Zhaba-script documentation

Владимир Мацюк

2022

Contents

Introduction

Zhaba-script (Russian: ˈʐabə, жаба(toad)) - is a multi-paradigm, high-level, statically typed, interpreted or source to source compiled language, designed to be as performant as C, but as beautiful as Python.

Motivation

C is one of my favorite languages because of it's power and performance, but at the same time it's syntax is over 50 years old and doest't have lots of amazing features of modern programming languages. For example to simply loop over int range you have to use something like this: `for (int i = 0; i < n; ++i)`. Can we do better? In python you can use `for i in range(0, n)`, this is already a huge improvement, but can we do even better? Of course! Zhaba-script solution to this task looks like this: `@ i 0..n`. Another example is function pointers: the C syntax for them is very unintuitive: `int (*foo) (int, int)`. It puts variable name inside of type definition, which can be very confusing. Zhaba-script function pointer looks like this: `F<int int int> foo`. This is relatively simple examples, but them can show what zhaba-script is tries to achieve.

So, the main goal of zhaba-script is to make your programs smaller while also maintain readability and performance. To do this, zhaba-script is using C and C++ semantic concepts and bringing them with short python-like syntax.

Syntax

The zhaba-script syntax is the most different and interesting part from other programming languages and mostly resembles python, which does't use `{}` to declare blocks of code. But zhaba-script takes a step forward by removing almost all unnecessary syntax elements like `,` in some places. For example in this expression: `print(1, 2, 4)` it is obvious where commas should be so you don't need to explicitly write them. Other syntax elements like `;` or `:` are not required, but their use is put to make your code even shorter. Also most common keywords such as `if` or `return` are reduced to simple symbols, to make code shorter and even more readable. Other very big feature is ability to overload any operator and even create your onw new ones.

List of Zhaba-script features

Complier & dev environment:

- Interpretation throw bytecode
- Interpretation in web environment
- Syntax highlighter VSCode extension
- Web code editor
- Docs website
- C api

Basics:

- Basic types like int, bool or char
- Variables (and local redefinition)
- All C operators like (+ - * %)
- If, else, elif
- While loop
- C-style for loop
- Single and multi-line comments

- Functions
- Other files usage

More advanced features:

- Foreach loop
- Patten matching
- Functions overloading
- Any operator overloading
- Subscript `[]` and call `()` overload
- New operators creation
- Local functions and operators definition
- Pointers
- Function pointers
- References
- Object & array destructuring
- Custom types (classes/structs)
- Member functions
- Constructors
- Copy constructors (and implicit calls)
- Destructors (and implicit calls)
- Generic types

Standard library:

- `Vec<T>`, `Str`, `Map<K V>`, `Pair<A B>` `Option` - most common data containers
- `Result<T E>` and `panic` - for error handling
- `Rng`, `V2`, `Out`, `File` and more for other different use cases
- `c/std` - most necessary parts of C library is also ported
- And many other features!

Examples

Here is some of zhaba-script code examples:

Hello world

```
fn main: out 'Hi world'
```

FizzBuzz

```
use std

fn fizz_buzz int mx
  @ i 1..mx+1
    ? i %% 15: <'FizzBuzz'<
    | i %% 3: <'Fizz'<
    | i %% 5: <'Buzz'<
    \ <i<

fn main: fizz_buzz(20)
```

File reading

```
use file
use c/std

/** Read file in pure C */
fn fileReadC
  name := 'file-reader.zh'
  f := fopen(name 'r')

  ? f == (0 as FILEP)
    out 'File read error'
    exit(EXIT_FAILURE())

  ch := fgetc(f)
  @ ch != -1 as char
    printf('%c', va ch)
    ch = fgetc(f)

/** Read file in Zhaba-script (single line!) */
fn fileReadZH
  out open('file-reader.zh').unwrap().read()

fn main
```

```
  // fileReadC()
  fileReadZH()
```

Snake game:

```
/** Import Zhaba-script standart library */
use std
/** Import C standart library */
use c/std

fn main
  /** Initialize game data */
  sx := 20 sy := 40 tick := 100
  screen := Vec<Str>(sx ' ' * sy)
  head := Pair<int int>(10 20)
  snake := Vec<Pair<int int> >(1 head)
  rng := Rng()

  /** Lambda function to put x in range 0..mx */
  mod := (int x int mx) -> (x % mx + mx) % mx

  /** Place 2 random items */
  screen[mod(rng() sx)][mod(rng() sy)] = chr('@')
  screen[mod(rng() sx)][mod(rng() sy)] = chr('@')

  ch := chr('d')
  @ ch != chr('x')
    /** Delay and input */
    start := clock()
    @ dif(clock() start) < tick
      ? ! !kbhit()
        t := getch()
        ? t in $'wasdx': ch = t

    /** Clear screen */
    clrscr()
    /** Move head */
    ?? ch
      'w': head.a = mod(head.a - 1 sx)
      'a': head.b = mod(head.b - 1 sy)
      's': head.a = mod(head.a + 1 sx)
      'd': head.b = mod(head.b + 1 sy)

    ?? screen[head.a][head.b]
      /** Collide with self */
      'o':
```

```
      ? ch == chr('x'): out 'Exit'
      \ out 'Game over('
      ch = chr('x')
    /** Just go */
    ' ':
      {a b} := snake[0]
      screen[a][b] = chr(' ')
      snake.erase(0)
    /** Eat item */
    '@': screen[mod(rng() sx)][mod(rng() sy)] = chr('@')

  /** Add new head position to snake */
  snake += head
  screen[head.a][head.b] = chr('o')

  screen.println()
```

Links

The best way to feel zhaba-script is to use this playground, to play with examples right now! -> https://wgmlgz.github.io/zhaba/

Also there are docs with syntax highlighting -> https://wgmlgz.github.io/zhaba/?page=docs

Code in your's favorite frog programming language in your favorite IDE -> https://marketplace.visualstudio.com/items?itemName=wgmlgz.zhaba-script

Zhaba-script project

This chapter is about zhaba-script as a project and tools, which was used to build it.

Project

Zhaba-script compiler is a C++ project, which uses CMake. Is also depends on nlohmann/json for json processing and parsing and CLIUtils/CLI11 for cli arguments parsing.

Setup

If you want to run/develop zhaba-script on your own machine here is instruction for you:

- Set environment variable `zhstd` to `repo_path/std`
- If you are using VSCode you can install this extension for syntax highlighting
- To develop
- This is a CMake project, so you need to check how to set up it in your IDE
  - VSCode: I am using vscode with CMake extension, so to set up project run command `CMake: Configure`, and to add run arguments add `"cmake.debugConfig": { "args": [ your args here ] }` to settings.json
  - CLion: You probably can just open it with none or some minimal configuration
- use compiled binary to run your `.zh` files with `./zhaba <filename.zh>`
- To use zhaba-script
- Download the latest binary from releases / or use web IDE

Compiling zhaba-script code

You can run zhaba-script programs in 3 ways:

1. Translate to C and run C program.
2. Run program directly throw bytecode.
3. Use online complier without any installation and setup.

If you choose first 2 ways you can use `zhaba` binary to do that. By default `zhaba` will translate your program to C (to a temporary file `zhaba_tmp.c`) and run it with `gcc`, you can change compiler with `-c` option, if you don't want to do that you can set `-b` flag and run bytecode directly.

