PROLUA

Working Draft

Jeremy OTHIENO
https://github.com/supranove

Chapter I. Introduction

[TODO] Add cycle diagram: input (lua code) > lua2prolog > prolua > output (result and environment)

I.a. Why Lua?

[TODO] Why Lua? Simple, powerful yet expressive, hasn't been done, etc.

I.b. Why Prolog?

[TODO] Why Prolog? Power of expression, ease of use, etc.

I.c. Constraints

The interpreter will work for Lua 5.1 and below. This is because I'm not familiar with some of the changes made in later versions of Lua, most notably the change in how the environment is managed. I'll have to do a bit more research to be able to update Prolua.

Of the eight basic types of values in Lua, only only **numbers**, **strings**, **booleans**, **tables**, **functions** and **nil** will be implemented. The **userdata** and **thread** types will be excluded, as well as the features that depend on these types, such as coroutines.

I.d. Notation

[TODO] Explain notation and how to read it.

Chapter II. Syntax

Our first order of business is to define an abstract syntax specific to Prolua, from Lua's concrete syntax given below in EBNF

```
chunk ::= {stat [`;´]} [laststat [`;´]]

block ::= chunk

stat ::=
      varlist `=´ explist |
      functioncall |
      do block end |
      while exp do block end |
      repeat block until exp |
      if exp then block {elseif exp then block} [else block] end |
      for Name `=´ exp `,´ exp [`,´ exp] do block end |
      for namelist in explist do block end |
      function funcname funcbody |
      local function Name funcbody |
      local namelist [`=´ explist]

laststat ::= return [explist] | break

funcname ::= Name {‘.’ Name} [‘:’ Name]

varlist ::= var {‘,’ var}

var ::= Name | prefixexp ‘[’ exp ‘]’ | prefixexp ‘.’ Name

namelist ::= Name {‘,’ Name}

explist ::= exp {‘,’ exp}

exp ::=
      nil | false | true | Number | String | ‘...’ | function |
      prefixexp | tableconstructor | exp binop exp | unop exp

prefixexp ::= var | functioncall | ‘(’ exp ‘)’

functioncall ::=  prefixexp args | prefixexp ‘:’ Name args

args ::=  ‘(’ [explist] ‘)’ | tableconstructor | String

function ::= function funcbody

funcbody ::= ‘(’ [parlist] ‘)’ block end

parlist ::= namelist [‘,’ ‘...’] | ‘...’

tableconstructor ::= ‘{’ [fieldlist] ‘}’

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= ‘[’ exp ‘]’ ‘=’ exp | Name ‘=’ exp | exp

fieldsep ::= ‘,’ | ‘;’

binop ::=
      ‘+’ | ‘-’ | ‘*’ | ‘/’ |
      ‘^’ | ‘%’ | ‘..’ | ‘<’ | ‘<=’ | ‘>’ | ‘>=’ | ‘==’ | ‘~=’ | and | or

unop ::= ‘-’ | not | ‘#’
```

a. Sets

Let ***Expression*** be the set of all possible expressions in Lua, and ***ExpressionList*** a list of expressions such that

$$\text{expressionlist}:\frac{}{[]\ \in\ \text{ExpressionList}}\qquad \text{expressionlist}:\frac{e\ \in\ \text{Expression},\quad es\ \in\ \text{ExpressionList}}{e::es\ \in\ \text{ExpressionList}}$$

Let ***Name*** be the set of all possible identifer names in Lua, and ***NameList*** a list of names such that

$$\text{namelist}:\frac{}{[]\ \in\ \text{NameList}}\qquad \text{namelist}:\frac{n\ \in\ \text{Name},\quad ns\ \in\ \text{NameList}}{n::ns\ \in\ \text{NameList}}$$

Let ***ParameterName*** be an extension of ***Name*** to include the name "**...**", then let ***ParameterNameList*** be a list of parameter names such that

$$\text{parlist}:\frac{}{[]\ \in\ \text{ParameterNameList}}$$

$$\text{parlist}:\frac{p\ \in\ \text{ParameterName},\quad ps\ \in\ \text{ParameterNameList}}{p::ps\ \in\ \text{ParameterNameList}}$$

Let ***Variable***, a subset of ***Expression***, be the set of all possible variables in Lua, and ***VariableList*** a list of variables such that

$$\text{variablelist}:\frac{}{[]\ \in\ \text{VariableList}}\qquad \text{variablelist}:\frac{v\ \in\ \text{Variable},\quad vs\ \in\ \text{VariableList}}{v::vs\ \in\ \text{VariableList}}$$

Let ***Value***, a subset of ***Expression***, be the set of all possible values in Lua, and ***ValueList*** a list of values such that

$$\text{valuelist}:\frac{}{[]\ \in\ \text{ValueList}}\qquad \text{valuelist}:\frac{v\ \in\ \text{Value},\quad vs\ \in\ \text{ValueList}}{v::vs\ \in\ \text{ValueList}}$$

