

PROLUA

Working Draft

Chapter I. Introduction

[TODO] Add cycle diagram: input (lua code) > validate input > lua2prolog > prolua chunk > prolua syntax analysis > prolua evaluation > prolua output (result and environment)

I.a. Why Lua?

Lua is a well documented, lightweight, simple and somewhat [popular](#) [WIKIPEDIA] 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** being one. This makes it a particularly good choice for **modelling** other programming languages. It also seems that no Lua interpreter has been implemented in Prolog, or any logic programming language for that matter. Challenge accepted.

I.c. Constraints

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 it is not guaranteed that they will work. 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 work.

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 other programming languages, it isn't well suited for writing whole libraries. Although not impossible, completely rewriting a library in Prolog is very time consuming. 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**, **tables**, **functions** and **nil** will be implemented. The **userdata** and **thread** types will be excluded, as well as the language features that depend on these types, such as **coroutines**. A few frequently used functions from the standard library will be implemented; see §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 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.

Chapter II. Syntax

Our first order of business is to define an abstract syntax specific to Prolua so that we can have a general idea of the **form** that a Prolua program will take. To do so, we'll need to analyse Lua's concrete syntax given below in EBNF and come up with an abstract syntax of our own

```
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

$$\begin{array}{l} \text{expressions:} \frac{}{[] \in \text{Expressions}} \quad \text{expressions:} \frac{e \in \text{Expression}, \quad es \in \text{Expressions}}{e::es \in \text{Expressions}} \end{array}$$

[TODO] Explain the :: notation! (inductively defined data structures...)

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

$$\begin{array}{l} \text{names:} \frac{}{[] \in \text{Names}} \quad \text{names:} \frac{n \in \text{Name}, \quad ns \in \text{Names}}{n::ns \in \text{Names}} \end{array}$$

Let **Parameter** be the set of all possible parameter names that is defined as an extension of **Name** to include "..." (three dots), then let **Parameters** be a list of parameter names such that

$$\begin{array}{l} \text{parameter:} \frac{}{\text{Parameter} = \text{Name} \cup \{\dots\}} \\ \text{parameters:} \frac{}{[] \in \text{Parameters}} \quad \text{parameters:} \frac{p \in \text{Parameter}, \quad ps \in \text{Parameters}}{p::ps \in \text{Parameters}} \end{array}$$

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

$$\begin{array}{l} \text{variables:} \frac{}{[] \in \text{Variables}} \quad \text{variables:} \frac{v \in \text{Variable}, \quad vs \in \text{Variables}}{v::vs \in \text{Variables}} \end{array}$$

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

$$\begin{array}{l} \text{values:} \frac{}{[] \in \text{Values}} \quad \text{values:} \frac{v \in \text{Value}, \quad vs \in \text{Values}}{v::vs \in \text{Values}} \end{array}$$

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

$$\begin{array}{l} \text{statements:} \frac{}{[] \in \text{Statements}} \quad \text{statements:} \frac{s \in \text{Statement}, \quad ss \in \text{Statements}}{s::ss \in \text{Statements}} \end{array}$$

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

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

II.b. Values and Types

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

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

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

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

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

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

The type **table** implements associative arrays, i.e. arrays that can be indexed with any value except nil. Essentially, a table is a list of **<key, value>** pairs known as **fields** where each key can be an expression except **nil**

$$\text{field:} \frac{v \in \text{Expression}}{v \in \text{Field}} \quad \text{field:} \frac{k \in \text{Name} \cup \text{Expression} \setminus \{\text{niltype}(\text{nil})\}, v \in \text{Expression}}{\langle k, v \rangle \in \text{Field}} \\ \text{fields:} \frac{}{[] \in \text{Fields}} \quad \text{fields:} \frac{f \in \text{Field}, fs \in \text{Fields}}{f::fs \in \text{Fields}} \quad \text{table:} \frac{fs \in \text{Fields}}{\text{table}(fs) \in \text{Object}}$$

A **function** or **closure**, holds a list of parameter names, a block of statements, and an environment table, all of which can be empty. The environment table, if specified, will initialise the function's **execution context** when it is called

$$\text{function:} \frac{ps \in \text{Parameters}, ss \in \text{Statements}, r \in \{\text{Reference} \cup []\}}{\text{function}(ps, ss, r) \in \text{Object}}$$

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

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

Execution **contexts** and pool **addresses** are detailed in §III.a. *The Execution Environment*.

II.c. Expressions

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

$$\text{expression: } \frac{fs \in \text{Fields} \cup \text{Expressions}}{\text{tableconstructor}(fs) \in \text{Expression}}$$

An expression can be **enclosed** in parentheses

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

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

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

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

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

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

$$\text{expression: } \frac{}{\dots \in \text{Expression}}$$

Calling a **unary operator** requires the operator's name and the expression to be evaluated. Included, albeit not mentioned in the concrete syntax, is the **type** operator

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The **while-do** statement executes a block of code while a given expression is considered true

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

A **repeat-until** statement executes a block of code until a given expression is considered true

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

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

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

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, stopping when this value is **nil**.

The [Lua documentation](#) details workarounds for both versions using **while-do** and so no abstract syntax for either statement is specified in Prolua.

Declaring a local variable creates a variable with a given value in the current environment table. To be able to create a field in the environment, we need to know the field key, which in this case is the variable name. If no value is specified, then **nil** is implied.

$$\text{statement} : \frac{n \in \text{Name}, e \in \text{Expression}}{\text{localvariable}(n, e) \in \text{Statement}}$$

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

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

The **break** statement explicitly breaks a loop

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

II.e. Intrinsic functions

[TODO] Explain intrinsic functions.

error, type, print, tonumber, next, pairs, ipairs

The **ipairs** iterator function is generated by **lua2prolog** from the following 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 **pairs** iterator is inspired by the following code ...

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

... and relies on the **next** function, which is implemented in Prolua. It's pseudocode is

Now consider the following Lua program¹ 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
```