You can show help information with `-h` flag:

```
Zhaba-script compiler
Usage: zhaba.exe [OPTIONS] file

Positionals:
  file TEXT REQUIRED          input file to compile

Options:
  -h,--help                   Print this help message and exit
  -b,--B                      sets build target to bytecode, doesn't depend on ext
  --exp_parser_logs           shows logs from expression parser
  --show_ast                  shows ast
  --show_st                   shows syntax tree
  --show_c                    shows resulted C code
  --orig                      shows original source code
  --stack_trace               traces stack, if build target is bytecode
  --show_bytecode             shows resulted bytecode, if build target is bytecode
  --pure                      doesn't print logs
  -c,--compiler TEXT [gcc]    compiler, used to compile file, if target isn't byte
```

Basics

Identifiers

In most languages identifiers (variable names, function names, etc.) are litter followed by letter or number (`_` counted as letter). Zhaba-script identifiers can be, in addition to the usual identifiers, also a sequence of these characters: `~,.+-*\%<>=^&:;|/!#$@?`. Note, that you can choose on or another and cannot't mix them.

Keywords

Zhaba-script doesn't have regular keywords. Instead, all keywords are context depended. For example `fn` is used to declare functions and `?` for if statement, but you can use them as variables when them are not in expected context.

```
fn main
  int fn = 5 op = 4 ? = 1 @ = 7
  out(fn + op + ? + @) // 17
```

Code blocks

Unlike Rust, JS or C Zhaba-script uses both Python-style indentation to indicate blocks of code. A new block of code stats when the indentation is different from previous line, and closes when indentation of current line is less then previous line and matched indentation of some past opened block. In this case that like would go to the matching block. Here is some example of that:

```
fn test int mx
  ? mx > 50     // block 1 starts
    out 1       // block 2 starts
    out 2
    ? mx < 100  // block 3 starts
      out 3     // block 3 ends
    out 4
    ? mx > 100  // block 4 starts
      out 5     // block 4 ends
      out 6     // block 1, 2 and 4 ends
```

But also you can start new block with `:`. In this case content after will go to the next block. If the next line has same indentation as this line block will start and end at the same line.

```
fn test int mx
  ? mx > 50: out 1
    out 2
    ? mx < 100: out 3
    out 4
    ? mx > 100
      out 5
      out 6
```

If you want write multiple expressions at the same line you can use `;` to separate them.

```
fn test int mx
  ? mx > 50
    out 1
    out 2
    ? mx < 100: out 3
    out 4
    ? mx > 100: out 5; out 6
```

You can also use `;;` to close block for very short, but unreadable code.

```
fn test int mx: ?mx > 50:out 1;out 2; ?mx < 100:out 3;;out 4; ?mx>100:out 5; out 6
```

Here is some code to test examples from above:

```
fn test int mx
  ? mx > 50     // block 1 starts
    out 1       // block 2 starts
    out 2
    ? mx < 100  // block 3 starts
      out 3     // block 3 ends
    out 4
    ? mx > 100  // block 4 starts
      out 5     // block 4 ends
      out 6     // block 1, 2 and 4 ends

fn main
  // test(0)    // nothing
  test(200)  // 1 2 4 5 6
```

```
  // test(100)  // 1 2 4
  // test(60)   // 1 2 3 4
```

Implicit commas

One of the zhaba-script's interesting features is implicit commas. Implicit commas act like regular commas, but inserted implicitly into expressions between 2 tokens if this condition is met:

Left token is `)`, literal (for example string literal or int literal) or identifier. Right token is `(`, literal (for example string literal or int literal) or identifier.

Some examples:

```
a = 2   b = 3
a = 2 , b = 3
//    ^
// on left there is `2` and on right there is `b`, so `,` is inserted
// note that there is no implicit `,` between `b` and `=` because `=` is operator
```

Using other files

To use other files in your programs write `use` followed by `'filename path'`. If you want to use file at the same directory you can remove `''` and `.zh`. If there is no file with that name compiler will search in `std` directory for it.

```
use 'path/filename.zh'
use filename
```

Examples:

```
use std
use 'std.zh'
use 'da/da'
use 'da/da.zh'
```

Types

The concept of type is very important in zhaba-script. Every variable, function argument, and function return value must have a type in order to be compiled. Also, every expression (including literal values) is implicitly given a type by the compiler before it is evaluated. Some examples of types include `int` to store integer values, `f64` to store floating-point values (also known as scalar data types). You can also create your own types. The type specifies the amount of memory that will be allocated for the variable (or expression result), the kinds of values that may be stored in that variable, how those values (as bit patterns) are interpreted, and the operations that can be performed on it.

Integer numbers

For signed integers there are `i8`, `i16`, `i32` and `i64`. The number at the end of `i` represents the amount of bits allocated to one integer. Integers can also be unsigned, `u8`, `u16`, `u32` and `u64` are used for that. There is also `int` type which is alias for `i64`. To create `int`, write sequence of numbers, and if you want to specify it type, you can use `i`, `i8`, `i16`, `i32`, `i64`, `u`, `u8`, `u16`, `u32` or `u64` suffix. If you write number without any suffix the type will be `i64` (not `i32` like in most languages, for example C).

```
_u8 = 0u8 - 1u8
_u16 = 0u16 - 1u16
_u32 = 0u32 - 1u32
_u64 = 0u64 - 1u64

_i8 = 0i8 - 1i8
_i16 = 0i16 - 1i16
_i32 = 0i32 - 1i32
_i64 = 0i64 - 1i64

< 'Unsinged -1:' < < _u8 < _u16 < _u32 < _u64 <
< 'Singed -1:' < < _i8 < _i16 < _i32 < _i64 <
/**
 * Unsinged -1:
 * 255 65535 4294967295 18446744073709551615
 * Singed -1:
 * -1 -1 -1 -1
 */
```

Int literals are written in base-10 system by default, but you can also use binary or hexadecimal systems. To use them use `0x` or `0b` prefix for hexadecimal and binary systems respectively.

```
< 100 < // 100
< 0xff < // 255
< 0b101 < // 5
```

Floating point numbers

For floating point or scalar numbers zhaba-script have `f32` and `f64` types. `f32` is equivalent to `float` in C and `f64` is equivalent to `double`. You can create floating point numbers by putting `.` somewhere in number. You can also use `f`, `f32` or `f64` suffices to specify a type. If you write number without any suffix the type will be `f64` (not `f32` or `float` like in most languages, for example C).

```
< 0.5 <    // 0.5
< 1f <     // 1.0
< 1. <     // 1.0
< .1 <     // 0.1
< 1.1 <    // 1.1
< 1.1f64 < // 1.1
< 1.1f32 < // 1.1
```

Other types

Zhaba-script string literals have `str` (`Str` from std is a different type) type which is equivalent to `char*`. For single characters we have `char` (wow). There is also `bool` type with `true` and `false` literals. But zhaba-script have more short `tru` and `fls` versions.

Custom types

You can also create your own types with custom logic. More information is here TODO.

Lval & rval semantics

Like C++ zhaba-script have lvalue (lval) and rvalue(rval) concept. The difference is that you can use address-of (`&`) operator and assign value to it `a = b` with lval and cannot with rval.

These expressions are concentered lval:

- Variables
- Object members `a.b`
- Dereferenced expressions `*a`
- References

All the others expressions are considered to be rval.

Rval to lval conversion

Zhaba-script can convert rval to lvar if you pass parameter which requires reference and is not lval. This conversion is also made if you try to access member of rval object.

```
fn testRef intR a
  a += 2

fn main
  a := 3
  testRef(a)
  testRef(5)
  out a
```

Read more about rval and lval in C++ to understand this concept more here.

