PROLUA

Working Draft

Jeremy OTHIENO
https://github.com/supranove

Chapter I. Introduction

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

I.a. Why Lua?

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

I.b. Why Prolog?

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

I.c. Constraints

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

Of the eight basic types of values in Lua, only **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**.

No garbage collection yet.
[TEMPORARY] No generic for loops until I can get closures to work properly.

No standard library > no I/O operators. Although a few functions from the standard library will be implemented (see intrinsic functions)

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 such that

$$\text{expressions}: \frac{}{[] \in \text{Expressions}} \qquad \text{expressions}: \frac{e \in \text{Expression},\ es \in \text{Expressions}}{e::es \in \text{Expressions}}$$

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

$$\text{names}: \frac{}{[] \in \text{Names}} \qquad \text{names}: \frac{n \in \text{Name},\ ns \in \text{Names}}{n::ns \in \text{Names}}$$

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

$$\text{parameter}: \frac{}{\text{Parameter} = \text{Name} \cup \{...\}}$$

$$\text{parameters}: \frac{}{[] \in \text{Parameters}} \qquad \text{parameters}: \frac{p \in \text{Parameter},\ ps \in \text{Parameters}}{p::ps \in \text{Parameters}}$$

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

$$\text{variables}: \frac{}{[] \in \text{Variables}} \qquad \text{variables}: \frac{v \in \text{Variable},\ vs \in \text{Variables}}{v::vs \in \text{Variables}}$$

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

$$\text{values}: \frac{}{[] \in \text{Values}} \qquad \text{values}: \frac{v \in \text{Value},\ vs \in \text{Values}}{v::vs \in \text{Values}}$$

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

$$\text{statements}: \frac{}{[] \in \text{Statements}} \qquad \text{statements}: \frac{s \in \text{Statement},\ ss \in \text{Statements}}{s::ss \in \text{Statements}}$$

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

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

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

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

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

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

Number represents **real** numbers

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

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. 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

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

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**

$$\texttt{field}: \frac{\texttt{v} \in \texttt{Expression}}{\texttt{v} \in \texttt{Field}} \qquad \texttt{field}: \frac{\texttt{k} \in \texttt{Name} \cup \texttt{Expression} \backslash \{\texttt{niltype(nil)}\}, \ \texttt{v} \in \texttt{Expression}}{\langle \texttt{k}, \texttt{v} \rangle \in \texttt{Field}}$$

$$\texttt{fields}: \frac{}{[] \in \texttt{Fields}} \qquad \texttt{fields}: \frac{\texttt{f} \in \texttt{Field}, \ \texttt{fs} \in \texttt{Fields}}{\texttt{f::fs} \in \texttt{Fields}} \qquad \texttt{table}: \frac{\texttt{fs} \in \texttt{Fields}}{\texttt{table(fs)} \in \texttt{Object}}$$

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

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

In Lua, **tables** and **functions** are objects: variables do not contain these values but **references** to them. In Prolua, a reference is a ⟨**type, 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

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

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

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

An expression can be **enclosed** in parentheses

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

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

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

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

$$\texttt{expression}: \frac{\texttt{e} \in \texttt{Expression},\ \texttt{k} \in \texttt{Expression} \setminus \{\texttt{niltype(nil)}\}}{\texttt{access(e, k)} \in \texttt{Expression}}$$

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

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

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

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

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

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

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

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

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

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

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, and the same is done in Prolua

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

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

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

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

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

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

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

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

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

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

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

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

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

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, 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] Define intrinsic functions.
error, print, type, ipairs, pairs, etc...

Now 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
```

...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, both clauses on their own don't 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 **stack** of execution contexts and a memory **pool** to store objects.

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

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

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 pushed onto the stack and when the same function exits, the context is popped from the stack and eventually destroyed, as well as the values included therewithin.

To say that the stack contains execution contexts is misleading. In reality, when an execution context is created it is stored in the memory pool and a reference to it is pushed onto the stack. It is therefore more correct to say that the stack contains **references** to execution contexts. The reason we store references on the stack, and not actual contexts, 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. If multiple closures store a copy of the same environment multiple times, then the memory footprint grows rapidly.

With that said, the **stack** of references is defined syntactically as

$$\texttt{stack}:\frac{}{[] \in \texttt{Stack}} \qquad \texttt{stack}:\frac{r_i \in \texttt{Reference} \quad \forall i \geq 0}{\texttt{stack}(r_0 :: r_1 :: r_2 :: \ \ldots\ :: r_n) \in \texttt{Stack}}$$

The memory **pool** is memory reserved for **contexts**, **tables** and **functions**. A pool is a collection of contiguous memory blocks defined as ⟨**address, reference count, data**⟩ 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

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

A collection of memory blocks is defined syntactically as

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

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.

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

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.

Explain why the design of the memory pool and the list data structure (linearity) yields poor performance!

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

$$\texttt{environment}: \frac{\texttt{stack} \in \texttt{Stack},\ \ \texttt{pool} \in \texttt{Pool}}{\langle \texttt{stack},\ \texttt{pool} \rangle \in \texttt{Environment}}$$

A few predicates that help manipulate the stack and pool are defined in the **standard.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 Prolua'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

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

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 since it is a fact, and therefore needs no proof. The recurrence relation also states that **n!**, such that **n** is greater than zero, is equal to **n** $\times$ **(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** $\times$ **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.
```

