# PROLUA

**Working Draft**

Jeremy OTHIENO
https://github.com/supranove

# Chapter I. Introduction

Prolua is a simple Lua interpreter written in Prolog.

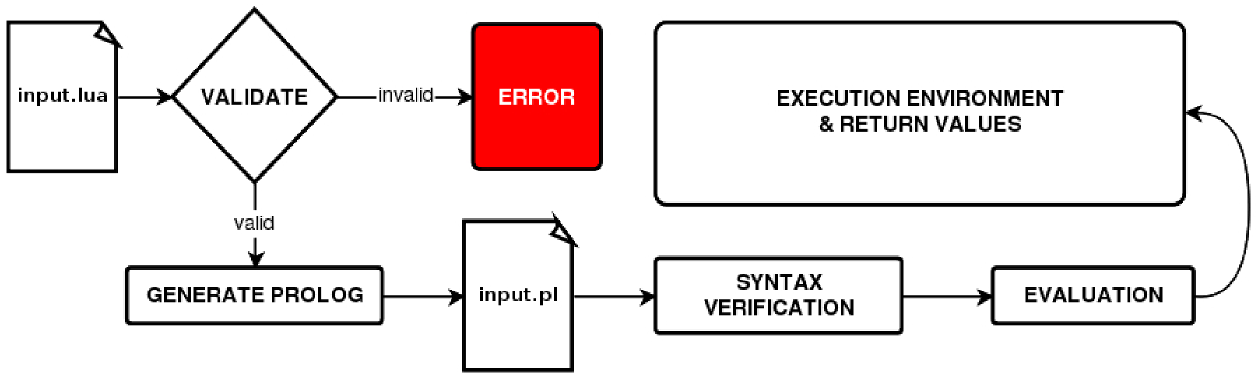


FIGURE I. A simplistic overview of the Prolua interpretation cycle.

## I.a. Why Lua?

Lua is a simple, well documented, and somewhat popular programming language. Its small concrete syntax coupled with a fairly straightforward and well-defined semantics makes it a relatively simple programming language to study.

## I.b. Why Prolog?

Prolog is a general purpose logic programming language geared towards a specific subset of problems. Computational linguistics is one such problem domain and this makes Prolog a particularly good choice for modelling natural languages and, as we'll soon find out, other programming languages. I could not seem to find any Lua interpreter implemented in Prolog, or any language that follows the logic programming paradigm for that matter. Challenge accepted.

## I.c. Constraints

At the moment, Prolua interprets Lua programs that adhere to the **Lua 5.1 specification only**. Lua programs written to work with Lua 5.0 and below have not been tested, and are not guaranteed to work within Prolua. Because I'm not familiar with some of the underpinnings in Lua 5.2 and above, most notably the change in how the environment is managed, Lua programs using features added in the 5.2 specification will not function either.

Most programming languages come with a standard library that provides extra features and functionality, greatly improving and in most cases facilitating, the use of the language. While Prolog does an excellent job at modelling languages, it isn't well suited for writing whole libraries. Although not impossible, completely rewriting Lua's standard library in Prolog is time consuming, and not the objective of this project. As such, Lua's standard library as well as its C API are not implemented. This means that of the eight basic types of values in Lua, only numbers, strings, booleans, nil, tables and functions will be implemented. The userdata and thread types will be excluded, as well as the features that depend on these types, such as coroutines. A few frequently used functions from the standard library will be implemented; please refer to §II.e. *Intrinsic functions* for more details.

Garbage collection is implemented via a reference counting algorithm, a relatively simple algorithm where each block in Prolua's memory pool keeps a count of how many references it has such that when this reference count reaches zero, the memory block is disposed of. The simplicity of this implementation comes at the cost of its efficiency. This subject is not in the scope of this document, and so I invite the reader to peruse this article [WIKIPEDIA] for more information.

## I.d. Notation

There is more than one notation used to formalise the semantics of a programming language, each having a specific methodology. They are

- **Denotational** semantics, where the meaning of a program is formalised by constructing mathematical objects to describe the meanings of expressions in the language.
- **Axiomatic** semantics, where the meaning of a program is formalised by a set of assertions about properties of a system and how they are affected by program execution.
- **Operational** semantics in which we verify properties of a program by constructing proofs from logical statements about its execution.

Operational semantics will be used to formalise Prolua's evaluation semantics since the concept behind it is similar to how one would prove a goal in Prolog by proving its subgoals.

The idea is to draw a **conclusion** from a conjunction of logical statements, or **premises**, such that a property is valid if and only if the truth of its premises logically entails the truth of its conclusion, and each step, sub-argument, or logical operation in the argument is valid.

As an example let's prove a simple property: the factorial of a positive integer which is defined by the following recurrence relation

$$\begin{cases} \mathbf{1} & \textbf{if } \mathbf{n = 0,} \\ \mathbf{(n - 1)! \times n} & \textbf{if } \mathbf{n > 0.} \end{cases}$$

This relation stipulates that **0!** is equal to one

$$\frac{}{\texttt{factorial(0)} \Rightarrow 1} \quad \text{where} \quad \frac{\texttt{premise}}{\texttt{conclusion}}$$

No premises are given for the base case **0!** since it is irreducible (it is not necessary to prove the value **1**). The relation then states that **n!** such that **n** is greater than zero, is equal to **n × (n - 1)!**. In the notation that will be used, this is written as

$$\frac{n > 0 \ \wedge \ \texttt{factorial}(n - 1) \Rightarrow m \ \wedge \ nm = n \times m}{\texttt{factorial}(n) \Rightarrow nm}$$

which is read as "if **n** is greater than zero, and **(n - 1)!** results in **m**, and **nm** is equal to **n × m** (the premises), then **n!** results in **nm** (the conclusion)". In Prolog, this can be written as

```prolog
factorial(0, 1).
factorial(N, NM) :-
   N > 0,
   N1 is N - 1,
   factorial(N1, M),
   NM is N * M.
```

Fairly straightforward, yes?

Prolog is a language that relies heavily on induction, so much so that constructing and manipulating complex data structures requires some sort of recursion.

When defining inductive data types, the concatenation operator, denoted as '**::**' (two juxtaposed colons), is used. The best way to explain it is by example so let's take the defintion of a list of expressions (documented later)

$$\texttt{expressions:}\frac{}{\texttt{[]} \in \texttt{Expressions}} \qquad \texttt{expressions:}\frac{\texttt{e} \in \texttt{Expression} \ \wedge \ \texttt{es} \in \texttt{Expressions}}{\texttt{e::es} \in \texttt{Expressions}}$$

The first proof says that an empty list '**[]**' (two juxtaposed brackets) is a list of expressions, while the second states that a list of expressions is also the concatenation of an expression with a list of expressions. To give a more concrete example, let's construct the list [1, 2, 3, 4, 5] step-by-step:

1. $L_0$ = []
2. $L_1$ = 5::$L_0$ = 5::[] = 5 (for simplicity, let's suppose that for all expressions **e**, **e**::[] = **e**)
3. $L_2$ = 4::$L_1$ = 4::5
4. $L_3$ = 3::$L_2$ = 3::4::5
5. $L_4$ = 2::$L_3$ = 2::3::4::5
6. $L_5$ = 1::$L_4$ = 1::2::3::4::5

where $L_i$ $\forall i$ is a list of expressions. It is also important to note that the concatenation operator always returns a list.

If you still have trouble understanding, think of how you would construct the same list in Prolog (without using the append predicate, of course).

# Chapter II. Syntax

The syntax of a programming language defines how expressions accepted by the language may be combined to **form** a correct program, disregarding any meaning implied by the program. Programming language syntaxes comes in two flavors:

- **Concrete** syntax in which the language's syntax defined by a context-free grammar.
- **Abstract** syntax which pertains to a specific evaluator of the language.

In simple terms, the concrete syntax consists of rules and expressions that define the way programs look like to the programmer while the abstract syntax, which derives from a given concrete syntax, defines the way programs look like to an evaluator.
It is important to note that the abstract syntax can vary from implementation to implementation so for example, the GNU C++ compiler may generate a completely different abstract syntax from Microsoft's Visual C++ compiler for the same C++ program; but both syntaxes *must* derive from C++'s concrete syntax. So for the reader that feels that my abstract syntax is a bit too cumbersome, feel free to change it to your liking.

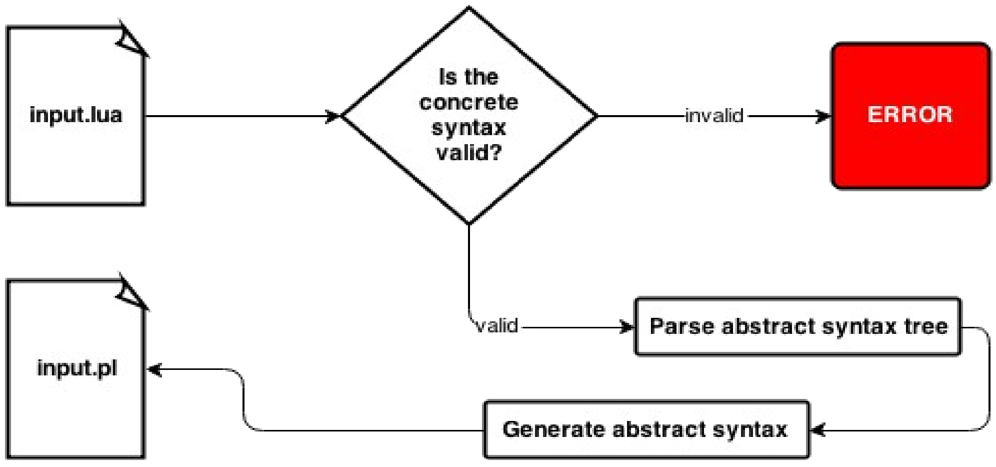


FIGURE II. From source code to abstract syntax: an illustration of what goes on between the **VALIDATE** and **GENERATE PROLOG** steps shown in FIGURE I.

