SER502 Project Milestone 2 Document

Team4

March 30, 2018

1 Team Members

- Zelin Bao
- Yuan Cao
- Yiru Hu
- Lei Zhang

2 Language Name

The name of language is **Luna**, and the programming file suffix is ***.lu**, byte-code file suffix is ***.luo**.

3 Tools

Tools provided by **Luna**:

- Compiler: **lunac**
- Interpreter: **luna**
- Installer for compiler and interperter.

4 Design and Grammar

4.1 Primary Types

The primary types of Luna include `bool, int, double, list`.

- `bool`: the value of `bool` can be either `true` or `false`.
- `int`: the range of integers in Luna is `[-2147483648, 2147483647]`($[-2^{31}, 2^{31}-1]$).

- `double`: Luna does not have `float` type of floating numbers, which means all floating numbers use double precision definition.
- `list`: `list` is the only data structure has been provided in Luna, `list` can be assigned to a variable or used as function parameter. Luna only support 1-dimension list. The element of `list` can be `bool, int` or `double`.

4.2 Variable Declaration and Assignment

4.2.1 Variable Declaration

The basic form of variable declaration in Luna is:

```
[type] [name] = [value].
```

Every variable in Luna must have a type and default value when it has been created, and the type of a variable cannot be changed until it have been destroyed.
The declaration of list is different from others types. In Luna, we use `[` and `]` to denote a constant value of list. Moreover, Luna also support initial a list by variables. For example:
Statement:

```
list lst = [1, 2, 3, 4]
```

created a `list` named `lst` by value `[1, 2, 3, 4]`.
Statement:

```
list lst = [a, b, 3, 4]
```

created a list used the value of variable `a` and `b`.

4.2.2 Variable Assignment

One variable can assign it's value to another variables in Luna. And that assignment operation is deep-copied. For example:

```
list lst = [1, 2, 3, 4]
list lst2 = lst \\ lst2 = [1, 2, 3, 4]
lst2 = [4, 5, 6, 7] \\ lst = [1, 2, 3, 4]
```

4.3 Operators

The operators of Luna include `++, --, +, -, *, /, ==, !=, >, <, >=` and `<=`.

4.3.1 Unary Operator

Unary operators only have one argument in right-side, and it can only work for data type `int`.

- `++`: increase the value of variable in left side by 1. Example:

```
a++ //If a = 1, return 2
```

- `--`: decrease the value of variable in left side by 1. Example:

```
a-- //If a = 1, return 0
```

4.3.2 Binary Operator

Both variables and constant values can be used in binary operators. Only integer and double types are supported.

- `+`: Addition operator. The priority of addition operator is lower than `*` and `/` operator and equal to `-` operator. Example:

```
1 + 2 // return 3
a + b // if a = 1, b = 3, return 4
```

- `-`: Subtraction operator. The priority of subtraction operator is lower than `*` and `/` operator and equal to `+` operator. Example:

```
1 - 2 // return -1
a - b // if a = 1, b = 3, return -2
```

- `*`: Multiplication operator. The priority of multiplication operator is higher than `+` and `-` operator and equal to `/` operator. Example:

```
1 * 0 // return 0
a * b // if a = 1, b = 3, return 3
```

- `/`: Division operator. The priority of division operator is higher than `+` and `-` operator and equal to `*` operator. The right-side value of division operator cannot be `0`, and the return value type of division operator is always `double`. Example:

```
0 / 1 // return 0.0
a / b // if a = 1, b = 2, return 0.5
a / 0 // invalid, error
```

4.3.3 Comparison Operator

All comparison operators are binary operator, and the return type of all comparison operator is `bool`. Data type `int, bool, double` can be used in both sides of all comparison operator. Moreover, the data types of both sides must be same.

- `==`: Equal-to operator. This operator will check whether or not the left-side value is equal to right-side value. If the check result is correct, it will return `true`. If the check result is wrong, it will return `false`. Example:

```
1 == 2 // return false
a == b // if a = 1, b = 1, return true
true == 1 // invalid, error
```

- `!=`: Not equal-to operator. This operator will check whether or not the left-side value is not equal to right-side value. If the check result is correct, it will return `true`. If the check result is wrong, it will return `false`. Example:

```
1 != 2 // return true
a != b // if a = 1, b = 1, return false
```

- `>, <, >=` and `<=`: This sequence of operators will check the value relationship between values in two-sides. Example:

```
1 <= 2 // return true
a > b // if a = 1, b = 2, return false
```

4.4 If-else Block

`if-else condition` is a statement to determine whether this condition is true. if this condition is true, execute this block of code, if this condition is false, execute another block of code. The format of if-else condition structure in Luna programming language is as follows:

```
if(boolean expression(s))

    statement(s)

end.
```

or

```
if(boolean expression(s))
    statement(s)
else
    statement(s)
end.
```

- `boolean expression` is an expression producing boolean value. boolean expression is always composed by the combination of comparison expressions, boolean value, connected by logical operator.

- `boolean values` is true or false, true can be replaced by 1, false can be replaced by 0.

- `statement(s)` is a block of code can be executed.