...and its generated Prolua program (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)])]),

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

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([]).
```

It's clearly not a very smart idea to manually write a Prolua program ...

The output are two Prolog base clauses that state that a **chunk** is a list of terms which resemble the documented abstract syntax, and that there are no **arguments**. However, neither of the clauses mean much since no relationships between the terms have been defined. That is where semantics will come into play.

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

Chapter III. Semantics

As I mentioned before, 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 because the **length** operator expects an expression as its parameter, but semantically wrong because the length of a nil value cannot be calculated. The semantics is in charge of restricting the parameter 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 how should it be interpreted? Should the result of its evaluation return a 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 nothing more than a string of text. The goal of the semantics of a language is to **give a meaning** to this textual form, define **how** it should be evaluated and eventually what values it may return, if any.

III.a. The Execution Environment

Before we can define the evaluation semantics of expressions and statements in Prolua, we need to detail the environment in which the execution of a program takes place. This shall be known as the **execution environment** in Prolua, and is composed of a **path** to the current execution context and a memory **pool** to store objects.

An **execution context** is a dictionary of **<key, value>** pairs within which each key is unique

$$\text{context: } \frac{}{\text{context}([\])\in \text{Context}} \quad \text{context: } \frac{k_i \in \text{Parameter}, \ v_i \in \text{Value} \ \forall i \geq 0}{\text{context}(\langle k_0, v_0 \rangle :: \langle k_1, v_1 \rangle :: \dots :: \langle k_n, v_n \rangle) \in \text{Context}}$$

Execution contexts are used to implement **lexical scoping** in Prolua. For example, when a function is called, an execution context -- where the function's arguments and local variables will be stored -- is created in the execution context graph and when the same function exits, the path to the node corresponding to the execution context is removed but the node isn't destroyed. 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.

The negative impact of this is that the DAG will keep growing until the program terminates, and this means that execution context lookup times will increase as the graph grows.

Execution contexts are stored in a single block in the memory pool in the form of a directed acyclic graph (DAG) which allows for an efficient node lookup time even in worst case scenarios. Each node in the DAG is located via a **path** from the root node, i.e. the global execution context, to the actual node. Syntactically, a path is nothing more than a list of integers

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

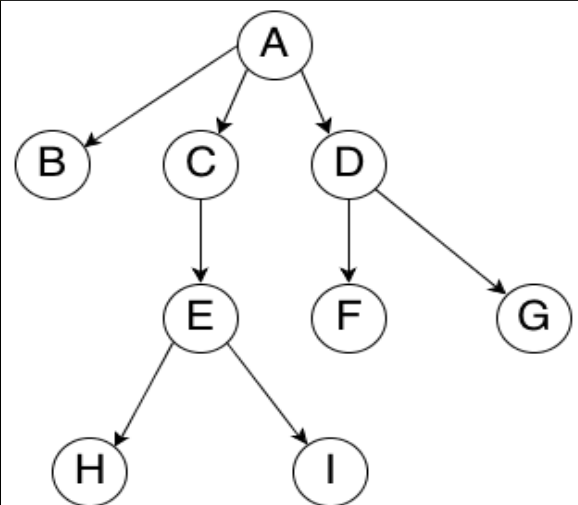
Suppose we have the following program and the corresponding execution context graph, as well as paths to each execution context.

CORRECT THIS GRAPH: Function scopes are created only when the function is called!