Type casting

To cast one type to another use `as` operator and you can only cast types with the same size.

```
exp as TypeName
```

Expressions

Expressions in Zhaba-script are very different from expressions in most other languages. They consist of operators and literals. Brackets `()` are used only to manipulate the operator's priorities and detect function calls. Here is some basic expression example:

```
// In this expression out is prefix operator and 'Hi frogs' is a str literal.
out 'Hi frogs'
```

Inline tuples

Any operator in Zhaba-script accepts inline tuple which in fact just a collection of other expressions separated by `,`. So this expression

```
//  sum
//    \
//  (1, 2)
sum(1, 2)
```

is in fact a prefix operator that actepts inline tuple of 1 and 2. But the main goal of Zhaba-script is minimizing code size so in most cases `,` are optional. And because of that, it is possible to write expression above like that:

```
sum(1 2)
```

Inline tuples can also be empty, and to create one of those just use `()`:

```
// In this case `begin` is a prefix operator that accepts empty inline tuple
//   begin
//      \
//      ()
begin()
```

Binary (infix) operators

Binary operator is an operator that accepts 2 arguments from different sides:

```
//     +
//    / \
//   2   2
2 + 2

//     ..
//    / \
//   0  10
0..10

//       *
//      / \
//     +   3
//    / \
//   1   2
1 + 2 * 3
```

But binary operators can also accept more or less than 2 arguments. In this case, they have 1 argument on the left side and others on the right size. At first, it may seem useless, but in fact all member function calls are implicitly converted to binary operators:

```
// .push_back acts like binary operator

//   .push_back
//     /  \
//   vec   1
vec.push_back(1)

//   .push_back
//     /  \
//   vec  (1 2)
vec.reverse(1 2)

//   .pop_back
//     /  \
//   vec  ()
vec.pop_back()
```

Operator precedence

Zhaba-script built-in operator precedence table:

Precedence	Operator	Description
2	`a.b`	Member access
3	`*a`	Dereference
	`&a`	Address-of
	`-a`	Unary minus
	`+a`	Unary plus
	`!a`	Logical NOT
4	`a as T`	type cast
5	`a*b` `a/b` `a%b`	Multiplication, division, and remainder
6	`a+b` `a-b`	Addition and subtraction
9	`<` `<=` `>` `>=`	For relational operators < and ≤ and > and ≥ respectively
10	`==` `!=`	For equality operators = and - respectively
14	`&&`	Logical AND
15	`||`	Logical OR
16	`=`	Assignment
17	`,` ` `	Inline tuple comma, Implicit inline tuple comma

Zhaba-script std operator precedence table:

Precedence	Operator	Description
5	`a%%b`	Is b an a divisor
9	`a..b`	range creation
16	`+=` `-=`	Compound assignment by sum and difference
	`*=` `/=` `%=`	Compound assignment by product, quotient, and remainder

Variables

Variables are essential part of most of programming languages and zhaba-script is not an exception! They allow to access memory by variable name. All variables in zhaba-script are mutable.

Zhaba-script has multiple ways to declare a variable, but the preferred one is using `:=` operator:

```
name := exp
```

This syntax is inspired by Go and variable type is inferred from `exp` type. But there are other ways for variable declaration which are more C-like:

```
TypeName name1[ = exp] [name2[ = exp] [...name3[ = exp]]]
```

If exp type is different you will get compile error:

```
int c = 37 d = 5i8

// 5 |   int c = 37 d = 5i8
//   |                 ^
//   | Types (i64 i8) for '='(init) are different
```

You can also infer type automatically with `auto`:

```
auto name1 = exp [name2 = exp [...name3 = exp]]
```

Types can be different:

```
auto e = 6i8 f = 'aboba'
```

Note, that this syntax doesn't call any constructors, so use `:=` when it's possible, but when you just need to get memory (for example in constructor definition), this is 100% valid way to do this.

```
valid := Vec<int>()
out(valid.size) // 0

Vec<int> wrong
out(wrong.size) // undefined (can be any value)
```

Object and array destructuring

You can also declare variables with destructuring using `:=` operator. Zhaba-script provides 2 ways of destructuring: by names and by indexes.

To destructure value by names wrap variable names with `{}` like so:

```
{a b} := exp
```

This syntax is equivalent to this:

```
tmp := exp // `tmp` isn't public name and can be different
a := tmp.a
b := tmp.b
```

But when dealing with arrays destructuring by indexes might be more useful. To do that use `[]`.

```
[a1 a2 a3] := exp
```

This syntax is equivalent to this:

```
tmp := exp // `tmp` isn't public name and can be different
a1 := tmp[0]
a2 := tmp[1]
a3 := tmp[2]
```

Note that with `{}` you must use valid variables identificators, but `[]` allows to put any expression inside them, so you can nest `{}` and `[]` inside `[]`:

```
[a {b c} d] := exp
```

if, else, elif statements

In zhaba-script you can use `?` to create if statements, `|` for elif(else if) and `\` for else.

Syntax

```
         ? <condition> <if block>
optional | <condition> <else if block>
optional | <condition> <else if block>
         | ...
optional \ <else block>
```

Examples:

Regular style:

```
? 2 + 2 == 4
  out 'Seems'
  out 'legit'
```

One liner:

```
? 666 > 2: out '666 is greater then 2'
```

Nested also works:

```
? 1 < 2
  ? 2 > 1
    out 'Double check'
```

if else

```
? 2 + 2 == 4: out 'cool!'
\ out 'how?'
```

if elseif elseif else

```
? i %% 15: out 'FizzBuzz'
| i %% 3: out 'Fizz'
| i %% 5: out 'Buzz'
\ out i
```

other variant

```
? i %% 15
  out 'FizzBuzz'
| i %% 3
  out 'Fizz'
| i %% 5
  out 'Buzz'
\ out i
```

Loops

Zhaba-script has 3 types of loops: while, for and foreach. To create a loop, use the `@` symbol and after that write your loop expression. Loop type is defined by a number of arguments, that have been passed to the loop: 1-while, 2-foreach, 3-for.

Syntax

```
@ <exp>
 <body>
```

While

While loop accepts 1 argument and runs while its body while the condition is still true.

Examples

```
@ i < 5: out 'infinite loop!!!'
```

```
@ i < 5
  out 'hi'
```

For

For loop is similar to the C for loop and accepts 3 argument: init, condition and loop expression.

Examples

```
@ i:=0 i<100 i+=1
  out i
```

The above syntax produces code equivalent to:

```
i:=0
@ i<100
  out i
  i+=1
```

Foreach

Foreach loop executes a for loop over a range and accepts 2 arguments: a variable name and range expression. Variable name must be identifier and range expression must have defined `iter` prefix operator, which produced expression that must have `.begin()` and `.end()` methods. Value provided by `.begin()` must be able to be compared with `.end()` value by `!=` operator and also must be able to be incremented by prefix `++` operator. Produced code is equivalent to this:

```
__range := iter(<range-expression>)
__cur := __range.begin()
__end := __range.end()
@ __cur != __end
  exp := *__cur
  <body>
  ++__cur
```

Examples

```
use 'range.zh'
@ i 0..10
  out i
```

```
@ i some_vector
  out *i
```

Functions

Functions are the core of programming in zhaba-script. To define your function use the `fn` keyword followed by return type (don't write it if the function doesn't anything), then function_name and finally set of arguments which are pairs of type and name. Functions can be declared at the global level, but also in local scope (like variables). To return something from a function use `<<<` and to call if write its name followed by `()`.

Syntax

Declaration:

```
fn [return-type(optional)] function-name [type name [type name [...type name]]]
  code-block
```

Call with args:

```
 function_name(args)
```

Call without args:

```
 function_name()
```

Examples

```
/** int return type 2 args */
fn int plus int a int b
  <<< a + b // to return use <<<

/** char return type 1 argument */
fn char aschar str s: <<< *(s as charP) // body at the same line

/** No return type 1 argument */
fn fizz_buzz int mx
  auto r = 0..mx
  @ r.nxt()
    auto i = r.cur
    ? i %% 3 && i %% 5: out 'FizzBuzz'
```

```
    | i %% 3: out 'Fizz'
    | i %% 5: out 'Buzz'
    \ out i

/** No return type 1 argument */
fn func1 int mx
  fn func2: <<< 5
  < (func2()  + func2()) <

/** No return type and no args */
fn main
  fizz_buzz(50) // call example
```

Return emergency exit

If your function has return type, but it reached it's end without returning anything zhaba-script will print `reached function end without returning anything <function name>` and then exit. This will happen not only in interpretation mode but in C translated program too.

```
fn int da
  res := 1 + 4
  // <<< res
fn main
  da() // reached function end without returning anything
```

Pointers

While Zhaba-script is designed for fast development, but the other goal is to do this without sacrificing the performance. Unlike JS or python there is no garbage collector, and you need to manually manage memory like in C or C++. A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Under the hood pointer is just a 64-bit integer or `i64` in zhaba-script.