[TODO] Explain the intermediate steps, especially why the AST is necessary wrt. source code analysis.

Our first order of business is to define the abstract syntax specific to Prolua so that we can have a general idea of the input that the Prolua evaluator will process. Lua's concrete syntax is given below in EBNF and the corresponding abstract syntax is detailed throughout the rest of this chapter.

```
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 | ‘#’
```

## II.a. Sets

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

$$\texttt{expressions:}\frac{}{\texttt{[]} \in \texttt{Expressions}} \qquad \texttt{expressions:}\frac{\texttt{e} \in \texttt{Expression} \;\land\; \texttt{es} \in \texttt{Expressions}}{\texttt{e::es} \in \texttt{Expressions}}$$

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

$$\texttt{names:}\frac{}{\texttt{[]} \in \texttt{Names}} \qquad \texttt{names:}\frac{\texttt{n} \in \texttt{Name} \;\land\; \texttt{ns} \in \texttt{Names}}{\texttt{n::ns} \in \texttt{Names}}$$

Let **Parameter**, the set of all possible parameter names and an extension of **Name** to include "**...**" be defined as **Name** $\cup$ **{...}**. Then let **Parameters** be a list of parameter names such that

$$\texttt{parameters:}\frac{}{\texttt{[]} \in \texttt{Parameters}} \qquad \texttt{parameters:}\frac{\texttt{p} \in \texttt{Parameter} \;\land\; \texttt{ps} \in \texttt{Parameters}}{\texttt{p::ps} \in \texttt{Parameters}}$$

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

$$\texttt{variables:}\frac{}{\texttt{[]} \in \texttt{Variables}} \qquad \texttt{variables:}\frac{\texttt{v} \in \texttt{Variable} \;\land\; \texttt{vs} \in \texttt{Variables}}{\texttt{v::vs} \in \texttt{Variables}}$$

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

$$\texttt{values:}\frac{}{\texttt{[]} \in \texttt{Values}} \qquad \texttt{values:}\frac{\texttt{v} \in \texttt{Value} \;\land\; \texttt{vs} \in \texttt{Values}}{\texttt{v::vs} \in \texttt{Values}}$$

Also, let **ObjectValue** be the subset of Value that contains only **tables** and **functions**.

Let **Statement** be the set of all possible statements in Lua, and **Statements** a list of statements such that

$$\texttt{statements:}\frac{}{\texttt{[]} \in \texttt{Statements}} \qquad \texttt{statements:}\frac{\texttt{s} \in \texttt{Statement} \;\land\; \texttt{ss} \in \texttt{Statements}}{\texttt{s::ss} \in \texttt{Statements}}$$

Let **Reference** be the set of all references to tables and functions

$$\texttt{reference:}\frac{\texttt{type} \in \{\texttt{table, function}\} \;\land\; \texttt{address} \in \mathbb{Z}_+}{\texttt{referencetype(type, address)} \in \texttt{Reference}}$$

Let **Dictionary**$_{T,U}$ be a set of ⟨**key, value**⟩ pairs where **T** and **U** are the type sets for **key** and **value**, respectively

$$\texttt{dictionary:}\frac{}{\texttt{[]} \in \texttt{Dictionary}_{T,\,U}} \qquad \texttt{dictionary:}\frac{\texttt{k} \in \texttt{T} \;\land\; \texttt{v} \in \texttt{U} \;\land\; \texttt{d} \in \texttt{Dictionary}_{T,\,U}}{\langle \texttt{k, v} \rangle\texttt{::d} \in \texttt{Dictionary}_{T,\,U}}$$

## II.b. Values and Types

Since Lua is a dynamically-typed language, values are responsible for keeping their types, and not variables. This is the same in Prolua as you'll come to see that values are encapsulated within their assigned type.

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

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

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

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

**Number** represents **real** (mathematically) numbers

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

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. We'll suppose that the character set is the set of all 8-bit ASCII characters. A string is then considered to be a concatenation of characters

$$\text{string}: \frac{}{[\,] \in \texttt{String}} \qquad \text{string}: \frac{c \in \texttt{Character} \ \land \ s \in \texttt{String}}{c :: s \in \texttt{String}} \qquad \text{value}: \frac{s \in \texttt{String}}{\texttt{stringtype}(s) \in \texttt{Value}}$$

The type **table** implements associative arrays, a set of ⟨**key, value**⟩ pairs known as **fields** where each **key** can be an expression except **nil**. A metatable can also be assigned to a table.
Let **TableFieldKey**, the set of all possible field keys, be defined as **Name** ∪ **Expression** \ **{nil}**

$$\text{table}: \frac{fs \in \texttt{Dictionary}_{\texttt{TableFieldKey, Expression}} \ \land \ r \in \texttt{Reference}}{\texttt{table}(fs, r) \in \texttt{Object}}$$

A **function** or **closure**, holds a list of parameter names, a block of statements, and a path to the execution context it was defined in

$$\text{function}: \frac{ps \in \texttt{Parameters} \ \land \ ss \in \texttt{Statements} \ \land \ \sigma_{\texttt{function}} \in \texttt{Path}}{\texttt{function}(ps, ss, \sigma_{\texttt{function}}) \in \texttt{Object}}$$

In Lua, **tables** and **functions** are objects: variables do not contain these values but **references** to them. In Prolua, a reference is a ⟨**type, address**⟩ pair where **type** is the type of the object being referenced, while **address** is its memory address

$$\text{value}: \frac{type \in \{\texttt{table}, \texttt{function}\} \ \land \ address \in \mathbb{Z}_{+}}{\texttt{referencetype}(type, address) \in \texttt{Value}}$$

For more information on execution **contexts**, context **paths** and memory **addresses**, see §III. *Semantics*.

## II.c. Expressions

The **table constructor** expression creates a new table from a list of fields or a list of expressions

$$\text{expression}: \frac{\texttt{fs} \in \texttt{Dictionary}_{\texttt{TableFieldKey, Expression}}}{\texttt{tableconstructor(fs)} \in \texttt{Expression}} \qquad \text{expression}: \frac{\texttt{es} \in \texttt{Expressions}}{\texttt{tableconstructor(es)} \in \texttt{Expression}}$$

An expression can be **enclosed** in parentheses

$$\text{expression}: \frac{\texttt{e} \in \texttt{Expression}}{\texttt{enclosed(e)} \in \texttt{Expression}}$$

**Variables** and **local variables** access a location in the execution environment where values can be stored or read from

$$\text{expression}: \frac{\texttt{n} \in \texttt{Name}}{\texttt{variable(n)} \in \texttt{Expression}} \qquad \text{expression}: \frac{\texttt{n} \in \texttt{Name}}{\texttt{localvariable(n)} \in \texttt{Expression}}$$

We define the **access** expression which, much like the **variable** expression, accesses a memory address. This time, the address corresponds to a **table field** indexed with a given key

$$\text{expression}: \frac{e_r \in \texttt{Expression} \ \wedge \ e_k \in \texttt{Expression} \setminus \{\texttt{niltype(nil)}\}}{\texttt{access}(e_r,\ e_k) \in \texttt{Expression}}$$

A **variadic expression**, represented by three dots "**...**", is a placeholder for a list of values

$$\text{expression}: \frac{}{\texttt{...} \in \texttt{Expression}}$$

A **unary operator** contains the name of the operator and the operand expression

$$\text{expression}: \frac{\texttt{op} \in \{\texttt{unm, not, len}\} \ \wedge \ \texttt{e} \in \texttt{Expression}}{\texttt{unop(op, e)} \in \texttt{Expression}}$$

Almost like a unary operator, a **binary operators** contains the name of the operator and two operand expressions

$$\text{expression}: \frac{\begin{gathered}\texttt{op} \in \{\texttt{add, sub, mul, div, mod, pow, eq, lt, le, and, or, concat}\} \\ e_{lhs}, e_{rhs} \in \texttt{Expression}\end{gathered}}{\texttt{binop}(\texttt{op}, e_{lhs}, e_{rhs}) \in \texttt{Expression}}$$

A **function definition** creates a closure from a list of parameter names and statements

$$\text{expression}: \frac{\texttt{ps} \in \texttt{Parameters} \ \wedge \ \texttt{ss} \in \texttt{Statements}}{\texttt{functiondef(ps, ss)} \in \texttt{Expression}}$$

**Function calls** require an expression that evaluates into a callable object and a list of expressions that will be used as function arguments

$$\text{expression}: \frac{\texttt{e} \in \texttt{Expression} \ \wedge \ \texttt{es} \in \texttt{Expressions}}{\texttt{functioncall(e, es)} \in \texttt{Expression}}$$

