Monolog

inunix3

Contents

Original Implementation

Usage

```
1.  monolog run FILENAME
2.  monolog scan FILENAME
3.  monolog parse FILENAME
4.  monolog repl
```

1. Run the specified program named `FILENAME`. On success, it returns 0 or the last exit code used by builtin `exit()` function, or -1 in case of failure.
2. Load the specified program named `FILENAME` and print tokens.
3. Load the specified program named `FILENAME` and dump the AST.
4. Run the REPL.

The REPL is powered by the isocline library, which enhances editing experience. All available keybindings can be seen in its README.

Building

To build Monolog, you need CMake.

Monolog is written in C11 without any extensions, so it should be compilable by any C11 compliant compiler.

Using CMake

1. Create folder for building and `cd` into it.

```
mkdir build; cd build
```

3. Generate building script:

```
cmake -DCMAKE_BUILD_TYPE=Release -G "GENERATOR" ..
```

If you also want to compile unit tests in `tests/`, add `-DBUILD_TESTS=ON`.

4. After successful generation type `cmake --build ..`
5. If compiled successfully, you can find the binary in `src/`. If you have enabled building of tests, those can be found in the `tests/`.

Reference

Introduction

Monolog is a simple, interpreted C-like imperative programming language.

```
// Minimal hello world

println("Hello, World!");
```

Monolog is **statically typed**, so every declared variable or function must have an explicitly specified type.

NOTE: For description of grammar, a EBNF dialect is used, developed by W3C.

Expressions and statements

```
expression      ::= literal | identifier | nil | binary | unary |
                    suffix | subscript | grouping | function-call
literal         ::= integer-literal | string-literal
integer-literal ::= [0-9]+
string-literal  ::= '"' char '"'
char            ::= /* any Unicode character */

identifier ::= ([a-zA-Z] | '_') ([a-zA-Z] | '_' | [0-9])*
nil        ::= 'nil'

binary   ::= expression binary-op expression
unary    ::= unary-op expression
suffix   ::= expression suffix-op
grouping ::= '(' expression ')'

statement-separated ::= (variable-declaration | return-statement |
                         break-statement | continue-statement | expression) ';'

statement ::= if-statement | while-statement | for-statement |
              statement-separated | function-declaration | block-statement
```

Monolog is based on expressions and statements:

- An expressions consists from combinations of operators, constants, variables and functions, and its value is computable.
- An statement specifies an action. It can contain expressions. Statements are separated by `;`.
- Both can cause **side effects** - modification of other state than expression's/statement's value.

Statements are executed sequentially.

Meaning of expressions

Meaning of an expression depends on how it is being used. For example,

```
int a = list[5];
```

expression `list[5]` returns the 5th element from `list`.

But the expression

```
list[5] = 115;
```

expression `list[5]` does not return a value, but returns a reference to the 5th element, where 115 is being stored.

The same applies for variables and string indexation.

Data types

```
type-specifier ::= int-type | string-type | void-type | option-type | list-type
int-type       ::= 'int'
string-type    ::= 'string'
void-type      ::= 'void'
option-type    ::= type-specifier '?'
list-type      ::= '[' type-specifier (',', expression)? ']'
```

Monolog does not support user-defined types, but has some builtin ones:

1. whole numbers `int` - 64-bit signed integer.
2. strings `string` - mutable array of chars (`int`s).
3. option type `T?`, where `T` is any type.
4. lists `[T]`, where `T` is any type.
5. empty type `void`.

Recursive declarations like `int???????` or `[[[int?]?]]` are supported.

Integer

Primitive type `int` is intended for working with integers.

It contains a 64-bit signed whole number.

String

Compound type `string` is a resizable array of chars (which are `int`s).

Every string has a property **length** - number of chars.

Empty type

Primitive type `void` represents an empty value.

Option type

Option type `T?` is a compound type, which:

1. Contains a value of type `T`,
2. or contains a special value `nil`, which signifies emptiness.

List

List `[T]` is a compound data type, which contains zero or more elements of type `T`, thus it is also a **homogenous** data type.

