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

No garbage collection.

I.d. Notation

[TODO] Explain notation and how to read it.

Chapter II. Syntax

Our first order of business is to define an abstract syntax specific to Prolua 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 and come up with an abstract syntax of our own. Below is Lua's concrete syntax in EBNF

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

block ::= chunk

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

laststat ::= return [explist] | break

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

varlist ::= var {‘,’ var}

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

namelist ::= Name {‘,’ Name}

explist ::= exp {‘,’ exp}

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

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

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

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

function ::= function funcbody

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

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

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

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

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

fieldsep ::= ‘,’ | ‘;’

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

unop ::= ‘-’ | not | ‘#’
```

II.a. Sets

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

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

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

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

Let ***ParName*** be the set of all possible parameter names and an extension of ***Name*** to include the name "**...**", then let ***Parlist*** be a list of parameter names such that

$$\text{parname}: \frac{}{\text{Parname} = \text{Name} \cup \{"..."\}} \qquad \text{parlist}: \frac{}{[] \in \text{Parlist}}$$

$$\text{parlist}: \frac{p \in \text{Parname},\ \ ps \in \text{Parlist}}{p::ps \in \text{Parlist}}$$

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

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

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

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

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

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

Let a ***Reference*** be a positive non-zero integer, and ***Referencestack*** a stack of references such that

$$\text{referencestack}: \frac{}{[] \in \text{ReferenceStack}}$$

$$\text{referencestack}: \frac{r \in \text{Reference},\ \ rs \in \text{ReferenceStack}}{r::rs \in \text{ReferenceStack}}$$

II.b. Values and Types

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

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

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

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

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

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

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

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

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

In Lua, tables are not passed by value but by **reference**: a positive non-zero integer that indexes a table's position in the environment

$$\text{value}: \frac{n \in \mathbb{N}_+^*}{\text{referencetype}(n) \in \text{Value}}$$

Functions are defined syntactically as

$$\text{value}: \frac{ps \in \text{Parlist},\ \ ss \in \text{Statementlist},\ \ rs \in \text{Referencestack}}{\text{functiontype}(ps, ss, rs) \in \text{Value}}$$

II.c. Expressions

An expression can be **enclosed**, for lack of a better word, in parentheses

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

Variables store values. To be able to retrieve a value of a variable, we need the variable's name

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

We define the table **field** expression which, much like the variable expression, retrieves the address or a value for a given key in a given table

$$\text{expression}: \frac{t, k \in \text{Expression}}{\text{field}(t, k) \in \text{Expression}}$$

Lua defines a **variadic expression** represented by three dots "**...**" which is a placeholder for a list of values.

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

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

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

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

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

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

