CS1101S Programming Methodology
Dissecting the Meta-Circular Evaluator

Disclaimer: This summary note has been compiled to aid my own understanding, and I hereby affirm that any mistakes found in this document is solely due to my own human error, and not committed on purpose in order to "snake". Also, this essay starts off very professionally but after a few pages the sanity of the author has started to unravel. As a result, this paragraph is a meta-circular paragraph which talks about this meta-circular essay which talks about a meta-circular evaluator which is a Source Program that talks about a Source Program.

1. The Environment Model

1.1. Abstraction for Frames

A **Frame** in the meta-circular evaluator is defined as a Pair containing a List of Names in its head and a List of Values in its tail. The following are the Constructor and Accessor functions for Frames.

make_frame (Constructor Function)

```
function make_frame(names, values) {
    return pair(names, values);
}
```

Accessor Functions for Frames

```
function frame_names(frame) {
    return head(frame);
}
function frame_values(frame) {
    return tail(frame);
}
```

1.2. Abstraction for Environments

An **Environment** is defined as a List of Frames, with the first Frame in the head and the enclosing Environment in the tail. To add a new frame to an environment, we use the **enclose_by** function. As a result, an empty Environment is just null.

enclose_by (Constructor Function)

```
function enclose_by(frame,env) {
   return pair(frame,env);
}
```

Whenever a (1) function is called, or a (2) block is evaluated, the Environment is extended using **extend_environment** by adding a Frame that contains a List of Names and a List of Values.

extend_environment (Creator Function)

```
function extend_environment(names, vals, base_env) {
```

The function first check if the number of Names is equal to the number of Values. This makes sense so that there is a 1-to-1 correspondence between the Names and the Value in the new Frame so that binding can occur.

```
    if (length(names) === length(vals)) {
```

Now we can create a new Frame. Note that in this Frame, all the values of Values are paired with the Boolean value true as well, probably to indicate that they are mutable data.

```
        return enclose_by(
                make_frame(names,
                   map(x => pair(x, true), vals)),
                base_env);
```

Note that the length of the List of Names is dependent on the number of arguments you have supplied when you call a certain function. Hence, if length of Names is different from the length of Values, it is wholly due to the fact that you supplied the wrong number of arguments.

```
    } else if (length(names) < length(vals)) {
        error("Too many arguments supplied: " +
              stringify(names) + ", " +
              stringify(vals));
    } else {
        error("Too few arguments supplied: " +
              stringify(names) + ", " +
              stringify(vals));
    }
}
```

Accessor Functions for Environments

```
function first_frame(env) {
   return head(env);
}
function enclosing_environment(env) {
   return tail(env);
}
function is_empty_environment(env) {
   return is_null(env);
}
```

1.2.1. Setting Up the Global Environment

The nullary function **setup_environment()** will Return an Environment with special properties and hence known as the Global Environment.

setup_environment()

A **primitive function** is defined as a list with its name as the first element and its function value, or function definition expression, as the second element. Similarly, a **primitive constant** is a list with its name as the first element and its value as the second element.

```
const the_empty_environment = null;
const primitive_functions = list(
      list("+",          (x,y) => x + y  ), …
      );
const primitive_constants = list(
      list("undefined", undefined),
      list("math_PI"  , math_PI)
     );
```

Every environment needs a consolidated List of Names and a List of Values. By making use of Retriever Functions it manages to extract all these Lists from the above constants.

```
function setup_environment() {
    const primitive_function_names =
        map(f => head(f), primitive_functions);
    const primitive_function_values =
        map(f => make_primitive_function(head(tail(f))),
            primitive_functions);
    const primitive_constant_names =
        map(f => head(f), primitive_constants);
    const primitive_constant_values =
        map(f => head(tail(f)),
            primitive_constants);
```

Now, by definition, the Global Environment's "enclosing environment" is the empty set. Hence, it is created by adding one Frame to an originally empty Environment, whre elements a consolidated List of Names is bound one-to-one to corresponding elements in the consolidated List of Elements.

```
    return extend_environment(
             append(primitive_function_names,
                  primitive_constant_names),
             append(primitive_function_values,
                  primitive_constant_values),
             the_empty_environment);
}
const the_global_environment = setup_environment();
```