Declaring pointers

Unlike C, in zhaba-script you can append `P` to the type to declare a pointer.

```
int var
intP ptr
```

In this example var is variable and ptr is a pointer.

Address-of operator `&`

The address of a variable can be obtained by preceding the name of a variable with an ampersand sign '&', known as address-of operator. You can use address-of operator only with lval expression, because getting address of rval expression can result in undefined behavior. Read more about rval/lval TODO here. For example:

```
ptr = &var;
```

This would assign the address of variable `var` to `ptr`; by preceding the name of the variable `var` with the address-of operator `&`, we are no longer assigning the content of the variable itself to foo, but its address.

Dereference operator `*`

As just seen, a variable which stores the address of another variable is called a pointer. Pointers are said to "point to" the variable whose address they store.

An interesting property of pointers is that they can be used to access the variable they point to directly. This is done by preceding the pointer name with the dereference operator `*`. The operator itself can be read as "value pointed to by".

Therefore, following with the values of the previous example, the following statement:

```
baz = *ptr;
```

This could be read as: "`baz` equal to value pointed to by `ptr`".

Heap

The heap is a large pool of memory that can be used dynamically. This is memory that is not automatically managed – you have to explicitly allocate (using malloc), and deallocate (e.g. free) the memory.

`malloc`

Declaration:

```
lop malloc int size
```

Description:

`malloc` is a built in operator that allocates the requested memory and returns a pointer to it.

`free`

Declaration:

```
lop free int ptr
```

Description:

`free` is a built in operator that deallocates the space previously allocated by `malloc`.

Heap in interpreter

Zhaba-script virtual machine tries to make work with pointers as safe as possible, and to achieve that it does some slow checks.

Pointer arithmetics

When dealing with continuos memory like array or string you can shift pointer forward and backward with `+` and `-`. Using this operators will shift pointer at the size of the underlying type multiplied by shift size.

Example:

```
size := 10
p := malloc(sizeof(int) * size) as intP
out 'ptr + x'
@ i 0..size: *(p + i) = i
@ i 0..size: out(*(p + i))

out 'x + ptr'
@ i 0..size: *(i + p) = i
@ i 0..size: out(*(i + p))

out 'ptr - x'
t := p + 5
t = t - 2
out(*t)
```

Custom data types

Zhaba-script custom data types are similar to rust structs. To define a data type, enter the keyword `type` and name the entire data type. Then, inside the next block, define the type and names of the pieces of data, which are called members. To access member use `.` operator.

Syntax

```
type TypeName
  MemberType1 name [name[ name[ ...name]]]
  MemberType2 name [name[ name[ ...name]]]
  MemberType3 name [name[ name[ ...name]]]
  ...
```

Examples

This is an example of Vector data structure.

```
type VecInt
  int size
  int capacity
  intP head

fn main
  VecInt v
  < v.size <
```

Code style

In zhaba-script code style, you should name your custom types in `PascalCase` and members in `snake_case`.

Member functions

Zhaba-script support member functions call. To define some of these functions you can use the keyword `impl` and then enter the type for which you are writing implementation. Note that this type can be any type, even primitive (like `int`). Inside this block just write regular functions. To call member function write `object.function_name(args)`

Syntax

```
impl TypeName
  fn func-header
    func-body
  fn func-header
    func-body
  fn func-header
    func-body
  ...
```

To access the object for which you are writing implementation you can use `slf` who is the first argument that is provided implicitly to your function. `slf` has the type of a pointer to a type written after `impl`. Continuing the example about the Vector, this is some example of `impl` and `slf` usage.

```
impl VecInt // implementation block for VecInt
  fn intP begin
    <<< slf.head

  fn intP end
    <<< slf.atP(slf.size)

  fn push_back int val
    ? slf.size == slf.capacity: slf.double_capacity()
    *slf.atP(slf.size) = val
    &slf.size += 1
```

Incomplete types

When declaring type you sometimes need to reference itself in his body (in structures like trees for example). However it is not possible because this would mean that type need to include infinitely many copies of it.

```
.-type--------------------.
| .--type------------.     |
| | .-type-----.     |     |
| | |  ...type |     |     |
`--------------------------'
```

To solve this problem you can use pointer or reference to type itself, because it's size is known and fixed.

```
.--------type--------.
| | ptr |            |
`--------------------'
```

So, if you try to reference a type in it self, you will get `Incomplete type error`, but referencing pointer or reference is perfectly fine.

```
type Node
  NodeP lhs // Ok
  Node rhs /** error :( -> test.zh:5:2
            *  5 |   Node rhs
            *    |   ^^^^^
            *    | Incomplete type is not allowed here (you probably need to make
            */
```

Generic types

Zhaba-script supports generic types. Generics is the idea to allow type (int, str, ... etc and user-defined types) to be a parameter to types. For example `Vec<T>` allows to store dynamic array of any type without need to make new implementations every time. Note, that you don't have to name your type as `T`. Generics are implemented by replacing types, so, for example if you try so sort types for which comparison operator is not defined, compiler will give you an error.

Syntax:

```
type TypeName [TypeName1 [TypeName2 [TypeName3]]]
  MemberType1 [name1 [name2 [...name3]]]
  MemberType2 [name1 [name2 [...name3]]]
  MemberType3 [name1 [name2 [...name3]]]
  ...
```

Examples:

```
type Vec T
  int size
  int capacity
  TP head
```

```
type Tp3 T1 T2 T3
  T1 v1
  T2 v2
  T3 v3
```

To use generic types use following syntax:

```
TypeName<T1[ T2[ ...T3]]>
```

Examples:

```
Vec<char> data

Tp3<char i32 bool> data
```

Note

When you are using generic type inside another generic type you need to separate closing `>`.

```
Vec<Vec<char>> data // wrong
Vec<Vec<char> > data // correct
```

Generic types and member functions

Like a normal types, generic types can also have member functions. To define them create `impl` block followed by the type name, but without generic parameters.

Syntax:

```
impl TypeName
  fn func-header
    func-body
  fn func-header
    func-body
  fn func-header
    func-body
  ...
```

Examples:

```
impl Vec
  fn free_mem
    ? ! !(slf.head as int): free(slf.head as int)
    slf.size = 0
```

```
    slf.capacity = 0
    slf.head = 0 as TP
```

Generic types and operators overloading

Generic `impl` block allows you not only to create member function, but also to overload prefix, postfix or binary operators. They would behave like other overloaded operators and would not have implicit `slf` argument. The main difference and benefit is that you can use generic parameters in them.

```
impl Vec
  op 17 += Vec<T>R slf T val: slf.push_back(val)

  op 17 += Vec<T>R slf Vec<T> other
    @ i other: slf.push_back(*i)

  lop put Vec<T>R slf: slf.print()
  lop out Vec<T>R slf: slf.print(); out ''

  ...
```

References

A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable. Internally references are converted to pointers with `&` and `*` operators when needed.

Creating References in zhaba-script

Think of a variable name as a label attached to the variable's location in memory. You can then think of a reference as a second label attached to that memory location. Therefore, you can access the contents of the variable through either the original variable name or the reference. But zhaba-script doesn't allows to store them directly, only returning them or using them as function arguments. To create reference use `R` at the end of variable name.