Function definitions are defined as

$$\text{expression}: \frac{ps \in \text{Parlist}, \quad ss \in \text{Statementlist}}{\text{function}(ps, ss) \in \text{Expression}}$$

Function calls are defined as

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

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, and the same is done in Prolua

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

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

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

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

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

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

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

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

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

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

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

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** loop which repeats a block of code while a control variable runs through an arithmetic progression

$$\texttt{statement}: \frac{\texttt{n} \in \texttt{Name},\ \ \texttt{e}_{\texttt{initial}}, \texttt{e}_{\texttt{end}}, \texttt{e}_{\texttt{increment}} \in \texttt{Expression},\ \ \texttt{ss} \in \texttt{Statementlist}}{\texttt{for(n,e}_{\texttt{initial}}\texttt{,e}_{\texttt{end}}\texttt{,e}_{\texttt{increment}}\texttt{,ss)} \in \texttt{Statement}}$$

The second flavor is the **generic for** loop which works over iterator functions. On each iteration, the iterator function is called to produce a new value, stopping when this value is **nil**

$$\texttt{statement}: \frac{\texttt{ns} \in \texttt{Namelist},\ \ \texttt{es} \in \texttt{Explist},\ \ \texttt{ss} \in \texttt{Statementlist}}{\texttt{for(ns, es, ss)} \in \texttt{Statement}}$$

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.

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

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

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

The **break** statement explicitly breaks a loop

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

Now consider the following Lua program[1]...

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

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

t.min = toCelsius(5);

local i = 1;

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

t.max = toCelsius(5^5);

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

...and its generated Prolua program (formatted for readability)

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

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

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

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

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

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

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

The output is a Prolog fact that states that a **chunk** is a sequence of terms, which curiously resemble the previously defined abstract syntax. However, the fact on its own doesn't mean much since no relationships between the terms have been defined. This is where the Prolua interpreter comes in to 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**. To be able to execute our program, we need to define the evaluation semantics of a Prolua program

III.a. The environment

<ets, rs, add> $\Rightarrow_{env}$ <et::ets, r::rs>
<ets, rs, make> $\Rightarrow_{env}$ <et::ets, r::rs>
<ets, rs, make(keys, values)> $\Rightarrow_{env}$ <et::ets, r::rs>
<ets, gettable(reference)> $\Rightarrow_{env}$ Table
<ets, getvalue(reference, key)> $\Rightarrow_{env}$ Value
<ets, setvalue(reference, key, value)> $\Rightarrow_{env}$ ets_1
<ets, keyexists(reference, key)> $\Rightarrow_{env}$ {true, false}

III.b. Expression evaluation

When evaluating expressions, we should take note that the evaluation can result in left and right values. Take for example the following assignment

$$x = 3$$

The variable x should evaluate into a memory address where the value '3' will be stored. The variable is also known as a left value and the value '3' is known as a right value.

By this train of thought, only two of the defined expressions can be left-hand expressions:
variable
index

All expressions are right-hand expressions, even if they don't return results.

Evaluating a list of expressions

Values are considered expressions and their evaluation usually results in the same value