DOCUMENTATION PASSED THIS POINT HAS NOT BEEN REVISED YET

III.c. Expression evaluation

[TODO] Explain the evaluation of an expression.
(ENV, e, ENV1, r)

Expressions can be left or right-hand side due to the **assignment** statement and this affects the result of an evaluation. Left-hand side expressions return an address where we can store or read data from, while a right-hand side expression returns zero or more values. In Lua, all expressions are right-hand side, but only the **variable** and **access** expressions can be left-hand side since they're the only constructs where values can be stored.

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 addresses

$$\frac{}{\langle ENV_0,\ expressions([])\rangle \overset{lhs}{\Rightarrow} \langle ENV_0,\ []\rangle}$$

If an error occurs while evaluating an expression, do not evaluate any remaining expressions and return the error

$$\frac{\langle ENV_0,\ e\rangle \overset{lhs}{\Rightarrow} \langle ENV_1,\ \textbf{ERROR}\rangle}{\langle ENV_0,\ expressions(e::es)\rangle \overset{lhs}{\Rightarrow} \langle ENV_1,\ \textbf{ERROR}\rangle}$$

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

$$\frac{\langle ENV_0,\ e\rangle \overset{lhs}{\Rightarrow} \langle ENV_1,\ \langle ECID,\ K\rangle\rangle \wedge \langle ENV_1,\ expressions(es)\rangle \overset{lhs}{\Rightarrow} \langle ENV_2,\ \textbf{ERROR}\rangle}{\langle ENV_0,\ expressions(e::es)\rangle \overset{lhs}{\Rightarrow} \langle ENV_2,\ \textbf{ERROR}\rangle}$$

If no error occurs during evaluation, then a list of addresses is returned

$$\frac{\langle ENV_0,\ e\rangle \overset{lhs}{\Rightarrow} \langle ENV_1,\ \langle ECID,\ K\rangle\rangle \quad \langle ENV_1,\ expressions(es)\rangle \overset{lhs}{\Rightarrow} \langle ENV_2,\ AS\rangle}{\langle ENV_0,\ expressions(e::es)\rangle \overset{lhs}{\Rightarrow} \langle ENV_2,\ \langle ECID,\ K\rangle::AS\rangle}$$

Evaluating a **list of right-hand side expressions** is almost similar in that it returns a list of values (instead of memory addresses), 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 an expression is used as the last (or the only) element of a list of right-hand side expressions then no adjustment is made

$$\frac{\langle ENV_0,\ e\rangle \overset{rhs}{\Rightarrow} \langle ENV_1,\ VS\rangle}{\langle ENV_0,\ expressions(e::[])\rangle \overset{rhs}{\Rightarrow} \langle ENV_1,\ VS\rangle}$$

In all other contexts, the result of the evaluation of the right-hand side expression is truncated to one element, thereby discarding all values but the first

$$\frac{\forall\, es \neq [] \quad \langle ENV_0,\ e\rangle \overset{rhs}{\Rightarrow} \langle ENV_1,\ V_e :: VS_e\rangle \quad \langle ENV_1,\ expressions(es)\rangle \overset{rhs}{\Rightarrow} \langle ENV_2,\ VS_{es}\rangle}{\langle ENV_0,\ expressions(e::es)\rangle \overset{rhs}{\Rightarrow} \langle ENV_2,\ V_e :: VS_{es}\rangle}$$

Evaluating a **value** -- a subset of expressions -- returns the same value, nothing more

$$\frac{}{\langle ENV_0,\ V\rangle \overset{rhs}{\Rightarrow} \langle ENV_0,\ V::[]\rangle \quad V \in Value}$$