Syntax

```
TypeNameR
```

Passing references as arguments

References are mostly used when you want to use some variable without copping it. Here is example of this:

```
use std

fn add_copy int a
  ++a
fn add_ref intR a
  ++a

fn main
  t := 5
  < t < // 5
  add_ref(t)
  < t < // 6
```

```
  add_copy(t)
  < t < // 6
```

Returning references

You can also return reference and then modify it's content. For example `Vec<T>` returns a reference when you call `[]` or `.at()`.

```
/** From std/vec.zh */
impl Vec
  fn TP atP int pos: <<< (slf.head + pos) as TP
  fn TR at int pos: <<< *slf.atP(pos)

fn main
  v := Vec<int>(5)
  v[3] = 4
  < v < // [0 0 0 4 0]
```

References and lval & rval semantics

References are referring to some valid variable or object in memory. Because of that they cannot be created from rval expression. Read more about rval/lval TODO here.

```
use std

fn add_ref intR a
  ++a

fn main
  a := 5
  add_ref(a) // Ok
  add_ref(5 + 5)
  /**
   *  error :( -> test.zh:9:12
   *  9 |   add_ref(5 + 5)
   *    |             ^
   *    | Expression must be lval to be able pass by reference
   */
```

Objects lifetime

A lifetime is a construct the compiler uses to ensure create and destroy object. Specifically, a variable's lifetime begins when it is created and ends when it is destroyed. Later, more specific mechanisms and the time of creation and destruction of objects will be described.

Example

```
fn main
  i := 0  // lifetime for `v` starts. ------.
  > i                                   //  |
                                        //  |
  ? i == 4:                             //  |
    a := 7  // `a` lifetime starts. -------.|
    < (a + i) <                         // ||
    // `a` lifetime ends. -----------------'|
                                        //  |
  // `i` lifetime ends. -------------------'
```

Constructor

By default, newly created object is just contains uninitialized space for it's members, but you can initialize it with special function called `ctor` (constructor). Constructor is just a function with any args that returns 1 object of it's constructed type. To create constructor use `impl` block. To call constructor use type name as function and call it

```
impl Vec
  fn ctor int size // constructor for `Vec` that takes `int size` as argument
    Vec<T> slf
    slf.head = 0 as TP
    slf.size = 0
    slf.capacity = 0
    @ slf.capacity < size: slf.double_capacity()
    slf.size = size
    slf.capacity = size
    <<< slf // don't forget to return!

fn main
```

```
  v := Vec<int>(4) // calling `Vec<int>` constructor with `int size` = 4
  < v <
```

Note, that constructors does't have implicit `slf` argument so, for example you can use them like this:

```
impl Vec
  fn ctor: <<< Vec<T>(0)
```

Copy constructor

Copy constructor is a special constructor that used when compiler needs a copy of some other object, but you can use them by yourself. Copy constructors must have only 1 argument of reference to returning type.

Example

```
impl Vec
  fn ctor Vec<T>R other
    out 'copy!'
    slf := Vec<T>(other.size)
    @ i 0..other.size: slf.at(i) = other.at(i)
    <<< slf
```

When copy constructor is called?

- Variable declaration with `:=`
- Assignment with `=`
- Passing arguments fo functions without `P` or `R` (by copy)

Destructor

A `dtor` (destructor) is a special member function that is called when the lifetime of an object ends. The purpose of the destructor is to free the resources that the object may have acquired during its lifetime

```
impl Vec
  fn dtor
    ? ! !(slf.head as int): free(slf.head as int)
    slf.size = 0
    slf.capacity = 0
    slf.head = 0 as TP
```

When destructor is called?

- Local scope variables at scope end
- All accessible local variables and function args when function returns except return value
- When expression return value is not used

Return and destruction

As follows from the description above local object is not destructed and not copied when it is used as function return value, but it can be reallocated in stack memory. So it is important to not store references or pointers to it's members or itself, but pointers to heap are 100% valid.

Operators overloading

Zhaba-script allows you to overload any prefix, postfix or infix operator. Remember about correct identifiers. You can read more about operators and expressions here. And here - C++ operators precedence table for reference.

Syntax

Declaration:

```
// Infix (binary operators):
// put operator precedence instead of 5
op 5 [return-type(optional)] operator-name [type name [type name [...type name]]]
  code-block

// Prefix operators:
lop [return-type(optional)] operator-name [type name [type name [...type name]]]
  code-block

// Suffix operators:
rop [return-type(optional)] operator-name [type name [type name [...type name]]]
  code-block
```

Examples:

```
/** From operators.zh */
op 5  bool %% int a int b: <<< !(a % b);

op 17 += intR a int b: a = a + b
op 17 -= intR a int b: a = a - b
op 17 /= intR a int b: a = a / b
op 17 %= intR a int b: a = a % b
op 17 *= intR a int b: a = a * b

lop ++ intR val: val += 1
lop -- intR val: val -= 1
lop ++ u8R val: val = val + 1u8
lop -- u8R val: val = val - 1u8


/** From range.zh */
op 9 Range .. int begin int end
```

```
  Range slf
  slf.begin = begin
  slf.end = end
  <<< slf

rop Range .. int begin: <<< begin..9223372036854775807

lop Range .. int end: <<< -9223372036854775807..end

op 10 bool == int i Range r
  ? r.begin < r.end: <<< r.begin <= i && i < r.end
  \ <<< r.end <= i && i < r.begin
```

Advanced overloading

Zhaba-script also allows you to overload Function call `()` and Subscript `[]` operators. To do that you need to define member functions named `call` and `sub` respectively.

Note, that you can pass any number of arguments, even zero.

Syntax

```
impl <Type>
  fn <ReturnType> sub <args>
    <code_block>
  fn <ReturnType> call <args>
    <code_block>
```

Examples:

```
/** From vec.zh */

/** Vector subscript - v[id] */
impl Vec
  fn TR sub int id
    ? id < 0: id += slf.size
    <<< slf.at(id)

/** Generate array from Range */
impl Range
  fn Vec<int> call
    res := Vec<int>()
```

```
    @ i, *slf: res += i
    <<< res
```

Pattern matching

Zhaba-script has pattern matching feature which allows to compare some value with a number of different values and it is something between C `switch-case` and rust `match`. If you want to match any expression, use `_`, this branch will allays be matched.

Syntax

```
?? <test-exp>
  <match1>
    <block>
  <match2>
    <block>
  <match3>
    <block>
```

Example

```
use std

fn main
  a := 4

  ?? a
    2: < '2' <
    0..2: < '0..2' <
    6..8: < '6..8' <
    -1..5: < '-1..5' <
    _: < '_' <
```

How it works?

Pattern matching statement converted to if statement like so:

```
switch := <test-exp>

? switch == <match1>
  <block1>
| switch == <match2>
```

```
  <block2>
| switch == <match3>
  <block3>
```

Because pattern matching uses `==` operator, you can overload it to match your custom types. For example you can match `Range` with `int` (declared at std/range):

```
/** int in range */
op 10 bool == int i Range r
  ? r.begin < r.end: <<< r.begin <= i && i < r.end
  \ <<< r.end <= i && i < r.begin
```

Advanced functions

Zhaba-script support first-class functions, and that means that you can pass functions as arguments to other functions, returning them as the values from other functions, and assigning them to variables or storing them in data structures. Zhaba-script has anonymous functions (function literals) as well.

Functions type

All functions have type of `F<...>`, inside `< >` there is return type and types of function arguments.

```
F<ReturnType [Arg1Type [Arg2Type [...]]]>
```

Referencing function

To do something with your functions you need to reference it. Just write function name like it is a variable. In function is overloaded, the last overload will be referenced.

Storing function:

```
fn int test int a int b: <<< a + b

fn main
  f := test
  out f(2 4)
```

Passing function as argument:

```
fn int test F<int int int> f
  <<< f(1 6)

fn int sum int a int b
  <<< a + b

fn main
  out test(sum)
```

Lambda function expressions

An lambda function expression is a compact alternative to a traditional function expression, but is limited and can't be used in all situations.