```

01  a = "Hello, World"
02
03  function square(x)
04      return x * x
05  end
06
07  function g(f)
08
09      local function nested(x)
10
11      end
12  end
13
14  do
15      a = 32
16      do
17          local a = a
18          local z = 32
19          a = a + z
20      end
21      do
22          local y = a
23          a = y * 2
24          y = a * 2
25      end
26  end
27
28  return closure(square(a))

```



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]

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

Explain stack.

[TODO] Explain better.

This linearity, coupled with the fact that execution contexts are not destroyed, makes for poor performance and is the reason why Prolua should only be used as a learning tool. **[TODO] Show fibonacci execution time growth...**

The memory **pool** is memory reserved for **contexts**, **tables** and **functions**. A pool is a collection of contiguous memory blocks defined as $\langle \text{address}, \text{reference count}, \text{data} \rangle$ triplets, such that **address** is a unique positive integer used to retrieve the memory block, **reference count** is the number of references made to the memory block, and **data** is what is stored in the block

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

A collection of memory blocks is defined syntactically as

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

A **pool** contains an offset counter which is used to determine the address of the next memory block to be added to the pool, incremented each time data is added to the pool. When an object is removed, the counter is not decremented because this may overwrite previous addresses.

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

The fundamental difference between the stack and heap is that data created in an execution context on the stack is destroyed when the context is popped from the stack, while data created in the heap is deallocated when it's no longer referenced and is therefore independent of the scope.

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 the reason why execution contexts are stored in an execution context graph and not as memory blocks, as this would result in rapid expansion of the memory pool.

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

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

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 a walk in the park, and is left as an exercise to the reader.

III.b. Notation

There is more than one notation used to formalise the semantics of a programming language, each having a specific purpose. 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 is used to formalise Prolog's evaluation semantics because the concept 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**. A property is said to be 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. This is noted as

$$\frac{\text{premises}}{\text{conclusion}}$$

As an example, let's prove the factorial of a positive integer defined by the following recurrence relation

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

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

$$\frac{}{\text{factorial}(0) \Rightarrow 1}$$

No premises are given for the base case **0!** since it is a fact, and therefore needs no proof. The recurrence relation also states that **n!** for all **n** greater than zero, is equal to **n × (n - 1)!**

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

which can be read as "if **n** is greater than zero, and **(n - 1)!** results in **m**, and **nm** is equal to **n × m**, then **n!** results in **nm**". The same problem can be solved in Prolog as

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

III.c. Expression evaluation

The evaluation of an expression is a $\langle \mathbf{ENV}_0, \mathbf{e}, \mathbf{ENV}_n, \mathbf{vs} \rangle$ quadruplet where

- \mathbf{ENV}_0 is the initial execution environment in which the expression will be evaluated,
- \mathbf{e} is an expression to evaluate,
- \mathbf{ENV}_n is the state of the execution environment after evaluation, and
- \mathbf{vs} is a list of zero or more values returned by the evaluation of \mathbf{e} .

Lua is a language with assignable variables (and table fields) so the result of an evaluation is not always the same. It is necessary to distinguish between **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 \mathbf{vs} is a $\langle \mathbf{locator}, \mathbf{key} \rangle$ 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 expression (excluding **nil**). Right-hand side expression evaluation is denoted by $\overset{\text{rhs}}{\Rightarrow}$ and \mathbf{vs} , in this case, is a list of zero or more values.