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

$$\frac{\begin{array}{c} fs \in Fields \quad \langle ENV_0,\ fields(fs)\rangle \overset{rhs}{\Rightarrow} \langle ENV_1,\ VS\rangle \\ T = table(VS) \\ \langle ENV_1,\ add(T)\rangle \overset{heap}{\Rightarrow} \langle ENV_2,\ Address\rangle \end{array}}{\langle ENV_0,\ tableconstructor(fs)\rangle \overset{rhs}{\Rightarrow} \langle ENV_2,\ referencetype(table,\ Address)::[]\rangle}$$

A list of expressions is a subset of a list of fields, by the definition given in the concrete syntax. Consequently, the **table constructor** can also create a new table from a list of expressions. Evaluating a list of expressions generates a list of values but no keys, so a key must be generated for each value we wish to store in the table

$$\frac{\begin{array}{c} es \in Expressions \quad \langle ENV_0,\ expressions(es)\rangle \overset{rhs}{\Rightarrow} \langle ENV_1,\ VS\rangle \\ build(VS) \overset{map}{\Rightarrow} Mapping \quad T = table(Mapping) \\ \langle ENV_1,\ add(T)\rangle \overset{heap}{\Rightarrow} \langle ENV_2,\ Address\rangle \end{array}}{\langle ENV_0,\ tableconstructor(es)\rangle \overset{rhs}{\Rightarrow} \langle ENV_2,\ referencetype(table,\ Address)::[]\rangle}$$

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

$$\frac{\langle ENV_0,\ e\rangle \overset{rhs}{\Rightarrow} \langle ENV_1,\ V::VS\rangle}{\langle ENV_0,\ enclosed(e)\rangle \overset{rhs}{\Rightarrow} \langle ENV_1,\ V::[]\rangle} \qquad \frac{\langle ENV_0,\ e\rangle \overset{rhs}{\Rightarrow} \langle ENV_1,\ []\rangle}{\langle ENV_0,\ enclosed(e)\rangle \overset{rhs}{\Rightarrow} \langle ENV_1,\ []\rangle}$$

And like always if there's an error, return it

$$\frac{\langle ENV_0,\ e\rangle \overset{rhs}{\Rightarrow} \langle ENV_1,\ \textbf{ERROR}\rangle}{\langle ENV_0,\ enclosed(e)\rangle \overset{rhs}{\Rightarrow} \langle ENV_1,\ \textbf{ERROR}\rangle}$$

Evaluating a left-hand side variable means retrieving an execution context and key that correspond to the variable. If the variable name, which serves as a key is defined in the current execution context, then we return the context identifer and the variable name

$$\frac{\langle \texttt{EC, keyExists(n)} \rangle \overset{\text{env}}{\Rightarrow} \texttt{true} \quad \texttt{getIdentifier(EC)} \overset{\text{env}}{\Rightarrow} \texttt{ECID}}{\langle \texttt{EC::ECS, variable(n)} \rangle \overset{\text{lhs}}{\Rightarrow} \langle \texttt{EC::ECS, } \langle \texttt{ECID, n} \rangle \rangle}$$

If the variable name cannot be found in the current execution context, then we check the next

$$\frac{\langle \texttt{EC, keyExists(n)} \rangle \overset{\text{env}}{\Rightarrow} \texttt{false} \quad \langle \texttt{ECS, variable(n)} \rangle \overset{\text{lhs}}{\Rightarrow} \langle \texttt{ECS, } \langle \texttt{ECID, n} \rangle \rangle}{\langle \texttt{EC::ECS, variable(n)} \rangle \overset{\text{lhs}}{\Rightarrow} \langle \texttt{EC::ECS, } \langle \texttt{ECID, n} \rangle \rangle}$$

A key always exists in the global execution context, even if it wasn't explicitly defined

$$\frac{\texttt{getIdentifier(EC)} \overset{\text{env}}{\Rightarrow} \texttt{ECID}}{\langle \texttt{EC::[], variable(n)} \rangle \overset{\text{lhs}}{\Rightarrow} \langle \texttt{EC::[], } \langle \texttt{ECID, n} \rangle \rangle}$$

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

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

If there's an error finding the address, return it

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

Getting the memory address of a table field is just like finding the address of a variable, with a few extra steps. We're given two expressions that should evaluate into a table reference and a field key. We evaluate both expressions and then return the reference-key pair, all the while watching out for errors. If evaluating the expression that returns a key reference returns an error then that error is returned

$$\frac{\langle \texttt{ETS}, \texttt{RS}, \texttt{r} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ETS}_1, \textbf{ERROR} \rangle}{\langle \texttt{ETS}, \texttt{RS}, \texttt{access(r, k)} \rangle \overset{\text{lhs}}{\Rightarrow} \langle \texttt{ETS}_1, \textbf{ERROR} \rangle}$$

