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 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

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

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

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

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

$$\mathtt{parname:}\frac{}{\mathtt{Parname} = \mathtt{Name} \cup \{\mathtt{'...'}\}} \qquad \mathtt{parlist:}\frac{}{\mathtt{[]} \in \mathtt{Parlist}} \qquad \mathtt{parlist:}\frac{\mathtt{p} \in \mathtt{Parname},\quad \mathtt{ps} \in \mathtt{Parlist}}{\mathtt{p::ps} \in \mathtt{Parlist}}$$

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

$$\mathtt{varlist:}\frac{}{\mathtt{[]} \in \mathtt{Varlist}} \qquad \mathtt{varlist:}\frac{\mathtt{v} \in \mathtt{Variable},\quad \mathtt{vs} \in \mathtt{Varlist}}{\mathtt{v::vs} \in \mathtt{Varlist}}$$

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

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

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

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

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

$$\mathtt{referencestack:}\frac{}{\mathtt{[]} \in \mathtt{Referencestack}} \qquad \mathtt{referencestack:}\frac{\mathtt{r} \in \mathtt{Reference},\quad \mathtt{rs} \in \mathtt{Referencestack}}{\mathtt{r::rs} \in \mathtt{Referencestack}}$$

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. Essentially, a table is a list of key-value pairs known as **fields** where each key can be an expression except **nil**

$$\text{fields}: \frac{}{[\,] \in \text{Fields}}$$

$$\text{fields}: \frac{k \in \text{Name} \cup \text{Expression}\backslash\{\text{niltype}(\text{nil})\},\ \ v \in \text{Expression},\ \ fs \in \text{Fields}}{\langle k,\ v\rangle::fs \in \text{Fields}}$$

$$\text{table}: \frac{fs \in \text{Fields}}{\text{table}(fs) \in \text{Value}}$$

In Lua, tables are not passed by value but by **reference**: a positive non-zero integer that indicates a table's address in memory, or in our case, a table's position in the environment. Consequently, a table is **not** considered to be a value that can be assigned to a variable, only the reference to it

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

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

The **table constructor** expression creates a new table in the environment

$$\mathtt{expression:}\frac{\mathtt{fs} \in \mathtt{Fields}}{\mathtt{tableconstructor(fs)} \in \mathtt{Expression}}$$

An expression can be **enclosed** in parentheses

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

Variables expression access a memory address where where values can be stored or read from

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

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

$$\mathtt{expression:}\frac{\mathtt{r,\ k} \in \mathtt{Expression}}{\mathtt{access(r,\ k)} \in \mathtt{Expression}}$$

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

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

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

$$\mathtt{expression:}\frac{\mathtt{n} \in \{\mathtt{type,\ unm,\ not,\ len}\},\quad \mathtt{e} \in \mathtt{Expression}}{\mathtt{unop(n,\ e)} \in \mathtt{Expression}}$$

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

$$\mathtt{expression:}\frac{\begin{array}{c}\mathtt{n} \in \{\mathtt{add,\ sub,\ mul,\ div,\ mod,\ pow,\ eq,\ lt,\ le,\ gt,\ ge,\ and,\ or,\ concat}\}\\ \mathtt{e}_{\mathtt{lhs}},\mathtt{e}_{\mathtt{rhs}} \in \mathtt{Expression}\end{array}}{\mathtt{binop}(\mathtt{n},\mathtt{e}_{\mathtt{lhs}},\mathtt{e}_{\mathtt{rhs}}) \in \mathtt{Expression}}$$

Function definitions store parameter names as well as a block of statements to be executed when the function is called

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

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

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

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** 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.

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

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

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

The **break** statement explicitly breaks a loop

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

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 base clause (or fact) that states that a sequence of statements is a **chunk**, and we can see that the terms used to define the statements resemble the documented abstract syntax. However, this fact on its own doesn't mean much since no relationships between the terms have been defined. That 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, make> $=>_{env}$ <ets::et, r::rs>
<ets, rs, make(fields)> $=>_{env}$ <ets::et, r::rs>
<ets, rs, make(keys, values)> $=>_{env}$ <ets::et, r::rs>
<ets, gettable(reference)> $=>_{env}$ Table
<ets, keyexists(reference, key)> $=>_{env}$ {true, false}
<ets, getvalue(reference, key)> $=>_{env}$ Value
<ets, setvalue(reference, key, value)> $=>_{env}$ ets_1
<ets, setvalues(addresses, values)> $=>_{env}$ ets_1

The <R, K> pair will be referred to as a memory address

III.b. Expression evaluation

Expressions can be left or right-hand side due to the **assignment** statement, and this determines the result of an evaluation. Left-hand side expressions return a memory 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 **variable** and **access** expressions can be left-hand side since they're the only data types that permit us to store values.