Operators

```
binary-op ::= '+' | '-' | '*' | '/' | '%' | '<' | '>' | '+=' |
              '-=' | '#=' | '<=' | '>=' | '==' | '!=' | '&&' | '||'
unary-op  ::= '-' | '+' | '*' | '!' | '#' | '$' | '++' | '--'
suffix-op ::= '++' | '--'
subscript ::= expression '[' expression ']'
```

Monolog supports binary, unary and suffix operators. Each type has a set of supported operators.

Arithmetic operators

Binary

Left side	Right side	Operator	Operation	Side effects	Result type
`int`	`int`	`+`	addition	NO	`int`
`int`	`int`	`-`	subtraction	NO	`int`
`int`	`int`	`*`	multiplication	NO	`int`
`int`	`int`	`/`	division	NO	`int`
`int`	`int`	`%`	division by modulo	NO	`int`

Unary

Left side	Operator	Operation	Side effects	Result type
`int`	`+`	returns value of the expression	NO	`int`
`int`	`-`	changes the sign to the opposite	NO	`int`

Logic and relation operators

These operations return 1 if the expression is true, 0 otherwise.

Binary

Left side	Right side	Operator	Operation	Side effects	Result type
`int`	`int`	`==`	compares values whether they are equal	NO	`int`
`int`	`int`	`!=`	compares values whether they are different	NO	`int`
`int`	`int`	`<`	if the first operand is lesser than the second one	NO	`int`
`int`	`int`	`>`	if the first operand is greater than the second one	NO	`int`
`int`	`int`	`<=`	if the first operand is equal to or lesser than the second one	NO	`int`
`int`	`int`	`>=`	if the first operand is equal to or greater than the second one	NO	`int`
`int`	`int`	`&&`	boolean AND	NO	`int`
`int`	`int`	`||`	boolean OR	NO	`int`

Unary

Right side	Operator	Operation	Side effects	Result type
`int`	`!`	logical negation	NO	`int`

List Operators

Binary

Left side	Right side	Operator	Operation	Side effects	Result type
`[T]`	`T`	`+=`	push value to the end of list	YES	`void`
`[T]`	`int`	`-=`	pop N values from the end of list	YES	`void`
`[T]`	`int`	`#=`	resize the list	YES	`void`

- `-=`: if the specified number of elements to pop is greater or equal to the size of list, the whole list will be cleared.
- `*=`: if the specified number of elements to pop is greater than the actual, new elements will be appended and they will be zeroed. If its lesser, remainder will be popped.

Unary

Right side	Operator	Operation	Side effects	Result type
`[T]`	`#`	returns the number of elements	NO	`int`

Suffix

Left side	Operator	Operation	Side effects	Result type
`[T]`	`[int]`	returns reference to the specified element	NO	`T`

- `[int]`: this operator expects an expression inside brackets to be of type `int`. The value must be in the range $[0, N)$, where N is the number of elements in the list.

Option Type Operators

Binary

Left side	Right side	Operator	Operation	Side effects	Result type
`T?`	`nil`	`==`	check, whether the option object is empty	NO	`T`

Unary

Right side	Operator	Operation	Side effects	Result type
`T?`	`*`	return reference to the stored value	NO	`T`

- this operator cannot be used on empty option objects.

String Operators

Binary

Operace `==` a `!=` return 1 if the expression is true, 0 otherwise.

Left side	Right side	Operator	Operation	Side effects	Result type
`string`	`string`	`+`	concatenates the right string to the left one	NO	`string`
`string`	`string`	`==`	compare lengths and contents of strings if they are equal	NO	`int`
`string`	`string`	`!=`	compare lengths or contents of string if they are not equal	NO	`int`

Unary

Right side	Operator	Operation	Side effects	Result type
`string`	`#`	returns the length of string	NO	`int`

Suffix

Left side	Operator	Operation	Side effects	Result type
`string`	`[int]`	returns reference to the specified char	NO	`int`

- `[int]`: this operator expects an expression inside brackets to be of type `int`. The value must be in the range $[0, N)$, where N is the length of the string.

Conversion Operators

Unary

Right side	Operator	Operation	Side effects	Result type
`int`	`$`	converts the integer to a string (creates new string)	NO	`string`

Assignment Operators

Left side	Right side	Operator	Operation	Side effects	Result type
variable of type `T`	`T`	`=`	assign a new value to the variable	YES	`T`
element in list of type `T`	`T`	`=`	assign a new value to the element in list	YES	`T`
variable or element in list of type `T?`	`T`	`=`	store a new value in the option object	YES	`T`
char in a string	`int`	`=`	assign a new char	YES	`int`

When assigning, the new value is being **copied**. For example, when we assign the value of `a` to variable `b`, future modifications of `b` will not affect the value of `a`.