## II.d. Statements

The unit of execution in Lua, and therefore Prolua, is called a **chunk** which is a sequence of statements that are executed sequentially. Lua handles a chunk as the body of an anonymous function with a variable number of arguments

$$\texttt{chunk:}\frac{\texttt{ss} \in \texttt{Statements}}{\texttt{chunk(ss)}}$$

The **assignment** statement in Lua allows for multiple assignments in one call. Lua's syntax defines a list of variables on the left side and another list of expressions on the right but in Prolua, these will both be lists of expressions that evaluate into memory addresses and values, respectively

$$\texttt{statement:}\frac{\texttt{es}_{\texttt{lhs}}, \texttt{es}_{\texttt{rhs}} \in \texttt{Expressions}}{\texttt{assign}(\texttt{es}_{\texttt{lhs}}, \ \texttt{es}_{\texttt{rhs}}) \in \texttt{Statement}}$$

**Function calls** were previously defined as expressions but can also be executed as statements, in which case all return values except errors are discarded

$$\texttt{statement:}\frac{\texttt{e} \in \texttt{Expression} \ \wedge \ \texttt{es} \in \texttt{Expressions}}{\texttt{functioncall(e, es)} \in \texttt{Statement}}$$

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

$$\texttt{statement:}\frac{\texttt{ss} \in \texttt{Statements}}{\texttt{do(ss)} \in \texttt{Statement}}$$

The **while-do** statement executes a block of instructions while the condition expression is true

$$\texttt{statement:}\frac{\texttt{e} \in \texttt{Expression} \ \wedge \ \texttt{ss} \in \texttt{Statements}}{\texttt{while(e, ss)} \in \texttt{Statement}}$$

A **repeat-until** statement executes a block of instructions until the condition expression is true

$$\texttt{statement:}\frac{\texttt{e} \in \texttt{Expression} \ \wedge \ \texttt{ss} \in \texttt{Statements}}{\texttt{repeat(e, ss)} \in \texttt{Statement}}$$

An **if-else** conditional statement evaluates one of two statements based on a condition expression

$$\texttt{statement:}\frac{\texttt{e} \in \texttt{Expression} \ \wedge \ \texttt{s}_{\texttt{true}}, \texttt{s}_{\texttt{false}} \in \texttt{Statement}}{\texttt{if}(\texttt{e}, \texttt{s}_{\texttt{true}}, \texttt{s}_{\texttt{false}}) \in \texttt{Statement}}$$

In Lua, **for** loops come in two flavors. The first is the **numeric for** statement which repeats a block of code while a control variable runs through an arithmetic progression, and the second is the **generic for** statement which works over iterator functions in such a way that on each iteration, the iterator function is called to produce a new value, halting the loop when this value is **nil**.
The Lua documentation details workarounds for both types using a **while-do** statement and so no abstract syntax for either statement is specified in Prolua.

The **return** statement returns zero or more values from a function

$$\texttt{statement:}\ \frac{\texttt{es} \in \texttt{Expressions}}{\texttt{return(es)} \in \texttt{Statement}}$$

The **break** statement explicitly breaks a loop

$$\texttt{statement:}\ \frac{}{\texttt{break} \in \texttt{Statement}}$$

## II.e. Intrinsic functions

Intrinsic functions are functions implemented in Lua's standard library without which some of the features described in Lua's reference manual would not work, or because I thought they were so common they needed to be included in Prolua. These functions are directly implemented (hardcoded) into Prolua's semantics which is why there's only a handful of them. They are:

- The **type** function returns the type of an expression.
- The **tonumber** function converts an expression into a numeric value. It is necessary for handling type coercion between numbers and string.
- The **tostring** function converts an expression into a string of characters. It is used by the **print** function, which prints a string to the standard output.
- The **error** and **assert** functions are used for error handling.
- The **getmetatable** and **setmetatable** functions are used to manipulate metatables.
- The **rawget** and **rawset** functions are used to get and set values in a table, respectively, while bypassing any defined metamethods.
- The **ipairs**, **next** and **pairs** iterator functions are mainly used in the generic for loop.

The **next** iterator is directly implemented as part of Prolua's semantics. Its pseudocode is

```
function next(table, key)
   result := nil
   if table is not empty then
      if key is nil then
         result := the first <k, v> entry in table
      else
         if key exists in table then
            field := the <k, v> entry where k = key
            if field is not the last <k, v> entry in table then
               result := the <k, v> entry that comes after field in table
            end if
         else
            result := Error! An invalid key was passed to 'next'
         end if
      end if
   end if
   return result
end function
```

The **pairs** iterator is inspired by the following Lua code

```
function pairs(table)
   return next, table, nil
end
```

The **ipairs** iterator function is generated by **lua2prolog** based on the following Lua code

```
function ipairs(table)
   return function(table, index)
      index = index + 1
      local value = table[index]
      if value then
         return index, value
      end
   end, table, 0
end
```

The remaining intrinsics will not be enumerated but for more information, I suggest reading through the **semantics.pl** source file for their implementations.

Now let's consider the following Lua program[1] passed to **lua2prolog** with no command line arguments...

```
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
```

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

...and the generated abstract syntax (formatted for readability)

```
chunk([
assign([variable('toCelsius')], [functiondef(['fahrenheit'], [return([binop(mul,
enclosed(binop(sub, variable('fahrenheit'), numbertype(32))), enclosed(binop(div,
numbertype(5), numbertype(9))))])])]),

assign([variable('t')], [tableconstructor(fields([[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)])]),

assign([localvariable('i')], [numbertype(2)]),

while(enclosed(binop(lt, variable('i'), numbertype(4))), [assign([access(variable('t'),
variable('i'))], [functioncall(variable('toCelsius'), [binop(pow, numbertype(5),
enclosed(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(pow, 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'))])
]).
arguments([]).
```

Notice just how much Prolua's abstract syntax differs from Lua's concrete syntax?

It's clearly not a very smart idea to manually write the abstract syntax of a Prolua program and this example illustrates one of the fundamental differences between each syntax type: the concrete syntax is meant to be easily read and processed by a human being while the abstract syntax is meant to be easy for the evaluator to process, usually at the expense of readability.

Now that we have a valid Prolua program -- a base clause that states that a **chunk** is a list of terms, and another that states that there're no **arguments** -- there is still no way of knowing what this program actually means since no relationships between the terms have been defined. This is where semantics will come into play.

# Chapter III. Semantics

As I mentioned earlier, the abstract syntax dictates the form of a valid Prolua program, describing nothing about its **behavior** or **usage restrictions**. Suppose we have the following term

```
unop(len, niltype(nil))
```

The expression is syntactically correct since the length operator expects an expression as its operand, but semantically flawed because the length of a nil value cannot be calculated. The semantics of a language is thereby responsible for checking for correct usage of elements in the language so in the above example, the operand is restricted to a subset of expressions, namely strings and tables. Now suppose we fix the expression by changing the nil value to a string

```
unop(len, stringtype('Hello, World'))
```

The expression is now syntactically and semantically correct, but what is the result of its evaluation? Zero, eight, a boolean value, a function, or maybe even the answer to life? We simply do not know because as it stands, the expression is still nothing more than a string of text. The goal of the semantics of a language is to **give a meaning** to this textual form by defining **how** it should be evaluated and eventually return a value, if any.

Before the evaluation semantics of expressions and statements in Prolua can be defined, it is necessary to detail the environment in which the execution of a program occurs. This shall be known as the **execution environment** in Prolua and it is composed of a **memory pool** and **execution contexts**.

## III.a. The Execution Context

The execution environment contains multiple execution contexts which are used to implement **lexical scoping**. An execution context has a storage duration, as well as a symbol table.

The symbol table is a dictionary of ⟨**identifier, value**⟩ pairs where **identifier** ∈ **Parameter**, is a variable and **value** ∈ **Value** is the value assigned to it.