There are differences between lambda functions and traditional functions, as well as some limitations:

Lambda function return type is automatically inferred and cannot be explicitly set. Lambda function body can only be an expression, so, for example, ifs or loops cannot be inside lambda function body.

Syntax

To create lambda function write arguments list wrapped in parentheses followed by `->` and expression.

```
(Type1 arg1[, Type2 arg2[, ...]]) -> expression
```

Examples

```
/** Storing lambda */
f := (intR i) -> i*i // f type is `F<int intR>`

out iota(0 10).filter((intR i) -> i %% 2).map(f)
/**   Inline lambda   ^^^^^^^^^^^^^^^^^^  */

sum := (int a int b) -> a + b
out sum(1 5)
```

Standard library

Some languages like python or javascript declare their functionality in their core, and others like C++ do this in standard library. For example `Map` in javascript is available at any moment without including or importing anything, and in C++ you need to `#include <map>` to use `std::map`. Zhaba-script takes C++ approach, so standard library is very important and contains many of cool zhaba-script features.

Using standard library

If you want to use all standard library at once, write `use std` at the top of your file, but if you only need one or two of it's features, it is good idea to use only needed modules, like `use map` or `use vec`.

Standard library content

File	Description
std/avl.zh	`AVLTree<T>`: avl binary search tree
std/complex.zh	`V2`: complex numbers
std/err.zh	`Err`: `Str` error wrapper
std/file.zh	filesystem
std/frog.zh	cool frog image
std/pair.zh	`Pair<A B>`: pair container
std/result.zh	`Result<T E>`: recoverable errors container type
std/map.zh	`Map<K V>`: wrapper for avl tree
std/operators.zh	some advanced operators like `%%` or `+=`
std/io.zh	overloaded `<` and `>` operators for input/output
std/range.zh	`Range`: int range
std/rng.zh	`Rng`: random number generator
std/str.zh	`Str`: dynamic string

File	Description
std/util.zh	utilities functions
std/vec.zh	`Vec<T>`: dynamic generic array
std/std.zh	Includes all standard library

Range

```
type Range: int begin end
```

A (half-open) range bounded inclusively below and exclusively above [start..end).

The range `start..end` contains all values with `start <= x < end`. It is empty if `start >= end`. To create range which includes `end` you can use `..=` operator.

Creating range

Range can be created using `..` or `..=` operator with it's prefix, postfix or infix overloads. Prefix and postfix operators create infinite (limited by i64 min/max values) ranges:

```
10..  // 10..9223372036854775807
..40  // -9223372036854775807..40
3..10 // 3..10
3..=10 // 3..11
```

Usage

Range can be used in foreach loops:

```
@ i 5..10 : < i
// 5 6 7 8 9
```

Range can be used in patter matching:

```
t := 5
?? t
  10.. : out '10..'
  15.. : out '15..'
  ..3: out '..3'
  3..10: out '3..10'
  0..2: out '0..2'
```

```
  5: out 5
  _: out 'any'

// 3..10
```

Range can be also used to index array, more about that at next chapter.

P.S

Inspired by Rust and Python!

Input and output

By default zhaba-script input system has `out` and `put` prefix operators for basic types. `out` will print your expression followed by newline character and `put` will just print without newline.

```
out 'hi'
out 2
put(5 - 2)
```

For input there are several functions, when called they will return inputted type:

```
in_i8()
in_i16()
in_i32()
in_i64()
in_u8()
in_u16()
in_u32()
out_u64()
in_u64()
in_str()
in_char()
in_bool()
in_f32()
in_f64()
```

However zhaba-script has different, more handy way for input and output. To achieve this there are overloaded `<` and `>` operators. To use them write `use std` at the top of your file.

Here is usage example:

```
< 1 // outputs ' ' at end
< 1
< 1 < 2 < 3 < 4 <  // you can chain `<`, last `<` will print newline character

v := 0
> v // user typed '54' for example
< v < // input items must be lval or ref

/** output:
```

```
1 1 1 2 3 4
54

*/
```

Iterators

An Iterator is an object that can be used to loop through collections, like `Vec<T>` or `Map<K V>`. It is called an "iterator" because "iterating" is the technical term for looping.

Iterators isn't a part of zhaba-script, but they are rather implemented throw standard library.

Using iterators

When using iterators we have begin and end, begin is included in collection and end isn't.

Zhaba-script iterators has this convention:

First, get iterator range with `iter()` function. Then, get begin and end iterators from it. And finally increment iterator using prefix `++` operator, while you didn't reach the end. You can check if iterator reached end using `!=` binary operator. To get content of iterator use overloaded prefix operator `*`.

Here is example if doing that:

```
use std

fn main
  v := iota(0 10)

  iters := iter(v)
  begin := iters.begin()
  end := iters.end()
  cur := begin
  @ cur != end
    val := *cur
    out val
    ++cur
```

Iterators + foreach

Using iterators like that isn't cool, so zhaba-script has foreach loop for that:

```
use std

fn main
  v := iota(0 10)

  @ cur v
    out cur
```

`Vec<T>` type

Definition

```
type Vec T
  int size
  int cap
  TP head
```

The elements are stored contiguously, which means that elements can be accessed not only through iterators, but also using offsets to regular pointers to elements. This means that a pointer to an element of a vector may be passed to any function that expects a pointer to an element of an array.

The storage of the vector is handled automatically, being expanded and contracted as needed. Vectors usually occupy more space than static arrays, because more memory is allocated to handle future growth. This way a vector does not need to reallocate each time an element is inserted, but only when the additional memory is exhausted.

`Vec` member functions

Functions	Description
`fn ctor int size`	Constructor, creates vector with given size
`fn ctor int size T default`	Constructor, creates vector with given size initialized with given value
`fn ctor`	Constructor, creates empty vector
`fn ctor Vec<T>R other`	Copy constructor
`fn dtor`	Destructor
`fn TP atP int pos`	Access pointer to specified element
`fn TR at int pos`	Access specified element
`fn print`	Prints vector content in this form: `[0 1 3 4]`
`fn println`	

Functions	Description
	Prints every vector element from new line, doesn't print `[]`
`fn double_cap`	Doubles vector's capacity
`fn push_back T val`	Adds an element to the end
`fn pop_back`	Removes the last element
`fn VecIter<T> begin`	Returns an iterator to the beginning
`fn VecIter<T> end`	Returns an iterator to the end
`lop VecIterRange<T> iter Vec<T>R slf`	Returns iterator range
`fn TR front`	Access the first element
`fn TR back`	Access the last element
`op 17 += Vec<T>R slf T val`	Append single item to vector
`op 17 += Vec<T>R slf Vec<T>R other`	Append other vector item to given vector
`op 2 Vec<T> ,, T a T b`	Short syntax for creating vector from multiple items: `0,,1,,2,,3`
`op 2 Vec<T>R ,, Vec<T>R v T a`	Short syntax for creating vector from multiple items: `0,,1,,2,,3`
`lop out Vec<T>R slf`	Outputs vector with '\n'
`lop put Vec<T>R slf`	Outputs vector
`rop Out < Vec<T>R i`	Outputs vector
`lop Out < Vec<T>R i`	Outputs vector
`op 9 Out < Out o Vec<T>R i`	Outputs vector
`fn TR sub int id`	Access specified element
`fn Vec<T> sub int begin int end`	Get subvector from indexes [begin, end)

Functions	Description
`fn Vec<T> sub Range<int> r`	Get subvector from indexes [begin, end), from range
`fn Vec<T> sub`	Get subvector from 0..size
`op 9 bool < Vec<T>R a Vec<T>R b`	Lexicographically compares the values in the vector
`lop TP partition TP lo TP hi`	Partitions vector
`lop qsort TP lhs TP rhs`	Qsort algorithm
`fn sort`	Sort vector's content
`lop sort Vec<T>R slf`	Sort vector's content
`fn Vec<T> map F<T TR> f`	Creates a new vector with all elements that pass the provided function
`fn Vec<T> filter F<bool TR> f`	Creates a new vector populated with the results of calling a provided function on every element

