desarrollador. Pero después de un poco de experiencia en la codificación con Berry, encontrará que hay errores comunes que son difíciles de encontrar y que el compilador podría ayudarlo a detectar. El modo `estricto` realiza verificaciones adicionales **en tiempo de compilación** sobre algunos errores comunes.

Este modo está habilitado con `import strict` o cuando se ejecuta Berry con la opción `-s`: `berry -s`

`var` obligatorio para variables locales

Este es el error más común, una variable asignada sin `var` es global si ya existe una variable global o local en caso contrario. El modo estricto rechaza la asignación si no hay un global con el mismo nombre.

No más permitido:

```
def f()
  i = 0    # this is a local variable
  var j = 0
end
```

syntax_error: stdin:2: strict: no global 'i', ¿quiso decir 'var i'?

Pero todavía funciona para globales:

```
g_i = 0
def f()
  g_i = 1
end
```

Sin anulación de elementos integrados

Berry permite anular una función incorporada. Sin embargo, esto generalmente no es deseable y es una fuente de errores difíciles de encontrar.

```
map = 1
syntax_error: stdin:1: estricto: redefinición de 'map' incorporado
```

Múltiples `var` con el mismo nombre no permitidos en el mismo ámbito

Berry toleraba la declaración múltiple de una variable local con el mismo nombre. Esto ahora se considera como un error (incluso sin modo estricto).

```
def f()
  var a
  var a   # redefinición de a
end
syntax_error: stdin:3: redefinición de 'a'
```

No ocultar la variable local del alcance externo

En Berry puedes declarar variables locales con el mismo nombre en el ámbito interno. La variable en el ámbito interno oculta la variable del ámbito externo durante la duración del ámbito.

La única excepción son las variables que comienzan con el punto '.' que se pueden enmascarar desde el alcance externo. Este es el caso de la variable local oculta `.it` cuando se incrustan múltiples `for`.

```
def f()
  var a    # variable en el ámbito externo
  if a
    var a  # redefinición de a en ámbito interno
  end
end
syntax_error: stdin:4: estricto:  redefinición de 'a' desde el ámbito externo
```

8.2 Miembros virtuales

Los miembros virtuales le permiten agregar de forma dinámica y programática miembros y métodos a clases y módulos. Ya no está limitado a los miembros declarados en el momento de la creación.

Esta función está inspirada en `__getattr__()` / `__setattr__()` de Python. La motivación proviene de la integración de LVGL a Berry en Tasmota. La integración necesita cientos de constantes en un módulo y miles de métodos asignados a funciones C. La creación estática de atributos y métodos funciona, pero consume una cantidad significativa de espacio de código.

Esta característica permite crear dos métodos:

Método Berry	Descripción
`` `member` ``	`(nombre:cadena) -> any` Debería devolver el valor del `nombre`
`` `setmember` ``	`(nombre:cadena, valor:any) especificado -> nil` Debería almacenar el 'valor' en el miembro virtual con el 'nombre' especificado

Módulo `undefined`

La función `member()` debe ser capaz de distinguir entre un miembro con un valor `nil` y el miembro que no existe. Para evitar cualquier ambigüedad, la función `member()` puede indicar que el miembro no existe de dos maneras:

- generar una excepción - o `import undefined` y devolver el módulo `undefined`. Esto se usa como un marcador para que la VM sepa que el atributo no existe, mientras se beneficia de excepciones consistentes.

Ejemplo de un objeto dinámico al que puede agregar miembros, pero devolvería un error si el miembro no se agregó previamente.

```
class dyn
    var _attr
    def init()
        self._attr = {}
    end
    def setmember(nombre, valor)
        self._attr[nombre] = valor
    end
```

(continues on next page)

(continued from previous page)

```
    def member(nombre)
        if self._attr.contains(nombre)
            return self._attr[nombre]
        else
            import undefined
            return undefined
        end
    end
end
```

Ejemplo de uso:

```
a = dyn()
a.a
```

attribute_error: el objeto 'dyn' no tiene el atributo 'a' stack traceback: stdin:1: en función *main*

a.a = 1 a.a

1

a.a = nil a.a

Llamada implícita de `member()`

Cuando se ejecuta el siguiente código `a.b`, Berry VM hace lo siguiente:

- Obtiene el objeto llamado `a` (local o global), genera una excepción si no existe
- Comprueba si el objeto `a` es de tipo `módulo`, `instancia` o `clase`. Genera una excepción de lo contrario
- Comprueba si el objeto `a` tiene un miembro llamado `b`. En caso afirmativo, devuelve su valor, en caso negativo, procede a continuación
- Si el objeto `a` es del tipo `clase`, genera una excepción porque los miembros virtuales no funcionan para métodos estáticos (clase)
- Comprueba si el objeto `a` tiene un miembro llamado `member` y es una `función`. En caso afirmativo, lo llama con el parámetro `"b"` como cadena. Si no, genera una excepción
- Comprueba el valor de retorno. Si es el módulo `undefined` genera una excepción que indica que el miembro no existe

Llamada implícita de `setmember()`

Cuando se ejecuta el siguiente código `ab = 0` (mutador), Berry VM hace lo siguiente:

- Obtiene el objeto llamado `a` (local o global), genera una excepción si no existe
- Comprueba si el objeto `a` es de tipo `módulo`, `instancia` o `clase`. Genera una excepción de lo contrario
  - Si `a` es del tipo `clase`, comprueba si existe el miembro `b`. En caso afirmativo, cambia su valor. Si no, genera una excepción. (los miembros virtuales no funcionan para clases o métodos estáticos)
  - Si `a` es del tipo `instancia`, comprueba si existe el miembro `b`. En caso afirmativo, cambia su valor. Si no, procede a continuación
    - Comprueba si `a` tiene un miembro llamado `setmember`. Si es así, lo llama, si no, genera una excepción.

- Si `a` es de tipo `módulo`. Si el módulo no es de solo lectura, crea o cambia el valor (`setmember` nunca se llama para un módulo de escritura). Si el módulo es de solo lectura, entonces se llama a `setmember` si existe.

Manejo de excepciones

Para indicar que un miembro no existe, `member()` devolverá `undefined` después de `import undefined`. También puede generar una excepción en `member()`, pero tenga en cuenta que Berry podría intentar llamar a métodos como `tostring()` que aterrizarán en su método `member()` si no existen como métodos estáticos. Para indicar que un miembro no es válido, `setmember()` debe generar una excepción o devolver `undefined`. Devolver cualquier otra cosa como `nil` indica que la asignación fue exitosa. Tenga en cuenta que puede recibir nombres de miembros que no sean identificadores válidos de Berry. La sintaxis `a.("<->")` llamará a `a.member("<->")` con un nombre de miembro virtual que no es léxicamente válido, es decir, no se puede llamar en código normal, excepto mediante el uso indirecto formas como `introspect` o `member()`.

Especificidades para las clases

El acceso a los miembros del objeto de clase no desencadena miembros virtuales. Por lo tanto, no es posible tener métodos estáticos virtuales.

Especificidades de los módulos

Los módulos admiten la lectura de miembros estáticos con `member()`. Al escribir en un miembro, el comportamiento depende de si el módulo es de escritura (en la memoria) o de solo lectura (en el firmware). Si se puede escribir en el módulo, los nuevos miembros se agregan directamente al módulo y nunca se llama a `setmember()`. Si el módulo es de solo lectura, se llama a `setmember()` cada vez que intenta cambiar o crear un miembro. Entonces es su responsabilidad almacenar los valores en un objeto separado como un global.

Ejemplo

```
class T
    var a
    def init()
        self.a = 'a'
    end

    def member(nombre)
        return "miembro "+nombre
    end

    def setmember(nombre, valor)
        print("Almacenar '"+nombre+"': "+str(valor))
    end
end
t=T()
```

Ahora intentémoslo:

```
t.a
```

'a'

```
t.b
```

'miembro b'

```
t.foo
```

'miembro foo'

```
t.bar = 2
```

Almacenar 'bar': 2

Esto también funciona para los módulos:

```
m = module()
m.a = 1
m.member = def (nombre)
    return "miembro "+nombre
end
m.setmember(nombre, valor)
    print("Almacenar '"+nombre+"': "+str(valor))
end
```

Intentemoslo:

```
m.a
```

1

```
m.b
```

'miembro b'

```
m.c = 3   # la asignación es válida por lo que no se llama a `setmember()
m.c
```

3

Ejemplo más avanzado:

```
class A
    var i

    def member(n)
      if n == 'ii' return self.i end
      return nil     # lo hacemos explícito aquí, pero esta línea es opcional
    end

    def setmember(n, v)
      if n == 'ii' self.i = v end
    end
  end
a=A()
```

(continues on next page)

(continued from previous page)

```
a.i      # devuelve nil
a.ii     # i llama implícitamente `a.member("ii")`
```

attribute_error: el objeto 'A' no tiene atributo 'ii'
stack traceback:
stdin:1: en función *main*

```
# devuelve un excepción ya que el miembro es nulo (considerado inexistente)

a.ii = 42    # llama implícitamente `a.setmember("ii", 42)`
a.ii         # llama implícitamente `a.member("ii")` and returns `42`
```

42

```
a.i          #  la variable concreta también fue cambiada
```

42

8.3 Cómo empaquetar un módulo

Esta guía lo lleva a través de las diferentes opciones de empaquetado de código para su reutilización utilizando la directiva de "import" de Berry.

Comportamiento de `import`

Cuando se utiliza `import <modulo> [as <nombre> ]`, suceden los siguientes pasos:

- Hay una caché global de todos los módulos ya importados. Si `<modulo>` ya fue importado, `import` devuelve el valor en caché ya devuelto por la primera llamada a `import`. No se realizan otras acciones.
- `import` busca un módulo de nombre `<modulo>` en el siguiente orden:

1. en módulos nativos incrustados en el firmware en tiempo de compilación
2. en el sistema de archivos, comenzando con el directorio actual, luego iterando en todos los directorios desde `sys.path`: busque el archivo `<nombre>`, entonces `<nombre>.bec` (código de bytes compilado), luego `<nombre>.be`. Si `BE_USE_SHARED_LIB` está habilitado, también busca bibliotecas compartidas como `<nombre>.so que` o `<nombre>.dll` aunque esta opción generalmente no está disponible en MCU.

- Se ejecuta el código cargado. El código debe terminar con una declaración `return`. El objeto devuelto se almacena en la memoria caché global y se pone a disposición de la persona que llama (en el ámbito local o global).
- Si el objeto devuelto es un `módulo` y si el módulo posee un miembro `init`, entonces se toma un paso adicional. La función `<modulo>.init(m)` se llama pasando como argumento el propio objeto del módulo. El valor devuelto por `init()` reemplaza el valor en el caché global. Tenga en cuenta que `init()` se llama como máximo una vez durante la primera `importación`.

Nota: una función `init(m)` implícita siempre está presente en todos los módulos, incluso si no se declaró ninguno. Esta función implícita no tiene ningún efecto.

Empaquetado de un módulo

Aquí hay un ejemplo simple de un módulo:

Archivo `demo_modulo.be`:

```
# modulo simple
# use `import demo_modulo`

demo_module = module("demo_module")

demo_modulo.foo = "bar"

demo_modulo.decir_hola = def ()
    print("Hola Berry!")
end

return demo_modulo      # devuelve el módulo como salida de import
```

Ejemplo de uso:

```
import demo_modulo

demo_modulo
<module: demo_modulo>

demo_module.decir_hola()
```

Hola Berry!

```
demo_modulo.foo
```

'bar'

```
demo_modulo.foo = "baz"   # el módulo se puede escribir, aunque esto es muy desaconsejado
demo_modulo.foo
```

'baz'

Empaquetar un singleton (mónada)

El problema de usar módulos es que no tienen variables de instancia para realizar un seguimiento de los datos. Están diseñados esencialmente para bibliotecas sin estado.

A continuación, encontrará una forma elegante de empaquetar una clase única devuelta como una "declaración de importación".

Para ello, utilizamos diferentes trucos. Primero, declaramos la clase para el singleton como una clase interna de una función, esto evita que se contamine el espacio de nombres global con esta clase. Es decir, la clase no será accesible por otro código.

En segundo lugar, declaramos una función `init()` del módulo que crea la clase, crea la instancia y la devuelve.

Según este esquema, `import <modulo>` en realidad devuelve una instancia de una clase oculta.

Ejemplo de `demo_monad.be`:

```
# monada simple
# use `import demo_monad`

demo_monad = module("demo_monad")

#  el módulo tiene un solo miembro `init()` y delega todo a la clase interna
demo_monad.init = def (m)

    # inncer class
    class my_monad
        var i

        def init()
            self.i = 0
        end

        def say_hello()
            print("Hola Berry!")
        end
    end

    # rdevolver una sola instancia para esta clase
    return my_monad()
end

return demo_monad      # evuelve el módulo como la salida de importación, que␣
↪eventualmente se reemplaza por el valor de retorno de 'init()'
```

Ejemplo:

```
import demo_monad
demo_monad
<instance: my_monad()>     # es una instancia no un modulo

demo_monad.say_hello()
```

Hola Berry!

```
demo_monad.i = 42        #  puedes usarlo como cualquier instancia
demo_monad.i
```

42

```
demo_monad.j = 0         # hay una fuerte verificación de miembros en comparación con␣
↪los módulos
```

Attribute_error: la clase 'my_monad' no puede asignarse al atributo 'j' stack traceback: stdin:1: en función *main*

8.4 Solidificación

La solidificación es el proceso de capturar estructuras y códigos Berry compilados (clases, módulos, mapas, listas…) y almacenarlos en el firmware. Reduce drásticamente el uso de la memoria, pero tiene algunas limitaciones.

Módulo `solidify`

La solidificación es manejada por el módulo `solidify`. Este módulo no está compilado por defecto debido a su tamaño (~10kB). Debe compilar con la directiva `#define BE_USE_SOLIDIFY_MODULE 1`.

El módulo tiene un solo miembro `dump(x)` que toma un solo argumento (el objeto a solidificar) y envía a `stdout` el código solidificado.

De forma predeterminada, solidify agrega todas las constantes de cadena al grupo global. En su lugar, puede generar cadenas débiles (elegibles para la poda por parte del enlazador) estableciendo el segundo argumento en "verdadero".

Por defecto, `solidify.dump` genera el código solidificado en la salida estándar. Puede especificar un archivo como tercer argumento. El archivo debe estar abierto en modo de escritura y no está cerrado para que pueda concatenar varios objetos.

```
solidify.dump(object:any, [, strings_weak:bool, file_out:file]) -> nil
```

Solidificación de funciones

Puede solidificar una sola función.

Ejemplo:

```
> def f() return "hello" end
> import solidify
> solidify.dump(f)
```

```
/********************************************************************
** Solidified function: f
********************************************************************/
be_local_closure(f,   /* name */
  be_nested_proto(
    0,                          /* nstack */
    0,                          /* argc */
    0,                          /* varg */
    0,                          /* has upvals */
    NULL,                       /* no upvals */
    0,                          /* has sup protos */
    NULL,                       /* no sub protos */
    1,                          /* has constants */
    ( &(const bvalue[ 1]) {     /* constants */
    /* K0   */  be_nested_str(hello),
    }),
    &be_const_str_f,
    &be_const_str_solidified,
    ( &(const binstruction[ 1]) {  /* code */
      0x80060000,  //  0000  RET    1   K0
    })
```

(continues on next page)

(continued from previous page)

```
  )
);
/********************************************************************/
```

Para compilar utilizando cadenas débiles (es decir, cadenas que el enlazador puede eliminar si el objeto no está incluido en el ejecutable de destino), use `solidify.dump(f, true)`:

```
/********************************************************************
** Solidified function: f
********************************************************************/
be_local_closure(f,   /* name */
  be_nested_proto(
    0,                          /* nstack */
    0,                          /* argc */
    0,                          /* varg */
    0,                          /* has upvals */
    NULL,                       /* no upvals */
    0,                          /* has sup protos */
    NULL,                       /* no sub protos */
    1,                          /* has constants */
    ( &(const bvalue[ 1]) {     /* constants */
    /* K0   */  be_nested_str_weak(hello),
    }),
    be_str_weak(f),
    &be_const_str_solidified,
    ( &(const binstruction[ 1]) {  /* code */
      0x80060000,  //  0000  RET    1   K0
    })
  )
);
/********************************************************************/
```

Solidificación de clases

Cuando solidifica una clase, incrusta todos los subelementos. También se agrega un código auxiliar `C` para crear la clase y agregarla al ámbito global.

```
>  class demo
      var i
      static foo = "bar"

      def init()
          self.i = 0
      end

      def say_hello()
          print("Hello Berry!")
      end
  end
> import solidify
> solidify.dump(demo)
```

```
/********************************************************************
** Solidified function: init
********************************************************************/
be_local_closure(demo_init,   /* name */
  be_nested_proto(
    1,                          /* nstack */
    1,                          /* argc */
    2,                          /* varg */
    0,                          /* has upvals */
    NULL,                       /* no upvals */
    0,                          /* has sup protos */
    NULL,                       /* no sub protos */
    1,                          /* has constants */
    ( &(const bvalue[ 2]) {     /* constants */
    /* K0   */  be_nested_str(i),
    /* K1   */  be_const_int(0),
    }),
    &be_const_str_init,
    &be_const_str_solidified,
    ( &(const binstruction[ 2]) {  /* code */
      0x90020101,  //  0000  SETMBR R0  K0  K1
      0x80000000,  //  0001  RET    0
    })
  )
);
/*******************************************************************/

/********************************************************************
** Solidified function: say_hello
********************************************************************/
be_local_closure(demo_say_hello,   /* name */
  be_nested_proto(
    3,                          /* nstack */
    1,                          /* argc */
    2,                          /* varg */
    0,                          /* has upvals */
    NULL,                       /* no upvals */
    0,                          /* has sup protos */
    NULL,                       /* no sub protos */
    1,                          /* has constants */
    ( &(const bvalue[ 1]) {     /* constants */
    /* K0   */  be_nested_str(Hello_X20Berry_X21),
    }),
    &be_const_str_say_hello,
    &be_const_str_solidified,
    ( &(const binstruction[ 4]) {  /* code */
      0x60040001,  //  0000  GETGBL R1  G1
      0x58080000,  //  0001  LDCONST    R2  K0
      0x7C040200,  //  0002  CALL   R1  1
      0x80000000,  //  0003  RET    0
    })
  )
);
```

(continues on next page)

(continued from previous page)

```
/********************************************************************/

/********************************************************************
** Solidified class: demo
********************************************************************/
be_local_class(demo,
    1,
    NULL,
    be_nested_map(4,
    ( (struct bmapnode*) &(const bmapnode[]) {
        { be_const_key(i, -1), be_const_var(0) },
        { be_const_key(say_hello, 2), be_const_closure(demo_say_hello_closure) },
        { be_const_key(init, -1), be_const_closure(demo_init_closure) },
        { be_const_key(foo, 1), be_nested_str(bar) },
    })),
    (bstring*) &be_const_str_demo
);
/********************************************************************/

void be_load_demo_class(bvm *vm) {
    be_pushntvclass(vm, &be_class_demo);
    be_setglobal(vm, "demo");
    be_pop(vm, 1);
}
```

Las subclases también son compatibles.

```
> class demo_sub : demo
      var j

      def init()
          super(self).init()
          self.j = 1
      end
  end
> solidify.dump(demo_sub)
```

```
/********************************************************************
** Solidified function: init
********************************************************************/
be_local_closure(demo_sub_init,   /* name */
  be_nested_proto(
    3,                          /* nstack */
    1,                          /* argc */
    0,                          /* varg */
    0,                          /* has upvals */
    NULL,                       /* no upvals */
    0,                          /* has sup protos */
    NULL,                       /* no sub protos */
    1,                          /* has constants */
    ( &(const bvalue[ 3]) {     /* constants */
    /* K0   */  be_nested_str(init),
```

(continues on next page)