Every Lua expression is a right-hand side expression, but only the **variable** and **access** expressions can return **lvalues** which makes 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 returns an empty list of lvalues.

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

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

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

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

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

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

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

Notice that the environment is never discarded since it's useful to know its current state when debugging an error.

Evaluating a left-hand side variable returns the lvalue $\langle \text{ECID}, \text{name} \rangle$ where **ECID** is the identifier of 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 identifier and the variable name

$$\frac{\text{getContext}(\sigma_{\text{top}}, \rho) \Rightarrow \Phi \wedge \text{keyExists}(\Phi, n) \Rightarrow \text{true}}{\langle \langle \sigma, \rho \rangle, \text{variable}(n) \rangle \xRightarrow{\text{lhs}} \langle \langle \sigma, \rho \rangle, \langle \Phi_{\text{identifier}}, n \rangle \rangle}$$

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

$$\frac{\text{getContext}(\sigma_{\text{top}}, \rho) \Rightarrow \Phi \wedge \text{keyExists}(\Phi, n) \Rightarrow \text{false} \wedge \text{popContext}(\sigma) \Rightarrow \sigma_1 \mid \sigma_1 \neq [] \wedge \langle \langle \sigma_1, \rho \rangle, \text{variable}(n) \rangle \xRightarrow{\text{lhs}} \langle \langle \sigma_1, \rho \rangle, \langle \text{identifier}, n \rangle \rangle}{\langle \langle \sigma, \rho \rangle, \text{variable}(n) \rangle \xRightarrow{\text{lhs}} \langle \langle \sigma, \rho \rangle, \langle \text{identifier}, n \rangle \rangle}$$

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

$$\frac{\text{getContext}(\sigma_{\text{top}}, \rho) \Rightarrow \Phi \wedge \text{keyExists}(\Phi, n) \Rightarrow \text{false} \wedge \text{popContext}(\sigma) \Rightarrow []}{\langle \langle \sigma, \rho \rangle, \text{variable}(n) \rangle \xRightarrow{\text{lhs}} \langle \langle \sigma, \rho \rangle, \langle \Phi_{\text{identifier}}, n \rangle \rangle}$$

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

$$\frac{\langle \text{ENV}_0, e_r \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, r::[] \rangle \wedge \text{getObjectType}(r) \Rightarrow \text{table} \wedge \langle \text{ENV}_1, e_k \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_2, v_k::vs \rangle}{\langle \text{ENV}_0, \text{access}(e_r, e_k) \rangle \xRightarrow{\text{lhs}} \langle \text{ENV}_2, \langle r, v_k \rangle \rangle}$$

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

$$\frac{\langle \text{ENV}_0, e_r \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, r::[] \rangle \wedge \text{getObjectType}(r) \Rightarrow \text{type} \mid \text{type} \neq \text{table}}{\langle \text{ENV}_0, \text{access}(e_r, e_k) \rangle \xRightarrow{\text{lhs}} \langle \text{ENV}_2, \text{ERROR} \rangle}$$

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

$$\frac{\langle \text{ENV}_0, e_r \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, \text{ERROR} \rangle}{\langle \text{ENV}_0, \text{access}(e_r, e_k) \rangle \xRightarrow{\text{lhs}} \langle \text{ENV}_1, \text{ERROR} \rangle}$$

Likewise, if the second expression results in an error even when the first was successfully evaluated, the error is returned

$$\frac{\langle \text{ENV}_0, e_r \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, r::[] \rangle \wedge \text{getObjectType}(r) \Rightarrow \text{table} \wedge \langle \text{ENV}_1, e_k \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_2, \text{ERROR} \rangle}{\langle \text{ENV}_0, \text{access}(e_r, e_k) \rangle \xRightarrow{\text{lhs}} \langle \text{ENV}_2, \text{ERROR} \rangle}$$

Evaluating a **list of right-hand side expressions** is almost similar but it 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 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

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

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

$$\frac{es \neq [] \wedge \langle \text{ENV}_0, e \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, v_e::\text{values} \rangle \wedge \langle \text{ENV}_1, \text{expressions}(es) \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_2, \text{values}_{es} \rangle}{\langle \text{ENV}_0, \text{expressions}(e::es) \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_2, v_e::\text{values}_{es} \rangle}$$