If the expression that should evaluate into a table key fails, then the error is returned

$$\frac{\langle \texttt{ETS}, \texttt{RS}, \texttt{r} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ETS}_1, \texttt{R::[]} \rangle \quad \langle \texttt{ETS}_1, \texttt{RS}, \texttt{k} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ETS}_2, \textbf{ERROR} \rangle}{\langle \texttt{ETS}, \texttt{RS}, \texttt{access(r, k)} \rangle \overset{\text{lhs}}{\Rightarrow} \langle \texttt{ETS}_2, \textbf{ERROR} \rangle}$$

If both expressions evaluate correctly, then the reference-key pair is returned

$$\frac{\langle \texttt{ETS}, \texttt{RS}, \texttt{r} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ETS}_1, \texttt{R::[]} \rangle \quad \langle \texttt{ETS}_1, \texttt{RS}, \texttt{k} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ETS}_2, \texttt{V::VS} \rangle}{\langle \texttt{ETS}, \texttt{RS}, \texttt{access(r, k)} \rangle \overset{\text{lhs}}{\Rightarrow} \langle \texttt{ETS}_2, \langle \texttt{R}, \texttt{V} \rangle \rangle}$$

And just like variables, **returning the value of a table field** is made simple once we know the field's memory address. We do have to watch out for errors

$$\frac{\langle \texttt{ETS}, \texttt{RS}, \texttt{access(r, k)} \rangle \overset{\text{lhs}}{\Rightarrow} \langle \texttt{ETS}_1, \textbf{ERROR} \rangle}{\langle \texttt{ETS}, \texttt{RS}, \texttt{access(r, k)} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ETS}_1, \textbf{ERROR} \rangle}$$

If there's no error retrieving an address, then we can safely return a value

$$\frac{\langle \texttt{ENV}_0, \texttt{access(r, k)} \rangle \overset{\text{lhs}}{\Rightarrow} \langle \texttt{ENV}_1, \langle \texttt{ECID}, \texttt{n} \rangle \rangle \quad \langle \texttt{ENV}_1, \texttt{getValue(ECID, n)} \rangle \overset{\text{env}}{\Rightarrow} \texttt{V}}{\langle \texttt{ETS}, \texttt{RS}, \texttt{access(r, k)} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ETS}_1, \texttt{V::[]} \rangle}$$

Variadic expressions evaluate into a list of values. The evaluation of a variadic expression consists of finding out whether the key "**...**" is defined in the current scope. If it is, then its values are returned
Check semantics.pl source for updated version!

$$\frac{\langle \texttt{EC}, \texttt{keyExists(...)} \rangle \overset{\text{env}}{\Rightarrow} \texttt{true} \quad \langle \texttt{EC}, \texttt{getValue(...)} \rangle \overset{\text{env}}{\Rightarrow} \texttt{VS}}{\langle \texttt{EC::ECS}, \texttt{...} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{EC::ECS}, \texttt{VS} \rangle}$$

If on the other hand the key is undefined in the current scope, then we check the outer scope and so forth. Eventually, this lookup will end since the "**...**" key, like any other, is guaranteed to exist in the global scope

$$\frac{\langle \texttt{EC}, \texttt{keyExists(...)} \rangle \overset{\text{env}}{\Rightarrow} \texttt{false} \quad \langle \texttt{ECS}, \texttt{...} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ECS}, \texttt{VS} \rangle}{\langle \texttt{EC::ECS}, \texttt{...} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{EC::ECS}, \texttt{VS} \rangle}$$

Unary operators
Binary operators

When a **function is defined**, a closure is created with given parameters, statements and an empty execution environment attached to it. It is then stored in the current context and a reference to it is returned

$$\frac{\texttt{ENV}_f = []}{\langle \texttt{ENV}_0, \texttt{functiondef(ps, ss)} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ENV}_0, \texttt{function(ps, ss, ENV}_f\texttt{)::[]} \rangle}$$

On the other hand, a **function call** creates a new scope in which it will evaluate a function. The created scope is then discarded 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

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

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

$$\frac{\langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{V::VS}\rangle \quad \forall \mathtt{V} \notin \mathtt{Function}}{\langle \mathtt{ENV}_0,\ \mathtt{functioncall(e,\ es)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathbf{ERROR}\rangle}$$