Let ***Statement*** be the set of all possible statements (instructions) in Lua, and ***StatementList*** a list of statements such that

$$\text{statementlist}:\frac{}{[]\ \in\ \text{StatementList}}\qquad \text{statementlist}:\frac{s\ \in\ \text{Statement},\quad ss\ \in\ \text{StatementList}}{s::ss\ \in\ \text{StatementList}}$$

b. Values and Types

Nil is a type of value whose main property is to be different from any other value; it usually represents the absence of a useful value

$$\text{value}: \frac{v \in \{\text{nil}\}}{\text{niltype}(v) \in \text{Value}}$$

Boolean values are defined as **false** and **true**.

$$\text{value}: \frac{v \in \{\text{false, true}\}}{\text{booleantype}(v) \in \text{Value}}$$

Number represents **real** numbers.

$$\text{value}: \frac{v \in \mathbb{R}}{\text{numbertype}(v) \in \text{Value}}$$

A **string** represents arrays of 8-bit characters. There's no **character** type in Lua but to be able to define the syntax of a string, we need to define what a character is. Unfortunately, the character set is too large to enumerate so we'll simplify by supposing that it's the set of all 8-bit ASCII characters. A string is then considered to be a concatenation of characters

$$\text{string}: \frac{}{[] \in \text{String}} \qquad \text{string}: \frac{c \in \text{Character}, \quad s \in \text{String}}{c{::}s \in \text{String}}$$

$$\text{value}: \frac{s \in \text{String}}{\text{stringtype}(s) \in \text{Value}}$$

The type **table** implements associative arrays, i.e. arrays that can be indexed with any value except nil. Tables can contain values of all types including 'nil', in which case the table field is deleted

$$\text{table}: \frac{}{[] \in \text{Table}} \qquad \text{table}: \frac{k \in \text{Value} \setminus \{\text{niltype(nil)}\}, \quad v \in \text{Value}, \quad t \in \text{Table}}{\langle k, v \rangle{::}t \in \text{Table}}$$

$$\text{value}: \frac{t \in \text{Table}}{\text{tabletype}(t) \in \text{Value}}$$

Function bodies are considered values in Lua and are defined syntactically as

$$\text{value}: \frac{ps \in \text{ParameterNameList}, \quad ss \in \text{StatementList}}{\text{functionbody}(ps, ss) \in \text{Value}}$$

b. Operators

Calling a **unary operator** requires the operator's name and the expression to be evaluated

$$\text{expression}: \frac{n \in \{\text{negative, not, length}\}, e \in \text{Expression}}{\text{unop}(n, e) \in \text{Expression}}$$

Almost like a unary operator, calling **binary operators** requires the name of the operator and two expressions to be evaluated

$$\text{expression}: \frac{n \in \{\text{add, subtract, multiply, divide, modulo, exponent}\}, \quad e_{lhs}, e_{rhs} \in \text{Expression}}{\text{binop}(n, e_{lhs}, e_{rhs}) \in \text{Expression}}$$

$$\text{expression}: \frac{n \in \{\text{equal, lt, le, gt, ge, and, or, concatenate}\}, \quad e_{lhs}, e_{rhs} \in \text{Expression}}{\text{binop}(n, e_{lhs}, e_{rhs}) \in \text{Expression}}$$

c. Statements.

The unit of execution in Lua is called a **chunk** which is simply a sequence of statements that are executed sequentially

$$\text{statement}: \frac{ss \in \text{StatementList}}{\text{chunk}(ss) \in \text{Statement}}$$

Lua allows multiple **assignments**. Therefore, the syntax for assignment defines a list of variables on the left side and a list of expressions on the right. Syntactically, the assignment operator takes two lists of expressions which it will evaluate into appropriate values

$$\text{statement}: \frac{es_{lhs}, es_{rhs} \in \text{ExpressionList}}{\text{assign}(es_{lhs}, es_{rhs}) \in \text{Statement}}$$

Function calls can be executed as statements, in which case all return values are thrown away

$$\text{statement}: \frac{e \in \text{Expression}, \quad es \in \text{ExpressionList}}{\text{functioncall}(e, es) \in \text{Statement}}$$

The **do** statement allows us to explicitly delimit a block of statements to produce a single statement

$$\text{statement}: \frac{ss \in \text{StatementList}}{\text{do}(ss) \in \text{Statement}}$$

A **while** loop is defined as

$$\text{statement}: \frac{e \in \text{Expression}, \quad ss \in \text{StatementList}}{\text{while}(e, ss) \in \text{Statement}}$$

A **repeat-until** loop is similar to a while loop

$$\text{statement}: \frac{e \in \text{Expression}, \quad ss \in \text{StatementList}}{\text{repeat}(e, ss) \in \text{Statement}}$$

An **if-else** condition takes an expression that it evaluates into a boolean value, a statement that will be executed if the aforementioned value is true, and another if it is false

$$\text{statement:}\ \frac{e \in \text{Expression},\quad s_{\text{true}}, s_{\text{false}} \in \text{Statement}}{\text{if}(e, s_{\text{true}}, s_{\text{false}}) \in \text{Statement}}$$

We can then define an **if** condition (without the **else**) as "if(e, s_{true}, do([]))", an **if-elseif** condition recursively as "if(e, s_{true}, $\text{if}_i(e_i, s_{\text{truei}}, \text{do}([]))$)", and finally an **if-elseif-else** condition as "if(e, s_{true}, $\text{if}_i(e_i, s_{\text{truei}}, s_{\text{false}})$)".