4.5 Iteration Blocks

The primary loops of Luna are `while loop, for loop`. In this situation, you can execute a block of code several number of times. In Luna, statements are executed sequentially:

the first statement in a function is executed first, followed by the second, and so on

4.5.1 While-loop

In Luna language, a `while loop` statement repeatedly executes a target statement as long as a given boolean expression is true. The syntax of `while loop` in Luna programming language is as follows:

```
while([bool expression])
do
    //statement
end
```

Here, `statement(s)` may be a single statement or a block of statements. `condition` is boolean expression(s), and `true` or `false`. The loop iterates while the `condition(s)` is true. When the `condition(s)` became false, the program escapes the loop immediately.
There is something need to be mentioned that if the `condition` is false, the loop body will be skipped. Example:

```
int i = 1
while(i < 5)
do
    print("value:", i)
    i = i + 1
end

Result:
    value: 1
    value: 2
    value: 3
    value: 4
```

4.5.2 For-loop

Luna provides a `for` loop which allows you to elegantly create a loop that needs to execute a specific number of times.The syntax of `for` loop in Luna programming language is as follows:

```
for begin, range, step
do
    statement(s)
end
```

Here is the flow of control in a for loop:

- The `begin` variable is initialized first, and only once. This process allows you to declare and initialize `begin` variable.
- Next, the `range` value till which the loop continues to execute. It creates a condition check internally to compare between the initial value and `range` value.
- After the body of the `for` loop executes, the flow of the control jumps back up to the increment statement where it updates loop control variable by operating with `step` variable.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the `for` loop terminates.
  Example:

```
for i = 1, 5, 1
do
    print("value:", i)
end

Result:
    value: 1
    value: 2
    value: 3
    value: 4
```

There is another `for` loop called `for-each`, which allows you to do all elements in a list. The syntax as follows:

```
for item in list
do
    statement(s)
end
```

4.6 Comment

There are two kinds of `comment` in Luna programming language, including `//` and `/**/`.

- `//` could comment one line of code.
- `/**/` could comment a block of code.

4.7 Function and Recursion

4.7.1 Function Definition

The syntax of function definition in Luna is:

```
functiondef ::=type function funcbody
funcbody ::= ‘(’ [parlist] ‘)’ block end
```

Parameters in Luna have the syntax:

```
parlist ::= ...
parlist ::= var
parlist ::= var, parlist
var     ::= type name
```

An example function definition of geting the maximum of two values in Luna:

```
int function max(int num1, int num2)
    int result;
    if (num1 > num2) then
        result = num1;
    else
        result = num2;
    end
    return result;
end
```

4.7.2 Function Call

The syntax of the function call in Luna is:

```
functioncall ::= FunctionName args
```

And the syntax of the arguments in Luna should be:

```
args ::= '(' [explist] ')'
explist ::= exp|exp ',' explist
```

An example of call a function:

```
print(1+2);
```

The function can also be called as a statement. For example:

```
x=max(a,b);
```

In syntax the function call can be statement:

```
stat ::= functioncall
```

4.8 Program Startup

The entry point of a Luna Program should be like:

```
entry ::= main '(' [argint, arglist] ')' block end
```

The argint should be in **int** type and arglist should be **list** type.

5 Parser

5.1 Syntax

Here is the complete syntax of Luna in EBNF. A means 0 or more As, and [A] means an optional A.

```
program      ::= {functiondef} "main" '(' [var ',' var] ')' block "end"

block        ::= {stat} [retstat]

stat         ::= ';' |
                 var '=' expr |
                 var '=' list_expr|
                 id '=' expr |
                 id unary_op |
                 functioncall |
                 "do" block "end" |
                 "while" '(' exp ')' "do" block "end" |
                 "if" '(' exp ')' block ["else" block] "end" |
                 "for" id "in" id "do" block "end" |
                 "for" id "in" list_expr "do" block "end" |
                 "for" id '=' num ',' num ',' num "do" block "end"

retstat      ::= "return" id

var          ::= type id

functioncall ::= id args

args         ::= '(' [explist] ')'

explist      ::= expr | expr ',' explist

exp          ::= "false" | "true" | expr boolop expr

expr         ::= term {expr_op term} | functioncall

list_expr    ::= '[' [num] {',' num} ']'
```

```
term          ::= num {term_op num}

functiondef   ::= type "function" funcbody

funcbody      ::= '(' [parlist] ')' block "end"

parlist       ::= '...' | type id {',' type id}

expr_op       ::= '+' | '-'

term_op       ::= '*' | '/'

unary_op      ::= '++' | '--'

boolop        ::= '<' | '<=' | '>' | ">=" | "==" | "!="

type          ::= "bool" | "int" | "double" | "list"

num           ::= ['-'] digit{digit}

floatnum      ::= num | num '.' digit{digit}

digit         ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

id            ::= [a-z|A-Z]+
```

5.2 Parsing Technique

The scanner and parser of Luna will be implemented by tool Flex and Bison. Flex will accept a text file and generate a sequence of tokens. Bison will take these tokens and parse them to a Abstract Syntax Tree(AST) use above EBNF defination of Luna. Then, we will write a bytecode generator use that AST to generate our bytecode file. The bytecode generator will be written by C++. The basic idea of that generator is using DFS algorithm to traverse the AST and generate corresponding code for each node. `Vector` and `Stack` will be the primary data structure in bytecode generator.

5.3 Type Checking

To decrease run time error of Luna, we will do static type checking in bytecode generator. That means any static type error will be found and reported before bytecode generation operation. The compiler will print error information when type checking failure occured.

6 Bytecode and Interpreter

6.1 Bytocode Design

The Luna language will use a stack-based virtual machine in interpreter. Therefore, the byte-code generated by byte-code generator has following operations: `load, store, add, sub, mult div, print, read, copy_top, jump, jump_if, pop, cmp, cmp_eq, cmp_bg, cmp_lt, cmp_eq_bg cmp_eq_lt` and `nop`. These operations are used to describe all instructions in AST.

6.2 Interpreter

The interpreter of Luna will be implemented by C++. The primary data structure of interpreter is `Stack` and `Map`. Map is used to store variables. Stack is used to simulate the opreration order of AST. Interpreter has two pointers, one for stack, one for bytecode file. With the moving of bytecode file pointer, the stack will be pushed or removed objects.