If evaluating the function arguments returns an error, return it

$$\frac{\langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{function(ps,\ ss,\ ENV_f)::VS}\rangle \quad \langle \mathtt{ENV}_1,\ \mathtt{explist(es)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathbf{ERROR}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{functioncall(e,\ es)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathbf{ERROR}\rangle}$$

If the function arguments evaluate without a problem, then we can evaluate the actual function. A function with an empty execution environment attached to it inherits the environment from its callee

$$\frac{\begin{array}{c}\langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{function(ps,\ ss,\ [])::VS_e}\rangle \quad \langle \mathtt{ENV}_1,\ \mathtt{explist(es)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{VS_{es}}\rangle \\ \langle \mathtt{ENV}_2,\ \mathtt{addContext(ps,\ VS_{es})}\rangle \overset{\mathtt{env}}{\Rightarrow} \mathtt{ENV}_3 \\ \langle \mathtt{ENV}_3,\ \mathtt{statementlist(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{EC::ENV}_4,\ \mathtt{CTRL},\ \mathtt{VS_{ss}}\rangle \ \mathtt{st\ CTRL} \in \{\mathtt{continue},\ \mathtt{return},\ \mathtt{error}\}\end{array}}{\langle \mathtt{ENV}_0,\ \mathtt{functioncall(e,\ es)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_4,\ \mathtt{VS_{ss}}\rangle}$$

In the case where a function has it's own execution environment, this is the environment where the statement block will be evaluated

$$\frac{\begin{array}{c}\langle \mathtt{ENV}_0,\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_1,\ \mathtt{function(ps,\ ss,\ }\mathbf{ENV_f}\mathtt{)::VS_e}\rangle \quad \langle \mathtt{ENV}_1,\ \mathtt{explist(es)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{VS_{es}}\rangle \\ \langle \mathbf{ENV_f},\ \mathtt{addContext(ps,\ VS_{es})}\rangle \overset{\mathtt{env}}{\Rightarrow} \mathtt{ENV}_3 \\ \langle \mathtt{ENV}_3,\ \mathtt{statementlist(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{EC::ENV}_4,\ \mathtt{CTRL},\ \mathtt{VS_{ss}}\rangle \ \mathtt{st\ CTRL} \in \{\mathtt{continue},\ \mathtt{return},\ \mathtt{error}\}\end{array}}{\langle \mathtt{ENV}_0,\ \mathtt{functioncall(e,\ es)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ENV}_4,\ \mathtt{VS_{ss}}\rangle}$$

III.d. Statement evaluation

Evaluating a statement requires an environment, a reference to the current scope and the statement to evaluate. It returns the modified environment, a flag which controls the flow of the program, and zero or more return values.

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 with the **evaluation of a list of statements**. An empty list of statements does not modify the environment or return a value

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

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

$$\frac{\langle ENV_0,\ s\rangle \overset{stat}{\Rightarrow} \langle ENV_1,\ CTRL,\ VS\rangle \quad CTRL \in \{\texttt{return}, \texttt{break}, \texttt{error}\}}{\langle ENV_0,\ \texttt{statementlist(s::ss)}\rangle \overset{stat}{\Rightarrow} \langle ENV_1,\ CTRL,\ VS\rangle}$$

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

$$\frac{\langle ENV_0,\ s\rangle \overset{stat}{\Rightarrow} \langle ENV_1,\ \texttt{continue},\ VS_s\rangle \quad \langle ENV_1,\ \texttt{statementlist(ss)}\rangle \overset{stat}{\Rightarrow} \langle ENV_2,\ CTRL,\ VS_{ss}\rangle}{\langle ENV_0,\ \texttt{statementlist(s::ss)}\rangle \overset{stat}{\Rightarrow} \langle ENV_2,\ CTRL,\ VS_{ss}\rangle}$$

The **assignment statement** allows multiple assignments. Before assigning any values, all expressions are evaluated so that we have two lists, one with memory addresses and the other with values to store. If there's an error while evaluating the left-hand side expressions, return it

$$\frac{\langle ETS,\ RS,\ es_{lhs}\rangle \overset{lhs}{\Rightarrow} \langle ETS_1,\ \textbf{ERROR}\rangle}{\langle ETS,\ RS,\ \texttt{assign}(es_{lhs},\ es_{rhs})\rangle \overset{stat}{\Rightarrow} \langle ETS_1,\ \texttt{error},\ \textbf{ERROR})\rangle}$$