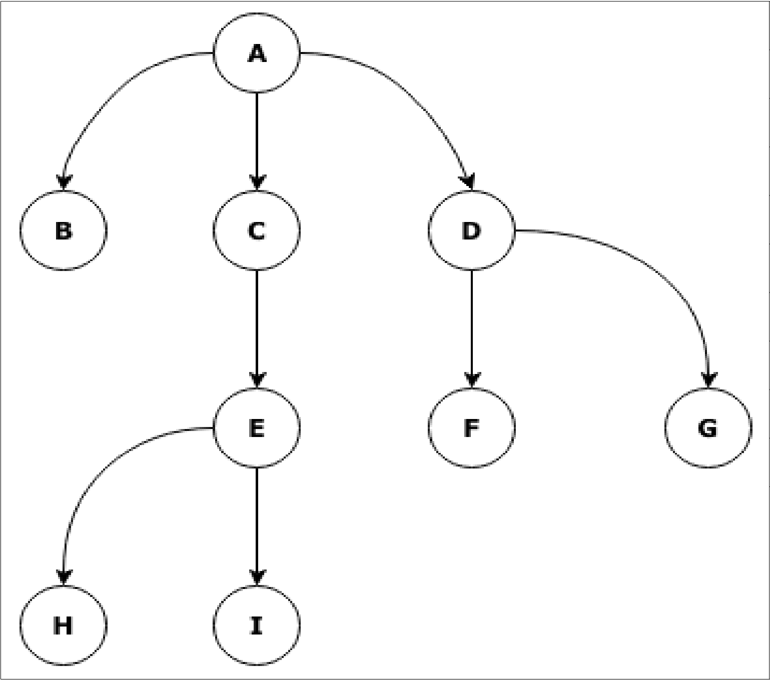
When a context is created, it has a lifetime during which it can exist in the memory before being disposed of. This lifetime depends on whether or not a closure is defined in the context, i.e. if a closure is defined in a given context, then that context should exist as long as the closure exists. If not, then the execution context is disposed of automatically when it is no longer needed.

$$\texttt{context:}\frac{\texttt{symbols} \in \texttt{Dictionary}_{\texttt{Parameter, Value}},\quad \lambda \in \mathbb{Z}_+}{\texttt{context(symbols, }\lambda\texttt{)} \in \texttt{Context}}$$

Execution contexts are stored in the form of a directed acyclic graph where each node is an execution context and is accessed via a **path** from the root node. Syntactically, a path is a list of positive non-zero integers

$$\texttt{path:}\frac{\texttt{i} \in \mathbb{Z}_+}{\texttt{path(i::[])} \in \texttt{Path}} \qquad \texttt{path:}\frac{\texttt{i} \in \mathbb{Z}_+,\ \texttt{is} \in \texttt{Path}}{\texttt{path(i::is)} \in \texttt{Path}}$$

Suppose we have the following diagram of an execution context graph, as well as paths to each execution context.



| NODE | PATH |
|---:|:---|
| A | [1] |
| B | [1, 1] |
| C | [1, 2] |
| D | [1, 3] |
| E | [1, 2, 1] |
| F | [1, 3, 1] |
| G | [1, 3, 2] |
| H | [1, 2, 1, 1] |
| I | [1, 2, 1, 2] |

FIGURE III. An example of an execution context graph and paths to its corresponding nodes.

To get from node A to G for example, we need to follow the path A → D → G. A is the first root node in the graph therefore it has the reference [1]; D is A's third child node (from left to right) so the path from A to D is [1, 3]; G is D's second child node so the path from A to D to G is [1, 3, 2].

[TODO1] Give a more concrete example with a simple Lua program.
[TODO2] Explain context switching wrt to stack data structures. (graph node's depth).

## III.b. The Memory Pool

The memory pool is a collection of contiguous memory blocks where the execution context graph and object data (tables and functions) are stored. A memory block is defined as a **⟨address, reference count, data⟩** triplet where **address** is a unique positive integer used to identify a memory block, **reference count** is the number of references made to the block, and **data** is what is stored in the block

$$\text{memoryblock}: \frac{\text{address},\ \text{references} \in \mathbb{Z}_+,\quad \text{data} \in \text{Context} \cup \text{ObjectValue}}{\langle \text{address},\ \text{references},\ \text{data} \rangle \in \text{MemoryBlock}}$$

A collection of memory blocks is defined syntactically as

$$\text{memoryblocks}: \frac{}{[] \in \text{MemoryBlocks}} \qquad \text{memoryblocks}: \frac{\text{b} \in \text{MemoryBlock},\quad \text{bs} \in \text{MemoryBlocks}}{\text{b::bs} \in \text{MemoryBlocks}}$$

The memory **pool** contains an offset counter which is used to determine the address of the currently available memory block to be allocated to an object, and this offset is incremented each time a new memory block is occupied. When a block is deleted from the pool, the counter is not decremented since previous memory addresses will be overwritten.

$$\text{pool}: \frac{\text{offset} \in \mathbb{Z}_+,\quad \text{blocks} \in \text{MemoryBlocks}}{\text{pool(offset, blocks)} \in \text{Pool}}$$

The memory pool is a **linear** data structure that grows whenever a new object is added to it. If a pool is composed of N memory blocks, then its search complexity is *O*(N), which becomes very inefficient for large values of N. This is a pretty significant limitation and is the reason Prolua should be used solely for educational purposes.

A convincing example of why linear data structures pose a problem can be seen when calculating large Fibonacci numbers with the **closures.lua** script provided in this project's samples. The relatively long execution time is caused by rapid expansion of an execution context's child nodes as each recursive call of the closure creates a new closure, each with its own execution context.

The **execution environment** can now be formally defined as

$$\text{environment}: \frac{\sigma \in \text{Path},\quad \rho \in \text{Pool}}{\langle \sigma,\ \rho \rangle \in \text{Environment}}$$

In cases where it is not necessary to detail the execution environment, **⟨α, ρ⟩** is shortened to **$\mathbf{ENV}_i$**, where **i** is the ith state of the execution environment.

A few predicates that help manipulate the environment are defined in the **environment.pl** file. They are not heavily documented but understanding them should be relatively easy, and is left as an exercise to the reader.

## III.c. Expression evaluation

The evaluation of an expression is a **⟨ENV$_0$, expression, ENV$_n$, result⟩** quadruplet where

- **ENV$_0$** is the initial execution environment in which the expression will be evaluated,
- **expression** is an expression to evaluate,
- **ENV$_n$** is the state of the execution environment after evaluation of **expression**, and
- **result** is a n-tuple of zero or more values returned by the evaluation of **expression**.

Lua is a language with assignable variables (and table fields) which means it has two kinds of expressions: **left-hand side expressions** which return **addresses** where data can be stored and read from; and **right-hand side expressions** which return **values**. The evaluation of a left-hand side expression is denoted by $\overset{\text{lhs}}{\Rightarrow}$ and its **result** is a **⟨locator, key⟩** pair hereinafter referred to as an **lvalue**, where **locator** is either a memory pool address, or a reference to an object in the pool; and **key** is a variable name or a non-nil expression.
Right-hand side expression evaluation is denoted by $\overset{\text{rhs}}{\Rightarrow}$ and its **result** is a list of zero or more values.

It is also important to note that every Lua expression is a right-hand side expression, but only the **variable** and **table field access** expressions can return **lvalues**, making them left-hand side expressions too.

Let's start off with the evaluation of a **list of left-hand side expressions**. Evaluating an empty list of left-hand side expressions in a given execution environment returns an empty list of lvalues, and does not modify the environment.

$$\frac{}{\langle \mathtt{ENV}_0,\ \mathtt{expressions([])}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ENV}_0,\ \mathtt{[]}\rangle}$$

If the list of expressions has one or more elements, then it is evaluated recursively

$$\frac{\langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \langle \mathtt{locator},\ \mathtt{key}\rangle\rangle \land \langle \mathtt{ENV}_1,\ \mathtt{expressions(es)}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{lvalues}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{expressions(e::es)}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \langle \mathtt{locator},\ \mathtt{key}\rangle\mathtt{::lvalues}\rangle}$$

However, if an error occurs while evaluating the first expression, then the remaining expressions are discarded and the error is returned

$$\frac{\langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathbf{ERROR}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{expressions(e::es)}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathbf{ERROR}\rangle}$$

Similarly, if we evaluate an expression that successfully returns an address then evaluate the next expression which returns an error, all previous lvalues are discarded and the error is returned

$$\frac{\langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \langle \mathtt{locator},\ \mathtt{key}\rangle\rangle \land \langle \mathtt{ENV}_1,\ \mathtt{expressions(es)}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathbf{ERROR}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{expressions(e::es)}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathbf{ERROR}\rangle}$$

The environment is never discarded because it's useful to know its state when debugging the error.

**DOCUMENTATION PASSED THIS POINT HAS NOT BEEN REVISED**

**Evaluating a left-hand side variable** returns the lvalue **⟨path, name⟩** where **path** is the path to the execution context in which the variable exists, and **name** is the identifier corresponding to the variable in the execution context. If the variable name exists in the current execution context, then we return the execution context's identifer and the variable name

$$\frac{\mathtt{getContext}(\sigma,\ \rho) \Rightarrow \Phi \ \wedge\ \mathtt{keyExists}(\Phi,\ \mathtt{n}) \Rightarrow \mathtt{true}}{\langle\langle\sigma,\ \rho\rangle,\ \mathtt{variable(n)}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle\langle\sigma,\ \rho\rangle,\ \langle\sigma,\ \mathtt{n}\rangle\rangle}$$

If the variable name does not exist in the current execution context, then we check the previous execution context. This is analogous to looking for the variable in an outer scope

$$\frac{\begin{array}{c}\mathtt{getContext}(\sigma,\ \rho) \Rightarrow \Phi \ \wedge\ \mathtt{keyExists}(\Phi,\ \mathtt{n}) \Rightarrow \mathtt{false} \ \wedge\ \mathtt{popContext}(\sigma) \Rightarrow \sigma_1 \mid \sigma_1 \neq [\,] \ \wedge \\ \langle\langle\sigma_1,\ \rho\rangle,\ \mathtt{variable(n)}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle\langle\sigma_1,\ \rho\rangle,\ \langle\mathtt{locator},\ \mathtt{n}\rangle\rangle\end{array}}{\langle\langle\sigma,\ \rho\rangle,\ \mathtt{variable(n)}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle\langle\sigma,\ \rho\rangle,\ \langle\mathtt{locator},\ \mathtt{n}\rangle\rangle}$$