(continued from previous page)

```
    /* K1   */  be_nested_str(j),
    /* K2   */  be_const_int(1),
    }),
    &be_const_str_init,
    &be_const_str_solidified,
    ( &(const binstruction[ 7]) {  /* code */
      0x60040003,  //  0000  GETGBL R1  G3
      0x5C080000,  //  0001  MOVE   R2  R0
      0x7C040200,  //  0002  CALL   R1  1
      0x8C040300,  //  0003  GETMET R1  R1  K0
      0x7C040200,  //  0004  CALL   R1  1
      0x90020302,  //  0005  SETMBR R0  K1  K2
      0x80000000,  //  0006  RET    0
    })
  )
);
/********************************************************************/

/********************************************************************
** Solidified class: demo_sub
********************************************************************/
extern const bclass be_class_demo;
be_local_class(demo_sub,
    1,
    &be_class_demo,
    be_nested_map(2,
    ( (struct bmapnode*) &(const bmapnode[]) {
        { be_const_key(init, -1), be_const_closure(demo_sub_init_closure) },
        { be_const_key(j, 0), be_const_var(0) },
    })),
    be_str_literal("demo_sub")
);
/********************************************************************/

void be_load_demo_sub_class(bvm *vm) {
    be_pushntvclass(vm, &be_class_demo_sub);
    be_setglobal(vm, "demo_sub");
    be_pop(vm, 1);
}
```

Solidificación de módulos

Cuando solidifica un módulo, incrusta todos los subelementos. También funciona con listas o mapas incrustados.

```
> def say_hello() print("Hello Berry!") end
> m = module("demo_module")
> m.i = 0
> m.s = "foo"
> m.f = say_hello
> m.l = [0,1,"a"]
> m.m = {"a":"b", "2":3}
```

(continues on next page)

(continued from previous page)

```
> import solidify
> solidify.dump(m)
```

```
/********************************************************************
** Solidified function: say_hello
********************************************************************/
be_local_closure(demo_module_say_hello,   /* name */
  be_nested_proto(
    2,                          /* nstack */
    0,                          /* argc */
    0,                          /* varg */
    0,                          /* has upvals */
    NULL,                       /* no upvals */
    0,                          /* has sup protos */
    NULL,                       /* no sub protos */
    1,                          /* has constants */
    ( &(const bvalue[ 1]) {     /* constants */
    /* K0   */  be_nested_str(Hello_X20Berry_X21),
    }),
    &be_const_str_say_hello,
    &be_const_str_solidified,
    ( &(const binstruction[ 4]) {  /* code */
      0x60000001,  //  0000  GETGBL R0  G1
      0x58040000,  //  0001  LDCONST    R1  K0
      0x7C000200,  //  0002  CALL   R0  1
      0x80000000,  //  0003  RET    0
    })
  )
);
/*********************************************************************/

/********************************************************************
** Solidified module: demo_module
********************************************************************/
be_local_module(demo_module,
    "demo_module",
    be_nested_map(5,
    ( (struct bmapnode*) &(const bmapnode[]) {
        { be_const_key(l, -1), be_const_simple_instance(be_nested_simple_instance(&be_
↪class_list, {
        be_const_list( *     be_nested_list(3,
    ( (struct bvalue*) &(const bvalue[]) {
        be_const_int(0),
        be_const_int(1),
        be_nested_str(a),
    }))    ) } )) },
        { be_const_key(m, 3), be_const_simple_instance(be_nested_simple_instance(&be_
↪class_map, {
        be_const_map( *     be_nested_map(2,
    ( (struct bmapnode*) &(const bmapnode[]) {
        { be_const_key(a, -1), be_nested_str(b) },
        { be_const_key(2, -1), be_const_int(3) },
```

(continues on next page)

(continued from previous page)

```
    }))    ) } )) },
        { be_const_key(i, 4), be_const_int(0) },
        { be_const_key(f, -1), be_const_closure(demo_module_say_hello_closure) },
        { be_const_key(s, -1), be_nested_str(foo) },
    }))
);
BE_EXPORT_VARIABLE be_define_const_native_module(demo_module);
/********************************************************************/
```

limitaciones de la solidificación

La solidificación funciona para muchos objetos: `clase`, `módulo`, `funciones` y constantes incrustadas u objetos como `int`, `real`, `string`, `list` y `map`.

Limitaciones:

- Los upvals no son compatibles. No puede solidificar un cierre que captura upvals del alcance externo
- La captura de variables globales requiere compilar con la opción `-g` "globales con nombre" (habilitada de forma predeterminada en Tasmota)
- Las constantes de cadena están limitadas a 255 bytes, cadenas largas (más de 255 caracteres no son compatibles, porque nadie nunca los necesitó)
- Los objetos solidificados son de solo lectura, esto tiene algunas consecuencias en las clases. Puede solidificar una clase con sus miembros estáticos cuando se crea, pero no puede solidificar una función que crea una clase derivada de otra clase o con miembros estáticos. La razón principal es que la configuración de la superclase o la asignación de miembros estáticos se implementa mediante el código mutante en la nueva clase, que no puede funcionar en una clase no mutante de solo lectura.
- El código solidificado puede depender del tamaño de "int" y "real" y es posible que no se transfiera a través de MCU con tipos de diferentes tamaños. Es posible que deba volver a solidificar para cada objetivo.

9. FFI

La **Interfaz de Función Externa** (FFI) es una interfaz para la interacción entre diferentes lenguajes. Berry proporciona un conjunto de FFI para realizar la interacción con el lenguaje C, este conjunto de interfaces también es muy fácil de usar en C++. La mayoría de las interfaces FFI son funciones y sus declaraciones se colocan en el archivo *berry.h*. Para reducir la cantidad de RAM utilizada, FFI también proporciona un mecanismo para generar una tabla hash fija durante la compilación de C. Este mecanismo debe utilizar herramientas externas para generar código C.

9.1 Conceptos básicos

La función interactiva más importante en FFI debería ser la función de llamar al código Berry y la función C mutuamente. Para darnos cuenta de cómo dos lenguajes llaman a las funciones del otro, primero debemos entender el mecanismo de paso de parámetros de la función de Berry.

9.1.1 Máquina virtual

A diferencia de los lenguajes compilados, el lenguaje Berry no puede ejecutarse directamente en una máquina física, sino en un entorno de software específico, que es **Máquina virtual** (VM). Similar a una computadora real, el código fuente en forma de texto no se puede ejecutar en una máquina virtual, sino que un compilador debe convertirlo en "código de bytes". La máquina virtual Berry se define como una estructura C `bvm`, el contenido de esta estructura es invisible para FFI. A través de algunas funciones de FFI, podemos crear e inicializar una máquina virtual. Introduciremos el uso de máquinas virtuales a través de un ejemplo sencillo:

```
void berry_test(void)
{
    bvm *vm = be_vm_new(); // Construir una VM
    be_loadstring(vm, "print('Hola Berry')"); // Compilar código de prueba
    be_pcall(vm, 0); // Función de llamada
    be_vm_delete(vm); // Destruir la VM
}
```

Este código da un ejemplo completo del uso de una máquina virtual. Primero, se llama a la función `be_vm_new` para construir una nueva máquina virtual, y luego todas las operaciones se completan en este objeto de máquina virtual. La función `be_vm_new` vinculará automáticamente la biblioteca estándar al crear una máquina virtual. La función de las líneas 4 a 5 es compilar el código fuente de una cadena en una función Berry y luego llamarla. Finalmente, se llama a la función `be_vm_delete` en la línea 6 para destruir la máquina virtual. Al ejecutar esta función obtendrá una línea de salida en la terminal:

```
Hola Berry
```

En todos los escenarios, la construcción de la máquina virtual, la carga de la biblioteca y el proceso de destrucción son los mismos que en las líneas 3, 4 y 6 del ejemplo anterior. Si es necesario, la forma de compilar o cargar el código fuente puede ser diferente. Por ejemplo, para el código fuente en forma de archivo, se puede compilar a través de la función `be_loadfile`. El código fuente se compilará en una función Berry y la función se almacenará en la parte superior de la pila. La función Berry se puede ejecutar llamando a la función FFI `be_pcall` o `be_call`. También puede usar REPL a través de la función `be_repl`. La interfaz del REPL se describirá en los capítulos correspondientes.

9.1.2 Pila virtual

Berry usa una pila virtual y funciones nativas escritas en C para pasar valores. Cada elemento de la pila es un valor Berry. Cuando el código Berry llama a una función nativa, siempre crea una nueva pila y empuja todos los parámetros a la pila. Esta pila virtual también se puede usar en código C para almacenar datos, y el recolector de elementos no utilizados no reclamará el valor almacenado en la pila.

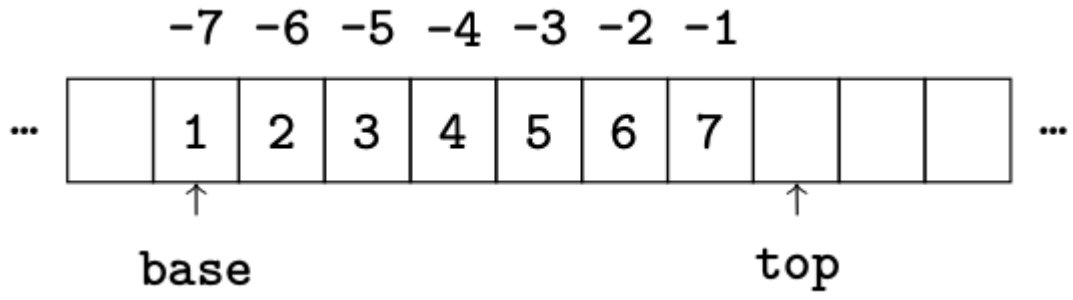

Fig. 2: Virtual_Stack

La pila virtual utilizada por Berry se muestra en la figura anterior.

La pila virtual crece de izquierda a derecha. Cuando el código Berry llama a una función nativa, obtendrá una pila inicial. La posición del primer valor de la pila se llama **base**, y la última posición se llama **superior** (top), en la función nativa solo el valor desde la parte inferior de la pila hasta la posición anterior a la parte superior de la pila puede ser accesible. La posición de la parte inferior de la pila es fija, mientras que la posición de la parte superior de la pila se puede mover y la parte superior de la pila siempre está vacía. El motivo de esta propiedad es que después de insertar el nuevo valor en la pila virtual, en la posición original de la parte superior de la pila se escribirá el nuevo valor, el puntero superior de la pila avanzará a la siguiente posición; por el contrario, si se extrae el valor en la parte superior de la pila virtual, el puntero superior de la pila disminuirá de 1. En este momento, aunque la posición del puntero superior de la pila es objetivamente un valor, este no es válido y se puede borrar en cualquier momento, por lo que la posición del puntero en la parte superior de la pila aún está vacía. Cuando la pila virtual está vacía, el puntero inferior `base` es igual al puntero superior `top`. La pila virtual no sigue estrictamente las reglas de funcionamiento de la pila: además de empujar y sacar, también se puede acceder a la pila virtual por índice, e incluso insertar o eliminar valores en cualquier posición. Hay dos formas de indexar elementos en la pila: una se basa en la parte inferior de la pila **Índice absoluto**, el valor del índice absoluto es un número entero positivo a partir de 1; el otro se basa en la parte superior de la pila **Índice relativo**, el valor del índice relativo es un número entero negativo a partir de 1. Tome la Figura anterior como ejemplo, el valor de índice 1, 2...8 es un índice absoluto, y el índice absoluto de un elemento es la distancia desde el elemento hasta el final de la pila. El valor de índice 1, 2... 8 es un índice relativo, y el valor de índice relativo de un elemento es el número negativo de la distancia desde el elemento hasta la parte superior de la pila. Si un valor de índice *index* es válido, entonces el elemento al que se refiere debe estar entre la parte inferior de la pila y la parte superior de la pila, lo que significa se cumple que la expresión:

```
1  abs(*index*)  *top*  *base* + 1.
```

Por conveniencia, estipulamos que el puntero inferior de la pila 'base' se usa como referencia, y su índice absoluto 1, y el valor anterior de 'base' no se considera (por lo general, 'base' no es la posición inferior de todo el pila). Por ejemplo, cuando regresa una función nativa, la ubicación donde se almacena el valor de retorno está justo antes de `base`, y la función nativa no suele acceder a estas ubicaciones.

Operar con pila virtual

Índice y tamaño de pila

Como se mencionó anteriormente, se pueden usar dos métodos de indexación para acceder a la pila virtual y el valor del índice debe ser válido. Al mismo tiempo, en muchos casos también es necesario introducir nuevos valores en la pila. En este caso, el programador debe asegurarse de que la pila no se desborde. Por defecto, Berry garantiza el espacio `BE_STACK_FREE_MIN` para que lo usen las funciones nativas. Este valor se puede modificar en el archivo *berry.h*. Su valor por defecto suele ser 10, que debería ser suficiente en la mayoría de los casos. Si realmente necesita expandir la pila, puede llamar a la función FFI `be_stack_require`. El prototipo de esta función es:

```
void be_stack_require(bvm *vm, int count);
```

El parámetro `count` es la cantidad de espacio necesario. Cuando el espacio restante en la pila virtual sea insuficiente, la capacidad de la pila se expandirá; de lo contrario, esta función no hará nada.

Advertencia: si se produce un desbordamiento de la pila, o si se utiliza un índice no válido para acceder a la pila, el programa fallará. Puede activar el interruptor de depuración `BE_DEBUG` (sección [sección::BE_DEBUG]), que activará la función de aserción, y puede obtener información de depuración en tiempo de ejecución para detectar errores como desbordamiento de pila o índice no válido.

Obtener valor de la pila

Hay un conjunto de funciones en FFI para obtener valores de la pila virtual. Estas funciones generalmente convierten los valores en la pila en valores simples compatibles con el lenguaje C y luego regresan. Los siguientes son FFI de uso común para obtener valores de la pila:

```
bint be_toint(bvm *vm, int index);
breal be_toreal(bvm *vm, int index);
int be_tobool(bvm *vm, int index);
const char* be_tostring(bvm *vm, int index);
void* be_tocomptr(bvm *vm, int index);
```

La forma de parámetro de estas funciones es la misma, pero el valor de retorno es diferente. Las primeras cuatro funciones son fáciles de entender. Al igual que sus nombres, la función de `be_toint` es convertir los valores en la pila virtual a valores enteros de C (`bint` suele ser un alias de tipo `int`) y devolverlos. La función de la última función `be_tocomptr` es sacar un valor de puntero de tipo general de la pila virtual. El significado específico de este puntero se explica por el propio programa C.

Estas funciones utilizan la misma forma de interpretar los parámetros: el parámetro `vm` es el puntero de la instancia de la máquina virtual; `index` es el índice del elemento a recuperar, que puede ser un índice relativo o un índice absoluto. No puede usar FFI para eliminar los tipos de datos complejos de Berry de la pila virtual, por lo que no puede eliminar un tipo de "map" o un tipo de "class" de la pila. Uno de los beneficios de este diseño es que no es necesario considerar la recolección de elementos no utilizados en las funciones nativas.

Función nativa

Una **Función nativa** está implementada por lenguaje C y puede ser llamada por código Berry. La función nativa puede ser una función ordinaria. En este caso, llamar a la función nativa no generará ningún espacio asignado dinámicamente, al igual que una llamada de función C normal. Las funciones nativas también pueden ser cierres, y se debe asignar espacio para variables libres al crear cierres nativos. En circunstancias normales, las funciones nativas simples son suficientes para satisfacer las necesidades. Ahorran más recursos que los cierres nativos y son más fáciles de usar.

Definir una función nativa

La función nativa en sí es una función C, pero todas tienen una forma específica. La definición de la función nativa es:

```
int a_native_function(bvm *vm)
{
    // hacer algo ...
}
```

La función nativa debe ser una función C cuyo parámetro sea un puntero a `bvm` y el valor de retorno sea `int`. Las funciones de Berry deben devolver un valor y las funciones nativas no son una excepción. A diferencia del valor de retorno del lenguaje C, el valor de retorno de la función nativa no es el valor transportado por la instrucción C `return`. Puede usar estos FFI para devolver el valor de la función nativa, y también hacen que la función C devuelva:

```
be_return(bvm *vm);
be_return_nil(bvm *vm);
```

Estos FFI son en realidad dos macros, y no es necesario usar la instrucción C `return` al usarlos. `be_return` pondrá la parte superior de la pila virtual

Usar una función nativa

Después de definir la función nativa, debe agregarse al intérprete de alguna manera antes de que pueda llamarse en código Berry. Una de las formas más sencillas es agregar funciones nativas a la tabla de objetos integrada de Berry. El proceso de configuración de objetos nativos como objetos incorporados de Berry se denomina **registración**. El FFI de la función nativa registrada es:

```
void be_regfunc(bvm *vm, const char *nombre, bntvfunc f);
```

vm es la instancia actual de la máquina virtual, nombre es el nombre de la función nativa y f es el puntero de la función nativa. El comportamiento específico de esta función está relacionado con el valor de la macro BE_USE_PRECOMPILED_OBJECT (aunque el FFI todavía está disponible cuando se utiliza la técnica de construcción en tiempo de compilación, no puede registrar dinámicamente las variables integradas. En este caso, consulte el método de registro de los objetos integrados. 1.3). La definición del tipo de función nativa bntvfunc es:

```
typedef int (*bntvfunc)(bvm*);
```

De hecho, el tipo bntvfunc es el tipo de puntero de función con el parámetro bvm y el tipo de valor devuelto int. La función be_regfunc debe llamarse antes de analizar el código fuente de Berry.

También puede insertar la función nativa en la pila virtual y luego usar una función FFI be_call para llamarla. Un uso más común es usar el objeto de función nativa en la pila virtual como valor de retorno.

Ejemplo completo

Finalizamos esta sección con un ejemplo sencillo. Aquí, vamos a implementar una función add que suma dos números y devuelve el resultado del cálculo. Primero, definimos una función nativa para implementar esta función:

```
static int l_add(bvm *vm)
{
    int top = be_top(vm); // Obtener el número de argumentos
    /* Verificar el número y tipo de argumentos */
    if (top == 2 && be_isnumber(vm, 1) && be_isnumber(vm, 2)) {
        breal x = be_toreal(vm, 1); // Obtener el primer argumento
        breal y = be_toreal(vm, 2); // Obtener el segundo argumento
        be_pushreal(vm, x + y); // Empuje el resultado a la pila
        be_return(vm); // Devuelve el valor en la parte superior de la pila
    }
    be_return_nil(vm); // Devuelve nil cuando algo sale mal
}
```

Por lo general, las funciones nativas no necesitan usarse fuera del archivo C, por lo que generalmente se declaran como tipos estáticos. Utilice la función be_top para obtener el índice absoluto de la parte superior de la pila virtual (valor top), que es la capacidad de la pila. Podemos llamar a be_top antes de que la función nativa realice la operación de pila virtual, en este momento la capacidad de la pila virtual es igual a la cantidad de parámetros reales. Para la función add, necesitamos dos parámetros para participar en la operación, así que verifica si el número de parámetros es 2 en la cuarta línea (top == 2). Y necesitamos verificar si los dos parámetros son de tipo numérico, por lo que debemos llamar a la función be_isnumber para verificar. Si todo es correcto, los parámetros se sacarán de la pila virtual, luego el resultado del cálculo se colocará en la pila y finalmente se devolverá usando be_return. Si la verificación del parámetro falla, se llamará a be_return_nil para devolver el valor de nil.

A continuación, registre esta función nativa en la tabla de objetos integrada. Para simplificar, lo registramos después de cargar la biblioteca:

```
bvm *vm = be_vm_new(); // Construir una VM
be_regfunc(vm, "myadd", l_add); // Registrar la función nativa "myadd"
```

La segunda línea es donde se registra la función nativa y la llamamos `myadd`. En este punto, la definición y el registro de la función nativa están completos. Como verificación, puede compilar el intérprete, luego ingresar el REPL y ejecutar algunas pruebas. Debería obtener resultados como este:

```
> myadd
<function: 0x562a210f0f90>
> myadd(1.0, 2.5)
3.5
> myadd(2.5, 2)
4.5
> myadd(1, 2)
3
```

Tipos y Funciones

Tipos

Esta sección presentará algunos tipos que deben usarse en FFI y son generalmente utilizados por funciones FFI. Generalmente, los tipos y declaraciones en FFI se pueden encontrar en el archivo *berry.h*. A menos que se especifique lo contrario en esta sección, la definición o declaración se proporciona en *berry.h* de forma predeterminada.