Similarly, if there's an error while evaluating the right-hand side expressions, handle it the same way

$$\frac{\langle ETS,\ RS,\ es_{lhs}\rangle \overset{lhs}{\Rightarrow} \langle ETS_1,\ VS_{lhs}\rangle \quad \langle ETS_1,\ RS,\ es_{rhs}\rangle \overset{rhs}{\Rightarrow} \langle ETS_2,\ \textbf{ERROR}\rangle}{\langle ETS,\ RS,\ \texttt{assign}(es_{lhs},\ es_{rhs})\rangle \overset{stat}{\Rightarrow} \langle ETS_2,\ \texttt{error},\ \textbf{ERROR})\rangle}$$

If no errors occured, then assign the values to the variables

$$\frac{\langle ETS,\ RS,\ es_{lhs}\rangle \overset{lhs}{\Rightarrow} \langle ETS_1,\ RS,\ VS_{lhs}\rangle \quad \langle ETS_1,\ RS,\ es_{rhs}\rangle \overset{rhs}{\Rightarrow} \langle ETS_2,\ VS_{rhs}\rangle \quad \langle ETS_2,\ \texttt{setvalues}(VS_{lhs},\ VS_{rhs})\rangle \overset{env}{\Rightarrow} ETS_3}{\langle ETS,\ RS,\ \texttt{assign}(es_{lhs},\ es_{rhs})\rangle \overset{stat}{\Rightarrow} \langle ETS_3,\ \texttt{continue},\ \texttt{[]})\rangle}$$

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 scope. When evaluation is done, the new scope is discarded. Even though a new scope is created, this doesn't mean that outer scopes won't be modified

$$\frac{\langle \mathtt{ENV}_0,\ \mathtt{addContext}\rangle \overset{\mathtt{env}}{\Rightarrow} \mathtt{ENV}_1 \quad \langle \mathtt{ENV}_1,\ \mathtt{statementlist(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{EC::ENV}_2,\ \mathtt{CTRL},\ \mathtt{VS}\rangle}{\langle \mathtt{ENV}_0,\ \mathtt{do(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ENV}_2,\ \mathtt{CTRL},\ \mathtt{VS}\rangle}$$

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

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

If the condition expression evaluates into either **nil** or **false**, then the while loop is broken

$$\frac{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{V::VS}\rangle \quad \mathtt{V} \in \{\mathtt{niltype(nil)},\ \mathtt{booleantype(false)}\}}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{while(e,\ ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{continue},\ []\rangle}$$

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

$$\frac{\begin{array}{l}\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{V::VS}\rangle \quad \mathtt{V} \notin \{\mathtt{niltype(nil)},\ \mathtt{booleantype(false)}\} \\ \langle \mathtt{ETS}_1,\ \mathtt{RS},\ \mathtt{do(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ETS}_2,\ \mathtt{C},\ \mathtt{VS}_{\mathtt{ss}}\rangle \quad \mathtt{C} \in \{\mathtt{return},\ \mathtt{error}\}\end{array}}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{while(e,\ ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ETS}_2,\ \mathtt{C},\ \mathtt{VS}_{\mathtt{ss}}\rangle}$$

If however the loop is broken by a **break** statement, then no more statements are evaluated but the **continue** control flag is returned

$$\frac{\begin{array}{l}\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{e}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{V::VS}\rangle \quad \mathtt{V} \notin \{\mathtt{niltype(nil)},\ \mathtt{booleantype(false)}\} \\ \langle \mathtt{ETS}_1,\ \mathtt{RS},\ \mathtt{do(ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ETS}_2,\ \mathtt{break},\ []\rangle\end{array}}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{while(e,\ ss)}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ETS}_2,\ \mathtt{C},\ \mathtt{VS}_{\mathtt{ss}}\rangle}$$

If the **while-do** loop flow is not broken, we keep evaluating its body until it is

$$\frac{\langle \text{ETS}, \text{RS}, \text{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1, \text{V::VS}\rangle \quad \text{V} \notin \{\text{niltype(nil)}, \text{booleantype(false)}\} \quad \langle \text{ETS}_1, \text{RS}, \text{do(ss)}\rangle \overset{\text{stat}}{\Rightarrow} \langle \text{ETS}_2, \text{continue}, []\rangle \quad \langle \text{ETS}_2, \text{RS}, \text{while(e, ss)}\rangle \overset{\text{stat}}{\Rightarrow} \langle \text{ETS}_3, \text{C}, \text{VS}_{ss}\rangle}{\langle \text{ETS}, \text{RS}, \text{while(e, ss)}\rangle \overset{\text{stat}}{\Rightarrow} \langle \text{ETS}_3, \text{C}, \text{VS}_{ss}\rangle}$$