`Vec` non-member functions

Functions	Description
`fn Vec<int> iota int begin int end`	Fills the range [begin end) with sequentially increasing values
`fn<T O> Vec<O> mp Vec<T>R v F<O TR> f`	Pseudo-generic function, same as `map` call: `mp<T O>(vec fn)`

Iterators

Vector also has iterators, which can be used as random access iterators. After `push_back` iterators may be invalid.

```
type VecIter T: TP ptr
```

```
type VecIterRange T: VecIter<T> begin end
```

`VecIter<T>` member functions

Functions	Description
`fn ctor TP ptr:`	Constructor
`lop ++ VecIter<T>R slf`	Next iterator
`op 6 VecIter<T> + VecIter<T> slf int i`	Sum 2 pointers
`op 10 bool != VecIter<T> a VecIter<T> b`	Check unequal
`lop TR * VecIter<T> slf`	Dereference

`VecIterRange<T>` member functions

Functions	Description
`fn ctor VecIter<T> begin VecIter<T> end`	Constructor
`fn VecIter<T> begin`	`begin` getter
`fn VecIter<T> end`	`end` getter

`Str` type

Definition

```
type Str
  Vec<char> data
  int size
```

Strings are objects that represent sequences of characters ending with null ('\0').

The standard string provides support for such objects with an interface similar to `Vec<T>`, but adding features specifically designed to operate with strings of single-byte characters.

Note that this class handles bytes independently of the encoding used: If used to handle sequences of multi-byte or variable-length characters (such as UTF-8), all members of this class (such as length or size), as well as its iterators, will still operate in terms of bytes (not actual encoded characters).

Iterators

`Str` rely on `Vec<T>`, so it uses it's iterators.

`str` member functions

Functions	Description
`fn char sub int pos`	
`fn ctor`	

`str` non-member functions

Functions	Description
`op 10 bool == char ch str s`	
`lop char chr str s`	

Functions	Description
`fn int len str s`	Return s

`Str` member functions

Functions	Description
`fn ctor`	Default constructor
`fn ctor str s`	Constructor from `str`
`fn ctor StrR s`	Copy constructor
`fn dtor`	Destructor
`fn str cstr`	Returns `str`
`lop VecIterRange<char> iter StrR s`	Returns iterator range
`fn charR at int pos`	Get character in string
`fn charR sub int pos`	Get character in string
`fn charP atP int pos`	Get pointer to character in string
`fn push_back char ch`	Append character to string
`fn pop_back`	Delete last character
`fn sort`	Sorts string

`Str` non-member functions

Functions	Description
`lop out StrR s`	Outputs string with '\n'
`lop put StrR s`	Outputs string
`op 17 += StrR a char ch`	Append to string
`op 17 += StrR a StrR b`	Append to string
`op 17 += StrR a str b`	Append to string

Functions	Description
`op 6 Str + StrR a StrR b`	Concatenate strings
`op 6 Str + str a str b`	Concatenate strings
`rop Out < StrR i`	Outputs string
`lop Out < StrR i`	Outputs string
`op 9 Out < Out o StrR i`	Outputs string
`op 9 bool < StrR a StrR b`	Compare strings
`op 5 Str * str s int i`	Python-like string multiplication
`lop Str $ str s`	Short syntax for `str` to `Str` conversion
`lop up StrR s`	Converts string to uppercase
`lop low StrR s`	Converts string to lowercase
`lop VecIterRange<char> iter str s`	Returns iterator range

Avl Tree

Definition

```
type AVLNode T
  int h
  T val
  AVLNode<T>P lhs rhs par
```

```
type AVLTree T: AVLNode<T>P root
```

Avl tree in zhaba-script standard library is an associative container that contains a sorted set of unique objects of type `T`. Sorting is done using the key comparison operator `<`. Search, removal, and insertion operations have logarithmic complexity. `AVLTree<T>` is very similar to `std::set` in C++.

Type `AVLTree<T>` is a wrapper for `AVLNode<T>` and `AVLNode<T>` functions aren't presented here, because they are concentered to be private.

`AVLTree<T>` member functions

Functions	Description
`fn ctor`	Default constructor
`fn dtor`	Destructor
`fn AVLIter<T> begin`	Returns an iterator to the beginning
`fn AVLIter<T> end`	Returns an iterator to the end
`lop AVLIterRange<T> iter AVLTree<T>R slf`	Returns iterator range
`fn insert T val`	Inserts element
`fn TR sub T val`	Access specified element
`op 17 += AVLTree<T>R slf T val`	Inserts single item

Functions	Description
`fn Vec<T> call`	Converts tree content to vector via inorder traversal
`fn show`	Pretty prints tree structure

Iterators

`AVLTree<T>` iterators provide inorder access to tree elements and aren't random access, so you can't jump any number of elements, only one at a time.

```
type AVLIter T: AVLNode<T>P next
```

```
type AVLIterRange T: AVLIter<T> begin end
```

`AVLIter<T>` member functions

Functions	Description
`fn ctor AVLNode<T>P root`	Constructor
`lop AVLNode<T>P ++ AVLIter<T>R slf`	Next iterator
`op 10 bool != AVLIter<T> a AVLIter<T> b`	Check unequal
`lop TR * AVLIter<T> it`	Dereference

`AVLIterRange<T>` member functions

Functions	Description
`fn ctor AVLIter<T> begin AVLIter<T> end`	Constructor
`fn AVLIter<T> begin`	`begin` getter
`fn AVLIter<T> end`	`end` getter

Map

Definition

```
type MapNode K V
  K key
  V val
```

```
type Map K V
  AVLTree<MapNode<K V> > tree
```

`Map<K V>` is a sorted associative container that contains key-value pairs with unique keys. Keys are sorted by using the comparison operator `<`. Search, removal, and insertion operations have logarithmic complexity. Maps wraps `AVLTree<T>` and uses `MapNode<K V>` for data storage. Map iterators are also `AVLTree<MapNode<K V> >` iterators.

`MapNode<K V>` member functions

Functions	Description
`fn ctor`	Default constructor
`fn ctor KR key VR val`	Constructor
`fn dtor`	Destructor
`op 4 bool < MapNode<K V>R a MapNode<K V>R b`	Comparator compares by `key` value
`lop put MapNode<K V>R slf`	Outputs content with '\n`
`lop out MapNode<K V>R node`	Outputs content
`rop Out < MapNode<K V>R i`	Outputs content
`lop Out < MapNode<K V>R i`	Outputs content
`op 9 Out < Out o MapNode<K V>R i`	Outputs content

`Map<K V>` member functions

Functions	Description
`fn ctor`	Default constructor
`fn dtor`	Destructor
`fn VR sub KR key`	Access specified element by `key`
`fn insert KR key VR val`	Inserts element
`fn Vec<MapNode<K V> > call`	Converts content to vector
`fn print`	Prints content
`lop put Map<K V>R slf`	Outputs content
`lop out Map<K V>R slf`	Outputs content
`rop Out < Map<K V>R i`	Outputs content
`lop Out < Map<K V>R i`	Outputs content
`op 9 Out < Out o Map<K V>R i`	Outputs content
`lop AVLIterRange<MapNode<K V> > iter Map<K V>R m`	Returns iterator range

Error handling

Errors are a fact of life in software, so Zhaba-script has a number of features for handling situations in which something goes wrong. Zhaba-script takes similar approach to Rust is this topic.

Zhaba-script groups errors into two major categories: recoverable and unrecoverable errors. For a recoverable error, such as a file not found error, we most likely just want to report the problem to the user and retry the operation. Unrecoverable errors are always symptoms of bugs, and so we want to immediately stop the program.