The **numeric for** loop is defined syntactically as

$$\text{statement:}\ \frac{n \in \text{Name},\quad e_{\text{initial}}, e_{\text{end}}, e_{\text{increment}} \in \text{Expression},\quad ss \in \text{StatementList}}{\text{for}(n, e_{\text{initial}}, e_{\text{end}}, e_{\text{increment}}, ss) \in \text{Statement}}$$

while the **generic for** loop is defined as

$$\text{statement:}\ \frac{ns \in \text{NameList},\quad es \in \text{ExpressionList},\quad ss \in \text{StatementList}}{\text{for}(ns, es, ss) \in \text{Statement}}$$

Declaring functions requires the name of the function as well as a function body value

$$\text{statement:}\ \frac{n \in \text{Name},\quad v \in \text{Value}}{\text{function}(n, v) \in \text{Statement}}$$

Declaring local functions has the same syntax, but different semantics

$$\text{statement:}\ \frac{n \in \text{Name},\quad v \in \text{Value}}{\text{localfunction}(n, v) \in \text{Statement}}$$

Declaring variables requires the name of the variable as well as a value that will be assigned to the variable. If no value is specified, then **nil** is implied.

$$\text{statement:}\ \frac{n \in \text{Name}}{\text{variable}(n) \in \text{Statement}} \qquad \text{statement:}\ \frac{n \in \text{Name},\quad v \in \text{Value}}{\text{variable}(n, v) \in \text{Statement}}$$

Declaring local variables is the same as declaring variables but, just like local function declaration, differs in semantics.

$$\text{statement:}\ \frac{n \in \text{Name}}{\text{localvariable}(n) \in \text{Statement}} \qquad \text{statement:}\ \frac{n \in \text{Name},\quad v \in \text{Value}}{\text{localvariable}(n, v) \in \text{Statement}}$$

The **return** statement returns one or more values from a function or a chunk

$$\text{statement:}\ \frac{es \in \text{ExpressionList}}{\text{return}(es) \in \text{Statement}}$$

With the abstract syntax defined, we now know the **form** that a Prolua program will take. For example, consider the following Lua program[1]

```
function toCelsius(fahrenheit)
   return (fahrenheit - 32)*(5 / 9);
end;

t = {min = 0, 0, 0, 0, max = 0}

t.min = toCelsius(5);
local i = 1;
while (i < 4) do
   t[i] = toCelsius(5^(i + 1));
   i = i + 1;
end;
t.max = toCelsius(5^5);

return t.min, t[1], t[2], t[3], t.max;
```

and its generated Prolua program (formatted for readability)

```
chunk([

assign([variable('toCelsius')], [functionbody([variable('fahrenheit')],
block([return([binop(multiply, binop(subtract, variable('fahrenheit'),
numbertype(32)), binop(divide, numbertype(5), numbertype(9)))])]))]),

assign([variable('t')], [tabletype([[stringtype('min'), numbertype(0)],
[numbertype(1), numbertype(0)], [numbertype(2), numbertype(0)], [numbertype(3),
numbertype(0)], [stringtype('max'), numbertype(0)]])]),

assign([access(variable('t'), stringtype('min'))],
[functioncall(variable('toCelsius'), [numbertype(5)])]),

localvariable('i', numbertype(1)),

while(binop(lt, variable('i'), numbertype(4)),
block([assign([access(variable('t'), variable('i'))],
[functioncall(variable('toCelsius'), [binop(exponent, numbertype(5), binop(add,
variable('i'), numbertype(1)))])]), assign([variable('i')], [binop(add,
variable('i'), numbertype(1))])])),

assign([access(variable('t'), stringtype('max'))],
[functioncall(variable('toCelsius'), [binop(exponent, numbertype(5),
numbertype(5))])]),

return([access(variable('t'), stringtype('min')), access(variable('t'),
numbertype(1)), access(variable('t'), numbertype(2)), access(variable('t'),
numbertype(3)), access(variable('t'), stringtype('max'))])

]).
```

We can see that the output is essentially a Prolog program with predicates that resemble the previously defined abstract syntax.

1 This is the **temperature.lua** program provided in the samples.

Chapter III. Semantics

As I mentioned before, the syntax describes the form of a Prolua program, but says nothing about its behavior...

[TODO] Describe Prolua semantics.