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

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

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

$$\frac{}{\langle \text{ENV}_0, \text{value} \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_0, \text{value}::[] \rangle \mid \text{value} \in \text{Value}}$$

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

$$\frac{fs \neq [] \wedge fs \in \text{Fields} \wedge \langle \langle \sigma, \rho \rangle, \text{fields}(fs) \rangle \xRightarrow{\text{rhs}} \langle \langle \sigma, \rho_1 \rangle, \text{map} \rangle \wedge \text{allocate}(\rho_1, \text{table}(\text{map})) \Rightarrow \langle \rho_2, \text{address} \rangle}{\langle \langle \sigma, \rho \rangle, \text{tableconstructor}(fs) \rangle \xRightarrow{\text{rhs}} \langle \langle \sigma, \rho_2 \rangle, \text{referencetype}(\text{table}, \text{address})::[] \rangle}$$

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

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

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

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

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

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

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

$$\frac{\langle \text{ENV}_0, e \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, v::\text{values} \rangle}{\langle \text{ENV}_0, \text{enclosed}(e) \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, v::[] \rangle} \quad \frac{\langle \text{ENV}_0, e \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, [] \rangle}{\langle \text{ENV}_0, \text{enclosed}(e) \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, [] \rangle}$$

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

$$\frac{\langle \text{ENV}_0, e \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, \text{ERROR} \rangle}{\langle \text{ENV}_0, \text{enclosed}(e) \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, \text{ERROR} \rangle}$$

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

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

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

$$\frac{\langle \text{ENV}_0, \text{variable}(n) \rangle \xRightarrow{\text{lhs}} \langle \text{ENV}_1, \text{ERROR} \rangle}{\langle \text{ENV}_0, \text{variable}(n) \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, \text{ERROR} \rangle}$$

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

$$\frac{\langle \text{ENV}_0, \text{access}(e_r, e_k) \rangle \xRightarrow{\text{lhs}} \langle \text{ENV}_1, \langle r, k \rangle \rangle \wedge \text{getField}(\text{ENV}_1, r, k) \Rightarrow \text{value}}{\langle \text{ENV}_0, \text{access}(e_r, e_k) \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, \text{value}::[] \rangle}$$

We do have to watch out for errors

$$\frac{\langle \text{ENV}_0, \text{access}(e_r, e_k) \rangle \xRightarrow{\text{lhs}} \langle \text{ENV}_1, \text{ERROR} \rangle}{\langle \text{ENV}_0, \text{access}(e_r, e_k) \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, \text{ERROR} \rangle}$$

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

$$\frac{\text{getContext}(\sigma_{\text{top}}, \rho) \Rightarrow \Phi \wedge \text{keyExists}(\Phi, \dots) \Rightarrow \text{true} \wedge \text{getValue}(\Phi, \dots) \Rightarrow \text{values}}{\langle\langle\sigma, \rho\rangle, \dots\rangle \xRightarrow{\text{rhs}} \langle\langle\sigma, \rho\rangle, \text{values}\rangle}$$

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

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

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

$$\frac{\text{getContext}(\sigma_{\text{top}}, \rho) \Rightarrow \Phi \wedge \text{keyExists}(\Phi, \dots) \Rightarrow \text{false} \wedge \text{popContext}(\sigma) \Rightarrow []}{\langle\langle\sigma, \rho\rangle, \dots\rangle \xRightarrow{\text{lhs}} \langle\langle\sigma, \rho\rangle, \text{ERROR}\rangle}$$

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

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

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

$$\frac{\langle\text{ENV}_0, e\rangle \xRightarrow{\text{rhs}} \langle\text{ENV}_1, \text{ERROR}\rangle}{\langle\text{ENV}_0, \text{functioncall}(e, \text{es})\rangle \xRightarrow{\text{rhs}} \langle\text{ENV}_1, \text{ERROR}\rangle}$$

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

$$\frac{\langle\text{ENV}_0, e\rangle \xRightarrow{\text{rhs}} \langle\text{ENV}_1, v::\text{values}\rangle \mid v \notin \text{Reference}}{\langle\text{ENV}_0, \text{functioncall}(e, \text{es})\rangle \xRightarrow{\text{rhs}} \langle\text{ENV}_1, \text{ERROR}\rangle}$$