$$\frac{}{\langle \text{ETS, RS, V} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS, V} \rangle \quad \text{V} \in \{\text{Nil, Boolean, Number, String, Function}\}}$$

The only exception is the evaluation of a table that returns a reference to its location in the environment

$$\frac{\langle \text{ETS, RS, add(tabletype(T))} \rangle \overset{\text{env}}{\Rightarrow} \langle \text{ETS}_1\text{, R::RS} \rangle}{\langle \text{ETS, RS, tabletype(T)} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1\text{, R} \rangle}$$

As I mentioned earlier, expressions can be **enclosed in parentheses**. This usually means that in the case the expression evaluates into a non-empty list of values, only the first value is returned

$$\frac{\langle \text{ETS, RS, e} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1\text{, V::VS} \rangle}{\langle \text{ETS, RS, enclosed(e)} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1\text{, V} \rangle}$$

In case the expression evaluates into an error, propagate the error

$$\frac{\langle \text{ETS, R, e} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1\text{, } \textbf{ERROR} \rangle}{\langle \text{ETS, R, enclosed(e)} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1\text{, } \textbf{ERROR} \rangle}$$

An enclosed expression returns no results if the evaluated expression returns none

$$\frac{\langle \text{ETS, R, e} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1\text{, []} \rangle}{\langle \text{ETS, R, enclosed(e)} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1\text{, []} \rangle}$$

Evaluating a left-hand side variable means retrieving a field in an environment table where a value can be stored or retrieved, a memory address of sorts. If the variable name, which serves as

a field key is defined in the current scope, then we return the reference to the current scope and the name of the variable

$$\frac{\langle \text{ETS}, \text{keyexists}(\text{R}, \text{n}) \rangle \overset{\text{env}}{\Rightarrow} \text{true}}{\langle \text{E}, \text{R::RS}, \text{variable}(\text{n}) \rangle \overset{\text{lhs}}{\Rightarrow} \langle \text{ETS}, \langle \text{R}, \text{n} \rangle \rangle}$$

If the variable name cannot be found in the current scope, then we check the outer scope. This process is repeated until we find a scope where the variable name is defined, or we reach the global scope

$$\frac{\begin{array}{c}\langle \text{ETS}, \text{keyexists}(\text{R}_i, \text{n}) \rangle \overset{\text{env}}{\Rightarrow} \text{false} \\ \langle \text{ETS}, \text{RS}, \text{variable}(\text{n}) \rangle \overset{\text{lhs}}{\Rightarrow} \langle \text{ETS}, \langle \text{R}_k, \text{n} \rangle \rangle \quad \forall\ \text{i}, \text{k} \in \mathbb{N} \text{ st. } 1 \leq \text{k} < \text{i}\end{array}}{\langle \text{ETS}, \text{R}_i\text{::RS}, \text{variable}(\text{n}) \rangle \overset{\text{lhs}}{\Rightarrow} \langle \text{ETS}, \langle \text{R}_k, \text{n} \rangle \rangle}$$

If a query is made in the global scope, always return the reference to the aforementioned scope and the variable name, even if it hasn't been defined in this scope

$$\frac{}{\langle \text{ETS}, \text{R::[]}, \text{variable}(\text{n}) \rangle \overset{\text{lhs}}{\Rightarrow} \langle \text{ETS}, \langle \text{R}, \text{n} \rangle \rangle}$$

To be able to **return the value of a variable**, we need to know its address and then return the value at this address. Luckily half of the work has been done and all we need to do is retrieve the value from a given reference and key. When an address evaluation results in an error, we still need to propagate it

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

If no error is returned, we can proceed to retrieve the value

$$\frac{\langle \text{ETS}, \text{RS}, \text{variable}(\text{n}) \rangle \overset{\text{lhs}}{\Rightarrow} \langle \text{ETS}_1, \langle \text{R}, \text{n} \rangle \rangle \quad \langle \text{ETS}_1, \text{getvalue}(\text{R}, \text{n}) \rangle \overset{\text{env}}{\Rightarrow} \text{V}}{\langle \text{ETS}, \text{RS}, \text{variable}(\text{n}) \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1, \text{V} \rangle}$$

Getting the address of a table field is just like finding the address of a variable, with a few extra steps. First we need to evaluate the expression that results in a table reference. If there's an error in its evaluation, nothing else is evaluated and the error is propagated

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

If the expression that evaluates into a table key also fails, then the same error propagation is done