El tipo `bvm` se utiliza para almacenar la información de estado de la máquina virtual Berry. Los detalles de este tipo no son visibles para los programas externos. Por lo tanto, esta definición solo se puede encontrar en el archivo *berry.h*:

```
typedef struct bvm bvm;
```

La mayoría de las funciones de FFI usan el tipo `bvm` como primer parámetro, porque todas operan en la máquina virtual internamente. Ocultar la implementación interna de `bvm` ayuda a reducir el acoplamiento entre el estándar FFI y la VM. Fuera del intérprete, normalmente solo se utilizan punteros `bvm`. Para crear un nuevo objeto `bvm`, use la función `be_vm_new` y destruya el objeto `bvm` usando la función `be_vm_delete`.

La definición del tipo de función nativa es:

```
typedef int (*bntvfunc)(bvm*);
```

Este tipo es un puntero de función nativo y algunas FFI que registran o agregan funciones nativas a la máquina virtual usan parámetros de este tipo. Las variables o parámetros de este tipo deben inicializarse con un nombre de función cuyo parámetro sea del tipo `bvm` y cuyo valor de retorno sea del tipo `int`.

Este tipo se usa cuando se registran funciones nativas en lotes o se construyen clases nativas. Se define como:

```
typedef struct {
    const char *nombre; // El nombre de la función u objeto
    bntvfunc funcion; // El puntero de función
} bnfuncinfo;
```

El miembro `nombre` de `bnfuncinfo` representa el nombre de una función u objeto, y el miembro `funcion` es un puntero de función nativo.

Este tipo es un tipo entero integrado de Berry. Se define en el documento *berry.h*. Por defecto, `bint` se implementa usando el tipo `long long`, y la implementación de `bint` se puede modificar cambiando el archivo de configuración.

Este es el tipo de número real incorporado de Berry, que en realidad es el tipo de punto flotante en lenguaje C. `breal` se define como:

```
#if BE_SINGLE_FLOAT != 0
    typedef float breal;
#else
    typedef double breal;
#endif
```

Puede usar la macro `BE_SINGLE_FLOAT` para controlar la implementación específica de `breal`: cuando el valor de `BE_SINGLE_FLOAT` es `0`, se usará la implementación de tipo `doble breal`, de lo contrario, la implementación de tipo `float` se utilizará para `breal`.

[sección::código de error]

Este tipo de enumeración se utiliza en algunos valores de retorno de FFI. La definición de este tipo es:

```
enum berrorcode {
    BE_OK = 0,
    BE_IO_ERROR,
    BE_SYNTAX_ERROR,
    BE_EXEC_ERROR,
    BE_MALLOC_FAIL,
    BE_EXIT
};
```

El significado de estos valores de enumeración son:

- `BE_OK`: No hay ningún error, la función se ejecuta con éxito.
- `BE_IO_ERROR`: Ocurrió un error de lectura de archivo cuando el intérprete estaba leyendo el archivo fuente. El error generalmente es causado por el ausencia del expediente.
- `BE_SYNTAX_ERROR`: Ocurrió un error de sintaxis cuando el intérprete estaba compilando el código fuente. Después de que ocurre este error, el intérprete no generará bytecode, por lo que no puede continuar ejecutándose el código de bytes.
- `BE_EXEC_ERROR`: Error de tiempo de ejecución. Cuando se produce este error, la ejecución de El código Berry se detiene y el entorno se restaura al máximo punto de recuperación reciente.
- `BE_MALLOC_FAIL`: Falló la asignación de memoria. Este error es causado por espacio de pila insuficiente.
- `BE_EXIT`: Indica que el programa sale y el valor no es un error. Ejecutar la función `exit` de Berry hace que el intérprete devuelva este valor.

Cabe señalar que cuando se produce un error `BE_MALLOC_FAIL`, ya no se puede realizar la asignación de memoria dinámica, lo que significa que ya no se pueden asignar objetos de cadena, por lo que la función que devuelve este error generalmente no brinda información más detallada sobre el error.

Funciones y Macros

Esta función se utiliza para crear una nueva instancia de máquina virtual. El prototipo de función es:

```
bvm* be_vm_new(void);
```

El valor de retorno de la función es un puntero a la instancia de la máquina virtual. `be_vm_new` es la primera función llamada cuando se crea el intérprete de Berry. Esta función hará mucho trabajo: solicitar memoria para la máquina virtual, inicializar el estado y los atributos de la máquina virtual, inicializar el GC (recolector de basura), la biblioteca estándar se carga en la instancia de la máquina virtual, etc.

La función `be_vm_delete` se usa para destruir una instancia de máquina virtual. El prototipo de la función es:

```
void be_vm_delete(bvm *vm);
```

El parámetro `vm` es el puntero del objeto de la máquina virtual que se va a destruir. La destrucción de la máquina virtual liberará todos los objetos de la máquina virtual, incluidos los valores de la pila y los objetos administrados por el GC. El puntero de la máquina virtual después de la destrucción será un valor no válido y ya no se podrá hacer referencia a él.

Esta función se utiliza para cargar un fragmento de código fuente del búfer y compilarlo en un código de bytes. El prototipo de la función es:

```
int be_loadbuffer(bvm *vm, const char *name, const char *buffer, size_t length);
```

El parámetro `vm` es el puntero de la máquina virtual. `name` es una cadena, que generalmente se usa para marcar la fuente del código fuente. Por ejemplo, la entrada del código fuente del dispositivo de entrada estándar puede pasar la cadena `"stdin"` a este parámetro, y la entrada del código fuente del archivo puede ser el nombre del archivo y se pasa a este parámetro. El parámetro `buffer` es el búfer para almacenar el código fuente. La organización de este búfer es muy similar a la cadena de C. Es una secuencia continua de caracteres, pero el búfer al que apunta `buffer` no requiere caracteres `'\0'` como terminador. El parámetro `longitud` indica la longitud del búfer. Esta longitud se refiere al número de bytes de texto de código fuente en el búfer.

Para dar un ejemplo simple, si queremos usar la función `be_loadbuffer` para compilar una cadena, el uso general es:

```
const char *str = "print('Hola Berry')";
be_loadbuffer(vm, "cadena", str, strlen(str));
```

Aquí usamos la cadena `"cadena"` para representar el código fuente, también puede modificarla a cualquier valor. Tenga en cuenta que la función `strlen` de la función de biblioteca estándar de C se usa aquí para obtener la longitud del búfer de cadena (en realidad, el número de bytes en la cadena).

Si la compilación es exitosa, `be_loadbuffer` compilará el código fuente en una función Berry y lo colocará en la parte superior de la pila virtual. Si la compilación encuentra un error, `be_loadbuffer` devolverá un valor de error de tipo `berrorcode` [ver Sección código de error] y, si es posible, almacenará la cadena de mensaje de error específica en la parte superior de la pila virtual.

`be_loadstring` es una macro definida como:

```
#define be_loadstring(vm, str) be_loadbuffer((vm), "string", (str), strlen(str))
```

Esta macro es solo un contenedor simple para la función `be_loadbuffer`. El parámetro `vm` es un puntero a la instancia de la máquina virtual, y el parámetro `str` es un puntero a la cadena de código fuente. Es muy conveniente usar `be_loadstring` para compilar cadenas, por ejemplo:

```
be_loadstring(vm, "print('Hola Berry')");
```

Esta forma de escribir es más concisa que usar `be_loadbuffer`, pero debe asegurarse de que la cadena termine con un carácter `'\0'`.

Esta función se utiliza para compilar un archivo de código fuente. El prototipo de función es:

```
int be_loadfile(bvm *vm, const char *nombre);
```

La funcionalidad de esta función es similar a la función `be_loadbuffer`, excepto que la función se compilará leyendo el archivo de código fuente. El parámetro `vm` es el puntero de la instancia de la máquina virtual y el parámetro `nombre` es el nombre del archivo de origen. Esta función llamará a la interfaz de archivo y, de forma predeterminada, utilizará funciones como `fopen` en la biblioteca estándar de C para manipular archivos.

Si usa la interfaz de archivo de la biblioteca estándar de C, puede usar nombres de archivo de ruta relativa o ruta absoluta. Si el archivo no existe, `be_loadfile` devolverá un error `BE_IO_ERROR` (Ve Seccion de código de error) y colocará el mensaje de error en la parte superior de la pila. Otros mensajes de error son los mismos que los de la función `be_loadbuffer`. Se recomienda usar la función `be_loadfile` para compilar el archivo fuente, en lugar de leer todos los archivos fuente en un búfer, y luego llamar a la función `be_loadbuffer` para compilar el código fuente. El primero leerá el archivo fuente en segmentos y solo creará un pequeño búfer de lectura en la memoria, ahorrando así más memoria.

La función `be_top` devuelve el valor de índice absoluto del elemento superior en la pila virtual. Este valor es también el número de elementos en la pila virtual (la capacidad de la pila virtual). Llame a esta función antes de agregar o quitar elementos en la pila virtual para obtener la cantidad de parámetros de la función nativa. El prototipo de esta función es:

```
int be_top(bvm *vm);
```

Esta función se suele utilizar para obtener el número de parámetros de una función nativa. Cuando se usa para este propósito, se recomienda llamar a `be_top` en la parte superior del cuerpo de la función nativa. P.ej:

```
static int native_function_example(bvm *vm)
{
    int argc = be_top(vm); // Obtener el número de argumentos
    // ...
}
```

La función `be_typename` convierte el tipo del objeto Berry en una cadena y lo devuelve. Por ejemplo, devuelve `"int"` para un objeto entero y `"function"` para un objeto función. El prototipo de esta función es:

```
const char* be_typename(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, e `index` es el índice del objeto a operar. La función `type` en la biblioteca estándar de Berry se implementa llamando a `be_typename`. Consulte la sección `baselib_type` para conocer la cadena de retorno correspondiente al tipo de parámetro.

La función `be_classname` se utiliza para obtener el nombre de clase de un objeto o clase. El prototipo de función es:

```
const char* be_classname(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, e `index` es el índice del objeto a operar. Si el valor en `index` es una instancia, la función `be_classname` devolverá la cadena del nombre de la clase a la que pertenece la instancia, y si el valor en `index` es una clase, devolverá directamente la cadena del nombre de la clase. En otros casos `be_classname` devolverá `NULL`.

La función `be_strlen` devuelve la longitud de la cadena Berry especificada. El prototipo de función es:

```
int be_strlen(bvm *vm, int index);
```

El parámetro vm es el puntero de la instancia de la máquina virtual, e index es el índice del objeto a operar. Esta función devuelve el número de bytes en la cadena en index (los caracteres '\0' al final de la cadena Berry no se cuentan). Si el valor de la posición index no es una cadena, la función be_strlen devolverá 0.

Aunque la cadena Berry es compatible con el formato de cadena C, no se recomienda utilizar la función strlen de la biblioteca estándar de C para medir la longitud de la cadena Berry. Para cadenas Berry, be_strlen es más rápido que strlen y tiene mejor compatibilidad.

La función be_strconcat se utiliza para empalmar dos cadenas Berry. El prototipo de función es:

```
void be_strconcat(bvm *vm, int index);
```

El parámetro vm es el puntero de la instancia de la máquina virtual. Esta función concatenará la cadena en la posición del parámetro de index con la cadena en la posición superior de la pila, y luego colocará la cadena resultante en la posición indexada por index.

La función be_pop extrae el valor en la parte superior de la pila. El prototipo de función es:

```
void be_pop(bvm *vm, int n);
```

El parámetro vm es el puntero de la instancia de la máquina virtual, y el parámetro n es la cantidad de valores que se extraerán de la pila. Tenga en cuenta que el valor de n no puede exceder la capacidad de la pila.

La función be_remove elimina un valor de la pila. Esta función eliminará un valor de la pila.

```
void be_remove(bvm *vm, int index);
```

El parámetro vm es el puntero de la instancia de la máquina virtual y el parámetro index es el índice del objeto que se eliminará. Después de que el valor en index se mueva, los siguientes valores se completarán y la capacidad de la pila se reducirá en uno. El valor de index no puede exceder la capacidad de la pila.

La función be_absindex devuelve el valor de índice absoluto de un valor de índice dado, y su prototipo de función es:

```
int be_absindex(bvm *vm, int index);
```

El parámetro vm es el puntero de la instancia de la máquina virtual y el parámetro index es el valor del índice de entrada. Si index es positivo, el valor de retorno de be_absindex es el valor de index. Si index es negativo, el valor de retorno de be_absindex es el valor de índice absoluto correspondiente a index. Cuando index es un valor negativo (índice relativo), su posición de índice no puede ser inferior a la parte inferior de la pila.

La función be_newlist crea un nuevo valor de list, y su prototipo de función es:

```
void be_newlist(bvm *vm);
```

El parámetro vm es el puntero de la instancia de la máquina virtual. Después de llamar con éxito a esta función, el nuevo valor de list se colocará en la parte superior de la pila. El valor list es una representación interna de una lista, que no debe confundirse con una instancia de la clase list.

La función be_newmap crea un nuevo valor map, y su prototipo de función es:

```
void be_newmap(bvm *vm);
```

El parámetro vm es el puntero de la instancia de la máquina virtual. Después de llamar con éxito a esta función, el nuevo valor del map se colocará en la parte superior de la pila. El valor map es una representación interna de una lista, que no debe confundirse con una instancia de la clase map.

La función be_getglobal empuja la variable global con el nombre especificado a la pila. Su prototipo de función es:

```
void be_getglobal(bvm *vm, const char *name);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual y el parámetro `name` es el nombre de la variable global. Después de llamar a esta función, la variable global llamada `name` se colocará en la parte superior de la pila virtual.

La función `be_setmember` se utiliza para establecer el valor de la variable miembro de la clase de objeto de instancia. El prototipo de función es:

```
void be_setmember(bvm *vm, int index, const char *k);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, el parámetro `index` es el índice del objeto de la instancia y el parámetro `k` es el nombre del miembro. Esta función copiará el valor en la parte superior de la pila al miembro `k` de la instancia de posición de índice. Tenga en cuenta que el elemento superior de la pila no aparecerá automáticamente.

La función `be_getmember` se utiliza para obtener el valor de la variable miembro de la clase de objeto de instancia. El prototipo de función es:

```
void be_getmember(bvm *vm, int index, const char *k);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, el parámetro `index` es el índice del objeto de la instancia y el parámetro `k` es el nombre del miembro. Esta función coloca el valor del miembro de la instancia de posición de índice `k` en la parte superior de la pila virtual.

La función `be_getindex` se utiliza para obtener el valor de `list` o `map`. El prototipo de función es:

```
void be_getindex(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, y el parámetro `index` es el índice del objeto a operar. Esta función se usa para obtener un elemento del contenedor `map` o `list` (valores internos, no instancias de las clases `map` o `list`), y el índice del elemento se almacena en la parte superior de la pila (el índice relativo es -1). Después de llamar a esta función, el valor obtenido del contenedor se colocará en la parte superior de la pila. Si no hay ningún subíndice señalado por el contenedor, el valor `nil` se colocará en la parte superior de la pila. Por ejemplo, si el elemento con el índice 1 en la pila virtual es una `list` y queremos extraer el elemento con el índice 0, entonces podemos usar el siguiente código:

```
be_pushint(vm, 0); //  Inserte el valor de índice 0 en la pila virtual
be_getindex(vm, 1); //  Obtener un elemento del contenedor de lista
```

Primero colocamos el valor entero `0` en la pila, y este valor se usará como índice para obtener el elemento del contenedor `list`. La segunda línea de código implementa para obtener elementos del contenedor `list`. El valor de índice del contenedor `list` en el ejemplo es 1 en la pila virtual. El elemento recuperado se almacena en la parte superior de la pila y podemos usar el índice relativo -1 para acceder a él.

La función `be_setindex` se utiliza para establecer un valor en `list` o `map`. El prototipo de función es:

```
void be_setindex(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, y el parámetro `index` es el subíndice del objeto a operar. Esta función se utiliza para escribir un elemento del contenedor `map` o `list`. El índice del valor que se va a escribir en la pila virtual es -1, y el índice del subíndice de la posición de escritura en la pila virtual es -2. Si el elemento con el subíndice especificado no existe en el contenedor, la operación de escritura fallará.

Suponiendo que la posición con el índice 1 en la pila virtual tiene un valor de `map`, y tiene un elemento con un subíndice de `"prueba"`, un ejemplo de configuración del elemento en el subíndice de `"prueba"` a `100` es:

```
be_pushstring(vm, "prueba"); // Empuja el índice "índice"
be_pushint(vm, 100);         // Empuja el valor 100
be_setindex(vm, 1);          // Establece el par clave-valor a map["prueba"] = 100
```

Primero debemos empujar el subíndice y el valor que se escribirá en la pila en orden. Para `map`, es un par clave-valor. En el ejemplo, las dos primeras líneas de código completan estas tareas. La tercera línea llama a la función `be_setindex` para escribir el valor en el objeto `map`.

La función `be_getupval` se utiliza para leer un valor ascendente del cierre nativo. El prototipo de función es:

```
void be_getupval(bvm *vm, int index, int pos);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual; `index` es el valor de índice de cierre nativo del valor ascendente que se va a leer; `pos` es la posición del upvalue en la tabla upvalue de cierre nativa (la numeración comienza desde 0). El valor leído se colocará en la parte superior de la pila virtual.

La función `be_setupval` se utiliza para establecer un valor superior del cierre nativo. El prototipo de función es:

```
void be_setupval(bvm *vm, int index, int pos);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual; `index` es el valor del índice de cierre nativo que se escribirá en upvalue; `pos` es la posición del upvalue en la tabla upvalue de cierre nativa (la numeración comienza desde 0). Esta función obtiene un valor de la parte superior de la pila virtual y lo escribe en el valor superior de destino. Una vez completada la operación, el valor superior de la pila no se extraerá de la pila.

La función `be_getsuper` se utiliza para obtener el objeto principal de la clase base o la instancia de la clase. El prototipo de función es:

```
void be_getsuper(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual; `index` es la clase u objeto a operar. Si el valor en `index` es una clase con una clase base, la función colocará su clase base en la parte superior de la pila; si el valor en `index` es un objeto con un objeto padre, la función tomará su padre. El objeto se coloca en la parte superior de la pila; de lo contrario, se coloca un valor de `nil` en la parte superior de la pila.

La función `be_data_size` se utiliza para obtener el número de elementos contenidos en el contenedor. El prototipo de función es:

```
int be_data_size(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual; `index` es el índice del objeto contenedor que se va a operar. Si el valor en `index` es un valor Map o List, la función devuelve el número de elementos contenidos en el contenedor; de lo contrario, devuelve `-1`.

La función `be_data_push` se usa para agregar un nuevo elemento al final del contenedor. El prototipo de función es:

```
void be_data_push(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual; `index` es el índice del objeto contenedor que se va a operar. El objeto en `index` debe ser un valor de Lista. Esta función obtiene un valor de la parte superior de la pila y lo agrega al final del contenedor. Una vez completada la operación, el valor en la parte superior de la pila no se extraerá de la pila.

La función `be_data_insert` se utiliza para insertar un par de elementos en el contenedor. El prototipo de función es:

```
void be_data_insert(bvm *vm, int index);
```

El parámetro vm es el puntero de la instancia de la máquina virtual; index es el índice del objeto contenedor que se va a operar. El objeto en index debe ser un valor de lista o un valor de mapa. El elemento insertado forma un par de pares clave-valor. El valor se almacena en la parte superior de la pila y la clave se almacena en el índice anterior en la parte superior de la pila. Cabe señalar que la clave insertada en el contenedor Mapa no puede ser un valor "nil" y la clave insertada en el contenedor Lista debe ser un valor entero. Si la operación es exitosa, la función devolverá bture, de lo contrario devolverá bfalse.

La función be_data_remove se utiliza para eliminar un elemento del contenedor. El prototipo de función es:

```
void be_data_remove(bvm *vm, int index);
```

El parámetro vm es el puntero de la instancia de la máquina virtual; index es el índice del objeto contenedor que se va a operar. El objeto en index debe ser un valor de lista o un valor de mapa. Para el contenedor de mapas, la llave para eliminar el elemento se almacena en la parte superior de la pila virtual (debe empujarse antes de llamar a la función); para el contenedor de lista, el índice del elemento que se va a eliminar se almacena en la parte superior de la pila virtual (debe estar antes de la llamada a la función). Si la operación es exitosa, la función devolverá btrue, de lo contrario devolverá bfalse.