If evaluating the function arguments returns an error, return it

$$\frac{\begin{array}{c} \langle\text{ENV}_0, e\rangle \xRightarrow{\text{rhs}} \langle\text{ENV}_1, v::\text{values}\rangle \mid v \in \text{Reference} \wedge \\ \langle\text{ENV}_1, \text{expressions}(\text{es})\rangle \xRightarrow{\text{rhs}} \langle\text{ENV}_2, \text{ERROR}\rangle \end{array}}{\langle\text{ENV}_0, \text{functioncall}(e, \text{es})\rangle \xRightarrow{\text{rhs}} \langle\text{ENV}_2, \text{ERROR}\rangle}$$

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

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

[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

III.d. Statement evaluation

The evaluation of a statement is a $\langle \text{ENV}_0, \mathbf{s}, \text{ENV}_n, \text{control}, \mathbf{vs} \rangle$ 5-tuple where

- **ENV₀** is the initial execution environment in which the statement will be evaluated,
- **s** is a statement to evaluate,
- **ENV_n** 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

$$\frac{}{\langle \text{ENV}_0, \text{statements}([]) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_0, \text{continue}, [] \rangle}$$

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

$$\frac{\langle \text{ENV}_0, \mathbf{s} \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_1, \text{CTRL}, \text{values} \rangle \mid \text{CTRL} \in \{\text{return}, \text{break}, \text{error}\}}{\langle \text{ENV}_0, \text{statements}(\mathbf{s}::\text{ss}) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_1, \text{CTRL}, \text{values} \rangle}$$

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

$$\frac{\langle \text{ENV}_0, \mathbf{s} \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_1, \text{continue}, _ \rangle \wedge \langle \text{ENV}_1, \text{statements}(\text{ss}) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_2, \text{CTRL}, \text{values} \rangle}{\langle \text{ENV}_0, \text{statements}(\mathbf{s}::\text{ss}) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_2, \text{CTRL}, \text{values} \rangle}$$

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.

$$\frac{\begin{array}{l} \langle \text{ENV}_0, \text{expressions}(\text{es}_{\text{lhs}}) \rangle \xRightarrow{\text{lhs}} \langle \text{ENV}_1, \text{lvalues} \rangle \wedge \\ \langle \text{ENV}_1, \text{expressions}(\text{es}_{\text{rhs}}) \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_2, \text{values} \rangle \wedge \langle \text{ENV}_2, \text{setValues}(\text{lvalues}, \text{values}) \rangle \Rightarrow \text{ENV}_3 \end{array}}{\langle \text{ENV}_0, \text{assign}(\text{es}_{\text{lhs}}, \text{es}_{\text{rhs}}) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_3, \text{continue}, [] \rangle}$$

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

$$\frac{\langle \text{ENV}_0, \text{expressions}(\text{es}_{\text{lhs}}) \rangle \xRightarrow{\text{lhs}} \langle \text{ENV}_1, \text{ERROR} \rangle}{\langle \text{ENV}_0, \text{assign}(\text{es}_{\text{lhs}}, \text{es}_{\text{rhs}}) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_1, \text{error}, \text{ERROR} \rangle}$$

$$\frac{\langle \text{ENV}_0, \text{expressions}(\text{es}_{\text{lhs}}) \rangle \xRightarrow{\text{lhs}} \langle \text{ENV}_1, _ \rangle \wedge \langle \text{ENV}_1, \text{expressions}(\text{es}_{\text{rhs}}) \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_2, \text{ERROR} \rangle}{\langle \text{ENV}_0, \text{assign}(\text{es}_{\text{lhs}}, \text{es}_{\text{rhs}}) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_2, \text{continue}, [] \rangle}$$

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 ...

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

... except errors

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

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

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

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

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

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

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

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

$$\frac{\langle \text{ENV}_0, e \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, v::\text{values} \rangle \mid v \notin \{\text{niltype}(\text{nil}), \text{booleantype}(\text{false})\} \wedge \langle \text{ENV}_1, \text{do}(ss) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_2, \text{CTRL}, \text{values}_{ss} \rangle \mid \text{CTRL} \in \{\text{return}, \text{error}\}}{\langle \text{ENV}_0, \text{while}(e, ss) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_2, \text{CTRL}, \text{values}_{ss} \rangle}$$