$$\frac{\langle \text{ETS}, \text{RS}, \text{t} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1, \text{R}_t \rangle \quad \langle \text{ETS}_1, \text{RS}, \text{k} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_2, \textbf{ERROR} \rangle}{\langle \text{ETS}, \text{RS}, \text{field}(\text{t}, \text{k}) \rangle \overset{\text{lhs}}{\Rightarrow} \langle \text{ETS}_2, \textbf{ERROR} \rangle}$$

If both expressions evaluate correctly, then both the reference and key are returned

$$\frac{\langle \text{ETS}, \text{RS}, \text{t} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1, \text{R}_t \rangle \quad \langle \text{ETS}_1, \text{RS}, \text{k} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_2, \text{V} \rangle}{\langle \text{ETS}, \text{RS}, \text{field}(\text{t}, \text{k}) \rangle \overset{\text{lhs}}{\Rightarrow} \langle \text{ETS}_2, \langle \text{R}_t, \text{V} \rangle \rangle}$$

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

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

If there's no error retrieving a reference and key, then we can safely return a value

$$\frac{\langle \text{ETS, RS, field(t, k)} \rangle \overset{\text{lhs}}{\Rightarrow} \langle \text{ETS}_1, \langle \text{R}_t, \text{V} \rangle \rangle \quad \langle \text{ETS}_1, \text{getvalue(R}_t, \text{V)} \rangle \overset{\text{env}}{\Rightarrow} \text{V}_2}{\langle \text{ETS, RS, field(t, k)} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1, \text{V}_2 \rangle}$$

Variadic expressions evaluate into a list of values if they exist. If the name "**...**" is defined in the current scope, then we return its list of values

$$\frac{\langle \text{ETS, keyexists(R, "...")} \rangle \overset{\text{env}}{\Rightarrow} \text{true} \quad \langle \text{ETS, getvalue(R, "...")} \rangle \overset{\text{env}}{\Rightarrow} \text{V}}{\langle \text{ETS, R::RS, "..."} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS, V} \rangle}$$

If on the other hand the name "**...**" does not exist in the current scope, then we check the outer scope. Eventually, this recursive lookup will end since the "**...**" name is guaranteed to exist in the global scope

$$\frac{\langle \text{ETS, keyexists(R, "...")} \rangle \overset{\text{env}}{\Rightarrow} \text{false} \quad \langle \text{ETS, RS, "..."} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS, V} \rangle}{\langle \text{ETS, R::RS, "..."} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS, V} \rangle}$$

Unary operators
Binary operators

When a **function is defined**, it inherits the environment of the function that created it

$$\frac{}{\langle \text{ETS, RS, function(ps, ss)} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS, functiontype(ps, ss, RS)} \rangle}$$

On the other hand, a **function call** creates a new scope in which it will evaluate a function body. The created scope is then discarded when evaluation is done and a value or an error may be returned. Remember that outer scopes can be modified too. If evaluating function arguments returns an error, propagate it

$$\frac{\langle \text{ETS}, \text{RS}, \text{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1, \text{functiontype(ns, ss, RS)}\rangle \quad \langle \text{ETS}_1, \text{RS}, \text{es}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_2, \textbf{ERROR}\rangle}{\langle \text{ETS}, \text{RS}, \text{functioncall(e, es)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_2, \textbf{ERROR}\rangle}$$

If function arguments evaluate without a problem, then we can evaluate the function body

$$\frac{\begin{array}{c}\langle \text{ETS}, \text{RS}, \text{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1, \text{functiontype(ns, ss, RS)}\rangle \quad \langle \text{ETS}_1, \text{RS}, \text{es}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_2, \text{vs}\rangle \\ \langle \text{ETS}_2, \text{RS}, \text{make(ns, vs)}\rangle \overset{\text{env}}{\Rightarrow} \langle \text{ET::ETS}_2, \text{RS}_1\rangle \\ \langle \text{ET::ETS}_2, \text{RS}_1, \text{ss}\rangle \overset{\text{evaluate}}{\Rightarrow} \langle \text{ET}_1\text{::ETS}_3, \text{C, V}\rangle \text{ st C} \in \{\text{continue, return, error}\}\end{array}}{\langle \text{ETS}, \text{RS}, \text{functioncall(e, es)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_3, \text{V}\rangle}$$

III.c. 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 control value which signals whether or not to continue execution, and possibly a return value.

Controls are return, break, continue, error.

Before assigning variables, we need to ???