Priority and associativity

Monolog respects conventional priority and associativity of operators.

The following table shows priorities and associativity of each operator. Operators are listed in descending order.

Priority	Operator(s)	Description	Associativity
1	`++ -- () []`	Suffix operators	left-to-right
2	`+ - ! # $ * ++ --`	Prefix operators	right-to-left
3	`* / %`	Multiplication, division	left-to-right
4	`+ -`	Addition, subtraction	left-to-right
5	`< <= > >=`	Relational operators	left-to-right
6	`== !=`	Equality, inequality	left-to-right
7	`&&`	Boolean AND	left-to-right
8	`||`	Boolean OR	left-to-right
9	`= += -= #=`	Assignment	right-to-left

Control Flow Statements

Monolog has basic statements for branching and looping:

- Branching

  - `if` - conditional code execution.
  - `else` - alternative path of branch.
- Looping
  - `while` - looping.
  - `for` - iterative looping.
- Miscellaneous
  - `return` - return from a function.
  - `break` - terminate loop.
  - `continue` - skip loop body.

if, else

```
if-statement   ::= 'if' '(' expression ')' statement? else-statement?
else-statement ::= 'else' statement?
```

- Check whether condition is true. If yes, execute the body.
- If the condition is false, execute the alternative body, given by ‘else.

while

```
while-statement ::= 'while' '(' expression ')' statement?
```

- Check, whether condition is true. If yes, execute the body.
- After execution of body, check the condition again. If it is still true, repeat the procedure. If not, terminate the loop.

for

```
for-statement ::= 'for' '(' init-clause? ';' condition? ';' iter-expr? ')' statement?
init-clause   ::= expression | variable-declaration
condition     ::= expression
iter-expr     ::= expression
```

- If `init-clause` exists, it is executed first.
- If `condition` exists, check, whether it is true. If `condition` does not exists, it’s replaced by 1.
- If the condition is true, body is executed.
- After executing body, `iter-expr` is executed if it exists.
- Again check the condition. If it’s still true, this procedure repeats. If not, loop terminates.

return

```
return-statement ::= 'return' expression?
```

- This statement can be used only inside functions.
- When used, execution will leave the current function and will continue after the place in code, where the function wa called.
- If the function has other type than `void`, this statement must have a return value, which will be returned to the caller.
- If the function has type `void`, this statement must not have a return value.

break

```
break-statement ::= 'break'
```

- This statement can be used only inside loops.

- When used, terminates the loop, and execution will continue immediatelly after the loop body.

continue

`continue-statement ::= 'continue'`

- This statement can be used only inside loops.
- When used, execution will jump to the start of loop body.
- After jumping, `while` will check the condition. `For` will not execute `iter-expr` clause, and will check the condition.

Name binding

Declaration is an introduction of one or more names, which have assigned meaning and properties.

Monolog supports declaration of **variables** and **functions**.

Variables

`variable-declaration ::= type-specifier identifier ('=' expression)? ';'`

V variable binds name to a specific value. Each variable has a type `type-specifier` specified by user, and optionally a default value located on the right side of `=`.

If a variable is declared without a defaut value, its default value will be one of these, depending on the type:

Type	Default value
`int`	`0`
`string`	`""`
`void`	-
`[T]`	`[]`
T?	`nil`

When a variable is used in expression, its value is substituted in the place of name.

Variable cannot be of type `void`. This is a semantic error.

```
// Declaration of a variable with type int and name "a", default value is 0.
int a;

// Declaration of a variable with type int and name "c",
// its value is a sum of variables a and b.
int c = a + b;

// Declaration of a variable with type string and name "city",
// its value is a string "Prague".
string city = "Prague";

// An option variable, it's empty (default value is `nil`).
string? name = nil;
name = "hello";

// Declaration of a list, its inner type is a list of option ints.
[[int?]] matrix;
```

Lists: syntactic sugar

When declaring a list, its default size can be specified:

```
[int, 5] list; // has 5 zeroed elements, each is of type int.
```