A key always exists in the global execution context -- the first context to be pushed onto the stack and the last to be popped -- even if it isn't explicitly defined

$$\frac{\mathtt{getContext}(\sigma,\ \rho) \Rightarrow \Phi \ \wedge\ \mathtt{keyExists}(\Phi,\ \mathtt{n}) \Rightarrow \mathtt{false} \ \wedge\ \mathtt{popContext}(\sigma) \Rightarrow [\,]}{\langle\langle\sigma,\ \rho\rangle,\ \mathtt{variable(n)}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle\langle\sigma,\ \rho\rangle,\ \langle\sigma,\ \mathtt{n}\rangle\rangle}$$

**Evaluating a left-hand side local variable** is as simple as returning the path to the current execution context and the variable's identifier

$$\frac{}{\langle\langle\sigma,\ \rho\rangle,\ \mathtt{localvariable(n)}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle\langle\sigma,\ \rho\rangle,\ \langle\sigma,\ \mathtt{n}\rangle\rangle}$$

**Finding the lvalue of an indexed table field** is not as straightforward as finding the lvalue of a variable. The expression has two operands, both expressions, which should evaluate into a reference to a table and a table field key, respectively

$$\frac{\langle\mathtt{ENV}_0,\ \mathtt{e}_r\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle\mathtt{ENV}_1,\ \mathtt{r}::[\,]\rangle \ \wedge\ \mathtt{getObjectType(r)} \Rightarrow \mathtt{table} \ \wedge\ \langle\mathtt{ENV}_1,\ \mathtt{e}_k\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle\mathtt{ENV}_2,\ \mathtt{v}_k::\mathtt{vs}\rangle}{\langle\mathtt{ENV}_0,\ \mathtt{access}(\mathtt{e}_r,\ \mathtt{e}_k)\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle\mathtt{ENV}_2,\ \langle\mathtt{r},\ \mathtt{v}_k\rangle\rangle}$$

If evaluating the first expression returns a reference to an object, but the object is not a table, then an error is returned

$$\frac{\langle\mathtt{ENV}_0,\ \mathtt{e}_r\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle\mathtt{ENV}_1,\ \mathtt{r}::[\,]\rangle \ \wedge\ \mathtt{getObjectType(r)} \Rightarrow \mathtt{type} \mid \mathtt{type} \neq \mathtt{table}}{\langle\mathtt{ENV}_0,\ \mathtt{access}(\mathtt{e}_r,\ \mathtt{e}_k)\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle\mathtt{ENV}_2,\ \mathbf{ERROR}\rangle}$$

If evaluating the first expression results in an error, then that error is returned

$$\frac{\langle\mathtt{ENV}_0,\ \mathtt{e}_r\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle\mathtt{ENV}_1,\ \mathbf{ERROR}\rangle}{\langle\mathtt{ENV}_0,\ \mathtt{access}(\mathtt{e}_r,\ \mathtt{e}_k)\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle\mathtt{ENV}_1,\ \mathbf{ERROR}\rangle}$$

Likewise, if the second expression results in an error then the error must be returned

$$\frac{\langle\mathtt{ENV}_0,\ \mathtt{e}_r\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle\mathtt{ENV}_1,\ \mathtt{r}::[\,]\rangle \ \wedge\ \mathtt{getObjectType(r)} \Rightarrow \mathtt{table} \ \wedge\ \langle\mathtt{ENV}_1,\ \mathtt{e}_k\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle\mathtt{ENV}_2,\ \mathbf{ERROR}\rangle}{\langle\mathtt{ENV}_0,\ \mathtt{access}(\mathtt{e}_r,\ \mathtt{e}_k)\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle\mathtt{ENV}_2,\ \mathbf{ERROR}\rangle}$$

Evaluating a **list of right-hand side expressions** returns a list of values, but not just anyhow. For the sake of brevity, the semantics for evaluating an empty list of right-hand side expressions as well as error management are skipped as they are quasi-identical to the semantics defined in the evaluation of a list of left-hand side expressions.
If the last or only expression in a list of right-hand side expressions is evaluated and it happens to return a list of values, then all the values are returned

$$\frac{\langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{values}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{expressions(e::[])}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{values}\rangle}$$

In all other cases, if the result of the evaluation of the right-hand side expression returns a list containing more than one value, then this list is truncated to only one element thereby discarding all values but the first. The list is then prepended to the result of the evaluation of the remaining expressions in the list

$$\frac{\mathtt{es} \neq \mathtt{[]} \ \wedge\ \langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{v}_{\mathtt{e}}\mathtt{::values}\rangle \ \wedge\ \langle \mathtt{ENV}_1,\ \mathtt{expressions(es)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{values}_{\mathtt{es}}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{expressions(e::es)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{v}_{\mathtt{e}}\mathtt{::values}_{\mathtt{es}}\rangle}$$

If an expression is not the last in the list of expressions to evaluate, and it returns an empty list of values, a **nil** value must explicitly be added to the list of return values

$$\frac{\mathtt{es} \neq \mathtt{[]} \ \wedge\ \langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{[]}\rangle \ \wedge\ \langle \mathtt{ENV}_1,\ \mathtt{expressions(es)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{values}_{\mathtt{es}}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{expressions(e::es)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{niltype(nil)::values}_{\mathtt{es}}\rangle}$$

A **value** is an expression in normal form, i.e. it cannot be evaluated any further. As such, evaluating a value simply returns a list containing that single value

$$\frac{}{\langle \mathtt{ENV}_0,\ \mathtt{value}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_0,\ \mathtt{value::[]}\rangle \mid \mathtt{value} \in \mathtt{Value}}$$

The **table constructor** creates a new table from a list of fields, stores it in the memory pool then returns a reference to it

$$\frac{\begin{array}{c}\mathtt{fs} \neq \mathtt{[]} \ \wedge\ \mathtt{fs} \in \mathtt{Fields} \ \wedge\ \langle\langle \sigma,\ \rho\rangle,\ \mathtt{fields(fs)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle\langle \sigma,\ \rho_1\rangle,\ \mathtt{map}\rangle \ \wedge \\ \mathtt{allocate}(\rho_1,\ \mathtt{table(map)}) \Rightarrow \langle \rho_2,\ \mathtt{address}\rangle\end{array}}{\langle\langle \sigma,\ \rho\rangle,\ \mathtt{tableconstructor(fs)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle\langle \sigma,\ \rho_2\rangle,\ \mathtt{referencetype(table,\ address)::[]}\rangle}$$

From the definition of a list of fields given in the concrete syntax, we can deduce that a list of expressions is a subset of a list of fields. Consequently, the **table constructor** can also create a new table from a list of expressions. Remember though, that evaluating a list of expressions generates a list of values only. This means that a key must be generated for each value we wish to store in the table

$$\frac{\begin{array}{c}\mathtt{es} \neq \mathtt{[]} \ \wedge\ \mathtt{es} \in \mathtt{Expressions} \ \wedge\ \langle\langle \sigma,\ \rho\rangle,\ \mathtt{expressions(es)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle\langle \sigma,\ \rho_1\rangle,\ \mathtt{values}\rangle \ \wedge \\ \mathtt{buildMap(values)} \Rightarrow \mathtt{map} \ \wedge\ \mathtt{allocate}(\rho_1,\ \mathtt{table(map)}) \Rightarrow \langle \rho_2,\ \mathtt{address}\rangle\end{array}}{\langle\langle \sigma,\ \rho\rangle,\ \mathtt{tableconstructor(es)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle\langle \sigma,\ \rho_2\rangle,\ \mathtt{referencetype(table,\ address)::[]}\rangle}$$

If the evaluation of the list of expressions returns an error, then that error will be propagated

$$\frac{\texttt{es} \neq [\,] \;\wedge\; \texttt{es} \in \texttt{Expressions} \;\wedge\; \langle\langle\sigma,\ \rho\rangle,\ \texttt{expressions(es)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle\langle\sigma,\ \rho_1\rangle,\ \textbf{ERROR}\rangle}{\langle\langle\sigma,\ \rho\rangle,\ \texttt{tableconstructor(es)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle\langle\sigma,\ \rho_1\rangle,\ \textbf{ERROR}\rangle}$$

And trivially, if the **table constructor** is passed an empty list of fields, then it creates an empty table in the memory pool

$$\frac{\texttt{allocate}(\rho,\ \texttt{table([]))} \Rightarrow \langle\rho_1,\ \texttt{address}\rangle}{\langle\langle\sigma,\ \rho\rangle,\ \texttt{tableconstructor(es)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle\langle\sigma,\ \rho_1\rangle,\ \texttt{referencetype(table, address)::[]}\rangle}$$

Expressions can be **enclosed in parentheses** which means that in case the enclosed expression evaluates into a list of values, only the first is returned. It stands to reason that if a list with less than two values is returned, no truncation is made

$$\frac{\langle \texttt{ENV}_0,\ \texttt{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ENV}_1,\ \texttt{v::values}\rangle}{\langle \texttt{ENV}_0,\ \texttt{enclosed(e)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ENV}_1,\ \texttt{v::[]}\rangle} \qquad \frac{\langle \texttt{ENV}_0,\ \texttt{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ENV}_1,\ \texttt{[]}\rangle}{\langle \texttt{ENV}_0,\ \texttt{enclosed(e)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ENV}_1,\ \texttt{[]}\rangle}$$