2. Evaluation of Entire Programs in a String

The heart of the meta-circular evaluator is the **parse_and_eval** function. It takes in an entire Source program as a String as argument. The String would be converted into an Array, or a parse tree, using the **parse** function. Then, the parse tree will be evaluated via the function **eval_toplevel**. The return result is the evaluated result of the Source program in the String.

parse_and_eval

```
function parse_and_eval(str) {
    return eval_toplevel(parse(str));
}
```

The function **eval_toplevel** basically ensures that there are no Return statements in the "program environment" of your input String. It takes in an List (parse tree) as argument. The parse tree is wrapped in a Block and regarded as the "program environment". The Block is then evaluated in the enclosing environment called the "global environment". Since Return statements are not allowed in the program environment of any Source program, if it is found that the program is a Return value, the evaluator will display an error. Otherwise, the value of the evaluated block will be displayed in the REPL.

eval_toplevel

```
function eval_toplevel(stmt) {
  const program_block = make_block(stmt);
  const value = evaluate(program_block,
                         the_global_environment);
  if (is_return_value(value)) {
     error("return not allowed " +
           "outside of function definitions");
  } else {
     return value;
  }
}
```

The function **evaluate** will take in a Parse Tree and return its evaluated value. How it does so will depend on the "nature" of the Parse Tree.

evaluate

If the statement corresponds to a Source Program that is a self-evaluating program (eg. Numbers, Strings and Booleans), **evaluate** will just return the statement itself.

```
function evaluate(stmt, env) {
  return is_self_evaluating(stmt)
       ?  stmt
```

If it is a "name", then the name must have been previously declared in the program itself, **evaluate** will look up the value of the name in the current environment using the function **lookup_name_value**, and return that value.

```
       : is_name(stmt)
        ? lookup_name_value(name_of_name(stmt), env)
```

If the statement is a constant declaration, variable declaration, assignment statement, conditional expression, function definition, sequence, block or return expression, it will be evaluated accordingly depending on its type. Each type of statement has a corresponding function that is able to evaluate the statement.

```
       : is_constant_declaration(stmt)
        ? eval_constant_declaration(stmt, env)
       : is_variable_declaration(stmt)
        ? eval_variable_declaration(stmt, env)
       : is_assignment(stmt)
        ? eval_assignment(stmt, env)
       : is_conditional_expression(stmt)
        ? eval_conditional_expression(stmt, env)
       : is_function_definition(stmt)
        ? eval_function_definition(stmt, env)
       : is_sequence(stmt)
        ? eval_sequence(sequence_statements(stmt), env)
       : is_block(stmt)
        ? eval_block(stmt, env)
       : is_return_statement(stmt)
        ? eval_return_statement(stmt, env)
```

If the statement is a function application (ie. a function call), the function that is currently being called is detected using **operator**, and it is evaluated first. The arguments of the function called is detected using **operands**. Finally, the function is applied to the arguments using the **apply** function, which returns the evaluation of the function application.

```
       : is_application(stmt)
        ? apply(evaluate(operator(stmt), env),
                list_of_values(operands(stmt), env))
```

If the statement does not belong to any type mentioned earlier, an error message will be displayed.

```
       : error(stmt, "Unknown statement type in evaluate: ");
}
```

2.1. Evaluating Constants

As mentioned earlier, constants like Numbers, Strings and Booleans are self-evaluating. The function **is_self_evaluating** performs the sole purpose of detecting whether a statement corresponds to a constant, and if it does, it tells **evaluate** to simply return the value of the constant and be done with it.

is_self_evaluating

```
function is_self_evaluating(stmt) {
    return is_number(stmt) ||
           is_string(stmt) ||
           is_boolean(stmt);
}
```

2.2. Evaluating Tagged Lists

All other statements that are not constants are considered to be tagged lists. The head of the statement, the_tag, indicates the type of statement. The function **is_tagged_list** plays the role of telling the evaluator what the type of statement is to be evaluated.

is_tagged_list

```
function is_tagged_list(stmt, the_tag) {
    return is_pair(stmt) && head(stmt) === the_tag;
}
```

Function is used to tell evaluate "What type of statement am I evaluating today?". For example,

```
function is_constant_declaration(stmt) {
   return is_tagged_list(stmt, "constant_declaration");
}
```

I see a pattern in the way tagged lists are evaluated. It typically involves a "Retriever Function" that makes use of **head** and **tail** to access the name and value of the tagged list within the parse tree. Then, the names and values are inserted into an "Evaluator Function", which would return the computed value of the entire statement.