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 results

$$\frac{}{\langle \texttt{ETS, RS, []} \rangle \overset{\texttt{lhs}}{\Rightarrow} \langle \texttt{ETS, []} \rangle}$$

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

$$\frac{\langle \texttt{ETS, RS, e} \rangle \overset{\texttt{lhs}}{\Rightarrow} \langle \texttt{ETS}_1\texttt{, } \textbf{ERROR} \rangle}{\langle \texttt{ETS, RS, e::es} \rangle \overset{\texttt{lhs}}{\Rightarrow} \langle \texttt{ETS}_1\texttt{, } \textbf{ERROR} \rangle}$$

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

$$\frac{\langle \texttt{ETS, RS, e} \rangle \overset{\texttt{lhs}}{\Rightarrow} \langle \texttt{ETS}_1\texttt{, } \langle \texttt{R, K} \rangle \rangle \quad \langle \texttt{ETS}_1\texttt{, RS, es} \rangle \overset{\texttt{lhs}}{\Rightarrow} \langle \texttt{ETS}_2\texttt{, } \textbf{ERROR} \rangle}{\langle \texttt{ETS, RS, e::es} \rangle \overset{\texttt{lhs}}{\Rightarrow} \langle \texttt{ETS}_1\texttt{, } \textbf{ERROR} \rangle}$$

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

$$\frac{\langle \texttt{ETS, RS, e} \rangle \overset{\texttt{lhs}}{\Rightarrow} \langle \texttt{ETS}_1\texttt{, } \langle \texttt{R, K} \rangle \rangle \quad \langle \texttt{ETS}_1\texttt{, RS, es} \rangle \overset{\texttt{lhs}}{\Rightarrow} \langle \texttt{ETS}_2\texttt{, VS} \rangle}{\langle \texttt{ETS, RS, e::es} \rangle \overset{\texttt{lhs}}{\Rightarrow} \langle \texttt{ETS}_1\texttt{, } \langle \texttt{R, K} \rangle\texttt{::VS} \rangle}$$

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. The first three evaluation steps are similar to the evaluation of left-hand side expressions and need no explanation

$$\frac{}{\langle \texttt{ETS, RS, []} \rangle \overset{\texttt{rhs}}{\Rightarrow} \langle \texttt{ETS, []} \rangle}$$

$$\frac{\langle \texttt{ETS, RS, e} \rangle \overset{\texttt{rhs}}{\Rightarrow} \langle \texttt{ETS}_1\texttt{, } \textbf{ERROR} \rangle}{\langle \texttt{ETS, RS, e::es} \rangle \overset{\texttt{rhs}}{\Rightarrow} \langle \texttt{ETS}_1\texttt{, } \textbf{ERROR} \rangle}$$

$$\frac{\langle \texttt{ETS, RS, e} \rangle \overset{\texttt{rhs}}{\Rightarrow} \langle \texttt{ETS}_1\texttt{, VS} \rangle \quad \langle \texttt{ETS}_1\texttt{, RS, es} \rangle \overset{\texttt{rhs}}{\Rightarrow} \langle \texttt{ETS}_2\texttt{, } \textbf{ERROR} \rangle}{\langle \texttt{ETS, RS, e::es} \rangle \overset{\texttt{rhs}}{\Rightarrow} \langle \texttt{ETS}_1\texttt{, } \textbf{ERROR} \rangle}$$

If an expression is used as the last (or the only) element of a list of expressions then no adjustment is made

$$\frac{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{VS}\rangle}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{e::[]}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{VS}\rangle}$$

In all other contexts, the result list is adjusted to one element, effectively discarding all values but the first

$$\frac{\forall \mathtt{es} \neq \mathtt{[]} \quad \langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{V}_e\mathtt{::VS}_e\rangle \quad \langle \mathtt{ETS}_1,\ \mathtt{RS},\ \mathtt{es}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \mathtt{ETS}_2,\ \mathtt{VS}_{es}\rangle}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{e::es}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \mathtt{ETS}_2,\ \mathtt{V}_e\mathtt{::VS}_{es}\rangle}$$

Values are a subset of expressions and therefore need to be evaluated. For most values, the evaluation returns the same value

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

The **table constructor** expression creates a new table in the environment and returns the reference to it

$$\frac{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{make(FS)}\rangle \overset{\text{env}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{R::RS}\rangle}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{tableconstructor(FS)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{R::[]}\rangle}$$