And like always, if the evaluation of the expression results in an error, the error is returned

$$\frac{\langle \texttt{ENV}_0,\ \texttt{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ENV}_1,\ \textbf{ERROR}\rangle}{\langle \texttt{ENV}_0,\ \texttt{enclosed(e)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ENV}_1,\ \textbf{ERROR}\rangle}$$

To be able to **return the value of a right-hand side variable**, we need to know its **lvalue** from which the value can then be retrieved

$$\frac{\langle \texttt{ENV}_0,\ \texttt{variable(n)}\rangle \overset{\text{lhs}}{\Rightarrow} \langle \texttt{ENV}_1,\ \langle \texttt{ECID},\ \texttt{n}\rangle\rangle \;\wedge\; \texttt{getValue}(\texttt{ENV}_1,\ \texttt{ECID},\ \texttt{n}) \Rightarrow \texttt{value}}{\langle \texttt{ENV}_0,\ \texttt{variable(n)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ENV}_1,\ \texttt{value::[]}\rangle}$$

If there's an error finding the **lvalue**, return it

$$\frac{\langle \texttt{ENV}_0,\ \texttt{variable(n)}\rangle \overset{\text{lhs}}{\Rightarrow} \langle \texttt{ENV}_1,\ \textbf{ERROR}\rangle}{\langle \texttt{ENV}_0,\ \texttt{variable(n)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ENV}_1,\ \textbf{ERROR}\rangle}$$

And just like variables, **returning the value of a table field** at a given index is made simple once we know the field's **lvalue**. All we have to do is find the actual table, and then the field in the table

$$\frac{\langle \texttt{ENV}_0,\ \texttt{access}(\texttt{e}_r,\ \texttt{e}_k)\rangle \overset{\text{lhs}}{\Rightarrow} \langle \texttt{ENV}_1,\ \langle \texttt{r},\ \texttt{k}\rangle\rangle \;\wedge\; \texttt{getField}(\texttt{ENV}_1,\ \texttt{r},\ \texttt{k}) \Rightarrow \texttt{value}}{\langle \texttt{ENV}_0,\ \texttt{access}(\texttt{e}_r,\ \texttt{e}_k)\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ENV}_1,\ \texttt{value::[]}\rangle}$$

We do have to watch out for errors

$$\frac{\langle \texttt{ENV}_0,\ \texttt{access}(\texttt{e}_r,\ \texttt{e}_k)\rangle \overset{\text{lhs}}{\Rightarrow} \langle \texttt{ENV}_1,\ \textbf{ERROR}\rangle}{\langle \texttt{ENV}_0,\ \texttt{access}(\texttt{e}_r,\ \texttt{e}_k)\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ENV}_1,\ \textbf{ERROR}\rangle}$$

**Variadic expressions** evaluate into a list of values. The evaluation of a variadic expression consists of finding out whether the key "**...**" exists in the execution environment. If it exists in the current execution context, then its values are returned

$$\frac{\mathtt{getContext}(\sigma_{top},\ \rho) \Rightarrow \Phi \ \wedge\ \mathtt{keyExists}(\Phi,\ \mathbf{...}) \Rightarrow \mathtt{true} \ \wedge\ \mathtt{getValue}(\Phi,\ \mathbf{...}) \Rightarrow \mathtt{values}}{\langle\langle\sigma,\ \rho\rangle,\ \mathbf{...}\rangle \overset{rhs}{\Rightarrow} \langle\langle\sigma,\ \rho\rangle,\ \mathtt{values}\rangle}$$

If it doesn't exist, then the previous execution context is checked

$$\frac{\begin{array}{c}\mathtt{getContext}(\sigma_{top},\ \rho) \Rightarrow \Phi \ \wedge\ \mathtt{keyExists}(\Phi,\ \mathbf{...}) \Rightarrow \mathtt{false} \ \wedge\ \mathtt{popContext}(\sigma) \Rightarrow \sigma_1 \mid \sigma_1 \neq [\,] \\ \langle\langle\sigma_1,\ \rho\rangle,\ \mathbf{...}\rangle \overset{rhs}{\Rightarrow} \langle\langle\sigma_1,\ \rho\rangle,\ \mathtt{values}\rangle\end{array}}{\langle\langle\sigma,\ \rho\rangle,\ \mathbf{...}\rangle \overset{rhs}{\Rightarrow} \langle\langle\sigma,\ \rho\rangle,\ \mathtt{values}\rangle}$$

If '**...**' is not defined in the global execution context, then an error is returned

$$\frac{\mathtt{getContext}(\sigma_{top},\ \rho) \Rightarrow \Phi \ \wedge\ \mathtt{keyExists}(\Phi,\ \mathbf{...}) \Rightarrow \mathtt{false} \ \wedge\ \mathtt{popContext}(\sigma) \Rightarrow [\,]}{\langle\langle\sigma,\ \rho\rangle,\ \mathbf{...}\rangle \overset{lhs}{\Rightarrow} \langle\langle\sigma,\ \rho\rangle,\ \mathbf{ERROR}\rangle}$$

Unary operators
Binary operators

When a **function is defined**, a closure is created with given parameters, a statement block and a copy of the current execution context stack attached to it. It is then stored in the memory pool and a reference to it is returned

The reason has to do with the way closures behave. A closure inherits the environment where it is defined, not where it is called so it has to keep a copy of the environment at the moment it is defined

$$\frac{\mathtt{allocate}(\rho_0,\ \mathtt{function}(ps,\ ss,\ \sigma)) \Rightarrow \langle\rho_1,\ \mathtt{address}\rangle}{\langle\langle\sigma,\ \rho_0\rangle,\ \mathtt{functiondef}(ps,\ ss)\rangle \overset{rhs}{\Rightarrow} \langle\langle\sigma,\ \rho_1\rangle,\ \mathtt{referencetype}(\mathtt{function},\ \mathtt{address})::[\,]\rangle}$$

On the other hand, a **function call** creates a new scope in which it will evaluate a function. The created scope is then discarded -- albeit not destroyed -- when evaluation is done and a list of values or an error may be returned. If the expression we try to call returns an error, return the error

$$\frac{\langle ENV_0,\ e\rangle \overset{rhs}{\Rightarrow} \langle ENV_1,\ \mathbf{ERROR}\rangle}{\langle ENV_0,\ \mathtt{functioncall}(e,\ es)\rangle \overset{rhs}{\Rightarrow} \langle ENV_1,\ \mathbf{ERROR}\rangle}$$

If we try to call an expression that is not a function or table, an error is returned

$$\frac{\langle ENV_0,\ e\rangle \overset{rhs}{\Rightarrow} \langle ENV_1,\ v::\mathtt{values}\rangle \mid v \notin \mathtt{Reference}}{\langle ENV_0,\ \mathtt{functioncall}(e,\ es)\rangle \overset{rhs}{\Rightarrow} \langle ENV_1,\ \mathbf{ERROR}\rangle}$$

If evaluating the function arguments returns an error, return it

$$\frac{\langle \texttt{ENV}_0,\ \texttt{e}\rangle \overset{\texttt{rhs}}{\Rightarrow} \langle \texttt{ENV}_1,\ \texttt{v::values}\rangle \mid \texttt{v} \in \texttt{Reference}\ \wedge \quad \langle \texttt{ENV}_1,\ \texttt{expressions(es)}\rangle \overset{\texttt{rhs}}{\Rightarrow} \langle \texttt{ENV}_2,\ \textbf{ERROR}\rangle}{\langle \texttt{ENV}_0,\ \texttt{functioncall(e, es)}\rangle \overset{\texttt{rhs}}{\Rightarrow} \langle \texttt{ENV}_2,\ \textbf{ERROR}\rangle}$$

If the expression being called evaluates into a reference that points to a function, and the function arguments evaluate without any errors, then the function's statement block is evaluated in its execution context stack

$$\frac{\begin{array}{c}\langle\langle\sigma_0,\ \rho_0\rangle,\ e\rangle \overset{\text{rhs}}{\Rightarrow} \langle\langle\sigma_1,\ \rho_1\rangle,\ \text{v::values}\rangle \mid \text{v} \in \text{Reference}\ \wedge \\ \text{getObject}(\rho_1,\ \text{v}) \Rightarrow \text{function}(\text{ps},\ \text{ss},\ \sigma_{f,0})\ \wedge \\ \langle\langle\sigma_1,\ \rho_1\rangle,\ \text{expressions}(\text{es})\rangle \overset{\text{rhs}}{\Rightarrow} \langle\langle\sigma_2,\ \rho_2\rangle,\ \text{arguments}\rangle\ \wedge \\ \text{pushContext}(\sigma_{f,0},\ \rho_2,\ \text{ps},\ \text{arguments}) \Rightarrow \langle\sigma_{f,1},\ \rho_3\rangle\ \wedge \\ \langle\langle\sigma_{f,1},\ \rho_3\rangle,\ \text{statements}(\text{ss})\rangle \overset{\text{stat}}{\Rightarrow} \langle\langle\sigma_{f,2},\ \rho_4\rangle,\ \text{CTRL},\ \text{vs}_{ss}\rangle \mid \text{CTRL} \in \{\text{continue},\ \text{return},\ \text{error}\}\end{array}}{\langle\langle\sigma_0,\ \rho_0\rangle,\ \text{functioncall}(e,\ \text{es})\rangle \overset{\text{rhs}}{\Rightarrow} \langle\langle\sigma_2,\ \rho_4\rangle,\ \text{values}\rangle}$$

[TODO] Add these when implementing metatables:
1. If the reference points to a table, then the metamethod **__call** has to be retrieved and called
2. If the metamethod does not exist, then the expression is not callable, and an error is raised

Implemented metamethods:
 x add
 x sub
 x mul
 x div
 x mod
 x pow
 x unm
 x concat
 x len (only available for userdata, put this in documentation)
 x gc (only available for userdata, put this in documentation)
 x eq
 x lt
 x le
 x index
 x newindex
 x call
 x tostring

## III.d. Statement evaluation

The evaluation of a statement is a **⟨ENV₀, s, ENVₙ, control, vs⟩** 5-tuple where

- **ENV₀** is the initial execution environment in which the statement will be evaluated,
- **s** is a statement to evaluate,
- **ENVₙ** is the state of the execution environment after evaluation,
- **control** is a flag that controls the flow of the evaluation of the program, and
- **vs** is a list of zero or more values returned by the evaluation of **s**.