2.2.1. Evaluating Names

In order the evaluate a Name, the evaluator first asks, "What is the name?", and this question is answered use of the **name_of_name**. Now, after retrieving the name, **evaluate** can look up the value of the name in the current environment using the function **lookup_name_value.**

name_of_name (Retriever Function)

```
function name_of_name(stmt) {
    return head(tail(stmt));
}
```

The following is how **lookup_name_value** works with respect to the Environment Model.

lookup_name_value

```
function lookup_name_value(name, env) {
```

The function **env_loop** is called immediately for the current environment.

```
    function env_loop(env) {
        function scan(names, vals) {
            return is_null(names)
                   ? env_loop(
                       enclosing_environment(env))
                   : name === head(names)
                     ? head(head(vals))
                     : scan(tail(names), tail(vals));
        }
```

If the entire Environment is empty (i.e. no Frames at all, meaning no Names at all! WTF?), then there is no need to even bother looking for the Name. It's not there.

```
        if (is_empty_environment(env)) {
            error(name, "Unbound name: ");
```

Or else, it will scan the first Frame first, looking for the name. Based on the definition of **scan**, if the name is not found in the first Frame, it will go to the Frames in the enclosing Environment in search for the Name.

```
        } else {
            const frame = first_frame(env);
            const value =  scan(frame_names(frame),
                                frame_values(frame));
```

However, if we manage to find the Name, and yet its corresponding value is **no_value_yet**, it means that we are actually calling the Name before it has even been declared (i.e. there has been no constant / variable declaration statement yet which assigns an actual value to the name!).

```
                    if (value === no_value_yet) {
                error(name, "Name used before declaration: ");
            } else {
                    return value;
                }
        }
    }
    return env_loop(env);
}
```

2.2.2. Evaluating Constant and Variable Declarations

Every constant and variable in a Source Program has a (1) name and a (2) value. The following "Retriever Functions" will access these names and values from the array.

Various Retriever Functions

```
function constant_declaration_name(stmt) {
   return head(tail(head(tail(stmt))));
}
function constant_declaration_value(stmt) {
   return head(tail(tail(stmt)));
}
function variable_declaration_name(stmt) {
   return head(tail(head(tail(stmt))));
}
function variable_declaration_value(stmt) {
   return head(tail(tail(stmt)));
}
```

In a normal Source program, constant and variable declarations fulfill the purpose of "binding" a value to a name. Hence, the "Evaluator Function" does the same thing by calling the function **set_name_value** to bind the name to the evaluated value of the constant / variable.

eval_constant_declaration / eval_variable_declaration (Evaluator Functions)

```
function eval_constant_declaration(stmt, env) {
    set_name_value(constant_declaration_name(stmt),
        evaluate(constant_declaration_value(stmt), env),
        env);
}

function eval_variable_declaration(stmt, env) {
    set_name_value(variable_declaration_name(stmt),
        evaluate(variable_declaration_value(stmt), env),
        env);
}
```

The following is how **set_name_value** works with respect to the environment model.

set_name_value

```
function set_name_value(name, val, env) {
```

When you want to bind a Value to a Name, it is already assumed that the name is inside the first Frame of the current Environment, i.e. it is found in the List of Names pertaining to the first Frame. However, if there is no names found in the first Frame, the binding cannot happen!

```
    function scan(names, vals) {
        return is_null(names)
            ? error("internal error: name not found")
```

This is basically a recursive process.

```
            : name === head(names)
             ? set_head(head(vals), val)
             : scan(tail(names), tail(vals));
    }
    const frame = first_frame(env);
    return scan(frame_names(frame),
            frame_values(frame));
}
```

2.2.3. Evaluating Conditional Expressions

Every conditional expression has a (1) predicate, (2) consequent expression and (3) alternative expression. The following Retriever Functions will return (1), (2) and (3) respectively.