This is a shortcut for:

```
[int] list;
list #= 5;
```

NOTE: to keep the code of the interpreter simpler, it's not possible to use this in inner lists:

```
// Not an error, but the nested list will not have 3 elements.
[[int, 3], 5] a;
```

As a workaround, `#=` can be used:

```
[[int], 5] a;

for (int i = 0; i < #a; ++i) {
    a[i] #= 3;
}
```

Functions

```
function-declaration ::= type-specifier identifier '(' param-decl-list ')' statement
param-decl           ::= type-specifier identifier
param-decl-list      ::= param-decl ','? | (param-decl ',')+ param-decl ','?

function-call ::= identifier '(' arg-list ')'
arg-list      ::= expression ','? | (expression ',')+ expression ','?
```

Function binds a name and a piece of code, that can use **parameters** (`param-decl-list`).

Calling a function means execute a specific function - its assigned piece of code - and if it has defined parameters, execute with **arguments**.

When function has defined parameters and it's called using the `function-call` syntax, in place of parameters values of arguments are substituted. Interpreter then will start to execute function's body, and will create variables (local to this body) with names of parameters and their values.

When calling a function, type of each argument must be the same/convertable to the type of the corresponding parameter.

```
// Function with name "foo", without any parameters, the return type is void,
// its body is a block
void foo() {
    println("Hello, World!");
}

// Function with parameters and a return type int.
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

// Function call
foo();
```

```
// Calling a function with parameters
int m = max(115, 94); // argumenty jsou a = 115, b = 94.

void bar() {
    // Use of variable and a function inside function,
    // which are declared in the global scope.
    if (max(m, 5)) {
        println("A");
    } else {
        println("B");
    }
}
```

Scoping

Scope is a part of source code, which has defined variables (i.e. name binding is applied).

Each program written in Monolog has at least one scope - a **global scope**. It has the same properties as the user-created scopes, except that it is created automatically by interpreter.

Each function creates a new scope for its parameters:

```
int sum(int x, int y) {
    return x + y;
}

int z = sum(arg1, arg2);

// Meaning:

int z;

{
    int x = arg1; // arg1 must be int
    int y = arg2; // arg2 must be int

    {
        z = x + y; // return x + y;
    }
}
```

When calling a function, a new scope is created right after the global scope, which helps to avoid situation, when function has access to the caller's scope and its parent scopes (except the global one).

Since the for loop allows to declare a variable, a new scope is created too, which is a subscope of the current scope:

```
int z;

for (int i = 0; i < 10; ++i) {
    z = z + i * i;
}

println($z);

// Meaning:

int z;
```

```
{
    int i = 0;

    while (i < 10) {
        z = z + i * i;
        ++i;
    }
}

println($z);
```

Name resolution

Name resolution means finding out which entity is associated with a given name.

Monolog performs name resolution so that it searches the current scope first, then, if a parent scope exists, searches there and repeats it until the global scope, where it searches too.

If the searching failed, this is a semantic error and the program is ill-formed

In case of functions, a function can be declared only in the global scope, so name resolution of function name is performed only in the global scope.

New scope can be created using **blocks**.

Block - statement grouping

`block-statement ::= '{' statement* '}'`

Block creates a new scope and a *lifetime scope*, and then sequentially executes each statement.

```
// global scope

int x;
int y;

// scope 1
{
    int z = x + y;

    // scope 2
    {
        string w = $x + $y + $z;
    }

}
```

Memory Model

Memory model in Monolog is based on **lifetime scopes**, which are closely related to normal scopes.

Lifetime Scope

Lifetime scope covers a whole scope, and manages values of variable, which were created/declared in the corresponding scope.

End of lifetime means that value or variable are freed from memory and stops existing, and the memory, that was occuppied, can be reused by the interpreter.

```
// global scope

int x;
int y;

// scope 1
{
    int z = x + y;

    // scope 2
    {
        string w = $x + $y +:$z;

        // lifetime of w ends here
    }

    // lifetime of z ends here
}

// end of the program
// lifetime of x and y ends here
```

Static and Dynamic Values

Depending on memory usage, values can be grouped as:

1. static
    - integers (`int`)
    - empty type (`void`)
    - `nil`
    - empty option types (`T?`)
2. dynamic
    - strings (`string`)
    - lists (`[T]`)
    - non-empty option types (`T?`)

Dynamic values are deallocated, when their lifetime is ended. If a dynamic value is a value of a variable, the value will be deallocated when the lifetime of the variable ends.