La función be_data_resize se utiliza para restablecer la capacidad del contenedor. El prototipo de función es:

```
void be_data_resize(bvm *vm, int index);
```

El parámetro vm es el puntero de la instancia de la máquina virtual; index es el índice del objeto contenedor que se va a operar. Esta función solo está disponible para contenedores de lista y la nueva capacidad se almacena en la parte superior de la pila virtual (debe ser un número entero).

La función be_iter_next se utiliza para obtener el siguiente elemento del iterador. El prototipo de función es:

```
int be_iter_next(bvm *vm, int index);
```

El parámetro vm es el puntero de la instancia de la máquina virtual; index es el índice del iterador a operar. El objeto iterador puede ser un iterador de un contenedor List o un contenedor Map. Para el iterador List, esta función empuja el valor del resultado de la iteración a la parte superior de la pila, mientras que para el iterador Map, empuja el valor clave a la posición anterior y la parte superior de la pila, respectivamente. Llamar a esta función actualizará el iterador. Si la función devuelve 0, la llamada falla, devuelve 1 para indicar que el iterador actual es un iterador de lista y devuelve 2 para indicar que el iterador actual es un iterador de mapa.

La función map_hasnext se usa para probar si hay otro elemento en el iterador. El prototipo de función es:

```
int map_hasnext(bvm *vm, int index)
```

El parámetro vm es el puntero de la instancia de la máquina virtual; index es el índice del iterador a operar. El objeto iterador puede ser un iterador de un contenedor List o un contenedor Map. Si hay más elementos iterables en el iterador, devuelve 1, de lo contrario, devuelve 0.

La función be_refcontains se usa para probar si hay una referencia al objeto especificado en la pila de referencia. Debe usarse junto con be_refpush y be_refpop. Esta API puede evitar la recursividad al atravesar objetos que tienen sus propias referencias. El prototipo de función es:

```
int be_refcontains(bvm *vm, int index);
```

El parámetro vm es el puntero de la instancia de la máquina virtual; index es el índice del objeto a operar. Esta función se utiliza para el valor de un tipo de instancia. Si hay una referencia al objeto en la pila de referencia, devuelve 1, de lo contrario, devuelve 0.

La función be_refpush inserta la referencia del objeto especificado en la pila de referencia. El prototipo de función es:

```
int be_refpush(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual; `index` es el índice del objeto a operar. Esta función se utiliza para el valor de un tipo de instancia.

La función `be_refpop` extrae el objeto en la parte superior de la pila de referencia. Esta función eliminará un valor de la pila.

```
int be_refpop(bvm *vm);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual. Esta función se usa en pares con `be_refpush`. El siguiente es el uso de la API de la pila de referencia para evitar el problema del recorrido recursivo infinito cuando se hace referencia al objeto mismo:

```
int list_traversal(bvm *vm)
{
    // ...
    if (be_refcontains(vm, 1)) {
        be_return(vm);
    }
    be_refpush(vm, 1);
    // Atravesando el contenedor, puede llamar a list_traversal recursivamente.
    be_refpop(vm);
    be_return(vm);
}
```

Este es un proceso transversal simplificado del contenedor List. Para obtener el código completo, consulte el código fuente de la función `m_tostring` en *be_listlib.c*. Asumimos que el índice del objeto List es 1. Primero, verificamos si la Lista ya existe en la pila de referencia (línea 4), y si la referencia ya existe, regresa directamente, de lo contrario, continúa con el procesamiento posterior. Para hacer utilizable `be_refcontains`, necesitamos usar `be_refpush` y `be_refpop` para procesar la pila de referencia antes y después de la operación transversal real (líneas 7 y 9).

La función `be_stack_require` prueba la cantidad de espacio libre en la pila y expande el espacio de la pila si es insuficiente. El prototipo de función es:

```
void be_stack_require(bvm *vm, int count);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual; `count` es la capacidad de pila libre requerida. Si la capacidad libre de la pila virtual asignada por la VM a la función nativa es inferior a este valor, se realizará una operación de expansión.

La función `be_isnil` devuelve si el valor indexado por el parámetro `index` en la pila virtual es `nil`, si lo es, devuelve `1`, de lo contrario, devuelve `0`. El prototipo de esta función es:

```
int be_isnil(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, e `index` es el índice del valor a medir.

La función `be_isbool` devuelve si el valor indexado por el parámetro `index` en la pila virtual es de tipo `bool`, si lo es, la función devuelve `1`, de lo contrario devuelve `0`. El prototipo de esta función es:

```
int be_isbool(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, e `index` es el índice del valor a medir.

La función `be_isint` devuelve si el valor indexado por el parámetro `index` en la pila virtual es de tipo entero, si lo es, devuelve `1`, de lo contrario, devuelve `0`. El prototipo de esta función es:

```
int be_isint(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, e `index` es el índice del valor a medir.

La función `be_isreal` devuelve si el valor indexado por el parámetro `index` en la pila virtual es un tipo de número real, si lo es, devuelve `1`, de lo contrario, devuelve `0`. El prototipo de esta función es:

```
int be_isreal(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, e `index` es el índice del valor a medir.

La función `be_isnumber` devuelve si el valor indexado por el parámetro `index` en la pila virtual es un número entero o un tipo de número real, si lo es, devuelve `1`, de lo contrario, devuelve `0`. El prototipo de esta función es:

```
int be_isnumber(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, e `index` es el índice del valor a medir.

La función `be_isstring` devuelve si el valor indexado por el parámetro `index` en la pila virtual es un tipo de cadena, si lo es, devuelve `1`, de lo contrario, devuelve `0`. El prototipo de esta función es:

```
int be_isstring(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, e `index` es el índice del valor a medir.

La función `be_isclosure` devuelve si el valor indexado por el parámetro `index` en la pila virtual es un tipo de cierre, si lo es, devuelve `1`, de lo contrario, devuelve `0`. El prototipo de esta función es:

```
int be_isclosure(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, e `index` es el índice del valor a medir.

La función `be_isntvclos` devuelve si el valor indexado por el parámetro `index` en la pila virtual es un tipo de cierre primitivo, si lo es, devuelve `1`, de lo contrario, devuelve `0`. El prototipo de esta función es:

```
int be_isntvclos(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, e `index` es el índice del valor a medir.

La función `be_isfunction` devuelve si el valor indexado por el parámetro `index` en la pila virtual es un tipo de función, si lo es, devuelve `1`, de lo contrario, devuelve `0`. El prototipo de esta función es:

```
int be_isfunction(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, e `index` es el índice del valor a medir. Hay tres tipos de funciones: cierre, función nativa y cierre nativo.

La función `be_isproto` devuelve si el valor indexado por el parámetro `index` en la pila virtual es de tipo `proto`, si lo es, devuelve `1`, de lo contrario, devuelve `0`. El prototipo de esta función es:

```
int be_isproto(bvm *vm, int index);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual, e `index` es el índice del valor a medir. El tipo `proto` es el prototipo de función del cierre de Berry.

La función `be_isclass` devuelve si el valor indexado por el parámetro `index` en la pila virtual es de tipo `class`, si lo es, devuelve `1`, de lo contrario devuelve `0`. El prototipo de esta función es:

```
int be_isclass(bvm *vm, int index);
```

El parámetro vm es el puntero de la instancia de la máquina virtual, e index es el índice del valor a medir.

La función be_isinstance devuelve si el valor indexado por el parámetro index en la pila virtual es de tipo instance, si lo es, devuelve 1, de lo contrario devuelve 0. El prototipo de esta función es:

```
int be_isinstance(bvm *vm, int index);
```

El parámetro vm es el puntero de la instancia de la máquina virtual, e index es el índice del valor a medir.

La función be_isbytes devuelve si el valor indexado por el parámetro index en la pila virtual es una instancia o subinstancia de la clase bytes; si lo es, devuelve 1; de lo contrario, devuelve 0. El prototipo de esta función es:

```
int be_isbytes(bvm *vm, int index);
```

La función be_islist devuelve si el valor indexado por el parámetro index en la pila virtual es de tipo list, si lo es, devuelve 1, de lo contrario devuelve 0. El prototipo de esta función es:

```
int be_islist(bvm *vm, int index);
```

El parámetro vm es el puntero de la instancia de la máquina virtual, e index es el índice del valor a medir.

La función be_ismap devuelve si el valor indexado por el parámetro index en la pila virtual es de tipo map, si lo es, devuelve 1, de lo contrario devuelve 0. El prototipo de esta función es:

```
int be_ismap(bvm *vm, int index);
```

El parámetro vm es el puntero de la instancia de la máquina virtual, e index es el índice del valor a medir.

La función be_iscomptr devuelve si el valor indexado por el parámetro index en la pila virtual es un tipo de puntero universal, si lo es, devuelve 1, de lo contrario, devuelve 0. El prototipo de esta función es:

```
int be_iscomptr(bvm *vm, int index);
```

El parámetro vm es el puntero de la instancia de la máquina virtual, e index es el índice del valor a medir.

```
bint be_toint(bvm *vm, int index);
```

Obtiene el valor de la posición de índice de index de la pila virtual y devuelve como un tipo entero. Esta función no comprueba la corrección del tipo. Si el valor es una instancia, se llama al método toint() si existe.

```
breal be_toreal(bvm *vm, int index);
```

Obtiene el valor de la posición de índice de index de la pila virtual y devuelve como un tipo de número de punto flotante. Esta función no comprueba la exactitud del tipo.

```
bint be_toindex(bvm *vm, int index);
```

Obtiene el valor de la posición de índice de index de la pila virtual y devuelve como un tipo entero. Esta función no comprueba la corrección del tipo. A diferencia de be_toint, el tipo de valor de retorno de be_toindex es int, mientras que el valor de retorno del primero es bint.

```
bbool be_tobool(bvm *vm, int index);
```

Obtiene el valor de la posición de índice de index de la pila virtual y devuelve como un tipo booleano. Si el valor indexado no es de tipo booleano, se convertirá de acuerdo con las reglas de la sección type_bool, y el proceso de conversión no hará que cambie el valor indexado. Si el valor es una instancia, se llama al método tobool() si existe.

```
const char* be_tostring(bvm *vm, int index);
```

Obtiene el valor de la posición de índice de `index` de la pila virtual y devuelve como un tipo de cadena. Si el valor indexado no es un tipo de cadena, el valor indexado se convertirá en una cadena y el proceso de conversión reemplazará el valor en la posición indexada en la pila virtual con la cadena convertida. La cadena devuelta por esta función siempre termina con los caracteres `'\0'`. Si el valor es una instancia, se llama al método `tostring()` si existe.

```
void* be_tocomptr(bvm* vm, int index);
```

Obtiene el valor de la posición de índice de `index` de la pila virtual y devuelve como un tipo de puntero general. Esta función no comprueba la exactitud del tipo.

```
const void* be_tobytes(bvm *vm, int index, size_t *len);
```

Obtiene el valor de la posición de índice de `index` de la pila virtual y devuelve como un búfer de bytes. Se devuelve el puntero del búfer y el tamaño se almacena en `*len` (a menos que `len` sea NULL). Esta función funciona solo para instancias de la clase `bytes`, o devuelve `NULL`.

```
void be_pushnil(bvm *vm);
```

Inserta un valor `nil` en la pila virtual.

```
void be_pushbool(bvm *vm, int b);
```

Inserta un valor booleano en la pila virtual. El parámetro `b` es el valor booleano que se insertará en la pila. Cuando el valor es `0`, significa falso, de lo contrario es verdadero.

```
void be_pushint(bvm *vm, bint i);
```

Inserta un valor entero `i` en la pila virtual.

```
void be_pushreal(bvm *vm, breal r);
```

Inserta un valor de punto flotante `r` en la pila virtual.

```
void be_pushstring(bvm *vm, const char *str)
```

Empuja la cadena `str` en la pila virtual. El parámetro `str` debe apuntar a una cadena C que termina con un carácter nulo `'\0'`, y no se puede pasar un puntero nulo.

```
void be_pushnstring(bvm *vm, const char *str, size_t n);
```

Inserta la cadena `str` de longitud `n` en la pila virtual. La longitud de la cadena está sujeta al parámetro `n` y el carácter nulo no se usa como marca final de la cadena.

```
const char* be_pushfstring(bvm *vm, const char *formato, ...);
```

Empuja la cadena formateada en la pila virtual. El parámetro `formato` es una cadena formateada que contiene el texto que se insertará en la pila, y el parámetro `formato` contiene una etiqueta, que puede ser reemplazada por el valor especificado por el parámetro adicional subsiguiente y formateada según sea necesario. De acuerdo con la etiqueta de la cadena `formato`, se pueden requerir una serie de parámetros adicionales, y cada parámetro adicional reemplazará la etiqueta `%` correspondiente en el parámetro `formato`.

espec ificador	Descripción	
d	Formato como entero con signo decimal (los números positivos no generan signo)	
f	Número de punto flotante de precisión simple o doble con formato decimal	
g	Número de punto flotante de precisión simple o doble con formato exponencial	
s	Formatear como cadena	
c	Formatear como un solo carácter	
p	Formatear como dirección de puntero	
%	Escapado como carácter % (sin parámetro)	

Tabla 12: Descripción del parámetro de la etiqueta ``formato``

La función `be_pushfstring` es similar a la función estándar de C `printf`, pero la función de formato de cadenas es relativamente básica y no admite operaciones como personalizar el ancho y los lugares decimales. Un ejemplo típico es:

```
be_pushfstring(vm, "%s: %d", "hola", 12); // ¡Bien, funciona!
be_pushfstring(vm, "%s: %.5d", "hola", 12); // Error, el ancho especificado no es␣
↪compatible.
```

Esto significa que `be_pushfstring` solo puede realizar operaciones de formateo simples. Si no se pueden cumplir los requisitos, se recomienda utilizar cadenas con formato `sprintf` para las operaciones.

```
void be_pushvalue(bvm *vm, int index);
```

Empuja el valor con el índice `index` en la parte superior de la pila virtual.

```
void be_pushntvclosure(bvm *vm, bntvfunc f, int nupvals);
```

Empuja un cierre nativo en la parte superior de la pila virtual. El parámetro `f` es el puntero de función C del cierre nativo, y `nupvals` es el número de valor superior del cierre.

```
void be_pushntvfunction(bvm *vm, bntvfunc f);
```

Empuja una función nativa en la parte superior de la pila virtual y el parámetro `f` es el puntero de la función nativa.

```
void be_pushclass(bvm *vm, const char *name, const bnfuncinfo *lib);
```

Empuja una clase nativa en la parte superior de la pila virtual. El parámetro `name` es el nombre de la clase nativa y el parámetro `lib` es la descripción del atributo de la clase nativa.

```
void be_pushcomptr(bvm *vm, void *ptr);
```

Empuja un puntero general en la parte superior de la pila virtual. El puntero general `ptr` apunta a una determinada área de datos de C. Dado que el recolector de elementos no utilizados de Berry no mantiene el contenido señalado por este puntero, los usuarios deben mantener el ciclo de vida de los datos ellos mismos.

```
void* be_pushbytes(bvm *vm, const void *buf, size_t len);
```

Empuja un búfer `bytes ()` que comience en la posición `buf` y de tamaño `len`. El búfer se copia en la memoria asignada de Berry, no necesita mantener el búfer válido después de esta llamada.

```
bbool be_pushiter(bvm *vm, int index);
```

Empuja un iterador en la parte superior de la pila virtual.

La función `be_pusherror` inserta un mensaje de error en la parte superior de la pila. Después de ejecutar el FFI, el intérprete volverá directamente a la posición que puede manejar el error, y el código inmediatamente siguiente no se ejecutará. El prototipo de función es:

```
void be_pusherror(bvm *vm, const char *msg);
```

El parámetro `vm` es el puntero de la instancia de la máquina virtual; `msg` es la cadena que contiene la información del error.

Mueve el valor en el índice `desde` a la posición del índice `hasta`. Esta función no elimina el valor de la posición del índice `desde`, solo modifica el valor de la posición del índice `hasta`.

Tecnología de construcción en tiempo de compilación

La tecnología de construcción en tiempo de compilación se implementa principalmente mediante *coc*, que se encuentra en la ruta *coc/coc* del directorio del código fuente del intérprete. La herramienta *coc* se usa para generar cadenas constantes y objetos constantes como código C, y se compilará en constantes cuando se compile el intérprete. En principio, la herramienta *coc* generará código a partir de la información de declaración del objeto constante (de acuerdo con un formato específico). El proceso calculará automáticamente el valor Hash y generará la tabla Hash.

El archivo *Makefile* en el directorio raíz del proyecto del intérprete compilará automáticamente esta herramienta y la ejecutará antes de compilar el código fuente del intérprete. El contenido de *Makefile* asegura que cuando se usa el comando `make`, el código para construir el objeto en tiempo de compilación siempre se actualizará a través de la herramienta (si necesita actualizarse). El código para construir objetos en tiempo de compilación se puede generar manualmente a través del comando `make prebuild`, que se almacena en la carpeta *generate*.

La construcción en tiempo de compilación se puede activar o desactivar modificando la macro `BE_USE_PRECOMPILED_OBJECT`. En cualquier caso, se llama a la herramienta *coc* para generar códigos de objetos constantes (los códigos no se usan cuando la construcción en tiempo de compilación está desactivada).

Usar el comando coc

La herramienta `coc` se utiliza para generar código para objetos constantes. El formato del comando es

```
tools/coc/coc -o <dst_path> <src_path(s)> -c <include_path>
```

La ruta de salida *dst_path* se utiliza para almacenar el código generado, y la ruta de origen *src_path* es una lista de rutas que deben escanearse en busca del código fuente (utilice espacios para separar varias rutas). `include_path` contiene un archivo de encabezado C escaneado para detectar directivas de compilación. `coc` intenta compilar solo las constantes necesarias. Dado que *generate* se usa como la ruta del código generado en el código fuente del intérprete, *dst_path* debe ser *generate*. Tomando el proyecto de intérprete estándar como ejemplo, el comando para usar la herramienta en `map_build` debe ser

```
tools/coc/coc -o generate default src -c default/berry_conf.h
```

El significado de este comando es: la ruta de salida es *generate*, y la ruta de origen es *src* y *default*.

Ruta de salida

Estrictamente hablando, la carpeta *generate* utilizada como ruta de salida no se puede colocar en ningún lado, debe almacenarse en un directorio principal que contenga la ruta. La ruta de inclusión se refiere a la ruta donde se buscará el archivo de encabezado en el proyecto. Tomando el código fuente del intérprete estándar como ejemplo, la ruta de inclusión es *src* y *default*. Por lo tanto, en el proyecto de intérprete estándar, la carpeta *generate* se almacena en el directorio raíz del código fuente del intérprete (el directorio principal de *src* y *default*).

El motivo de las reglas anteriores es que los siguientes códigos se utilizan en el código fuente del intérprete para hacer referencia a objetos constantes:

```
#include "../generate/be_fixed_xxx.h"
```

Si los lectores quieren definir objetos constantes por sí mismos, también necesitan usar dicho código para incluir los archivos de encabezado correspondientes. Esta sección presentará cómo usar estas herramientas para definir y usar objetos constantes.

Tabla de cadenas en tiempo de compilación

La tabla de cadenas en tiempo de compilación se utiliza para almacenar cadenas constantes. Las cadenas constantes son objetos que son transparentes para el script. No se crean ni destruyen cuando el intérprete se está ejecutando, pero siempre se almacenan como constantes en el segmento de datos del programa del intérprete. Si necesita usar una cadena como cadena constante, puede agregar el prefijo `be_const_str_` delante de la cadena en el código fuente del intérprete, y la declaración se puede colocar en cualquier parte del archivo fuente (incluidos los comentarios). Por ejemplo, para crear una cadena constante con el contenido `"cadena"`, debe declarar el símbolo `be_const_str_cadena` en el archivo fuente, y este símbolo también es el nombre de la variable que hace referencia a la cadena constante en el código C.

Todas las palabras clave crearán cadenas constantes. Si modifica el código relacionado con la palabra clave en el intérprete de Berry, también se debe modificar el código correspondiente en *coc*.

Si la cadena contiene símbolos especiales, se "escapean" automáticamente como `_XHH`, donde `HH` es la representación hexadecimal (en mayúsculas) del carácter. Por ejemplo `"` está representado por `_X3A`. Esta representación es biactiva, por lo que es fácil convertirla a la cadena original y desde ella.

Usar cadena constante

Normalmente, no hay necesidad de declarar cadenas constantes manualmente, ni de usarlas manualmente. Si realmente necesita llamar a la cadena constante manualmente, incluya el archivo de encabezado *be_constobj.h* para usar todas las variables de cadena constante (este archivo de encabezado tiene declaraciones para todas las cadenas constantes). El uso típico de cadenas constantes es construir objetos en tiempo de compilación. La declaración y definición de cadenas constantes en este proceso son manejadas automáticamente por la herramienta.

En cualquier caso, la función FFI `be_pushstring` debe usarse directamente para crear una cadena. Cuando una cadena tiene una cadena constante, no creará repetidamente un nuevo objeto de cadena, sino que usará directamente la cadena constante correspondiente.

De forma predeterminada, todas las cadenas utilizadas se referencian en una tabla hash global `m_const_string_table`. Sin embargo, algunos proyectos pueden tener muchas variantes de compilación para las que no se necesitan algunos conjuntos de cadenas. Si todas las constantes de cadena se almacenan en todas las variantes, esto crea una pérdida de tamaño de flash. Por esta razón, algunas cadenas pueden declararse como cadenas "débiles" en el sentido de tener una referencia "débil". En tal caso, la constante de cadena se declara en código C, pero no se incluye en el objeto de mapa global. Esto significa que el enlazador puede optar por no incluir las constantes de cadena si ningún código hace referencia a ellas. La desventaja es que si crea dinámicamente un objeto de cadena

con el mismo valor, se crea un nuevo objeto en la memoria (mientras que no lo haría para una constante de cadena normal). Para indicar cadenas débiles, use el modificador `strings: weak` (ver más abajo).

Construir objeto en tiempo de compilación

Los objetos construidos en tiempo de compilación también se denominan objetos constantes. La estructura de datos de estos objetos se construye cuando se compila el intérprete y no se puede modificar en tiempo de ejecución. `map_build` define un conjunto de reglas de declaración en la herramienta para generar código C para objetos constantes. La información de declaración del objeto constante se almacena directamente en el archivo fuente (*.*c*). Para distinguirlo de otro contenido, se debe incluir una información de declaración completa en el siguiente código de arranque:

```
@const_object_info_begin
@const_object_info_end
```

La información de declaración de objeto constante no se ajusta a la sintaxis del lenguaje C, por lo que debe colocarse en un comentario de varias líneas (incluido con `/* */`). Todos los objetos constantes tienen la misma forma de declaración. La estructura de declaración de un objeto constante se denomina "bloque de declaración de objeto", que se compone de

```
type object_name (attributes) {
    member_fields
}
```

`type` es el tipo de objeto constante, puede ser `map`, `class`, `module` o `vartab`. `object_name` es el nombre de la variable del objeto constante en lenguaje C. `attributes` es la lista de atributos de los objetos constantes. Un atributo se compone de nombre de atributo y valor de atributo. El nombre del atributo y el valor del atributo están separados por punto y coma, y varios atributos están separados por comas. Por ejemplo, la lista de atributos `scope: global, name: map` significa que el atributo `scope` de un objeto constante es `global`, y el atributo `name` es `map`. También `strings: weak` indica que se debe generar constantes de cadena débiles para los nombres de los campos de miembros o cualquier constante de cadena. `member_fields` es la lista de dominios miembros de objetos constantes. Un miembro se compone de nombre y valor, separados por comas. Cada línea puede declarar un miembro y varios miembros deben declararse en varias líneas.

La herramienta **coc** utiliza expresiones regulares para analizar el bloque de declaración de objetos. En el proceso de análisis, primero se comparará todo el bloque de declaración del objeto y se comparará la información "tipo" y "nombre_del_objeto". Para la información de `atributos` y `member_fields`, sae hará un análisis adicional. Para facilitar la implementación, la herramienta no tiene requisitos estrictos sobre la sintaxis del bloque de declaración de objetos y carece de un mecanismo completo de manejo de errores, por lo que debe asegurarse de que la sintaxis sea correcta al escribir el bloque de declaración de objetos.

Para facilitar la comprensión, ilustramos con una clase constante simple:

```
/* @const_object_info_begin
class be_class_map (scope: global, name: map) {
    .data, var
    init, func(m_init)
    tostring, func(m_tostring)
}
@const_object_info_end */
#include "../generate/be_fixed_be_class_map.h"
```

En este ejemplo, la información de declaración de toda la clase constante está en el comentario del lenguaje C, por lo que no afectará la compilación del código C. El bloque de declaración de objetos se coloca entre `@const_object_info_begin` y `@const_object_info_end` para garantizar que la herramienta **coc** detecte el bloque de declaración de objetos.

Dado que es una declaración de clase constante, el valor de *tipo* en el bloque de declaración de objeto es `class`, y `be_class_map` es el nombre de variable del objeto constante en el código C. Se declaran dos atributos en la lista de atributos del objeto (la parte encerrada entre paréntesis), y el significado de estos atributos se presentará en la sección "Clase de construcción en tiempo de compilación" de esta sección. Tres miembros están definidos en la lista de miembros entre llaves, varios miembros están separados por saltos de línea y el nombre del miembro y el valor del miembro están separados por una coma. Existen varios formatos legales para los nombres de miembros:

- Formato de nombre de variable Berry: comienza con una letra o guión bajo, seguido de varias letras, guiones bajos o números.
- Utilice "`.`" como primer carácter, seguido de letras, guiones bajos o números.
- Operadores sobrecargables, como "+", "-" y "<<", etc.

El valor de un miembro puede ser de los siguientes tipos:

- `var`: Este símbolo se compilará en un objeto entero (`be_const_var`), y el valor del objeto entero se incrementa automáticamente desde `0`. `var` está diseñado para la declaración de variables miembro en la clase, y es automática La función de numeración se utiliza para realizar el número de serie de las variables miembro.
- `func(símbolo)`: Declara funciones miembro nativas o métodos de objetos constantes. El símbolo se compilará en una función nativa con el valor (`be_const_func`), `symbol` es el puntero de función nativo correspondiente al miembro. `m_init` y `m_tostring` en el ejemplo son dos funciones nativas.
- `closure (símbolo)`: Declara funciones o métodos miembro de bytecode precompilados de objetos constantes. El símbolo se compilará en una función nativa con el valor (`be_const_closure`), `símbolo` es el nombre de la función solidificada. Ver `módulo solidificar` para saber cómo solidificar objetos.
- `nil()`: este símbolo se compilará en un valor nulo (`be_const_nil`).
- `int(valor)`: este símbolo se compilará en un objeto entero (`be_const_int`), el valor del objeto entero es `valor`.
- `real(valor)`: Este símbolo se compilará en un número real (`be_const_real`), el valor del objeto de número real es `valor`.
- `comptr(valor)`: Este símbolo se compilará en un puntero objeto (`be_const_comptr`), el valor del puntero es `valor` y se puede utilizar para pasar la dirección de una estructura global de C.
- `class (símbolo)`: este símbolo se compilará en un objeto de clase (`be_const_class`). `símbolo` es un puntero a este tipo de objeto, y el puntero debe apuntar a un objeto de tipo constante.
- `module (símbolo)`: este símbolo se compilará en un objeto de módulo (`be_const_module`). `símbolo` es un puntero al objeto del módulo, y el puntero debe apuntar a un objeto de módulo constante.
- `ctype_func(símbolo)`: Este símbolo se compilará en una función nativa (`be_const_ctype_func`). `símbolo` es un puntero al mapeo C definición. Esta característica es utilizada por berry_mapping

Para usar el objeto `be_class_map`, debemos incluir el archivo de encabezado correspondiente en el código C para garantizar que el objeto se compilará. La práctica habitual es incluir el archivo de cabecera correspondiente cerca del bloque de declaración del objeto. En el ejemplo, la línea 8 lo contiene. El archivo de encabezado correspondiente se puede usar para construir objetos `be_class_map` en tiempo de compilación.

Después de procesarlo con la herramienta **coc**, cada bloque de declaración de objeto se compilará en un archivo de encabezado llamado *be_fixed_be_xxx.h*, donde *xxx* es el nombre de la variable C del objeto. Para compilar objetos constantes en código C, debemos incluir los archivos de encabezado correspondientes. Por lo general, se recomienda incluir los archivos de encabezado correspondientes cerca del bloque de declaración de objetos. La octava línea del ejemplo contiene *be_fixed_be_class_map.h* para construir el objeto `be_class_map` en tiempo de compilación.

Construir mapa en tiempo de compilación

Los mapas construidos en tiempo de compilación también son objetos `map` constantes. Por lo general, no se declaran directamente mediante bloques de declaración de objetos, sino que se declaran en otras estructuras de construcción en tiempo de compilación. Al construir el `map` constante, la información del tipo de objeto constante debe ser `map`, que admite un atributo `scope`. Cuando el valor del atributo 'scope' es 'local', el objeto constante es 'estático', cuando el atributo es 'global', es 'externo', y el valor de este atributo es 'local' por defecto. Los `member_fields` del objeto `map` constante admiten especificaciones comunes de nombre de miembro/valo, y los valores de miembro solo se almacenan como datos sin una interpretación especial. El siguiente es un ejemplo del uso del bloque de declaración de objetos para declarar directamente un objeto `map` constante:

```
map map_name (scope: local/global) {
    init, func(m_init)
}
```

Construcción de Clases en tiempo de compilación

Para construir una clase en tiempo de compilación, use el bloque de declaración de objetos para declarar, y la información de tipo del objeto es `class`. Las propiedades declaradas de este objeto son `scope` y `name`. `scope` es el alcance de la variable C del objeto de declaración de atributos, cuando el valor es `local` (predeterminado), el alcance es `static`, cuando es `global`, el alcance es `extern`; `name` es el valor del atributo es ese nombre de clase, la clase anónima puede omitir este parámetro. Dado que la lista de atributos de una clase solo almacena métodos e índices de variables miembro, los `member_fields` de la clase construida en tiempo de compilación solo pueden usar los valores `var` y `func()`. Un bloque de declaración de clase de construcción simple en tiempo de compilación es:

```
class be_class_map (scope: global, name: map) {
    .data, var
    init, func(m_init)
    tostring, func(m_tostring)
}
```

Construcción de Módulos en tiempo de compilación

La información de tipo del bloque de declaración del bloque de construcción en tiempo de compilación es `module`.

```
module math (scope: global) {
    sin, func(m_sin)
    cos, func(m_cos)
    pi, real(M_PI)
}
```

Construcción de un dominio integrado en tiempo de compilación

```
vartab m_builtin (scope: local) {
    assert, func(l_assert)
    print, func(l_print)
    list, class(be_class_list)
}
```

Definición de gramática

Este capítulo dará algunas definiciones gramaticales relacionadas con Berry. Usamos **Extended Backus Normal Form** (EBNF) para definir o expresar la gramática. No usamos la gramática EBNF estricta para definir, pero hicimos muchas simplificaciones, pero estas simplificaciones no afectarán la comprensión de la gramática por parte de los lectores.

La definición EBNF de la gramática del lenguaje Berry es la siguiente:

```
(* program define *)
program = block;

(* block define *)
block = {statement};

(* statement define *)
statement = class_stmt | func_stmt | var_stmt | if_stmt | while_stmt |
         for_stmt | break_stmt | return_stmt | expr_stmt | import_stmt |
         try_stmt | throw_stmt | ';';
if_stmt = 'if' expr block {'elif' expr block} ['else' block] 'end';
while_stmt = 'while' expr block 'end';
for_stmt = 'for' ID ':' expr block 'end';
break_stmt = 'break' | 'continue';
return_stmt = 'return' [expr];

(* function define statement *)
func_stmt = 'def' ID func_body;
func_body = '(' [arg_field {',' arg_field}] ')' block 'end';
arg_field = ['*'] ID;

(* class define statement *)
class_stmt = 'class' ID [':' ID] class_block 'end';
class_block = {'var' ID {',' ID} | 'static' ['var'] ID ['=' expr] {',' ID ['=' expr] } |
↪'static' func_stmt | func_stmt};
import_stmt = 'import' (ID (['as' ID] | {',' ID}) | STRING 'as' ID);

(* exceptional handling statement *)
try_stmt = 'try' block except_block {except_block} 'end';
except_block = except_stmt block;
except_stmt = 'except' (expr {',' expr} | '..') ['as' ID [',' ID]];
throw_stmt = 'raise' expr [',' expr];

(* variable define statement *)
var_stmt = 'var' ID ['=' expr] {',' ID ['=' expr]};
```

(continues on next page)

(continued from previous page)

```
(* expression define *)
expr_stmt = expr [assign_op expr];
expr = suffix_expr | unop expr | expr binop expr | range_expr | cond_expr;
cond_expr = expr '?' expr ':' expr; (* conditional expression *)
assign_op = '=' | '+=' | '-=' | '*=' | '/=' |
            '%=' | '&=' | '|=' | '^=' | '<<=' | '>>=';
binop = '<' | '<=' | '==' | '!=' | '>' | '>=' | '||' | '&&' |
        '<<' | '>>' | '&' | '|' | '^' | '+' | '-' | '*' | '/' | '%';
range_expr = expr '..' [expr]
unop = '-' | '!' | '~';
suffix_expr = primary_expr {call_expr | ('.' ID) | '[' expr ']'};
primary_expr = '(' expr ')' | simple_expr | list_expr | map_expr | anon_func | lambda_
↪expr;
simple_expr =  INTEGER | REAL | STRING | ID | 'true' | 'false' | 'nil';
call_expr = '(' [expr {',' expr}] ')';
list_expr = '[' {expr ','} [expr] ']';
map_expr = '{' {expr ':' expr ','} [expr ':' expr] '}';
anon_func = 'def' func_body;

(* anonymous function *)
lambda_expr = '/' [arg_field {',' arg_field}] | {arg_field}] '->' expr;
```

El formato EBNF estándar se puede encontrar en materiales relacionados. Aquí hay una explicación de los detalles que necesitan atención al leer la gramática anterior. Los símbolos que han aparecido a la izquierda del signo igual son símbolos no terminales, y los demás son símbolos terminales. El terminador encerrado entre comillas ' es una cadena fija, que suele ser una palabra clave u operador de idioma. Hay varios terminadores que son inconvenientes para describir directamente en EBNF: `INTEGER` representa el valor nominal del entero; `REAL` representa el valor nominal del número real; `STRING` representa el valor literal de cadena; `ID` representa el identificador. Estos terminadores se pueden definir mediante expresiones regulares:

- `ENTERO`: `0x[a-fA-F0-9]+|\d+`.
- `REAL`: `(\d+\.?|\.\d)\d*([eE][+-]?\d+)?`.
- `CADENA`: `"(\\.|[^"])*"|'(\\.|[^'])*'`.
- `ID`: `[_a-zA-Z]\w*`

Los símbolos que aparecen secuencialmente en el EBNF estándar están separados por comas. Por intuición, uso espacios para implementar la función de coma. El símbolo de barra vertical "|" se pronuncia como "o", significa que los patrones izquierdo y derecho solo pueden coincidir con uno de ellos, o tiene la prioridad más baja. Por ejemplo, la gramática *a* 0 *a* 1 |*a* 2 significa la fórmula correspondiente *a* 0 *a* 1 o la combinación *a* 2 . Los corchetes indican que la subexpresión dentro de los paréntesis coincide 0 o 1 veces, las llaves indican que la subexpresión interna coincide 0 o más veces, y los paréntesis solo tienen la función de tomar la subexpresión interna como un todo.

La siguiente es la definición de gramática JSON admitida por el módulo JSON en la biblioteca estándar de Berry. El uso de EBNF aún cumple con las convenciones anteriores:

```
json = value;
value = object | array |
        string | number | 'true' | 'false' | 'null';
object = '{' [ string ':' value ] { ',' string ':' value } '}';
array = '[' [json] { ',' json } ']';
```

Los símbolos no terminales `cadena` y `número` también se pueden definir mediante expresiones regulares. http://www.json.org proporciona la gramática estándar de JSON, que también incluye las definiciones de `cadena` y `número`.

El soporte para números de la biblioteca Berry JSON es diferente del estándar. Los números JSON estándar deben comenzar con "-" o el número "0-9", mientras que la biblioteca Berry JSON también acepta números que comienzan con un punto decimal.

Intérprete del compilador

1. Visión general

El código fuente del intérprete de Berry está escrito con el estándar ISO C99 y el código central no depende de bibliotecas de terceros, por lo que tiene una gran versatilidad. Tome el sistema Ubuntu como ejemplo, ejecute el siguiente comando en la terminal para instalar el intérprete de Berry:

```
apt install git gcc g++ make libreadline-dev
git clone https://github.com/berry-lang/berry
cd berry
make
make install
```

El Makefile proporcionado en el repositorio de GitHub se crea con el compilador GCC. Otros compiladores también pueden compilar correctamente el intérprete de Berry. Los compiladores actualmente probados y disponibles incluyen GCC, Clang, MSVC, ARMCC e ICCARM. El compilador que compila el intérprete de Berry debe tener las siguientes características:

- Compilador C que soporta el estándar C99
- Compilador C++ compatible con el estándar C++11 (solo para compilación nativa)
- Plataforma de destino de 32 o 64 bits

El compilador de C++ solo se usa para compilar las herramientas *map_build*, por lo que no es necesario proporcionar un compilador cruzado de C++ para el intérprete de Berry al realizar la compilación cruzada, pero el usuario debe preparar un compilador nativo de C++ (a menos que el usuario pueda obtener el * archivo ejecutable de la herramienta map_build*).

2. portabilidad

Lo siguiente es cómo portar el intérprete de Berry al proyecto del usuario:

1. Agregue todos los archivos fuente en el directorio *src* al proyecto de usuario y el directorio debe agregarse a la ruta de inclusión
2. Los usuarios deben implementar por sí mismos archivos *predeterminados* que no sean *berry.c* en el directorio. Si las condiciones lo permiten, no necesitan modificarlos
3. Utilice la herramienta *map_build* para generar código de objeto constante y luego compilar

3. Soporte de plataforma

Actualmente, el intérprete de Berry ha sido probado en algunas plataformas. Los sistemas operativos Windows, Linux y MacOS que se ejecutan en CPU X86 pueden funcionar normalmente. Las plataformas integradas que se han probado incluyen Cortex M3/M0/M4/M7. El intérprete de Berry debería poder funcionar bien sobre la base de la biblioteca de tiempo de ejecución C necesaria. En la actualidad, cuando solo se compila el núcleo del lenguaje Berry, el código del intérprete generado por el compilador ARMCC tiene solo alrededor de 40 KiB, y el intérprete puede ejecutarse en un dispositivo con solo 8 KiB de RAM.

Guía de portabilidad

Archivo de configuración

El archivo de encabezado de configuración del intérprete de Berry es *berry_conf.h*. Este archivo incluye un lote de macros para la configuración y define algunos contenidos relacionados con la plataforma.

berry_conf.h Archivo

Interruptor de macro de configuración

Las macros de configuración presentadas en esta sección generalmente se usan para compilar conmutadores de algunos códigos fuente. Para facilitar la descripción, llamamos a esta macro "cambio de macro". Para el interruptor de macro, "on" se refiere a establecer el interruptor de macro en un valor distinto de cero, y "off" se refiere a establecer el valor del interruptor de macro en `0`.

Algunos interruptores macro tienen varios estados, no solo "encendido" o "apagado". Estos interruptores macro se usan generalmente para configuraciones con múltiples opciones. También hay algunas macros de configuración que no son conmutadores de macro. No importa cuál sea el valor de estas macros, no habrá código y, por lo tanto, no participará en la compilación. Estas macros se utilizan generalmente para configurar el valor de la cantidad.

[sección::BE_DEBUG]

Este conmutador de macro se utiliza para activar o desactivar la función de depuración del propio intérprete. Cuando el valor de `BE_DEBUG` es `0`, la depuración se desactiva; de lo contrario, se activará. La función de depuración mencionada aquí se refiere a la depuración del intérprete, no a la función de depuración del programa Berry. El valor predeterminado de `BE_DEBUG` es `0`. Si usa *Makefile* que viene con el proyecto del intérprete para compilar, este interruptor de macro se activará automáticamente cuando use el comando `make debug`.

Cuando se abre esta macro, se activarán algunas aserciones y se generará un mensaje de error cuando el intérprete encuentre un error que la aserción pueda detectar. Puede abrir `BE_DEBUG` al depurar el intérprete y cerrarlo al compilar la versión.

Este conmutador de macro configura el tipo de punto flotante utilizado por el tipo `breal`. Cuando el valor de la macro es `0`, se usará el tipo `doble` para implementar `breal`, de lo contrario, se usará el tipo `float` para implementar `breal`. Este interruptor de macro se puede activar en algunos entornos con bajo rendimiento o configuración de memoria. En la implementación predeterminada, este interruptor de macro está desactivado.

Esta macro configura la implementación del tipo `bint`. Cuando el valor de la macro es `0`, se usará el tipo `int` para implementar `bint`, cuando el valor sea `1`, se usará el tipo `long` para implementar `bint`, y cuando el el valor es `2`, se usará 64] Tipo de entero con signo de bit (`__int64` en Windows, `long long` en otras plataformas) implementa `bint`. El valor predeterminado de esta macro es `2`. Si desea reducir el uso de la memoria, puede establecer esta macro en `0` o `1` para habilitar el tipo de entero de 32 bits.

Esta macro se usa para configurar la información de depuración en tiempo de ejecución del código Berry. Tiene 3 valores disponibles: establecer en `0` para desactivar la salida del nombre de archivo y el número de línea de la información de depuración en tiempo de ejecución, y establecer en `1` para mostrar el nombre de archivo y el número de línea en la información de depuración en tiempo de ejecución, establecer a `2` Al usar `uint16_t` (entero de 16 bits) escriba para almacenar la información del número de fila. Su valor por defecto es `1`.

Establecer esta macro en `0` no almacenará el nombre del archivo ni la información del número de línea, por lo que el consumo de memoria es mínimo. Cuando se establece en `2`, consume menos memoria, pero si el programa es demasiado largo, `uint16_t` se desbordará.

Este modificador de macro configura la función de construir objetos en tiempo de compilación. Activar esta macro significa que la construcción de objetos en tiempo de compilación está habilitada. Esta macro está activada de forma predeterminada. Cuando esta macro está activada, los objetos nativos de la biblioteca estándar se generarán en tiempo de compilación y cuando esta macro está desactivada, los objetos de la biblioteca estándar se generarán en tiempo de ejecución.

Las funciones `be_regfunc` y `be_regclass` se verán afectadas por esta macro. La tabla de objetos integrada no se puede modificar cuando se utiliza la construcción de objetos en tiempo de compilación. En este momento, estas dos funciones no pueden registrar objetos en el ámbito integrado, pero registran objetos en el ámbito global.

Los objetos construidos durante el tiempo de compilación se almacenan junto con el código y no ocuparán recursos de RAM (o el área de lectura y escritura en la memoria). La tecnología de construcción durante el tiempo de compilación también puede reducir el tiempo de inicio del intérprete, por lo que se recomienda abrir esta macro. Consulte la sección [section::precompiled_build] para obtener más detalles sobre las técnicas de construcción en tiempo de compilación.

Esta macro define la capacidad máxima de pila de Berry, que se refiere a la cantidad de objetos de Berry. Cuando el código Berry usa más de esta cantidad de pila, dejará de ejecutar el programa y devolverá un mensaje de error. El valor predeterminado de esta macro es `2000`, que puede modificarse según la capacidad de memoria del sistema.

Este valor no afecta el uso de memoria de la pila Berry, porque la capacidad de la pila Berry se ajusta dinámicamente, por lo que no importa cuánto se establezca, no puede ayudar a reducir el uso de memoria. Su función principal es terminar la ejecución cuando el programa Berry consume demasiada pila. Es muy probable que estos programas sean incorrectos, por ejemplo, las llamadas a funciones recursivas sin condiciones de devolución seguirán consumiendo la pila.

Esta macro define el espacio mínimo disponible en la pila de Berry y su valor predeterminado es `10`. La función nativa puede insertar un valor en la pila de Berry. En este momento, la pila no crecerá automáticamente, así que asegúrese de que haya suficiente espacio en la pila para que la use la función nativa. No se recomienda modificar este valor, sino usar la función `be_stack_require` donde realmente se necesita más espacio de pila.

Para detectar errores de desbordamiento de pila al depurar el intérprete, puede abrir la macro `BE_DEBUG` (sección [sección::BE_DEBUG]).

Cuando se abre esta macro, el objeto de cadena corta guardará el valor hash de la cadena para mejorar la velocidad de ejecución, pero el tamaño de cada objeto de cadena aumentará en 4 bytes. Esta macro está desactivada de forma predeterminada y las pruebas actuales no han encontrado que abrir esta macro traiga una mejora significativa.

Este conmutador de macro se utiliza para activar o desactivar el módulo `string`, que está activado de forma predeterminada.

Este interruptor de macro se usa para habilitar o deshabilitar el módulo `json`, que está activado de forma predeterminada.

Este conmutador de macro se utiliza para habilitar o deshabilitar el módulo "matemático", que está activado de forma predeterminada.

Este interruptor de macro se usa para habilitar o deshabilitar el módulo `time`, que está activado de forma predeterminada.

Este conmutador de macro se utiliza para activar o desactivar el módulo `os`, que está activado de forma predeterminada.

Esta macro determina la función `abortar` utilizada internamente por el intérprete de Berry. Por defecto o cuando la macro no está definida, se utilizará la función `abortar` en la biblioteca estándar de C. Esta macro se define como `abortar` por defecto. Si el usuario necesita especificar explícitamente la función `abortar` utilizada por el intérprete, reemplace la definición de macro con la función requerida por el usuario. Esta función debe tener la misma forma que la declaración de la función `abortar` en la biblioteca estándar.

Esta macro determina la función `exit` utilizada internamente por el intérprete de Berry. Por defecto o cuando la macro no está definida, se utilizará la función `exit` en la biblioteca estándar de C. Esta macro se define como `salir` por defecto. Si el usuario necesita especificar explícitamente la función `salir` utilizada por el intérprete, reemplace la definición de macro con la función requerida por el usuario. Esta función debe tener la misma forma que la declaración de la función `exit` en la biblioteca estándar.

Esta macro determina la función `malloc` utilizada internamente por el intérprete de Berry. Por defecto o cuando la macro no está definida, se utilizará la función `malloc` en la biblioteca estándar de C. Esta macro se define como `malloc` por defecto. Si el usuario necesita especificar explícitamente la función `malloc` utilizada por el intérprete, reemplace la definición de macro con la función requerida por el usuario. Esta función debe tener la misma forma que la declaración de la función `malloc` en la biblioteca estándar.

Esta macro determina la función `libre` utilizada internamente por el intérprete de Berry. Por defecto o cuando la macro no está definida, se utilizará la función `free` en la biblioteca estándar de C. Esta macro se define como "gratis" por defecto. Si el usuario necesita especificar explícitamente la función "libre" utilizada por el intérprete, reemplace la definición de macro con la función requerida por el usuario. Esta función debe tener la misma forma que la declaración de la función "libre" en la biblioteca estándar.

Esta macro determina la función `realloc` utilizada internamente por el intérprete de Berry. Por defecto o cuando la macro no está definida, se utilizará la función `realloc` en la biblioteca estándar de C. Esta macro se define como `realloc` por defecto. Si el usuario necesita especificar explícitamente la función `realloc` utilizada por el intérprete, reemplace la definición de macro con la función requerida por el usuario. Esta función debe tener la misma forma que la declaración de la función `realloc` en la biblioteca estándar.

Esta macro se utiliza para definir la implementación de la función de aserción. De forma predeterminada, la función `assert` en la biblioteca estándar de C se usa para implementar la afirmación. Si el sistema de destino tiene inconvenientes para usar la función `assert()` en la biblioteca estándar para hacer una afirmación, puede modificar la definición de la macro `be_assert`. Una función de aserción correcta debe usar la siguiente declaración:

```
void assert(int condition);
```

Entre ellos, `condición` es la condición de afirmación. Si no se cumple la condición, se emitirá un mensaje de error y el programa finalizará. Por supuesto, la función "afirmar" generalmente se implementa mediante una macro.

Archivo *berry_port.c*

Este archivo implementa las funciones de E/S de bajo nivel del intérprete de Berry, incluida la entrada y salida estándar y la compatibilidad con el sistema de archivos. El archivo *berry_port.c* en el directorio *predeterminado* contiene un conjunto de soporte de E/S portátil. Las operaciones de archivo y la entrada y salida estándar se implementan mediante API en la biblioteca estándar de C. Las operaciones de ruta y carpeta son compatibles con las API estándar de Windows y POSIX. Este archivo también implementa un conjunto de funciones de operación de E/S basadas en FatFs para que los usuarios las usen directamente. Si necesita usar el intérprete de Berry en otros entornos, estas funciones deben implementarse por separado (es posible que solo deban implementarse parcialmente).

Esta sección presentará las funciones de las funciones implementadas en el archivo *berry_port.c* y guiará a los usuarios para implementar su propia versión.

```
void be_writebuffer(const char *buffer, size_t length);
```

Envíe un dato al dispositivo de salida estándar, el parámetro "búfer" es la primera dirección del bloque de datos de salida y "longitud" es la longitud del bloque de datos de salida. Esta función genera el archivo `stdout` de forma predeterminada. Dentro del intérprete, esta función generalmente se usa como una salida de flujo de caracteres, no como un flujo binario.

Las funciones `be_writebuffer` son muy versátiles y deben implementarse.

```
char* be_readstring(char *buffer, size_t size);
```

Ingrese un dato del dispositivo de entrada estándar y lea como máximo una fila de datos cada vez que se llame a esta función. El parámetro `buffer` es el búfer de datos pasado por la persona que llama, y la capacidad del búfer es `tamaño`. Esta función dejará de leer y regresará cuando se agote la capacidad del búfer; de lo contrario, regresará cuando se lea un carácter de nueva línea o un carácter de fin de archivo. Si la función se ejecuta con éxito, usará directamente el parámetro `buffer` como valor de retorno, de lo contrario devolverá `NULL`.

Esta función agregará los saltos de línea de lectura a los datos leídos, y cada vez que se llame a la función `be_readstring`, continuará leyendo desde la posición actual. Esta función solo se llama en la implementación de la función nativa `input`, y es posible que la función `be_readstring` no se implemente cuando no sea necesaria.

```
void* be_fopen(const char *filename, const char *modes);
```

Para abrir un archivo, `filename` es el nombre del archivo que se abrirá y `modos` es el método de apertura. La función devolverá un identificador de archivo o un puntero a la estructura de operación del archivo. El uso de esta función es similar a la función `fopen` en la biblioteca estándar de C. El nombre del archivo es una cadena de estilo C (que termina con un carácter `\0`), y el patrón debe admitir al menos las siguientes condiciones:

- `r`, `rt`: Para abrir un archivo de texto en modo de solo lectura, el archivo debe existir.
- `r+`, `rt+`: Abre un archivo de texto en modo lectura-escritura y crea un nuevo archivo si el archivo no existe.
- `rb`: abre un archivo binario en modo de solo lectura, el archivo debe existir.
- `rb+`: abre un archivo binario en modo de lectura y escritura y crea un nuevo archivo si el archivo no existe.
- `w`, `wt`: Crear y abrir un archivo de texto en modo de solo escritura, y el el archivo existente será eliminado.
- `w+`, `wt+`: Crea y abre un archivo de texto en modo lectura-escritura, y el el archivo existente será eliminado.
- `wb`: crea y abre un archivo binario en modo de solo escritura, y el el archivo existente será eliminado.
- `wb+`: Crea y abre un archivo binario en modo lectura-escritura, y el el archivo existente será eliminado.

De forma predeterminada, la función `fopen` en la biblioteca estándar de C se usa para implementar `be_fopen`. Si utiliza otros métodos para lograrlo, debe asegurarse de que se puedan lograr los modos de funcionamiento anteriores. Si no se requieren operaciones de archivo, esta función se puede dejar en blanco. Las operaciones de archivos aquí incluyen todos los escenarios, como usar la función `abrir` en el script, cargar el script desde un archivo (usando la función `be_loadfile`), etc.

```
int be_fclose(void *hfile);
```

Cierra un archivo, `hfile` es el identificador de archivo cerrado. La función de esta función es similar a la función `fclose` en la biblioteca estándar de C.

```
size_t be_fwrite(void *hfile, const void *buffer, size_t length);
```

Escribe un dato en el archivo especificado. El parámetro `hfile` es el identificador del archivo que se escribirá, `buffer` es el puntero de los datos que se escribirán, `length` es el número de datos que se escribirán (en bytes).

```
size_t be_fread(void *hfile, void *buffer, size_t length);
```

Leer un fragmento de datos del archivo especificado. El parámetro `hfile` es el identificador del archivo que se leerá, `buffer` es el puntero al búfer de lectura y `length` es el número de bytes que se leerán.

```
char* be_fgets(void *hfile, void *buffer, int size);
```

Lea una línea del archivo, similar a la función `fgets` en la biblioteca estándar de C. El parámetro `hfile` es el identificador del archivo que se va a leer, `buffer` es el puntero del búfer de lectura y `size` es la capacidad del búfer de lectura. Esta función regresará cuando se lean los bytes `size - 1`, los caracteres de nueva línea y los caracteres de fin de archivo, y el valor de retorno sea `buffer`.

```
int be_fseek(void *hfile, long offset);
```

Establezca la posición del puntero de lectura y escritura del archivo. El parámetro `hfile` es el identificador de archivo que se va a utilizar y `offset` es el valor que se va a establecer.

```
long int be_ftell(void *hfile);
```

Obtenga el puntero de lectura y escritura actual del archivo, el parámetro `hfile` es el identificador del archivo que se va a operar, y el valor de retorno de esta función es el puntero de lectura y escritura del archivo.

```
long int be_fflush(void *hfile);
```

Escriba los datos del búfer de archivo en el archivo. El parámetro `hfile` es el archivo a operar.

```
size_t be_fsize(void *hfile);
```

Obtenga el tamaño del archivo. El parámetro `hfile` es el archivo a operar.

1.2.2 Hacer una función nativa

La FFI en C (interfaz de función externa) de Berry opera en una pila virtual para interactuar con la máquina virtual. Si necesitamos hacer una función `add` para sumar dos números y usarla en Berry de esta manera:

```
result = add(1, 2)
```

Necesitamos saber cómo el código C obtiene los argumentos de la llamada a la función Berry y cómo devolver el valor.

Los argumentos de la función se almacenan en una pila, y desde el primer argumento hasta el último argumento de la función se almacenan desde la parte inferior de la pila hasta la parte superior de la pila. Si desea utilizar C para obtener elementos de la pila, utilice el siguiente conjunto de FFI:

```
int be_toint(bvm *vm, int index);
breal be_toreal(bvm *vm, int index);
int be_tobool(bvm *vm, int index);
const char* be_tostring(bvm *vm, int index);
void* be_tocomptr(bvm *vm, int index);
```

Si desea probar si un valor en la pila es de un tipo específico, use el siguiente conjunto de FFI:

```
int be_isnil(bvm *vm, int index);
int be_isbool(bvm *vm, int index);
int be_isint(bvm *vm, int index);
int be_isreal(bvm *vm, int index);
int be_isnumber(bvm *vm, int index);
```

(continues on next page)

(continued from previous page)

```
int be_isstring(bvm *vm, int index);
int be_isclosure(bvm *vm, int index);
int be_isntvclos(bvm *vm, int index);
int be_isfunction(bvm *vm, int index);
int be_isproto(bvm *vm, int index);
int be_isclass(bvm *vm, int index);
int be_isinstance(bvm *vm, int index);
int be_islist(bvm *vm, int index);
int be_ismap(bvm *vm, int index);
int be_iscomptr(bvm *vm, int index);
```

Si necesita enviar valores a la pila, use estos FFI:

```
void be_pushnil(bvm *vm);
void be_pushbool(bvm *vm, int b);
void be_pushint(bvm *vm, bint i);
void be_pushreal(bvm *vm, breal r);
void be_pushstring(bvm *vm, const char *str);
void be_pushnstring(bvm *vm, const char *str, size_t n);
const char* be_pushfstring(bvm *vm, const char *format, ...);
void be_pushvalue(bvm *vm, int index);
void be_pushntvclosure(bvm *vm, bntvfunc f, int nupvals);
void be_pushntvfunction(bvm *vm, bntvfunc f);
void be_pushclass(bvm *vm, const char *name, const bnfuncinfo *lib);
void be_pushcomptr(bvm *vm, void *ptr);
```

`index` es la posición del elemento en la pila, un valor positivo es el desplazamiento desde la parte inferior de la pila hasta la parte superior de la pila, y un valor negativo es el desplazamiento desde la parte superior de la pila hasta la parte inferior de la pila.

El valor de retorno utiliza dos FFI:

```
be_return(vm)
be_return_nil(vm)
```

Estos FFI son en realidad macros. `be_return` devuelve el objeto en la parte superior de la pila, y `be_return_nil` devuelve `nil`.

Estas FFI se definen en berry.h.

Ahora implementemos la función add:

```
int my_add_func(bvm *vm)
{
    /* comprobar que los argumentos son todos enteros */
    if (be_isint(vm, 1) && be_isint(vm, 2)) {
        bint a = be_toint(vm, 1); /* obtener el primer argumento */
        bint b = be_toint(vm, 2); /* obtener el segundo argumento */
        be_pushint(vm, a + b); /* empuja el resultado a la pila */
    } else if (be_isnumber(vm, 1) && be_isnumber(vm, 2)) { /* comprobar que los␣
↪argumentos son todos números */
        breal a = be_toreal(vm, 1); /* obtener el primer argumento */
        breal b = be_toreal(vm, 1); /* empuja el resultado a la pila */
        be_pushreal(vm, a + b); /* empuja el resultado a la pila */
```

(continues on next page)

(continued from previous page)

```
    } else { /* parámetros inaceptables */
        be_pushnil(vm); /* empuja nil a la pila */
    }
    be_return(vm); /* devuelve el resultado del cálculo */
}
```

Luego regístrelo en el lugar apropiado:

```
be_regcfunc(vm, "add", my_add_func);
```

1.2.3 Crear una instancia de un objeto `list` en una función nativa

La generación de clases nativas instanciadas en C puede ser engorrosa en comparación con los tipos simples. Esta sección guiará al lector a instanciar la clase `list`.

La clase `list` es un contenedor alrededor de la estructura de la lista, que tiene una propiedad `.data` para la estructura de la lista. Por lo tanto, primero necesitamos construir una estructura de lista:

```
be_newlist(vm);
```

La función `be_newlist` construye un valor de tipo `BE_LIST`. Entonces podemos operar sobre los datos:

```
be_pushint(vm, 100);
be_data_append(vm, -2);
be_pop(vm, 1); /* extraer el entero 100 */
```

Las dos primeras líneas de código se utilizan para añadir el entero `100` a la lista, y la tercera línea del entero `100` se extrae para facilitar las operaciones posteriores.

Dado que el tipo `BE_LIST` no se puede usar directamente en Berry, pero lo usa la clase `list`, tenemos que construir la clase `list` para él:

```
be_getglobal(vm, "list");
be_pushvalue(vm, -2); /* empuja los datos de la lista al principio */
be_call(vm, 1); /* llama al constructor */
```

El constructor de la clase `list` permite el uso del parámetro de tipo `BE_LIST`, que toma el argumento como datos de lista.

El código completo es el siguiente:

```
int m_listtest(bvm *vm)
{
    be_getglobal(vm, "list");
    be_newlist(vm);
    be_pushint(vm, 100);
    be_data_append(vm, -2);
    be_pop(vm, 1);
    be_call(vm, 1);
    be_pop(vm, 1); /* pop the arguments */
    be_return(vm);
}
```

Registre la función nativa en el lugar apropiado:

```
be_regcfunc(vm, "listtest", m_listtest);
```

1.2.4 Hoja de ruta

POR HACER

- Soporte multilínea REPL.
- Información de depuración en tiempo de ejecución.
- Protección de pila API.
- Soporte de operación de archivos.
- Tabla hash fija (basada en ROM).
- Soporte de destructor.
- Compatibilidad con módulos nativos (use `import xxx` para importar un módulo).
- Soporte de expresiones condicionales.
- Función anónima.
- Operación bit a bit.
- Sentencia de asignación compuesta.
- Alcance incorporado.
- Función de argumentos variables.
- Función nativa: `classof(obj)`.
- [STRIKEOUT:Función nativa: `copy(obj)`].
- Pila de llamadas de liberación automática.
- Expresión regular.
- GC optimizado para el uso de la pila (sin recursividad).
- Simplifica los mensajes de error de desbordamiento de pila.
- Expresión lambda.
- Manejo excepcional.
- Compatibilidad con archivos de bytecode.
- Iterador optimizado y sentencia `for`.
- Operador de conexión (redefine el operador de rango `..`).
  - Conexión de cadena, por ejemplo, `'cadena' .. expr`.
  - Lista de conexiones, por ejemplo, `[] .. expr`.
  - Método de serialización de lista (para conexión de cadena de alto rendimiento).
- Soporte de módulo completo.
  - Exportación/importación del módulo de archivo de script.
  - Compilación/importación del módulo de archivo de bytecode.
  - Carga del módulo nativo compartido (como *.so,*.dll).

- API de verificación de igualdad universal (como el operador == pero se usa en C).
  - Coincidencia de clave de mapa completa.
  - Coincidencia de valor de excepción completa.
- Compatibilidad con el depurador.
- Mecanismo completo de clases y objetos.
- Mensaje de error de soporte para la herramienta *map_build*.
- Llamada sobrecargada al operador `()`.

Versión de lanzamiento

V0.2.0

- Documentación china completa.
- Manejo excepcional.
- Soporte de módulo completo.

V0.1.0

- Interfaz de sistema de archivos portátil.
- Compatibilidad con objetos constantes precompilados.
- Módulo constante completo precompilado.
- [STRIKEOUT:Documentación más completa (chino)].
- [STRIKEOUT:Guía de portabilidad].

1.2.5 Requerimientos de Memoria

RAM requerida

- extrema: 4 KB
- baja: 8 KB
- recomendada: 16 KB

ROM/Flash requerida

- extrema: 64 KB
- baja: 128 KB
- recomendada: 256 KB

Explicación

- **extrema**: La capacidad de memoria requerida en la configuración mínima. Un valor inferior a este significa que es casi imposible ejecutar Berry.
- **baja**: la memoria mínima requerida para que el intérprete completo de Berry incluya algunos códigos de usuario y de terceros.
- **recomendado**: además del intérprete completo de Berry, se sportan muchos usuarios y código de terceros.

1.3 API documentation

Main berry entry API.

This file is part of the Berry default interpreter.

Author
skiars@qq.com, https://github.com/Skiars/berry

Copyright
(c) 2018-2022 Guan Wenliang MIT License (https://github.com/Skiars/berry/blob/master/LICENSE)

native-module member type specifier.

BE_CNIL

native-module member type specifier.

BE_CNIL

BE_CINT

BE_CINT

BE_CREAL

BE_CREAL

BE_CBOOL

BE_CBOOL

BE_CFUNCTION

BE_CFUNCTION

BE_CSTRING

BE_CSTRING

BE_CMODULE

BE_CMODULE

only linux.

BERRY_LOCAL

only linux

BERRY_LOCAL

BE_EXPORT_VARIABLE

BE_EXPORT_VARIABLE

debug hook typedefs.

BE_HOOK_LINE

debug hook typedefs.

BE_HOOK_LINE

BE_HOOK_CALL

BE_HOOK_CALL

BE_HOOK_RET

BE_HOOK_RET

BE_HOOK_EXCEPT

BE_HOOK_EXCEPT

Defines

BERRY_VERSION

do not modify the version number!

BE_INTEGER

BE_INTEGER

BE_INT_FMTLEN

BE_INT_FMTLEN

BE_INT_FORMAT

BE_INT_FORMAT

bbool

bbool

bfalse

bfalse

btrue

btrue

BERRY_API

API function mark.

be_native_module_nil(_name)

native module node definition macro

be_native_module_int(_name, _v)

be_native_module_int

be_native_module_real(_name, _v)

be_native_module_real

be_native_module_bool(_name, _b)

be_native_module_bool

be_native_module_function(_name, _f)

be_native_module_function

be_native_module_str(_name, _s)

be_native_module_str

be_native_module_module(_name, _m)

be_native_module_module

be_native_module_attr_table(name)

be_native_module_attr_table

be_native_module(name)

be_native_module

be_native_class(name)

be_native_class

be_extern_native_module(name)

native module declaration macro

be_extern_native_class(name)

native class declaration macro

be_define_native_module(_name, _init)

native module definition macro

be_define_local_const_str(_name, _s, _hash, _len)

support for solidified berry functions native const strings outside of global string hash

be_nested_const_str(_s, _hash, _len)

new version for more compact literals

be_local_const_str(_name)

be_local_const_str

BE_IIF(cond)

conditional macro see https://stackoverflow.com/questions/11632219/c-preprocessor-macro-specialisation-based-on-an-argument

BE_IIF_0(t, f)

conditional macro see https://stackoverflow.com/questions/11632219/c-preprocessor-macro-specialisation-based-on-an-argument

BE_IIF_1(t, f)

conditional macro see https://stackoverflow.com/questions/11632219/c-preprocessor-macro-specialisation-based-on-an-argument

be_local_const_upval(ins, idx)

be_local_const_upval

PROTO_SOURCE_FILE(n)

PROTO_SOURCE_FILE_STR(n)

PROTO_RUNTIME_BLOCK

conditional block in bproto depending on compilation options

PROTO_VAR_INFO_BLOCK

PROTO_VAR_INFO_BLOCK.

be_define_local_proto(_name, _nstack, _argc, _is_const, _is_subproto, _is_upval)

define bproto

be_nested_proto(_nstack, _argc, _varg, _has_upval, _upvals, _has_subproto, _protos, _has_const, _ktab, _fname, _source, _code)

new version for more compact literals

be_define_local_closure(_name)

be_define_local_closure

be_local_closure(_name, _proto)

new version for more compact literals

be_assert(expr)

the default assert definition

be_writestring(s)

be_writestring

Note: FFI function

---

be_writenewline()

be_writenewline

Note: FFI function

---

be_return(vm)

be_return

Note: FFI function

Parameters

- **vm** – virtual machine instance virtual machine instance

be_return_nil(vm)

be_return_nil

Note: FFI function

Parameters

- **vm** – virtual machine instance virtual machine instance

be_loadfile(vm, name)

be_loadfile

Note: FFI function

Parameters

- **vm** – virtual machine instance virtual machine instance
- **name** – (???)

be_loadmodule(vm, name)

be_loadmodule

Note: FFI function

Parameters

- **vm** – virtual machine instance virtual machine instance
- **name** –

be_loadstring(vm, str)

be_loadstring

Note: FFI function

Parameters

- **vm** – virtual machine instance virtual machine instance
- **str** – (???)

be_dostring(vm, s)

be_dostring

Note: FFI function

Parameters

- **vm** – virtual machine instance virtual machine instance
- **s** – (???)

Typedefs

typedef uint8_t **bbyte**

bbyte

typedef BE_INTEGER **bint**

bint

typedef double **breal**

breal

typedef struct *bvm* **bvm**

virtual machine structure

typedef int (***bntvfunc**)(*bvm**)

native function pointer

typedef struct *bntvmodobj* **bntvmodobj_t**

bntvmodobj_t

typedef struct *bntvmodule* **bntvmodule_t**

bntvmodule_t

typedef const struct bclass ***bclass_ptr**

we need only the pointer to `bclass` here

typedef *bclass_ptr* **bclass_array**[]

array of bclass* pointers, NULL terminated

typedef struct *bhookinfo* **bhookinfo_t**

bhookinfo_

typedef void (***bntvhook**)(*bvm* *vm, *bhookinfo_t* *info)

void (*bntvhook)(bvm *vm, bhookinfo *info)

Param vm
virtual machine instance

Param info

typedef void (***bobshook**)(*bvm* *vm, int event, ...)

Observability hook.

Param vm
virtual machine instance

Param event

typedef int (*`bctypefunc`)(*bvm**, const void*)

bctypefunc

Enums

enum `berrorcode`

error code definition

Values:

enumerator `BE_OK`

BE_OK

enumerator `BE_EXIT`

BE_EXIT

enumerator `BE_MALLOC_FAIL`

BE_MALLOC_FAIL

enumerator `BE_EXCEPTION`

BE_EXCEPTION

enumerator `BE_SYNTAX_ERROR`

BE_SYNTAX_ERROR

enumerator `BE_EXEC_ERROR`

BE_EXEC_ERROR

enumerator `BE_IO_ERROR`

BE_IO_ERROR

enum `beobshookevents`

beobshookevents

Values:

enumerator `BE_OBS_PCALL_ERROR`

called when be_callp() returned an error, most likely an exception

enumerator `BE_OBS_GC_START`

start of GC, arg = allocated size

enumerator **BE_OBS_GC_END**

end of GC, arg = allocated size

enumerator **BE_OBS_VM_HEARTBEAT**

VM heartbeat called every million instructions

enumerator **BE_OBS_STACK_RESIZE_START**

Berry stack resized

Functions

BERRY_API *bint* **be_str2int**(const char *str, const char **endstr)

(???)

Note: FFI function

Parameters

- **str** – (???)
- **endstr** – (???)

Returns

(???)

BERRY_API *breal* **be_str2real**(const char *str, const char **endstr)

(???)

Note: FFI function

Parameters

- **str** –
- **endstr** – (???)

Returns

(???)

BERRY_API const char ***be_str2num**(*bvm* *vm, const char *str)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance virtual machine instance
- **str** – (???)

Returns
(???)

BERRY_API int **be_top**(*bvm* *vm)

returns the absolute index value of the top element in the virtual stack

---

Note: FFI function This function returns the absolute index value of the top element in the virtual stack. This value is also the number of elements in the virtual stack (the capacity of the virtual stack). Call this function before adding or subtracting elements in the virtual stack to get the number of parameters of the native function.

Parameters
vm – virtual machine instance virtual machine instance

Returns
(???)

BERRY_API const char *__be_typename__(*bvm* *vm, int index)

converts the type of the Berry object into a string and returns it

---

Note: FFI function This function converts the type of the Berry object into a string and returns it. For example, it returns "int" for an integer object, and "function" for a function object.

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – index of the object to be operated

Returns
string corresponding to the parameter type (see: baselib_type)

BERRY_API const char *__be_classname__(*bvm* *vm, int index)

converts the type of the Berry object into a string and returns it.

Note: FFI function This function converts the type of the Berry object into a string and returns it. For example, it returns "int" for an integer object, and "function" for a function object

---

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – index of the object to be operated

Returns
string corresponding to the parameter type (see: baselib_type)

BERRY_API bbool **be_classof**(*bvm* *vm, int index)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – (???)

Returns
(???)

BERRY_API int **be_strlen**(*bvm* *vm, int index)

length of the specified Berry string

Note: FFI function This function returns the number of bytes in the string at index (the '\0' characters at the end of the Berry string are not counted). If the value of the index position is not a string, the be_strlen function will return 0. Although the Berry string is compatible with the C string format, it is not recommended to use the strlen function of the C standard library to measure the length of the Berry string. For Berry strings, be_strlen is faster than strlen and has better compatibility.

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – index of the object to be operated

Returns
length

BERRY_API void **be_strconcat**(*bvm* *vm, int index)

splice two Berry strings

Note: FFI function This function will concatenate the string at the parameter position of index with the string at the top position of the stack, and then put the resulting string into the position indexed by index.

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – (???)

BERRY_API void **be_pop**(*bvm* *vm, int n)

pops the value at the top of the stack

Note: FFI function Note that the value of n cannot exceed the capacity of the stack.

Parameters

- **vm** – virtual machine instance virtual machine instance
- **n** – number of values to be popped

BERRY_API void **be_remove**(*bvm* *vm, int index)

remove a value from the stack

Note: FFI function After the value at index is moved out, the following values will be filled forward, and the stack capacity will be reduced by one. The value of index cannot exceed the capacity of the stack.

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – the object to be removed

BERRY_API int **be_absindex**(*bvm* *vm, int index)

absolute index value of a given index value

Note: FFI function If index is positive, the return value is the value of index. If index is negative, the return value of textttbe_absindex is the absolute index value corresponding to index. When index is a negative value (relative index), its index position cannot be lower than the bottom of the stack.

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – index value

Returns

absolute index

BERRY_API bbool **be_isnil**(*bvm* *vm, int index)

value in virtual stack is nil

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is nil, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – value index

Returns

true/false

BERRY_API bbool **be_isbool**(*bvm* *vm, int index)

value in virtual stack is bool

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is bool, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – value index

Returns
true/false

BERRY_API bbool **be_isint**(*bvm* *vm, int index)

value in virtual stack is int

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is int, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – value index

Returns
true/false

BERRY_API bbool **be_isreal**(*bvm* *vm, int index)

value in virtual stack is real

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is real, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – value index

Returns
true/false

BERRY_API bbool **be_isnumber**(*bvm* *vm, int index)

value in virtual stack is number

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is number, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – value index

Returns
true/false

BERRY_API bbool **be_isstring**(*bvm* *vm, int index)

value in virtual stack is string

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is string, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance
- **index** – value index

Returns

true/false

BERRY_API bbool **be_isclosure**(*bvm* *vm, int index)

value in virtual stack is closure

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is closure, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance
- **index** – value index

Returns

true/false

BERRY_API bbool **be_isntvclos**(*bvm* *vm, int index)

value in virtual stack is primitive closure type

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is primitive closure type, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance
- **index** – value index

Returns

true/false

BERRY_API bbool **be_isfunction**(*bvm* *vm, int index)

value in virtual stack is function

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is function, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – value index

Returns
true/false

BERRY_API bbool **be_isproto**(*bvm* *vm, int index)

value in virtual stack is proto

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is proto, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – value index

Returns
true/false

BERRY_API bbool **be_isclass**(*bvm* *vm, int index)

value in virtual stack is class

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is class, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – value index

Returns
true/false

BERRY_API bbool **be_isinstance**(*bvm* *vm, int index)

value in virtual stack is instance

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is instance, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – value index

Returns
true/false

BERRY_API bbool **be_ismapinstance**(*bvm* *vm, int index)

value in virtual stack is instance

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is an instance of class map (or derived). If it is, it returns 1, otherwise it returns 0

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is an instance of class list (or derived). If it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – value index
- **vm** – virtual machine instance virtual machine instance
- **index** – value index

Returns
true/false

Returns
true/false

BERRY_API bbool **be_islistinstance**(*bvm* *vm, int index)

BERRY_API bbool **be_ismodule**(*bvm* *vm, int index)

value in virtual stack is module

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is module, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – value index

Returns
true/false

BERRY_API bbool **be_islist**(*bvm* *vm, int index)

value in virtual stack is list

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is list, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance
- **index** – value index

Returns
true/false

BERRY_API bbool **be_ismap**(*bvm* *vm, int index)

value in virtual stack is map

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is map, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance
- **index** – value index

Returns
true/false

BERRY_API bbool **be_iscomptr**(*bvm* *vm, int index)

value in virtual stack is universal pointer type

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is universal pointer type, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance
- **index** – value index

Returns
true/false

BERRY_API bbool **be_iscomobj**(*bvm* *vm, int index)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – value index

Returns
true/false

BERRY_API bbool **be_isderived**(*bvm* *vm, int index)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – value index

Returns
true/false

BERRY_API bbool **be_isbytes**(*bvm* *vm, int index)

value in virtual stack is instance or sub-instance of class bytes

---

Note: FFI function This function returns whether the value indexed by the parameter index in the virtual stack is instance or sub-instance of class bytes, if it is, it returns 1, otherwise it returns 0

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – value index

Returns
true/false

BERRY_API *bint* **be_toint**(*bvm* *vm, int index)

virtual stack to integer type

Note: FFI function Get the value of the index position of index from the virtual stack and return it as an integer type. This function does not check the correctness of the type. If the value is an instance, the method toint() is called if it exists.

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** – value index

Returns
(???)

BERRY_API *breal* **be_toreal**(*bvm* *vm, int index)

virtual stack to floating-point number type

Note: FFI function Get the value of the index position of index from the virtual stack and return it as an floating-point number type. This function does not check the correctness of the type.

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** –

Returns
(???)

BERRY_API int **be_toindex**(*bvm* *vm, int index)

virtual stack to integer type

Note: FFI function Get the value of the index position of index from the virtual stack and return it as an integer type. This function does not check the correctness of the type. Unlike be_toint, the return value type of be_toindex is int, while the return value of the former is bint.

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** –

Returns

(???)

BERRY_API bbool **be_tobool**(*bvm* *vm, int index)

virtual stack to Boolean type

Note: FFI function Get the value of the index position of index from the virtual stack and return it as a Boolean type. If the indexed value is not of Boolean type, it will be converted according to the rules in section type_bool, and the conversion process will not cause the indexed value to change. If the value is an instance, the method tobool() is called if it exists.

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** –

Returns

(???)

BERRY_API const char ***be_tostring**(*bvm* *vm, int index)

virtual stack to string

Note: FFI function Get the value of the index position of index from the virtual stack and return it as a string type. If the indexed value is not a string type, the indexed value will be converted to a string, and the conversion process will replace the value at the indexed position in the virtual stack with the converted string. The string returned by this function always ends with '\0' characters. If the value is an instance, the method tostring() is called if it exists.

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** –

Returns

(???)

BERRY_API const char *be_toescape(*bvm* *vm, int index, int mode)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** –
- **mode** –

Returns

(???)

BERRY_API void *be_tocomptr(*bvm* *vm, int index)

virtual stack to general pointer

Note: FFI function Get the value of the index position of index from the virtual stack and return it as a general pointer type. This function does not check the correctness of the type.

Parameters

- **vm** – virtual machine instance virtual machine instance
- **index** –

BERRY_API void be_moveto(*bvm* *vm, int from, int to)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance virtual machine instance
- **from** –
- **to** –

BERRY_API void be_pushnil(*bvm* *vm)

Push a nil value onto the virtual stack.

Note: FFI function

Parameters

vm – virtual machine instance

BERRY_API void **be_pushbool**(*bvm* *vm, int b)

Push a Boolean value onto the virtual stack.

Note: FFI function Push a Boolean value onto the virtual stack. The parameter b is the boolean value to be pushed onto the stack. When the value is 0, it means false, otherwise it is true.

Parameters

- **vm** – virtual machine instance
- **b** –

BERRY_API void **be_pushint**(*bvm* *vm, *bint* i)

Push an integer value i onto the virtual stack.

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **i** –

BERRY_API void **be_pushreal**(*bvm* *vm, *breal* r)

Push a floating point value r onto the virtual stack.

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **r** –

BERRY_API void **be_pushstring**(*bvm* *vm, const char *str)

Push the string str onto the virtual stack.

Note: FFI function Push the string str onto the virtual stack. The parameter str must point to a C string that ends with a null character '\0', and a null pointer cannot be passed in.

Parameters

- **vm** – virtual machine instance
- **str** –

BERRY_API void **be_pushnstring**(*bvm* *vm, const char *str, size_t n)

Push the string str of length n onto the virtual stack.

Note: FFI function Push the string str of length n onto the virtual stack. The length of the string is subject to the parameter n, and the null character is not used as the end mark of the string.

Parameters

- **vm** – virtual machine instance
- **str** –
- **n** –

BERRY_API const char ***be_pushfstring**(*bvm* *vm, const char *format, ...)

Push the formatted string into the virtual stack.

Note: FFI function Push the formatted string into the virtual stack. The parameter format is a formatted string, which contains the text to be pushed onto the stack, and the format parameter contains a label, which can be replaced by the value specified by the subsequent additional parameter and formatted as required. According to the label of the format string, a series of additional parameters may be required, and each additional parameter will replace the corresponding % label in the format parameter.