If there's an error while evaluating the left-hand expressions, propagate it

$$\frac{\langle E, R, es_{lhs}\rangle \overset{lhs}{\Rightarrow} \langle E_1, R, \textbf{ERROR}\rangle}{\langle E, R, assign(es_{lhs}, es_{rhs})\rangle \overset{evaluate}{\Rightarrow} \langle E_1, error, \textbf{ERROR}\rangle}$$

Similarly, if there's an error while evaluating the right-hand expressions, do the same

$$\frac{\langle E, R, es_{lhs}\rangle \overset{lhs}{\Rightarrow} \langle E_1, R, vs_{lhs}\rangle \quad \langle E_1, R, es_{rhs}\rangle \overset{rhs}{\Rightarrow} \langle E_2, R, \textbf{ERROR}\rangle}{\langle E, R, assign(es_{lhs}, es_{rhs})\rangle \overset{evaluate}{\Rightarrow} \langle E_2, error, \textbf{ERROR}\rangle}$$

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

$$\frac{\langle E, R, es_{lhs}\rangle \overset{lhs}{\Rightarrow} \langle E_1, R, vs_{lhs}\rangle \quad \langle E_1, R, es_{rhs}\rangle \overset{rhs}{\Rightarrow} \langle E_2, R, vs_{rhs}\rangle \quad \langle E_2, setvalues(vs_{lhs}, vs_{rhs})\rangle \overset{env}{\Rightarrow} \langle E_3\rangle}{\langle E, R, assign(es_{lhs}, es_{rhs})\rangle \overset{evaluate}{\Rightarrow} \langle E_3, continue, []\rangle}$$

Function calls can be statements too in which case all return values are discarded except errors. Just like function call expressions, if the arguments cannot be evaluated and an error is returned, propagate it

$$\frac{\langle ETS, RS, e\rangle \overset{rhs}{\Rightarrow} \langle ETS_1, functiontype(ns, ss, RS)\rangle \quad \langle ETS_1, RS, es\rangle \overset{rhs}{\Rightarrow} \langle ETS_2, \textbf{ERROR}\rangle}{\langle ETS, RS, functioncall(e, es)\rangle \overset{evaluate}{\Rightarrow} \langle ETS_2, error, \textbf{ERROR}\rangle}$$

Evaluating a function body can return an error. This value is not discarded

$$\frac{\langle ETS, RS, e\rangle \overset{rhs}{\Rightarrow} \langle ETS_1, functiontype(ns, ss, RS)\rangle \quad \langle ETS_1, RS, es\rangle \overset{rhs}{\Rightarrow} \langle ETS_2, vs\rangle \quad \langle ETS_2, RS, make(ns, vs)\rangle \overset{env}{\Rightarrow} \langle ET::ETS_2, RS_1\rangle \quad \langle ET::ETS_2, RS_1, ss\rangle \overset{evaluate}{\Rightarrow} \langle ET_1::ETS_3, error, \textbf{ERROR}\rangle}{\langle ETS, RS, functioncall(e, es)\rangle \overset{evaluate}{\Rightarrow} \langle ETS_3, error, \textbf{ERROR}\rangle}$$

If no errors occur, then evaluate the function call and discard the return values

$$\frac{\langle ETS, RS, e\rangle \overset{rhs}{\Rightarrow} \langle ETS_1, functiontype(ns, ss, RS)\rangle \quad \langle ETS_1, RS, es\rangle \overset{rhs}{\Rightarrow} \langle ETS_2, vs\rangle \quad \langle ETS_2, RS, make(ns, vs)\rangle \overset{env}{\Rightarrow} \langle ET::ETS_2, RS_1\rangle \quad \langle ET::ETS_2, RS_1, ss\rangle \overset{evaluate}{\Rightarrow} \langle ET_1::ETS_3, C, V\rangle \text{ st } C \in \{continue, return\}}{\langle ETS, RS, functioncall(e, es)\rangle \overset{rhs}{\Rightarrow} \langle ETS_3, continue, []\rangle}$$

The **do** statement evaluates a list of statements in a new scope. When evaluation is done, the new scope is discarded. We must also remember that statements in the new scope can modify outer scopes

$$\frac{\langle \text{ETS, RS, make} \rangle \overset{\text{env}}{\Rightarrow} \langle \text{ET::ETS, } \text{RS}_1 \rangle \quad \langle \text{ET::ETS, } \text{RS}_1\text{, ss} \rangle \overset{\text{evaluate}}{\Rightarrow} \langle \text{ET}_1\text{::ETS}_1\text{, C, V} \rangle}{\langle \text{ETS, RS, do(ss)} \rangle \overset{\text{evaluate}}{\Rightarrow} \langle \text{ETS}_1\text{, C, V} \rangle}$$

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 evaluation is halted

$$\frac{\langle \text{ETS, RS, e} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1\text{, } \textbf{ERROR} \rangle}{\langle \text{ETS, RS, while(e, ss)} \rangle \overset{\text{evaluate}}{\Rightarrow} \langle \text{ETS}_1\text{, error, } \textbf{ERROR} \rangle}$$

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

$$\frac{\langle \text{ETS, RS, e} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1\text{, C} \rangle \quad \text{C} \in \{\text{niltype(nil), booleantype(false)}\}}{\langle \text{ETS, RS, while(e, ss)} \rangle \overset{\text{evaluate}}{\Rightarrow} \langle \text{ETS}_1\text{, continue, []} \rangle}$$

If the condition expression 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, then we halt the evaluation and return a value