Expressions can be **enclosed in parentheses** which usually means that in case the expression evaluates into a non-empty list of values, only the first is returned. It stands to reason that if only one result is returned, no adjustment is made

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

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

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

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

$$\frac{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathbf{ERROR}\rangle}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{enclosed(e)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathbf{ERROR}\rangle}$$

Evaluating a left-hand side variable means retrieving a memory address that corresponds to the variable. 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

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

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

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

A variable name always exists in the global scope, even if it isn't explicitly defined. As such, if a query is made in the global scope, always return its reference and the variable name

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{access(r,\ k)}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \langle \mathtt{R},\ \mathtt{K}\rangle\rangle \quad \langle \mathtt{ETS}_1,\ \mathtt{getvalue(R,\ K)}\rangle \overset{\mathtt{env}}{\Rightarrow} \mathtt{V}}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{access(r,\ k)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{V::[]}\rangle}$$

Variadic expressions evaluate into a list of values if they exist. The evaluation of a variadic expressions consists of finding out whether the field key "**...**" is defined in the current scope. If it is, then its values are returned

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

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

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

Unary operators
Binary operators

When a **function is defined**, a function type is created with given parameters, statements and an empty reference stack attached to it.

$$\frac{}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{function(ps,\ ss)}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ETS},\ \mathtt{functiontype(ps,\ ss,\ [])::[]}\rangle}$$

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

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

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

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

If evaluating function arguments returns an error, propagate it

$$\frac{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{e}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{functiontype(ps,\ ss,\ RS}_f\mathtt{)::VS}\rangle \quad \langle \mathtt{ETS}_1,\ \mathtt{RS},\ \mathtt{es}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ETS}_2,\ \mathbf{ERROR}\rangle}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{functioncall(e,\ es)}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ETS}_2,\ \mathbf{ERROR}\rangle}$$

If the function arguments evaluate without a problem, then we can evaluate the actual function. A function without an environment attached to it implies the use of the global environment

$$\frac{\begin{array}{c}\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{e}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{functiontype(ps,\ ss,\ [])::VS}_e\rangle \quad \langle \mathtt{ETS}_1,\ \mathtt{RS},\ \mathtt{es}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ETS}_2,\ \mathtt{VS}_{es}\rangle \\ \langle \mathtt{ETS}_2,\ \mathtt{RS},\ \mathtt{make(ps,\ VS}_{es}\mathtt{)}\rangle \overset{\mathrm{env}}{\Rightarrow} \langle \mathtt{ETS}_3,\ \mathtt{RS}_1\rangle \\ \langle \mathtt{ETS}_3,\ \mathtt{RS}_1,\ \mathtt{ss}\rangle \overset{\mathrm{stat}}{\Rightarrow} \langle \mathtt{ETS}_4,\ \mathtt{C},\ \mathtt{VS}_{ss}\rangle \text{ st } \mathtt{C} \in \{\mathtt{continue},\ \mathtt{return},\ \mathtt{error}\}\end{array}}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{functioncall(e,\ es)}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ETS}_4,\ \mathtt{VS}_{ss}\rangle}$$

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

$$\frac{\begin{array}{c}\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{e}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{functiontype(ps,\ ss,\ RS}_f\mathtt{)::VS}_e\rangle \quad \langle \mathtt{ETS}_1,\ \mathtt{RS},\ \mathtt{es}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ETS}_2,\ \mathtt{VS}_{es}\rangle \\ \langle \mathtt{ETS}_2,\ \mathtt{RS}_f,\ \mathtt{make(ps,\ VS}_{es}\mathtt{)}\rangle \overset{\mathrm{env}}{\Rightarrow} \langle \mathtt{ETS}_3,\ \mathtt{RS}_1\rangle \\ \langle \mathtt{ETS}_3,\ \mathtt{RS}_1,\ \mathtt{ss}\rangle \overset{\mathrm{stat}}{\Rightarrow} \langle \mathtt{ETS}_4,\ \mathtt{C},\ \mathtt{VS}_{ss}\rangle \text{ st } \mathtt{C} \in \{\mathtt{continue},\ \mathtt{return},\ \mathtt{error}\}\end{array}}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{functioncall(e,\ es)}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ETS}_4,\ \mathtt{VS}_{ss}\rangle}$$

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 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

$$\frac{}{\langle \mathtt{ETS},\ \mathtt{RS},\ []\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ETS},\ \mathtt{continue},\ []\rangle}$$

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

$$\frac{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{s}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{C},\ \mathtt{VS}\rangle \quad \mathtt{C} \in \{\mathtt{return},\ \mathtt{break},\ \mathtt{error}\}}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{s::ss}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{C},\ \mathtt{VS}\rangle}$$