Parameters

- **vm** – virtual machine instance
- **format** –

Returns

(???)

BERRY_API void ***be_pushbuffer**(*bvm* *vm, size_t size)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **size** –

BERRY_API void **be_pushvalue**(*bvm* *vm, int index)

Push the value with index index onto the top of the virtual stack.

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **index** –

BERRY_API void **be_pushclosure**(*bvm* *vm, void *cl)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **cl** –

BERRY_API void **be_pushntvclosure**(*bvm* *vm, *bntvfunc* f, int nupvals)

Push a native closure onto the top of the virtual stack.

Note: FFI function Push a native closure onto the top of the virtual stack. The parameter f is the C function pointer of the native closure, and nupvals is the upvalue number of the closure.

Parameters

- **vm** – virtual machine instance
- **f** –
- **nupvals** –

BERRY_API void **be_pushntvfunction**(*bvm* *vm, *bntvfunc* f)

Push a native function onto the top of the virtual stack, and the parameter f is the native function pointer.

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **f** –

BERRY_API void **be_pushclass**(*bvm* *vm, const char *name, const *bnfuncinfo* *lib)

Push a native class onto the top of the virtual stack.

Note: FFI function Push a native class onto the top of the virtual stack. The parameter name is the name of the native class, and the parameter lib is the attribute description of the native class.

Parameters

- **vm** – virtual machine instance
- **name** –
- **lib** –

BERRY_API void **be_pushntvclass**(*bvm* *vm, const struct bclass *c)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **c** –

BERRY_API void **be_pushcomptr**(*bvm* *vm, void *ptr)

Push a general pointer onto the top of the virtual stack.

Note: FFI function Push a general pointer onto the top of the virtual stack. The general pointer ptr points to a certain C data area. Since the content pointed to by this pointer is not maintained by Berry's garbage collector, users have to maintain the life cycle of the data themselves.

Parameters

- **vm** – virtual machine instance
- **ptr** –

BERRY_API bbool **be_pushiter**(*bvm* *vm, int index)

Push an iterator onto the top of the virtual stack.

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **index** –

Returns

(???)

BERRY_API void **be_newlist**(*bvm* *vm)

creates a new list value

Note: FFI function After this function is successfully called, the new list value will be pushed onto the top of the stack. list value is an internal representation of a list, not to be confused with an instance of the list class.

Parameters

vm – virtual machine instance

BERRY_API void **be_newmap**(*bvm* *vm)

creates a new map value

Note: FFI function After this function is successfully called, the new map value will be pushed onto the top of the stack. map value is an internal representation of a list, not to be confused with an instance of the map class.

Parameters

vm – virtual machine instance

BERRY_API void **be_newmodule**(*bvm* *vm)

(???)

Note: FFI function

Parameters

vm – virtual machine instance

BERRY_API void **be_newcomobj**(*bvm* *vm, void *data, *bntvfunc* destroy)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **data** –
- **destroy** –

BERRY_API void **be_newobject**(*bvm* *vm, const char *name)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **name** –

BERRY_API bbool **be_copy**(*bvm* *vm, int index)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **index** –

Returns

BERRY_API bbool **be_setname**(*bvm* *vm, int index, const char *name)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **index** –
- **name** –

Returns

(???)

BERRY_API bbool **be_getglobal**(*bvm* *vm, const char *name)

pushes the global variable with the specified name onto the stack

Note: FFI function After this function is called, the global variable named name will be pushed onto the top of the virtual stack

Parameters

- **vm** – virtual machine instance
- **name** –

Returns

(???)

BERRY_API void **be_setglobal**(*bvm* *vm, const char *name)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **name** –

BERRY_API bbool **be_getbuiltin**(*bvm* *vm, const char *name)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **name** –

Returns
(???)

BERRY_API bbool **be_setmember**(*bvm* *vm, int index, const char *k)

set the value of the member variable of the instance object class

---

Note: FFI function This function will copy the value at the top of the stack to the member k of the index position instance. Note that the top element of the stack will not pop up automatically.

Parameters

- **vm** – virtual machine instance
- **index** – index of the instance object
- **k** – name of the member

Returns
(???)

BERRY_API bbool **be_getmember**(*bvm* *vm, int index, const char *k)

get the value of the member variable of the instance object class

---

Note: FFI function This function pushes the value of the member of the index position instance k onto the top of the virtual stack.

Parameters

- **vm** – virtual machine instance
- **index** – index of the instance object
- **k** – name of the member

Returns
(???)

BERRY_API bbool **be_getmethod**(*bvm* *vm, int index, const char *k)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **index** –
- **k** –

Returns
(???)

BERRY_API bbool **be_getindex**(*bvm* *vm, int index)

get the value of list or map

Note: FFI function This function is used to get an element from the map or list container (internal values, not instances of map or list classes), and the index of the element is stored at the top of the stack (relative index is 1). After calling this function, the value obtained from the container will be pushed onto the top of the stack. If there is no subscript pointed to by the container, the value of nil will be pushed onto the top of the stack.

Parameters

- **vm** – virtual machine instance
- **index** – index of the object to be operated

Returns
(???)

BERRY_API bbool **be_setindex**(*bvm* *vm, int index)

set a value in list or map

Note: FFI function This function is used to write an element of the map or list container. The index of the value to be written in the virtual stack is 1, and the index of the subscript of the write position in the virtual stack is 2. If the element with the specified subscript does not exist in the container, the write operation will fail.

Parameters

- **vm** – virtual machine instance
- **index** – index of the object to be operated

Returns
(???)

BERRY_API void **be_getupval**(*bvm* *vm, int index, int pos)

read an Up Value of the native closure

Note: FFI function The read Up Value will be pushed onto the top of the virtual stack.

Parameters

- **vm** – virtual machine instance
- **index** – the native closure index value of the Up Value to be read
- **pos** – position of the Up Value in the native closure Up Value table (numbering starts from 0)

BERRY_API bbool **be_setupval**(*bvm* *vm, int index, int pos)

set an Up Value of the native closure.

Note: FFI function This function obtains a value from the top of the virtual stack and writes it to the target Up Value. After the operation is completed, the top value of the stack will not be popped from the stack.

Parameters

- **vm** – virtual machine instance
- **index** – the native closure index value of the Up Value to be read
- **pos** – position of the Up Value in the native closure Up Value table (numbering starts from 0)

Returns

(???)

BERRY_API bbool **be_setsuper**(*bvm* *vm, int index)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **index** –

Returns

(???)

BERRY_API void **be_getsuper**(*bvm* *vm, int index)

get the parent object of the base class or instance of the class.

Note: FFI function If the value at index is a class with a base class, the function will push its base class onto the top of the stack; if the value at index is an object with a parent object, the function will take its parent The object is pushed onto the top of the stack; otherwise, a value of nil is pushed onto the top of the stack.

Parameters

- **vm** – virtual machine instance
- **index** – the class or object to be operated

BERRY_API int **be_data_size**(*bvm* *vm, int index)

get the number of elements contained in the container

Note: FFI function If the value at index is a Map value or List value, the function returns the number of elements contained in the container, otherwise it returns -1.

Parameters

- **vm** – virtual machine instance
- **index** – index of the container object to be operated

Returns
(???)

BERRY_API void **be_data_push**(*bvm* *vm, int index)

append a new element to the end of the container.

Note: FFI function The object at index must be a List value. This function gets a value from the top of the stack and appends it to the end of the container. After the operation is completed, the value at the top of the stack will not be popped from the stack.

Parameters

- **vm** – virtual machine instance
- **index** – index of the container object to be operate

BERRY_API bbool **be_data_insert**(*bvm* *vm, int index)

insert a pair of elements into the container

Note: FFI function The object at index must be a List value or a Map value. The inserted element forms a pair of key-value pairs. The value is stored at the top of the stack, and the key is stored at the previous index on the top of the stack. It should be noted that the key inserted into the Map container cannot be a nil value, and the key inserted into the List container must be an integer value. If the operation is successful, the function will return bture, otherwise it will return bfalse.

Parameters

- **vm** – virtual machine instance
- **index** – container object to be operated

Returns
(???)

BERRY_API bbool **be_data_remove**(*bvm* *vm, int index)

remove an element in the container

Note: FFI function The object at index must be a List value or Map value. For the Map container, the key to delete the element is stored on the top of the virtual stack (need to be pressed before the function call); for the List container, the index of the element to be deleted is stored on the top of the virtual stack (need to be before the function call) push into). If the operation is successful, the function will return bture, otherwise it will return bfalse.

Parameters

- **vm** – virtual machine instance
- **index** – container object to be operated.

Returns
(???)

BERRY_API bbool **be_data_merge**(*bvm* *vm, int index)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **index** –

Returns
(???)

BERRY_API void **be_data_resize**(*bvm* *vm, int index)

reset the capacity of the container

Note: FFI function This function is only available for List containers, and the new capacity is stored on the top of the virtual stack (must be an integer).

Parameters

- **vm** – virtual machine instance
- **index** – container object to be operated

BERRY_API void **be_data_reverse**(*bvm* *vm, int index)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **index** –

BERRY_API int **be_iter_next**(*bvm* *vm, int index)

get the next element of the iterator

Note: FFI function The iterator object may be an iterator of a List container or a Map container. For the List iterator, this function pushes the iteration result value onto the top of the stack, while for the Map iterator, it pushes the key value into the previous position and the top of the stack respectively. Calling this function will update the iterator. If the function returns 0, the call fails, returning 1 to indicate that the current iterator is a List iterator, and returning 2 to indicate that the current iterator is a Map iterator.

Parameters

- **vm** – virtual machine instance
- **index** – iterator to be operated

Returns
(???)

BERRY_API bbool **be_iter_hasnext**(*bvm* *vm, int index)

test whether there is another element in the iterator

Note: FFI function The iterator object may be an iterator of a List container or a Map container. If there are more iterable elements in the iterator, return 1, otherwise return 0.

Parameters

- **vm** – virtual machine instance
- **index** – iterator to be operated

Returns
(???)

BERRY_API bbool **be_refcontains**(*bvm* *vm, int index)

test whether there is a reference to the specified object in the reference stack

---

Note: FFI function This function is used to test whether there is a reference to the specified object in the reference stack. It must be used in conjunction with be_refpush and be_refpop. This API can avoid recursion when traversing objects that have their own references. This function is used for the value of an instance type. If there is a reference to the object in the reference stack, it returns 1, otherwise it returns 0.

Parameters

- **vm** – virtual machine instance
- **index** – object to be operated

Returns
(???)

BERRY_API void **be_refpush**(*bvm* *vm, int index)

Push the reference of the specified object onto the reference stack.

Note: FFI function This function is used for the value of an instance type.

Parameters

- **vm** – virtual machine instance
- **index** – object to be operated

BERRY_API void **be_refpop**(*bvm* *vm)

Pop the object at the top of the reference stack.

Note: FFI function This function is used in pairs with be_refpush

---

Parameters

vm – virtual machine instance

BERRY_API void **be_stack_require**(*bvm* *vm, int count)

tests the amount of free space on the stack and expands the stack space if it is insufficient

---

Note: FFI function If the free capacity of the virtual stack allocated by the VM to the native function is lower than this value, an expansion operation will be performed.

Parameters

- **vm** – virtual machine instance
- **count** – required free stack capacity.

BERRY_API bbool **be_getmodule**(*bvm* *vm, const char *k)

(???)

Note: FFI function

Parameters

- **vm** – virtual machine instance
- **k** –

Returns

(???)

BERRY_API bbool **be_iseq**(*bvm* *vm)

(???)

Note: relop operation API

Parameters

vm – virtual machine instance

Returns

(???)

BERRY_API bbool **be_isneq**(*bvm* *vm)

(???)

Note: relop operation API

Parameters
vm – virtual machine instance

Returns
(???)

BERRY_API bbool **be_islt**(*bvm* *vm)

(???)

Note: relop operation API

Parameters
vm – virtual machine instance

Returns
(???)

BERRY_API bbool **be_isle**(*bvm* *vm)

(???)

Note: relop operation API

Parameters
vm – virtual machine instance

Returns
(???)

BERRY_API bbool **be_isgt**(*bvm* *vm)

(???)

Note: relop operation API

Parameters
vm – virtual machine instance

Returns
(???)

BERRY_API bbool **be_isge**(*bvm* *vm)

(???)

Note: relop operation API

Parameters
vm – virtual machine instance

Returns
(???)

BERRY_API int **be_returnvalue**(*bvm* *vm)

(???)

Note: Function call/return API

Parameters
vm – virtual machine instance

Returns
(???)

BERRY_API int **be_returnnilvalue**(*bvm* *vm)

(???)

Note: Function call/return API

Parameters
vm – virtual machine instance

Returns
(???)

BERRY_API void **be_call**(*bvm* *vm, int argc)

(???)

Note: Function call/return API

Parameters

- **vm** – virtual machine instance
- **argc** –

BERRY_API int **be_pcall**(*bvm* *vm, int argc)

(???)

Note: Function call/return API

Parameters

- **vm** – virtual machine instance
- **argc** –

Returns
(???)

BERRY_API void **be_exit**(*bvm* *vm, int status)

(???)

Note: Function call/return API

Parameters

- **vm** – virtual machine instance
- **status** –

BERRY_API void **be_raise**(*bvm* *vm, const char *except, const char *msg)

(???)

Note: exception API

Parameters

- **vm** – virtual machine instance
- **except** –
- **msg** –

BERRY_API int **be_getexcept**(*bvm* *vm, int code)

(???)

Note: exception API

Parameters

- **vm** – virtual machine instance
- **code** –

Returns

(???)

BERRY_API void **be_dumpvalue**(*bvm* *vm, int index)

(???)

Note: exception API

Parameters

- **vm** – virtual machine instance
- **index** –

BERRY_API void **be_dumpexcept**(*bvm* *vm)
(???)

Note: exception API

Parameters
vm – virtual machine instance

BERRY_API void **be_stop_iteration**(*bvm* *vm)
(???)

Note: exception API

Parameters
vm – virtual machine instance

BERRY_API void **be_regfunc**(*bvm* *vm, const char *name, *bntvfunc* f)
register a native function

Note: exception API The specific behavior of this function is related to the value of the BE_USE_PRECOMPILED_OBJECT macro (although the FFI is still available when using the compile-time construction technique, it cannot dynamically register the built-in variables. In this case, please refer to the method of registering the built-in objects.

Parameters

- **vm** – virtual machine instance
- **name** – name of the native function
- **f** – pointer of the native function

BERRY_API void **be_regclass**(*bvm* *vm, const char *name, const *bnfuncinfo* *lib)
(???)

Note: exception API

Parameters

- **vm** – virtual machine instance
- **name** –
- **lib** –

BERRY_API *bvm* ***be_vm_new**(void)
Construct a VM.

Note: VM management API

Returns
(???)

BERRY_API void **be_vm_delete**(*bvm* *vm)

Destroy a VM.

Note: VM management API

Parameters
vm – virtual machine instance

BERRY_API void **be_set_obs_hook**(*bvm* *vm, *bobshook* hook)

(???)

Note: Observability hook

Parameters

- **vm** – virtual machine instance
- **hook** –

BERRY_API void **be_set_ctype_func_hanlder**(*bvm* *vm, *bctypefunc* handler)

(???)

Note: Observability hook

Parameters

- **vm** – virtual machine instance
- **handler** –

BERRY_API *bctypefunc* **be_get_ctype_func_hanlder**(*bvm* *vm)

(???)

Note: Observability hook

Parameters
vm – virtual machine instance

Returns
(???)

BERRY_API int **be_loadbuffer**(*bvm* *vm, const char *name, const char *buffer, size_t length)

load a piece of source code from the buffer and compile it into bytecode

Note: code load API f the compilation is successful, be_loadbuffer will compile the source code into a Berry function and place it on the top of the virtual stack. If the compilation encounters an error, be_loadbuffer will

return an error value of type berrorcode (Section errorcode), and if possible, will store the specific error message string at the top of the virtual stack.

Parameters

- **vm** – virtual machine instance
- **name** – string, which is usually used to mark the source of the source code
- **buffer** – buffer for storing the source code
- **length** – length of the buffer

Returns
(???)

BERRY_API int **be_loadmode**(*bvm* *vm, const char *name, bbool islocal)

(???)

Note: code load API

Parameters

- **vm** – virtual machine instance
- **name** –
- **islocal** –

Returns
(???)

BERRY_API int **be_loadlib**(*bvm* *vm, const char *path)

(???)

Note: code load API

Parameters

- **vm** – virtual machine instance
- **path** –

Returns
(???)

BERRY_API int **be_savecode**(*bvm* *vm, const char *name)

(???)

Note: code load API

Parameters

- **vm** – virtual machine instance

- **name** –

Returns
(???)

BERRY_API void **be_module_path**(*bvm* *vm)
(???)

Note: module path list API

Parameters
vm – virtual machine instance

BERRY_API void **be_module_path_set**(*bvm* *vm, const char *path)
(???)

Note: module path list API

Parameters

- **vm** – virtual machine instance
- **path** –

BERRY_API void ***be_pushbytes**(*bvm* *vm, const void *buf, size_t len)
Push a bytes() buffer.

Note: bytes operation

Parameters

- **vm** – virtual machine instance
- **buf** – starting at position
- **len** – size

BERRY_API const void ***be_tobytes**(*bvm* *vm, int index, size_t *len)
return virtual stack as a bytes buffer

Note: bytes operation Get the value of the index position of index from the virtual stack and return it as a bytes buffer. The pointer of the buffer is returned, and the size is stored in *len (unless len is NULL). This function works only for instances of the bytes class, or returns NULL.

Parameters

- **vm** – virtual machine instance
- **index** – index from the virtual stac
- **len** – size

BERRY_API void **be_sethook**(*bvm* *vm, const char *mask)

(???)

Note: debug API

Parameters

- **vm** – virtual machine instance
- **mask** –

BERRY_API void **be_setntvhook**(*bvm* *vm, *bntvhook* hook, void *data, int mask)

(???)

Note: debug API

Parameters

- **vm** – virtual machine instance
- **hook** –
- **data** –
- **mask** –

BERRY_API void **be_writebuffer**(const char *buffer, size_t length)

implement on berry_port.c

Note: basic character IO API Output a piece of data to the standard output device, the parameter `buffer` is the first address of the output data block, and `length` is the length of the output data block. This function outputs to the `stdout` file by default. Inside the interpreter, this function is usually used as a character stream output, not a binary stream.

Parameters

- **buffer** –
- **length** –

BERRY_API char ***be_readstring**(char *buffer, size_t size)

implement on berry_port.c

This function will add the read line breaks to the read data, and each time the `be_readstring` function is called, it will continue to read from the current position. This function is only called in the implementation of the native function `input`, and the `be_readstring` function may not be implemented when it is not necessary.

Note: basic character IO API Input a piece of data from the standard input device, and read at most one row of data each time this function is called. The parameter `buffer` is the data buffer passed in by the caller, and the capacity of the buffer is `size`. This function will stop reading and return when the buffer capacity is used

up, otherwise it will return when a newline character or end of file character is read. If the function executes successfully, it will directly use the `buffer` parameter as the return value, otherwise it will return `NULL`.

Parameters

- **buffer** –
- **size** –

Returns
(???)

struct **bnfuncinfo**

#include <berry.h> native function information

Public Members

const char ***name**

name

bntvfunc **function**

function

struct **bntvmodobj**

#include <berry.h> bntvmodobj

Public Members

const char ***name**

name

int **type**

type

union *bntvmodobj*::*value* **u**

u

union **value**

#include <berry.h> < value

Public Members

bint **i**
i

breal **r**
r

bbool **b**
b

bntvfunc **f**
f

const char ***s**
s

const void ***o**
o

struct **bntvmodule**

#include <berry.h> bntvmodule

Public Members

const char ***name**
native module name

const *bntvmodobj_t* ***attrs**
native module attributes

size_t **size**
native module attribute count

const struct bmodule ***module**
const module object

struct **bhookinfo**

#include <berry.h> bhookinfo

Public Members

int **type**
: current hook type

int **line**
: current line number

const char ***source**
: source path information

const char ***func_name**
: current function name

void ***data**
: user extended data

INDEX

B

P