Most languages don’t distinguish between these two kinds of errors and handle both in the same way, using mechanisms such as exceptions. Zhaba-script doesn’t have exceptions. Instead, it has the type `Result<T E>` for recoverable errors and the `panic` function that stops execution when the program encounters an unrecoverable error. Note, that you can can't use this in bytecode, because of dependence to stdlib.h. To use `panic` import `c/std` or `c/stdlib.c` (more about C api in next chapter) and to use `Result` import `result`.

Recoverable errors with `Result<T E>`

```
type Result T E
  bool is_err
  T val
  E err
```

Essentially this type allows you to combine 2 types: good type `T`, in which case all went OK, and bad type `E`, in which you might want to put some info about your error. `Result` is most handy for function return value. Here is an example from zhaba-script standard library (more about `Err` type later):

```
fn Result<File Err> open Str s Str mode
  f := fopen(s.cstr() mode.cstr())
  ? f != (0 as FILEP): <<< Result<File Err>(File(f))
  <<< Result<File Err>(Err('canno\'t open ' + s))
```

This function tries to open file using C api. If C function `fopen` returned valid pointer, we return `Result<File Err>(File(f))`, but if pointer is null, we return `Result<File Err>(Err('canno\'t open ' + s))`.

And then, to use file you can do look at `is_err` value:

```
f := open('file.txt')
? f.is_err: out 'everything is bad'
\ out f.val.read()
```

Another way to process recoverable errors is with `unwrap` function:

```
f := open('file.txt').unwrap()
out f.read()
```

This function will return good value, and otherwise will raise unrecoverable error.

Simple errors with `Err` type

```
type Err: Str val
```

In most cases, when error is accrued, you don't need to have some fancy information about the error itself and for that purposes zhaba-script has `Err` type, which simply wraps `Str`.

Unrecoverable errors with `panic`

Sometimes, when something very bad happened or when you unwrap `Result` value, you may want to immediately exit program execution. To do that zhaba-script has `panic` function which will print what went wrong end then stop program execution:

```
Program panic at zhaba_tmp.c:3782:
can't open some-invalid-file.txt
```

Interacting with C

Zhaba-script is a new language, and what if somebody want to write web server in it? How can that person do that? Of course, it is not possible to put everything in the language itself or it's standard library. To achieve this goal zhaba-script use the power and simplicity of C to be able to do anything. It means that almost everything you can do in C, you can do in zhaba-script.

Using json to import C

Zhaba-script uses json as universal bridge to C, however you can embed json into C directly, more about that later. To use this json file you can just `use file.json` to import it. Here is it's shape, written as typescript type:

```
type extern_c_def = {
  c_path: String
  types: ({ zh_def: String; c_name: String } | [String, String])[]
  functions: ({ zh_def: String; c_name: String } | [String, String])[]
}
```

- `c_path` is a path to C file, what will be imported
- `types` imported array of C types (structs), `zh_def` is type definition, written in zhaba-script without `type TypeName` and `c_name` is a name of a type from C.
- `functions` imported array of C functions, `zh_def` is function definition (or even operator), written in zhaba-script and `c_name` is a name of a C function.

Alternatively, you can express types and functions not like an object, but like a tuple array, where `zh_def` is a first element, and `c_name` is second element.

Here is an example that json file (suppose we want to import `stdio.h`):

```
{
  "c_path": "stdio.h",
  "types": [
    {
      "zh_def": "bool _", // In C standard File implementations is't defined,
      "c_name": "FILE" // so we can leave some useless value here
    }
  ],
  "functions": [
```

```
    {
      "zh_def": "fn FILEP fopen str filename str mode",
      "c_name": "fopen"
    },
    ["fn i32 fclose FILEP f", "fclose"] // Alternative array syntax
  ]
}
```

Importing C code directly

Writing this huge json files isn't cool, and zhaba-script is also about minimalism, so we can import C files directly! Comments are used to do that. Begin your comment with `@zh-type` or `@zh-fn` and follow it with type or function definition respectively. When importing files this way, you can think of imaginary json object with `c_path` equal to file path, and every time `@zh-...` found, content after that appended to `types` and `functions` arrays. `//` or `/* */` can be used for comments, but each comment can be used for only one definition. Any file extension like `.c`, `.h`, `.cpp`, `.cxx` is valid (other from `.zh` and `.json`). Here is the same example as above, but with this syntax:

```
#include <stdio.h>

// @zh-type ["bool _", "FILE"]

// @zh-fn ["fn FILEP fopen str filename str mode", "fopen"]
// @zh-fn ["fn i32 fclose FILEP f", "fclose"]
```

Advanced C api

Macros

Zhaba-script doesn't support macros, but it is possible to trick compiler into thinking that macro is a function and use it:

```
/** `panic` definition */

// @zh-fn ["fn panic str s", "panic_macro"]

#define panic_macro(s) panic_c(s, __FILE__, __LINE__)

void panic_c(const char* s, const char* file, int line) {
  fprintf(stderr, "Program panic at %s:%d:\n%s\n", file, line, s);
  exit(EXIT_FAILURE);
}
```

Constants

Another similar approach can be taking with global objects and constants, by defining getter-like macro, that expands to desirable value:

```
// @zh-fn ["fn EXIT_CODE EXIT_SUCCESS", "GET_EXIT_SUCCESS"]
#define GET_EXIT_SUCCESS() EXIT_SUCCESS
// @zh-fn ["fn EXIT_CODE EXIT_FAILURE ", "GET_EXIT_FAILURE "]
#define GET_EXIT_FAILURE() EXIT_FAILURE
```

Cross platform definitions

Another ability is to make cross platform definitions, because for zhaba-script compiler only the shape of function or type matters:

```
/** From `c/screen.c` */

// @zh-fn ["fn clrscr", "clrscr"]
void clrscr() {
#ifdef _WIN32
```

```
  HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
  COORD coord = {0, 0};
  DWORD count;
  CONSOLE_SCREEN_BUFFER_INFO csbi;
  GetConsoleScreenBufferInfo(hStdOut, &csbi);
  FillConsoleOutputCharacter(hStdOut, ' ', csbi.dwSize.X * csbi.dwSize.Y, coord,
                             &count);
  SetConsoleCursorPosition(hStdOut, coord);
#else
  printf("\033[2J\033[1; 1H");
#endif
}
```

Variadic functions

Many C functions like `printf` are variadic, but zhaba-script doesn't support them, but we `c/valist.c` provides `va` operators, that acts like `Vec<T>`'s operator `,,` but you need to put `va` in front of sequence too:

```
use c/std

fn main
  printf('number: %d, string: %s\n', va 55 va 'frog')
```

Note, that because `va` is operator, you need to put `,` in front of it.

C++

Zhaba-script isn't designed to target C++, but you can do that by changing C compiler to the C++ one with `-c` flag. You will probably also need to create some kind of wrappers for you C++ code.

Zhaba-script C library

Zhaba-script standard library defines most of necessary C functions and type like `FILE` or `printf`, in `c/std` module, and also has other cross-platform defined functions like `clrscr`.

Using C standard library

The best way to use C standard library is to import `c/std` module, but you can also import only needed modules, but some of them are depend on others, so it is recommended to simply use `c/std` module.

C Standard library content

File	Description
std/c/keyboard.c	Cross platform keyboard functions
std/c/math.c	Defines `<math.h>` for `f64`
std/c/preprocessor.c	Common preprocessor macros like `__FILE__` or `__LINE__`
std/c/screen.c	Cross platform terminal functions
std/c/stdio.c	`<stdio.h>`
std/c/stdlib.c	`<stdlib.h>`
std/c/time.c	Different time functions
std/c/valist.c	Definitions for working with C variadic functions
std/c/panic.c	Defined `panic`
std/c/std.zh	All C standard library