$$\frac{\begin{array}{c}\langle \text{ETS, RS, e} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1\text{, C} \rangle \quad \text{C} \notin \{\text{niltype(nil), booleantype(false)}\} \\ \langle \text{ETS}_1\text{, RS, do(ss)} \rangle \overset{\text{evaluate}}{\Rightarrow} \langle \text{ETS}_2\text{, C, V} \rangle \quad \text{C} \in \{\text{error, break, return}\}\end{array}}{\langle \text{ETS, RS, while(e, ss)} \rangle \overset{\text{evaluate}}{\Rightarrow} \langle \text{ETS}_2\text{, C, V} \rangle}$$

If the loop flow is not broken, we keep evaluating the body until it is, either when the condition expression evaluates to true, or the loop is explicitly broken

$$\frac{\begin{array}{c}\langle \text{ETS, RS, e} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1\text{, C} \rangle \quad \text{C} \notin \{\text{niltype(nil), booleantype(false)}\} \\ \langle \text{ETS}_1\text{, RS, do(ss)} \rangle \overset{\text{evaluate}}{\Rightarrow} \langle \text{ETS}_2\text{, continue, []} \rangle \\ \langle \text{ETS}_2\text{, RS, while(e, ss)} \rangle \overset{\text{evaluate}}{\Rightarrow} \langle \text{ETS}_3\text{, C, V} \rangle\end{array}}{\langle \text{ETS, RS, while(e, ss)} \rangle \overset{\text{evaluate}}{\Rightarrow} \langle \text{ETS}_3\text{, C, V} \rangle}$$

Repeat-until [TODO Explain me]

$$\frac{\langle ETS, RS, make\rangle \overset{env}{\Rightarrow} \langle ET::ETS, RS_1\rangle \quad \langle ET::ETS, RS_1, ss\rangle \overset{evaluate}{\Rightarrow} \langle E_1::ETS_1, C, V\rangle \quad C \in \{return, break, error\}}{\langle ETS, RS, repeat(e, ss)\rangle \overset{evaluate}{\Rightarrow} \langle ETS_1, C, V\rangle}$$

$$\frac{\langle ETS, RS, make\rangle \overset{env}{\Rightarrow} \langle ET::ETS, RS_1\rangle \quad \langle ET::ETS, RS_1, ss\rangle \overset{evaluate}{\Rightarrow} \langle E_1::ETS_1, continue, []\rangle \quad \langle ET_1::ETS_1, RS_1, e\rangle \overset{rhs}{\Rightarrow} \langle E_2::ETS_2, \textbf{ERROR}\rangle}{\langle ETS, RS, repeat(e, ss)\rangle \overset{evaluate}{\Rightarrow} \langle ETS_2, error, \textbf{ERROR}\rangle}$$

$$\frac{\langle ETS, RS, make\rangle \overset{env}{\Rightarrow} \langle ET::ETS, RS_1\rangle \quad \langle ET::ETS, RS_1, ss\rangle \overset{evaluate}{\Rightarrow} \langle E_1::ETS_1, continue, []\rangle \quad \langle ET_1::ETS_1, RS_1, e\rangle \overset{rhs}{\Rightarrow} \langle E_2::ETS_2, V\rangle \quad V \notin \{niltype(nil), booleantype(false)\}}{\langle ETS, RS, repeat(e, ss)\rangle \overset{evaluate}{\Rightarrow} \langle ETS_2, continue, []\rangle}$$

$$\frac{\langle ETS, RS, make\rangle \overset{env}{\Rightarrow} \langle ET::ETS, RS_1\rangle \quad \langle ET::ETS, RS_1, ss\rangle \overset{evaluate}{\Rightarrow} \langle E_1::ETS_1, continue, []\rangle \quad \langle ET_1::ETS_1, RS_1, e\rangle \overset{rhs}{\Rightarrow} \langle E_2::ETS_2, V\rangle \quad V \in \{niltype(nil), booleantype(false)\} \quad \langle ET_2::ETS_2, RS_1, repeat(e, ss)\rangle \overset{evaluate}{\Rightarrow} \langle E_3::ETS_3, Control, Return\rangle}{\langle ETS, RS, repeat(e, ss)\rangle \overset{evaluate}{\Rightarrow} \langle ETS_3, Control, Return\rangle}$$

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

$$\frac{\langle ETS, RS, e\rangle \overset{rhs}{\Rightarrow} \langle ETS_1, \textbf{ERROR}\rangle}{\langle ETS, RS, if(e, s_{true}, s_{false})\rangle \overset{evaluate}{\Rightarrow} \langle ETS_1, error, \textbf{ERROR}\rangle}$$

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

$$\frac{\langle ETS, RS, e\rangle \overset{rhs}{\Rightarrow} \langle ETS_1, C\rangle \quad C \in \{niltype(nil), booleantype(false)\} \quad \langle ETS_1, RS, s_{false}\rangle \overset{evaluate}{\Rightarrow} \langle ETS_2, C, V\rangle}{\langle ETS, RS, if(e, s_{true}, s_{false})\rangle \overset{evaluate}{\Rightarrow} \langle ETS_2, C, V\rangle}$$

Otherwise, the true statement is evaluated

$$\frac{\langle ETS, RS, e\rangle \overset{rhs}{\Rightarrow} \langle ETS_1, C\rangle \quad C \notin \{niltype(nil), booleantype(false)\} \quad \langle ETS_1, RS, s_{true}\rangle \overset{evaluate}{\Rightarrow} \langle ETS_2, C, V\rangle}{\langle ETS, RS, if(e, s_{true}, s_{false})\rangle \overset{evaluate}{\Rightarrow} \langle ETS_2, C, V\rangle}$$

numeric-for
generic-for

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

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

However if the value is valid, then we store it

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

The **return** statement is used to return one or more values from a function

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

$$\frac{}{\langle \texttt{ETS}, \texttt{RS}, \texttt{break} \rangle \overset{\text{evaluate}}{\Rightarrow} \langle \texttt{ETS}, \texttt{break}, [] \rangle}$$