Various Retriever Functions

```
function cond_expr_pred(stmt) { return list_ref(stmt, 1); }
function cond_expr_cons(stmt) { return list_ref(stmt, 2); }
function cond_expr_alt(stmt) { return list_ref(stmt, 3); }
```

To evaluate the conditional expression, the predicate is evaluated first, and the function **is_true** checks if the predicate returns true. If it does, the consequent expression is evaluated. Otherwise, the alternative expression is evaluated.

eval_conditional_expression (Evaluator Function)

```
function is_true(x) { return x === true; }

function eval_conditional_expression(stmt, env) {
   return is_true(evaluate(cond_expr_pred(stmt),
                  env))
         ? evaluate(cond_expr_cons(stmt), env)
         : evaluate(cond_expr_alt(stmt), env);
}
```

2.2.4. Evaluating Function Definition Expressions

Please take note that *evaluating* a function definition expression is not the same as *applying* a function definition expression. The former returns a function object, while the latter does something different entirely.

Every function definition expression is considered has (1) parameters and (2) body. To evaluate such an expression, a Creator Function is called to create a function object that corresponds to the function definition expression.

Various Retriever Functions

```
function function_definition_parameters(stmt) {
   return head(tail(stmt));
}
function function_definition_body(stmt) {
   return head(tail(tail(stmt)));
}
```

Functions can be further classified into Primitive and Compound Functions.

2.2.4.1. Evaluating Primitive Function Definitions

A primitive function is basically just a list with two elements, the_tag and its body / implementation (impl).

make_primitive_function (Creator Function)

```
function make_primitive_function(impl) {
    return list("primitive", impl);
}
```

primitive_implementation (Retriever Function)

```
function primitive_implementation(fun) {
   return list_ref(fun, 1);
}
```

2.2.4.2. Evaluating Compound Function Definitions

Unlike primitive functions, compound functions keep track of the values of its parameters, body and environment.

Its Creator Function, **make_compound_function**, hence creates a Parse Tree that contains information of the function's parameters, body as well as the environment in which the function resides in.

make_compound_function (Creator Function)

```
function make_compound_function(parameters, body, env) {
    return list("compound_function",
            parameters, body, env);
}
```

The following Retriever Functions are meant to be used on already-created compound functions, and NOT for function definition expressions. They will return the parameters, body and environment of the compound function respectively.

Various Retriever Functions

```
function function_parameters(f) {
        return list_ref(f, 1);
}
function function_body(f) {
        return list_ref(f, 2);
}
function function_environment(f) {
        return list_ref(f, 3);
}
```

To evaluate a function definition expression, the "Evaluator Function" will call **make_compound_function** such that a function object is created.

eval_function_definition (Evaluator Function)

```
function eval_function_definition(stmt, env) {
   return make_compound_function(
```

To generate the list of names for parameters, the function **name_of_name** is applied to every element in the list of function definition parameters, which is a list of names.

```
      map(name_of_name,
         function_definition_parameters(stmt)),
```

The body of the function definition is extracted from the stmt using a Retriever Function.

```
      function_definition_body(stmt),
      env);
}
```

2.2.5. Evaluating Sequences

A sequence of statements is basically a series of instructions to be carried out in order. To create a sequence the Creator Function simply takes a list of statements (where each element is a Parse Tree of any type) and makes a tagged list that is labelled "sequence".

make_sequence (Creator Function)

```
function make_sequence(stmts) {
   return list("sequence", stmts);
}
```

To evaluate a sequence, the evaluator first needs to know, "Where are my instructions?". The Retriever Function **sequence_statements** gives the evaluator the instructions that it needs, which is found as the 2nd element of the sequence tagged list.

sequence_statements (Retriever Function)

```
function sequence_statements(stmt) {
   return head(tail(stmt));
}
```