The **repeat-until** loop repeatedly executes a scoped block of statements until a given 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

$$\frac{\langle \text{ETS}, \text{RS}, \text{make}\rangle \overset{\text{env}}{\Rightarrow} \langle \text{ETS}_1, \text{RS}_1\rangle \quad \langle \text{ETS}_1, \text{RS}_1, \text{ss}\rangle \overset{\text{stat}}{\Rightarrow} \langle \text{ETS}_2, \text{C}, \text{VS}\rangle \quad \text{C} \in \{\text{return}, \text{error}\}}{\langle \text{ETS}, \text{RS}, \text{repeat(e, ss)}\rangle \overset{\text{evaluate}}{\Rightarrow} \langle \text{ETS}_2, \text{C}, \text{VS}\rangle}$$

If it's broken with the **break** statement then the **continue** control flag and a value are returned

$$\frac{\langle \text{ETS}, \text{RS}, \text{make}\rangle \overset{\text{env}}{\Rightarrow} \langle \text{ETS}_1, \text{RS}_1\rangle \quad \langle \text{ETS}_1, \text{RS}_1, \text{ss}\rangle \overset{\text{stat}}{\Rightarrow} \langle \text{ETS}_2, \text{break}, []\rangle}{\langle \text{ETS}, \text{RS}, \text{repeat(e, ss)}\rangle \overset{\text{evaluate}}{\Rightarrow} \langle \text{ETS}_2, \text{continue}, []\rangle}$$

If the condition expression results in an error, then the loop is broken

$$\frac{\langle \text{ETS}, \text{RS}, \text{make}\rangle \overset{\text{env}}{\Rightarrow} \langle \text{ETS}_1, \text{RS}_1\rangle \quad \langle \text{ETS}_1, \text{RS}_1, \text{ss}\rangle \overset{\text{stat}}{\Rightarrow} \langle \text{ETS}_2, \text{continue}, []\rangle \quad \langle \text{ETS}_2, \text{RS}_1, \text{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_3, \textbf{ERROR}\rangle}{\langle \text{ETS}, \text{RS}, \text{repeat(e, ss)}\rangle \overset{\text{stat}}{\Rightarrow} \langle \text{ETS}_3, \text{error}, \textbf{ERROR}\rangle}$$

The loop is broken if the condition expression is considered true

$$\frac{\langle \text{ETS}, \text{RS}, \text{make}\rangle \overset{\text{env}}{\Rightarrow} \langle \text{ETS}_1, \text{RS}_1\rangle \quad \langle \text{ETS}_1, \text{RS}_1, \text{ss}\rangle \overset{\text{stat}}{\Rightarrow} \langle \text{ETS}_2, \text{continue}, []\rangle \quad \langle \text{ETS}_2, \text{RS}_1, \text{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_3, \text{V::VS}\rangle \quad \text{V} \notin \{\text{niltype(nil)}, \text{booleantype(false)}\}}{\langle \text{ETS}, \text{RS}, \text{repeat(e, ss)}\rangle \overset{\text{stat}}{\Rightarrow} \langle \text{ETS}_3, \text{continue}, []\rangle}$$

If the condition expression is false, then we keep repeating the loop.

$$\frac{\langle \text{ETS}, \text{RS}, \text{make}\rangle \overset{\text{env}}{\Rightarrow} \langle \text{ETS}_1, \text{RS}_1\rangle \quad \langle \text{ETS}_1, \text{RS}_1, \text{ss}\rangle \overset{\text{stat}}{\Rightarrow} \langle \text{ETS}_2, \text{continue}, []\rangle \quad \langle \text{ETS}_2, \text{RS}_1, \text{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_3, \text{V::VS}\rangle \quad \text{V} \in \{\text{niltype(nil)}, \text{booleantype(false)}\} \quad \langle \text{ETS}_3, \text{RS}, \text{repeat(e, ss)}\rangle \overset{\text{stat}}{\Rightarrow} \langle \text{ETS}_4, \text{C}, \text{VS}\rangle}{\langle \text{ETS}, \text{RS}, \text{repeat(e, ss)}\rangle \overset{\text{stat}}{\Rightarrow} \langle \text{ETS}_4, \text{C}, \text{VS}\rangle}$$