Argument passing

Arguments are passed so that they are copied (passed by value) or passed by reference - mutation of the parameter inside the function will affect the value of the argument in the caller's scope.

If the value of an argument is a result of an expression and is not binded, this argument will be always copied. If, however, it is a variable, depending on its type, it will be passed by reference and any mutation will affect it. If the argument is an element from list/string, it will be always passed by reference.

Origin of the argument	Type	Passing method
Result of an expression	`T`	By copy
Variable	`T`, `T` != `int`, `void` or `nil`	By reference
Variable	`T`, `T` = `int`, `void` or `nil`	By copy
Element from list/string	`T`	By reference

Builtin functions

Monolog has builtin functions, that are available everywhere.

print

```
void print(string s);
```

Print a string `s` to standard output.

println

```
void println(string s);
```

Print a string `s` to standard output and print a newline character.

exit

```
void exit(int code);
```

Exit program with exit code `code`.

input_int

```
int? input_int();
```

Read an integer from the standard input stream. In case of failure (bad integer syntax or general I/O error), returns `nil`.

This function is blocking.

input_string

```
string? input_string();
```

Read a string from the standard input stream. In case of I/O error, returns `nil`.

This function is blocking.

random

```
int random();
```

Generate a random number in range $[0, M]$, where M is a number, which is **at least** 32767 or greater.

random_range

```
int random_range(int min, int max);
```

Generate a number in range $[min, max]$.

chr

```
string chr(int ch);
```

Convert an ASCII number `ch` and return a corresponding character as a string.

ord

```
int ord(string ch);
```

Convert the first character of string `ch` and return a corresponding ASCII number.

Empty string returns 0.

Appendix A

Grammar

Here is a complete grammar of Monolog. It is written in an EBNF dialect, developed by W3C.

```
expression ::= literal | identifier | nil | binary | unary |
               suffix | subscript | grouping | function-call

literal         ::= integer-literal | string-literal
integer-literal ::= [0-9]+
string-literal  ::= '"' char '"'
char            ::= /* any Unicode character */

identifier ::= ([a-zA-Z] | '_') ([a-zA-Z] | '_' | [0-9])*
nil        ::= 'nil'

binary    ::= expression binary-op expression
binary-op ::= '+' | '-' | '*' | '/' | '%' | '<' | '>' | '=' | '+=' |
              '-=' | '#=' | '<=' | '>=' | '==' | '!=' | '&&' | '||'

unary    ::= unary-op expression
unary-op ::= '-' | '+' | '*' | '!' | '#' | '$' | '++' | '--'

suffix    ::= expression suffix-op
suffix-op ::= '++' | '--'

subscript ::= '[' expression ']'
grouping  ::= '(' expression ')'

statement-separated ::= (variable-declaration | return-statement |
                         break-statement | continue-statement | expression) ';'

statement ::= if-statement | while-statement | for-statement |
              statement-separated | function-declaration | block-statement

type-specifier ::= int-type | string-type | void-type | option-type | list-type
int-type       ::= 'int'
string-type    ::= 'string'
void-type      ::= 'void'
option-type    ::= type-specifier '?'
list-type      ::= '[' type-specifier (',', expression)? ']'

if-statement   ::= 'if' '(' expression ')' statement? else-statement?
else-statement ::= 'else' statement?

while-statement ::= 'while' '(' expression ')' statement?

for-statement ::= 'for' '(' init-clause? ';' condition? ';' iter-expr? ')' statement?
init-clause   ::= expression | variable-declaration
condition     ::= expression
iter-expr     ::= expression

return-statement   ::= 'return' expression?
break-statement    ::= 'break'
continue-statement ::= 'continue'
```

```
variable-declaration ::= type-specifier identifier ('=' expression)? ';'

function-declaration ::= type-specifier identifier '(' param-decl-list ')' statement
param-decl           ::= type-specifier identifier
param-decl-list      ::= param-decl ','? | (param-decl ',')+ param-decl ','?

function-call ::= identifier '(' arg-list ')'
arg-list      ::= expression ','? | (expression ',')+ expression ','?

block-statement ::= '{' statement* '}'
```