The control flags are **return**, **break**, **continue** and **error**. Since the **return** and **break** are considered **last statements**, any statement that succeeds them is not evaluated. The **error** flag is raised when an error occurs during the evaluation of an expression or statement. If the interpreter is not interrupted, then the **continue** flag is set.

Just like expression evaluation, let's start off with the **evaluation of a list of statements**. An empty list of statements does not modify the environment or return a value

$$\frac{}{\langle \mathtt{ENV_0},\ \mathtt{statements([])}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV_0},\ \mathtt{continue},\ \mathtt{[]}\rangle}$$

If a statement raises the **return**, **break** or **error** flag, then the remaining statements are not evaluated and a value is returned

$$\frac{\langle \mathtt{ENV_0},\ \mathtt{s}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV_1},\ \mathtt{CTRL},\ \mathtt{values}\rangle \mid \mathtt{CTRL} \in \{\mathtt{return},\ \mathtt{break},\ \mathtt{error}\}}{\langle \mathtt{ENV_0},\ \mathtt{statements(s::ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV_1},\ \mathtt{CTRL},\ \mathtt{values}\rangle}$$

If the **continue** flag is returned, then we ignore the return value of the evaluated statement and continue evaluating the remaining statements

$$\frac{\langle \mathtt{ENV_0},\ \mathtt{s}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV_1},\ \mathtt{continue},\ \_\rangle \ \wedge\ \langle \mathtt{ENV_1},\ \mathtt{statements(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV_2},\ \mathtt{CTRL},\ \mathtt{values}\rangle}{\langle \mathtt{ENV_0},\ \mathtt{statements(s::ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV_2},\ \mathtt{CTRL},\ \mathtt{values}\rangle}$$

The **assignment statement** allows multiple assignments. All left and right-hand side expressions are evaluated prior to assignment. This will result in a list of lvalues and values, of varying lengths, and each value will be stored at the location corresponding to one lvalue.

$$\frac{\begin{array}{c}\langle \mathtt{ENV_0},\ \mathtt{expressions(es_{lhs})}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ENV_1},\ \mathtt{lvalues}\rangle \ \wedge \\ \langle \mathtt{ENV_1},\ \mathtt{expressions(es_{rhs})}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV_2},\ \mathtt{values}\rangle \ \wedge\ \langle \mathtt{ENV_2},\ \mathtt{setValues(lvalues,\ values)}\rangle \Rightarrow \mathtt{ENV_3}\end{array}}{\langle \mathtt{ENV_0},\ \mathtt{assign(es_{lhs},\ es_{rhs})}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV_3},\ \mathtt{continue},\ \mathtt{[]}\rangle}$$

If there's an error while evaluating the left or right-hand side expressions, return it

$$\frac{\langle \mathtt{ENV_0},\ \mathtt{expressions(es_{lhs})}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ENV_1},\ \mathbf{ERROR}\rangle}{\langle \mathtt{ENV_0},\ \mathtt{assign(es_{lhs},\ es_{rhs})}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV_1},\ \mathtt{error},\ \mathbf{ERROR}\rangle}$$

$$\frac{\langle \mathtt{ENV_0},\ \mathtt{expressions(es_{lhs})}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ENV_1},\ \_\rangle \ \wedge\ \langle \mathtt{ENV_1},\ \mathtt{expressions(es_{rhs})}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV_2},\ \mathbf{ERROR}\rangle}{\langle \mathtt{ENV_0},\ \mathtt{assign(es_{lhs},\ es_{rhs})}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV_2},\ \mathtt{continue},\ \mathtt{[]}\rangle}$$

**Function calls** were previously defined as right-hand side expressions but can be evaluated as statements too, in which case all return values are discarded ...

$$\frac{\langle \mathtt{ENV}_0,\ \mathtt{functioncall(e,\ es)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{VS}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{functioncall(e,\ es)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{continue},\ []\rangle}$$

... except errors

$$\frac{\langle \mathtt{ENV}_0,\ \mathtt{functioncall(e,\ es)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathbf{ERROR}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{functioncall(e,\ es)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{error},\ \mathbf{ERROR}\rangle}$$

The **do** statement evaluates a list of statements in a new execution context and when evaluation is done, the context is discarded

$$\frac{\mathtt{pushContext}(\sigma,\ \rho) \Rightarrow \langle \sigma_1,\ \rho_1\rangle \ \wedge\ \langle\langle \sigma_1,\ \rho_1\rangle,\ \mathtt{statements(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle\langle \sigma_1,\ \rho_2\rangle,\ \mathtt{CTRL},\ \mathtt{values}\rangle}{\langle\langle \sigma,\ \rho\rangle,\ \mathtt{do(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle\langle \sigma,\ \rho_2\rangle,\ \mathtt{CTRL},\ \mathtt{values}\rangle}$$

The **while** loop evaluates a condition and if it is true, executes a statement block. If the evaluation of the condition expression results in an error, then the **error** control flag is raised and the error is returned

$$\frac{\langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathbf{ERROR}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{while(e,\ ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{error},\ \mathbf{ERROR}\rangle}$$

If the condition expression evaluates into either **nil** or **false**, then the statement block is not executed since the condition for execution is no longer true

$$\frac{\langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{v::values}\rangle \mid \mathtt{v} \in \{\mathtt{niltype(nil)},\ \mathtt{booleantype(false)}\}}{\langle \mathtt{ENV}_0,\ \mathtt{while(e,\ ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{continue},\ []\rangle}$$

If the condition expression of a while-do statement does not evaluate into **nil** or **false**, then the statement block is evaluated. If the evaluation of the body results in an error or the loop is explicitly broken via a **return** statement, then evaluation is halted, the appropriate control flag is raised, and a value is returned

$$\frac{\begin{array}{c}\langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{v::values}\rangle \mid \mathtt{v} \notin \{\mathtt{niltype(nil)},\ \mathtt{booleantype(false)}\} \ \wedge \\ \langle \mathtt{ENV}_1,\ \mathtt{do(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{CTRL},\ \mathtt{values}_{ss}\rangle \mid \mathtt{CTRL} \in \{\mathtt{return},\ \mathtt{error}\}\end{array}}{\langle \mathtt{ENV}_0,\ \mathtt{while(e,\ ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{CTRL},\ \mathtt{values}_{ss}\rangle}$$

If the loop is broken with a **break** statement, then evaluation is halted but the **continue** flag is raised and no values are returned

$$\frac{\begin{array}{c}\langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{v::values}\rangle \mid \mathtt{v} \notin \{\mathtt{niltype(nil)},\ \mathtt{booleantype(false)}\} \ \wedge \\ \langle \mathtt{ENV}_1,\ \mathtt{do(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{break},\ []\rangle\end{array}}{\langle \mathtt{ENV}_0,\ \mathtt{while(e,\ ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{continue},\ []\rangle}$$

If the **while-do** loop is not broken, it is evaluated until either the condition expression is considered false, or the loop is explicitly broken

$$\frac{\langle \mathtt{ENV_0},\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV_1},\ \mathtt{v::values}\rangle \mid \mathtt{v} \notin \{\mathtt{niltype(nil)},\ \mathtt{booleantype(false)}\} \ \wedge \\ \langle \mathtt{ENV_1},\ \mathtt{do(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV_2},\ \mathtt{continue},\ []\rangle \ \wedge\ \langle \mathtt{ENV_2},\ \mathtt{while(e,\ ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV_3},\ \mathtt{CTRL},\ \mathtt{values_{ss}}\rangle}{\langle \mathtt{ENV_0},\ \mathtt{while(e,\ ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV_3},\ \mathtt{CTRL},\ \mathtt{values_{ss}}\rangle}$$