Other Retriever Functions will return the first instruction, and the remaining instructions.

first_statement / rest_statements (Retriever Function)

```
function first_statement(stmts) { return head(stmts); }
function rest_statements(stmts) { return tail(stmts); }
```

The following is how a sequence statement is evaluated. Take note that **eval_sequence** takes in the actual sequence statements as its argument and not the tagged list itself.

sequence_statements (Retriever Function)

```
function is_empty_sequence(stmts) { return is_null(stmts); }
function is_last_statement(stmts) { return is_null(tail(stmts)); }

function eval_sequence(stmts, env) {
```

If no instructions are given, the REPL will just show undefined, just like an empty program would.

```
   if (is_empty_sequence(stmts)) {
      return undefined;
```

If only one instruction is given, that statement will be evaluated in the current environment.

```
   } else if (is_last_statement(stmts)) {
      return evaluate(first_statement(stmts),env);
```

If more than one statement is given, the first statement is evaluated first. The evaluator then checks if the first statement is a return value. If it is, then it suffices to just return the value of the first statement and ignore the rest of the instructions.

```
   } else {
      const first_stmt_value =
         evaluate(first_statement(stmts),env);
      if (is_return_value(first_stmt_value)) {
         return first_stmt_value;
```

Otherwise, the evaluator has to continue to carry out the rest of the instructions by calling **eval_sequence** on the rest of the sequence recursively.

```
      } else {
         return eval_sequence(
            rest_statements(stmts),env);
      }
   }
}
```

2.2.6. Applying Functions

Every function application has two key details: (1) the function that is applied, or the **operator**, and (2) the arguments in which the function is applied to, or the **operands**. The following Retriever Functions help to extract (1) and (2) from the Parse Tree pertaining to a function application.

operator (Retriever Function)

```
function operator(stmt) {
   return head(tail(stmt));
}
```

Various Retriever Functions Pertaining to Operands

```
function operands(stmt) {
   return head(tail(tail(stmt)));
}
function first_operand(ops) {
   return head(ops);
}
function rest_operands(ops) {
   return tail(ops);
}
```

Now that we are able to dissect a function application Parse Tree into its respective components (Operators and Operands), we are able to **apply** the operator to the operands.

apply

```
function apply(fun, args) {
```

We first check if the Parse Tree fun is a tagged list with a tag "primitive function". If it is, then it will make use of a specialized function **apply_primitive_function** for its application.

```
   if (is_primitive_function(fun)) {
      return apply_primitive_function(fun, args);
```

Or else, if it is a "compound function" tagged list, we extract out its body first using the Retriever Function **function_body**.

```
   } else if (is_compound_function(fun)) {
      const body = function_body(fun);
```

The first question to ask is, "What are my names to be added into my new frame when the function is called?".

The function called **local_names** will extract all the names (ie. constants, variables and functions) declared *within the body of the function*.

```
      const locals = local_names(body);
```

Then, the function **insert_all** will create a combined List of names associated with the function, which comprises of both the function's parameters and the names declared locally in the function. These **names** will have to be bound to values in the frame created when the function is being called.

Take note that the elements corresponding to the function's parameters come first in the list, followed by the locally declared names.

```
const names = insert_all(function_parameters(fun),
                         locals);
```

Next, we need to ascertain "What are the values which I need to bind to each name in my new frame?"

All of the names declared locally in the function are temporarily given a special value called **no_value_yet.** Take note that **no_value_yet** is defined as () => null.

```
const temp_values = map(x => no_value_yet,
                        locals);
```

Now, we can obtain a list of **values**. The values of the functions's parameters are just the input arguments args, which are placed in the front of the list of values. The series of values **no_value_yet**, belonging to the locally declared names, are placed in the back.

```
const values = append(args, temp_values);
```

Now if you compare the list **names** and the list **values**, you obtain a one-to-one correspondence between each name and each value.