If control structures evaluate a condition expression that specifies a statement to run depending on the result. If the evaluation returns an error, no statement is evaluated and an error is returned

$$\frac{\langle \texttt{ETS}, \texttt{RS}, \texttt{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ETS}_1, \textbf{ERROR}\rangle}{\langle \texttt{ETS}, \texttt{RS}, \texttt{if(e, } s_{true}\texttt{, } s_{false}\texttt{)}\rangle \overset{\text{stat}}{\Rightarrow} \langle \texttt{ETS}_1, \texttt{error}, \textbf{ERROR}\rangle}$$

If the condition expression evaluates to either **nil** or **false**, then the false statement is evaluated

$$\frac{\langle \texttt{ETS}, \texttt{RS}, \texttt{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ETS}_1, \texttt{V}_e\texttt{::VS}\rangle \quad \texttt{V}_e \in \{\texttt{niltype(nil)}, \texttt{booleantype(false)}\} \quad \langle \texttt{ETS}_1, \texttt{RS}, s_{false}\rangle \overset{\text{stat}}{\Rightarrow} \langle \texttt{ETS}_2, \texttt{C}, \texttt{VS}_s\rangle}{\langle \texttt{ETS}, \texttt{RS}, \texttt{if(e, } s_{true}\texttt{, } s_{false}\texttt{)}\rangle \overset{\text{stat}}{\Rightarrow} \langle \texttt{ETS}_2, \texttt{C}, \texttt{VS}_s\rangle}$$

Otherwise, the true statement is evaluated

$$\frac{\langle \texttt{ETS}, \texttt{RS}, \texttt{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ETS}_1, \texttt{V}_e\texttt{::VS}\rangle \quad \texttt{V}_e \notin \{\texttt{niltype(nil)}, \texttt{booleantype(false)}\} \quad \langle \texttt{ETS}_1, \texttt{RS}, s_{true}\rangle \overset{\text{stat}}{\Rightarrow} \langle \texttt{ETS}_2, \texttt{C}, \texttt{VS}_s\rangle}{\langle \texttt{ETS}, \texttt{RS}, \texttt{if(e, } s_{true}\texttt{, } s_{false}\texttt{)}\rangle \overset{\text{stat}}{\Rightarrow} \langle \texttt{ETS}_2, \texttt{C}, \texttt{VS}_s\rangle}$$

Declaring a local variable creates a field inside the current scope with a given value. If the evaluation of the value expression returns an error, then propagate it. However if the value is valid, then it is stored

$$\frac{\langle \texttt{ETS}, \texttt{RS}, \texttt{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ETS}_1, \textbf{ERROR}\rangle}{\langle \texttt{ETS}, \texttt{RS}, \texttt{localvariable(n, e)}\rangle \overset{\text{stat}}{\Rightarrow} \langle \texttt{ETS}_1, \texttt{error}, \textbf{ERROR}\rangle}$$

$$\frac{\langle \texttt{ETS}, \texttt{R::RS}, \texttt{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ETS}_1, \texttt{V::VS}\rangle \quad \langle \texttt{ETS}_1, \texttt{setvalue(R, n, V)}\rangle \overset{\text{env}}{\Rightarrow} \texttt{ETS}_2}{\langle \texttt{ETS}, \texttt{R::RS}, \texttt{localvariable(n, e)}\rangle \overset{\text{stat}}{\Rightarrow} \langle \texttt{ETS}_2, \texttt{continue}, \texttt{[]}\rangle}$$

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

$$\frac{\langle \texttt{ENV}_0\texttt{, explist(es)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ENV}_1, \textbf{ERROR}\rangle}{\langle \texttt{ENV}_0\texttt{, return(es)}\rangle \overset{\text{stat}}{\Rightarrow} \langle \texttt{ENV}_1, \texttt{error}, \textbf{ERROR}\rangle}$$

$$\frac{\langle \texttt{ENV}_0\texttt{, explist(es)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ENV}_1, \texttt{VS}\rangle}{\langle \texttt{ENV}_0\texttt{, return(es)}\rangle \overset{\text{stat}}{\Rightarrow} \langle \texttt{ENV}_1, \texttt{return}, \texttt{VS}\rangle}$$

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 return the break control

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

[TODO] Explain evaluating a chunk

% Remember that a chunk is handled as the body of an anonymous function with a
% variable number of arguments. All we do is evaluate a function call to this
% anonymous function and print the return value and since we wish to view the
% return value, the function call is evaluated as an expression rather than a
% statement.

Garbage collection

The reference count is used to implement some form of garbage collection: when the reference count is zero, then the memory block is disposed of.