The **repeat-until** loop repeats the execution of a lexically scoped statement block until a condition expression is considered true, or the loop is explicitly broken. In case the loop is broken with a **return** statement, or an **error** arises, then a value and the appropriate control flag are returned

$$\frac{\mathtt{pushContext}(\sigma,\ \rho) \Rightarrow \langle \sigma_1,\ \rho_1\rangle \ \wedge \\ \langle\langle \sigma_1,\ \rho_1\rangle,\ \mathtt{statements(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle\langle \sigma_1,\ \rho_2\rangle,\ \mathtt{CTRL},\ \mathtt{values}\rangle \mid \mathtt{CTRL} \in \{\mathtt{return},\ \mathtt{error}\}}{\langle\langle \sigma,\ \rho\rangle,\ \mathtt{repeat(e,\ ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle\langle \sigma,\ \rho_2\rangle,\ \mathtt{CTRL},\ \mathtt{values}\rangle}$$

If it's broken with the **break** statement then the **continue** control flag is raised

$$\frac{\mathtt{pushContext}(\sigma,\ \rho) \Rightarrow \langle \sigma_1,\ \rho_1\rangle \ \wedge \\ \langle\langle \sigma_1,\ \rho_1\rangle,\ \mathtt{statements(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle\langle \sigma_1,\ \rho_2\rangle,\ \mathtt{break},\ []\rangle}{\langle\langle \sigma,\ \rho\rangle,\ \mathtt{repeat(e,\ ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle\langle \sigma,\ \rho_2\rangle,\ \mathtt{continue},\ []\rangle}$$

If the condition expression results in an error, then the loop is broken, the **error** control flag is raised and the error is returned

$$\frac{\mathtt{pushContext}(\sigma,\ \rho) \Rightarrow \langle \sigma_1,\ \rho_1\rangle \ \wedge \\ \langle\langle \sigma_1,\ \rho_1\rangle,\ \mathtt{statements(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle\langle \sigma_1,\ \rho_2\rangle,\ \mathtt{continue},\ []\rangle \ \wedge \\ \langle\langle \sigma_1,\ \rho_2\rangle,\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle\langle \sigma_1,\ \rho_3\rangle,\ \mathbf{ERROR}\rangle}{\langle\langle \sigma,\ \rho\rangle,\ \mathtt{repeat(e,\ ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle\langle \sigma,\ \rho_3\rangle,\ \mathtt{error},\ \mathbf{ERROR}\rangle}$$

The loop is broken if the condition expression holds true

$$\frac{\mathtt{pushContext}(\sigma,\ \rho) \Rightarrow \langle \sigma_1,\ \rho_1\rangle \ \wedge \\ \langle\langle \sigma_1,\ \rho_1\rangle,\ \mathtt{statements(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle\langle \sigma_1,\ \rho_2\rangle,\ \mathtt{continue},\ []\rangle \ \wedge \\ \langle\langle \sigma_1,\ \rho_2\rangle,\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle\langle \sigma_1,\ \rho_3\rangle,\ \mathtt{v::values}\rangle \mid \mathtt{v} \notin \{\mathtt{niltype(nil)},\ \mathtt{booleantype(false)}\}}{\langle\langle \sigma,\ \rho\rangle,\ \mathtt{repeat(e,\ ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle\langle \sigma,\ \rho_3\rangle,\ \mathtt{continue},\ []\rangle}$$

If the condition expression is false, then the loop continues

$$\frac{\mathtt{pushContext}(\sigma,\ \rho) \Rightarrow \langle \sigma_1,\ \rho_1\rangle \ \wedge \\ \langle\langle \sigma_1,\ \rho_1\rangle,\ \mathtt{statements(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle\langle \sigma_1,\ \rho_2\rangle,\ \mathtt{continue},\ []\rangle \ \wedge \\ \langle\langle \sigma_1,\ \rho_2\rangle,\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle\langle \sigma_1,\ \rho_3\rangle,\ \mathtt{v::values}\rangle \mid \mathtt{v} \in \{\mathtt{niltype(nil)},\ \mathtt{booleantype(false)}\} \ \wedge \\ \langle\langle \sigma,\ \rho_3\rangle,\ \mathtt{repeat(e,\ ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle\langle \sigma,\ \rho_4\rangle,\ \mathtt{CTRL},\ \mathtt{values}\rangle}{\langle\langle \sigma,\ \rho\rangle,\ \mathtt{repeat(e,\ ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle\langle \sigma,\ \rho_4\rangle,\ \mathtt{CTRL},\ \mathtt{values}\rangle}$$

When evaluating **if control structures**, the condition expression is evaluated first, which will determine which of the two statements will be evaluated next. If the evaluation of this condition expression returns an error, then the error control flag is raised and the error is returned

$$\frac{\langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathbf{ERROR}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{if(e,\ s_{true},\ s_{false})}\rangle \overset{\mathrm{stat}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{error},\ \mathbf{ERROR}\rangle}$$

If the condition expression evaluates to either **nil** or **false**, then **s<sub>false</sub>** is evaluated

$$\frac{\langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{v_e::values}\rangle \mid \mathtt{v_e} \in \{\mathtt{niltype(nil),\ booleantype(false)}\} \ \wedge \quad \langle \mathtt{ENV}_1,\ \mathtt{s_{false}}\rangle \overset{\mathrm{stat}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{CTRL},\ \mathtt{values_s}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{if(e,\ s_{true},\ s_{false})}\rangle \overset{\mathrm{stat}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{CTRL},\ \mathtt{values_s}\rangle}$$

Otherwise **s<sub>true</sub>** is evaluated

$$\frac{\langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{v_e::values}\rangle \mid \mathtt{v_e} \notin \{\mathtt{niltype(nil),\ booleantype(false)}\} \ \wedge \quad \langle \mathtt{ENV}_1,\ \mathtt{s_{true}}\rangle \overset{\mathrm{stat}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{CTRL},\ \mathtt{values_s}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{if(e,\ s_{true},\ s_{false})}\rangle \overset{\mathrm{stat}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{CTRL},\ \mathtt{values_s}\rangle}$$

**Declaring a local variable** creates a **⟨key, value⟩** pair in the current execution context, where **key** is the variable's name and **value** is set to **nil**.

$$\frac{\mathtt{setValue}(\langle\sigma,\ \rho\rangle,\ \sigma,\ \mathtt{n},\ \mathtt{niltype(nil)}) \Rightarrow \rho_1}{\langle\langle\sigma,\ \rho\rangle,\ \mathtt{localvariable(n)}\rangle \overset{\mathrm{stat}}{\Rightarrow} \langle\langle\sigma,\ \rho_1\rangle,\ \mathtt{continue},\ []\rangle}$$

The **return** statement is used to return one or more values from a function. It is handed a list of right-hand expressions that will evaluate into return values. If an error occurs during the evaluation of these expressions then it is returned, else a list of results is returned

$$\frac{\langle \mathtt{ENV}_0,\ \mathtt{expressions(es)}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathbf{ERROR}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{return(es)}\rangle \overset{\mathrm{stat}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{error},\ \mathbf{ERROR}\rangle} \qquad \frac{\langle \mathtt{ENV}_0,\ \mathtt{expressions(es)}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{values}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{return(es)}\rangle \overset{\mathrm{stat}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{return},\ \mathtt{values}\rangle}$$

The **break** statement does nothing more than break a loop. It does not return any values or modify the environment. In terms of its evaluation, all it does is raise the **break** control flag and return an empty list of values

$$\frac{}{\langle \mathtt{ENV}_0,\ \mathtt{break}\rangle \overset{\mathrm{stat}}{\Rightarrow} \langle \mathtt{ENV}_0,\ \mathtt{break},\ []\rangle}$$

## III.e. Program evaluation

The evaluation of a program is a ⟨**statements, arguments, ENV, results**⟩ quadruplet where

- **statements** is a list of statements to evaluate,
- **arguments** are the values that make up the anonymous function's variable argument list,
- **ENV** is the state of the environment after the evaluation of **statements**, and
- **results** is a list of one or more values returned by the evaluation of **statements**.

A chunk is handled as the body of an anonymous function with a variable number of arguments, so evaluating a Lua program is essentially evaluating a function call to this anonymous function.

In Prolua, things are done a little differently because evaluating a function call doesn't allow us to view a meaningful execution environment since contexts are popped from the stack and objects may be deleted from the memory pool when the function exits.
Instead, an environment is created and it comprises of a memory pool containg function objects (closures), as well as an execution context containing the variable argument list and references to the aforementioned functions. The statements that make up the chunk are then evaluated in this environment and when done, a result is returned as well as the state of the environment.
Semantically, this translates to

$$\frac{\texttt{loadEnvironment(args)} \Rightarrow \texttt{ENV}_0 \;\wedge\; \langle \texttt{ENV}_0,\ \texttt{statements(ss)} \rangle \overset{\texttt{stat}}{\Rightarrow} \langle \texttt{ENV}_1,\ \_,\ \texttt{values} \rangle}{\langle \texttt{ss},\ \texttt{args} \rangle \overset{\texttt{chunk}}{\Rightarrow} \langle \texttt{ENV}_1,\ \texttt{values} \rangle}$$

## IV. Conclusion

And that's all folks!
[TODO] Obviously not. Elaborate!

## V. References

**1. Lua 5.1 Reference Manual**
Roberto Ierusalimschy, Luiz Henrique de Figueiredo and Waldemar Celes.

**2. Cours "Sémantique des langages informatiques"**
Professor Didier Buchs, Université de Genève.

**3. Prolog — More Advanced** [PDF]
Professor J. Paul Gibson, Université Henri Poincaré.

**4. Reference counting** [WIKIPEDIA]