We can finally evaluate the body of the function. It has to first extend the environment by creating a new frame containing the bindings of each name in **names** to its corresponding value in **values**, and the enclosing environment is function_environment(fun), the environment that our function resides in. Evaluating the body will cause the constant and variable declarations inside the body to be evaluated, and hence the corresponding values for locally declared names will no longer be **no_value_yet**.

```
        const result =
           evaluate(body,
                    extend_environment(
                        names,
                        values,
                        function_environment(fun)));

        if (is_return_value(result)) {
           return return_value_content(result);
        } else {
            return undefined;
        }
     } else {
        error(fun, "Unknown function type in apply");
     }
}
```

The **apply** function requires a List of arguments as its second argument in order to function. The function **list_of_values** takes in a List of arguments (that are in its Array / Parse Tree form) and converts it into a List of actual values, so that they would be "available" for the operator to be applied on them.

list_of_values

```
function no_operands(ops) {
   return is_null(ops);
}

function list_of_values(exps, env) {
    if (no_operands(exps)) {
        return null;
    } else {
        return pair(evaluate(first_operand(exps), env),
                    list_of_values(rest_operands(exps), env));
    }
}
```

Applying a primitive function is much simpler, and it just makes use of **apply_in_underlying_javascript**, a pre-declared function.

apply_primitive_function

```
function apply_primitive_function(fun, argument_list) {
    return apply_in_underlying_javascript(
                primitive_implementation(fun),
                argument_list);
}
```

This is how **local_names** works. In order for a name to be included into the list of local names returned by that function, it has to be declared outside of any other block or function.

local_names

```
function local_names(stmt) {
```

The body of a function is usually a sequence of statements. Hence, the sequence of instructions is extracted out using the Retriever Function **sequence_statements**.

```
    if (is_sequence(stmt)) {
       const stmts = sequence_statements(stmt);
```

The base case is when the sequence is empty. Then we will obtain an empty list of names.

```
       return is_empty_sequence(stmts)
          ? null
```

Otherwise, we make use of recursion to extract the local names of the first statement, as well as the remaining statements, and then merge them together using the function **insert_all**.

```
          : insert_all(
                local_names(first_statement(stmts)),
                local_names(make_sequence(
                                     rest_statements(stmts))));
```

Another case is if there is only one statement in the sequence. If that is the case, it is ually just a constant or variable declaration. Retrieve the name using the respective Retriever Function, and we are done.

```
    } else {
       return is_constant_declaration(stmt)
          ? list(constant_declaration_name(stmt))
          : is_variable_declaration(stmt)
           ? list(variable_declaration_name(stmt))
           : null;
    }
}
```

Now to explain the function **insert_all**. It merges two lists of names together, but ensures that there are no multiple declarations of a constant or variable within the same frame.

insert_all

```
function insert_all(xs, ys) {
    return is_null(xs)
        ? ys
        : is_null(member(head(xs), ys))
          ? pair(head(xs), insert_all(tail(xs), ys))
          : error(head(xs), "multiple declarations of: ");
}
```

2.2.7. Evaluating Return Statements

A Return statement is another tagged list with its Return expression as the second element. Hence, the Retriever Function for obtaining the Return expression is:

return_statement_expression (Retriever Function)

```
function return_statement_expression(stmt) {
   return head(tail(stmt));
}
```

To evaluate a Return statement, a "return value" Parse Tree is created. This is done by evaluating the Return expression in the current environment.

eval_return_statement (Evaluator Function)

```
function eval_return_statement(stmt, env) {
    return make_return_value(
               evaluate(return_statement_expression(stmt),
                        env));
}
```

As mentioned earlier, a "return value" is also a Parse Tree, with its content as its second element.

make_return_value (Creator Function)

```
function make_return_value(content) {
    return list("return_value", content);
}
```

The content of the return value is access using a Retriever Function **return_value_content**.

return_value_content (Retriever Function)

```
function return_value_content(value) {
    return head(tail(value));
}
```

2.2.8. Evaluating Assignment Statements

Similar to constant and variable declaration statements, assignment statements have an associated Name and Value. Hence, the Retriever Functions to access the Name and Value are as follows:

Various Retriever Functions for Assignment Statements

```
function assignment_name(stmt) {
   return head(tail(head(tail(stmt))));
}
function assignment_value(stmt) {
   return head(tail(tail(stmt)));
}
```

The evaluation of an assignment statement using **eval_assignment** is the interesting bit. It basically evaluates the right-hand expression of the statement, which is the Value. It then assigns that Value to the left-hand name using a function **assign_name_value**.

eval_assignment (Evaluator Function)

```
function evil_assignment(stmt, env) {
    const value = evaluate(assignment_value(stmt), env);
    assign_name_value(assignment_name(stmt), value, env);
    return value;
}
```

The mechanism behind the assignment of a Value to the Name is rather important to understand as it has to do with the environment model.

assign_name_value

```
function assign_name_value(name, val, env) {
       function env_loop(env) {
```

What does **scan** do? Given a Name (hidden inside the environment) and a Value, the function **assign_name_value** wishes to assign the Value to the Name. Like Rihanna, it starts to sing "*Oh na-na, where's my Name*" and then traverses through the list of Names for the first frame in order to find the Name.

```
           function scan(names, vals) {
```

If it finds that the list of Names in the first frame is empty, it will go into the enclosing environment and continues singing "*Oh na-na, where's my Name*" as it traverses through the lists of Names for the enclosing frames.

```
               return is_null(names)
                   ? env_loop(
                       enclosing_environment(env))
```

Now, suppose that we have found our Name as the first element in the list of Names in the first frame.

```
                   : name === head(names)
```

Oh, what the hell? Now you need to check the corresponding entry in the list of Values. Apparently what a Value is defined as a *pair(value, Boolean)*, and you can only assign a value to a Name if the Boolean value is true. My guess is that only Variables will have true contained in their Value. I don't know anymore. I just need to feed my family man.

```
                   ? (tail(head(vals))
                       ? set_head(head(vals), val)
                       : error("no assignment " +
                                    "to constants allowed") )
```

Call **scan** recursively if your Name is not the first element in the list of Names.

```
                   : scan(tail(names), tail(vals));
           }
```

To assign a Value to the Name, the first thing this function does is to check if the current environment is empty. If it is, then the name is unbound, and an error message will appear.

```
           if (is_empty_environment(env)) {
               error(name, "Unbound name in assignment: ");
```

Otherwise, if the environment is not empty, then the function will use **scan** to find an instance of the Name in the environment.

```
           } else {
               const frame = first_frame(env);
               return scan(frame_names(frame),
                           frame_values(frame));
           }
       }
       return env_loop(env);
}
```

2.2.9. Evaluating Blocks

I am running out of saliva and I don't want to repeat myself anymore. The following is the constructor function for a block. It is a list with the first element that literally says "block".

make_block (Creator Function)

```
function make_block(stmt) {
   return list("block", stmt);
}
```

The body of the function is accessed through a Retriever Function **block_body** which retrieves the body of the function.

make_block (Creator Function)

```
function block_body(stmt) {
    return head(tail(stmt));
}
```

This is how you evaluate a block. Just use the function **eval_block.** Just beware, don't use the function **evil_block** because it is evil and will cause the entire Source Academy to crash.

eval_block (Evaluator Function)

```
function eval_block(stmt, env) {
```

First access the body of the block.

```
      const body = block_body(stmt);
```

Now, extract all the local names of the body, and assign the value **no_value_yet** to each of the local names. Man, maybe when I grow up and have a kid I will just name my child **no_value_yet**.

```
    const locals = local_names(body);
    const temp_values = map(x => no_value_yet,
                            locals);
```

Now, you can extend the environment and bind all the local names to **no_value_yet**. Then, evaluate the body which may or may not contain instructions on how to assign value to your kids. I mean, names.

```
    return evaluate(body,
              extend_environment(locals, temp_values, env));
}
```

3. Easter Eggs

One Source Academy user reported that when he was toying with the program's Missions, he tried to call the nullary function **find_sentient_life()** in the REPL, and what appeared was a very disturbing String return result:

```
find_sentient_life();

→ "Beware, humans. Martin Henz is actually a robot from Skynet aiming to take over
the world. RUN"
```