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

$$\frac{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{s}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{continue},\ \mathtt{VS}_s\rangle \quad \langle \mathtt{ETS}_1,\ \mathtt{RS},\ \mathtt{ss}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ETS}_2,\ \mathtt{C},\ \mathtt{VS}_{ss}\rangle}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{s::ss}\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{C},\ \mathtt{VS}_{ss}\rangle}$$

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

$$\frac{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{es}_{lhs}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathbf{ERROR}\rangle}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{assign}(\mathtt{es}_{lhs},\ \mathtt{es}_{rhs})\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{error},\ \mathbf{ERROR})\rangle}$$

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

$$\frac{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{es}_{lhs}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{VS}_{lhs}\rangle \quad \langle \mathtt{ETS}_1,\ \mathtt{RS},\ \mathtt{es}_{rhs}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ETS}_2,\ \mathbf{ERROR}\rangle}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{assign}(\mathtt{es}_{lhs},\ \mathtt{es}_{rhs})\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ETS}_2,\ \mathtt{error},\ \mathbf{ERROR})\rangle}$$

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

$$\frac{\begin{array}{c}\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{es}_{lhs}\rangle \overset{\mathtt{lhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{RS},\ \mathtt{VS}_{lhs}\rangle \quad \langle \mathtt{ETS}_1,\ \mathtt{RS},\ \mathtt{es}_{rhs}\rangle \overset{\mathtt{rhs}}{\Rightarrow} \langle \mathtt{ETS}_2,\ \mathtt{VS}_{rhs}\rangle \\ \langle \mathtt{ETS}_2,\ \mathtt{setvalues}(\mathtt{VS}_{lhs},\ \mathtt{VS}_{rhs})\rangle \overset{\mathtt{env}}{\Rightarrow} \mathtt{ETS}_3\end{array}}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{assign}(\mathtt{es}_{lhs},\ \mathtt{es}_{rhs})\rangle \overset{\mathtt{stat}}{\Rightarrow} \langle \mathtt{ETS}_3,\ \mathtt{continue},\ [])\rangle}$$

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

$$\frac{\langle \texttt{ETS, RS, functioncall(e, es)} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ETS}_1\texttt{, VS} \rangle}{\langle \texttt{ETS, RS, functioncall(e, es)} \rangle \overset{\text{stat}}{\Rightarrow} \langle \texttt{ETS}_1\texttt{, continue, []} \rangle}$$

... except errors.

$$\frac{\langle \texttt{ETS, RS, functioncall(e, es)} \rangle \overset{\text{rhs}}{\Rightarrow} \langle \texttt{ETS}_1\texttt{, } \mathbf{ERROR} \rangle}{\langle \texttt{ETS, RS, functioncall(e, es)} \rangle \overset{\text{stat}}{\Rightarrow} \langle \texttt{ETS}_1\texttt{, error, } \mathbf{ERROR} \rangle}$$

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

$$\frac{\langle \texttt{ETS, RS, make} \rangle \overset{\text{env}}{\Rightarrow} \langle \texttt{ETS}_1\texttt{, RS}_1 \rangle \quad \langle \texttt{ETS}_1\texttt{, RS}_1\texttt{, ss} \rangle \overset{\text{stat}}{\Rightarrow} \langle \texttt{ETS}_2\texttt{, C, VS} \rangle}{\langle \texttt{ETS, RS, do(ss)} \rangle \overset{\text{stat}}{\Rightarrow} \langle \texttt{ETS}_2\texttt{, C, VS} \rangle}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The loop is broken if the condition expression is considered true

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

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

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

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

$$\frac{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{e}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathbf{ERROR}\rangle}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{if}(\mathtt{e},\ \mathtt{s}_{\mathtt{true}},\ \mathtt{s}_{\mathtt{false}})\rangle \overset{\mathrm{stat}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{error},\ \mathbf{ERROR}\rangle}$$

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

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

Otherwise, the true statement is evaluated

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

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

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

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

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

$$\frac{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{es}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathbf{ERROR}\rangle}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{return(es)}\rangle \overset{\mathrm{stat}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{error},\ \mathbf{ERROR}\rangle}$$

$$\frac{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{es}\rangle \overset{\mathrm{rhs}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{VS}\rangle}{\langle \mathtt{ETS},\ \mathtt{RS},\ \mathtt{return(es)}\rangle \overset{\mathrm{stat}}{\Rightarrow} \langle \mathtt{ETS}_1,\ \mathtt{return},\ \mathtt{VS}\rangle}$$

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

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

III.d. No scope destruction?