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

$$\frac{\langle \text{ENV}_0, e \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, v::\text{values} \rangle \mid v \notin \{\text{niltype}(\text{nil}), \text{booleantype}(\text{false})\} \wedge \langle \text{ENV}_1, \text{do}(ss) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_2, \text{break}, [] \rangle}{\langle \text{ENV}_0, \text{while}(e, ss) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_2, \text{continue}, [] \rangle}$$

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

$$\frac{\langle \text{ENV}_0, e \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, v::\text{values} \rangle \mid v \notin \{\text{niltype}(\text{nil}), \text{booleantype}(\text{false})\} \wedge \langle \text{ENV}_1, \text{do}(\text{ss}) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_2, \text{continue}, [] \rangle \wedge \langle \text{ENV}_2, \text{while}(e, \text{ss}) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_3, \text{CTRL}, \text{values}_{\text{ss}} \rangle}{\langle \text{ENV}_0, \text{while}(e, \text{ss}) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_3, \text{CTRL}, \text{values}_{\text{ss}} \rangle}$$

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

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

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

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

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

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

The loop is broken if the condition expression holds true

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

If the condition expression is false, then the loop continues

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

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

$$\frac{\langle \text{ENV}_0, e \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, \text{ERROR} \rangle}{\langle \text{ENV}_0, \text{if}(e, s_{\text{true}}, s_{\text{false}}) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_1, \text{error}, \text{ERROR} \rangle}$$

If the condition expression evaluates to either **nil** or **false**, then **s_{false}** is evaluated

$$\frac{\begin{array}{c} \langle \text{ENV}_0, e \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, v_e::\text{values} \rangle \mid v_e \in \{\text{niltype}(\text{nil}), \text{booleantype}(\text{false})\} \wedge \\ \langle \text{ENV}_1, s_{\text{false}} \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_2, \text{CTRL}, \text{values}_s \rangle \end{array}}{\langle \text{ENV}_0, \text{if}(e, s_{\text{true}}, s_{\text{false}}) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_2, \text{CTRL}, \text{values}_s \rangle}$$

Otherwise **s_{true}** is evaluated

$$\frac{\begin{array}{c} \langle \text{ENV}_0, e \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, v_e::\text{values} \rangle \mid v_e \notin \{\text{niltype}(\text{nil}), \text{booleantype}(\text{false})\} \wedge \\ \langle \text{ENV}_1, s_{\text{true}} \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_2, \text{CTRL}, \text{values}_s \rangle \end{array}}{\langle \text{ENV}_0, \text{if}(e, s_{\text{true}}, s_{\text{false}}) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_2, \text{CTRL}, \text{values}_s \rangle}$$

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**.

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

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

$$\frac{\langle \text{ENV}_0, \text{expressions}(es) \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, \text{ERROR} \rangle}{\langle \text{ENV}_0, \text{return}(es) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_1, \text{error}, \text{ERROR} \rangle} \quad \frac{\langle \text{ENV}_0, \text{expressions}(es) \rangle \xRightarrow{\text{rhs}} \langle \text{ENV}_1, \text{values} \rangle}{\langle \text{ENV}_0, \text{return}(es) \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_1, \text{return}, \text{values} \rangle}$$

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

$$\frac{}{\langle \text{ENV}_0, \text{break} \rangle \xRightarrow{\text{stat}} \langle \text{ENV}_0, \text{break}, [] \rangle}$$

III.e. Program evaluation

The evaluation of a program is a $\langle \mathbf{statements}, \mathbf{arguments}, \mathbf{ENV}, \mathbf{results} \rangle$ 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 containing 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

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

Conclusion

And that's all folks!

[TODO] Elaborate.