Studio 09-D

Metacircular Evaluator

Jia Xiaodong

5 Nov 2019

What is it?

- *Meta*
  - About itself.
- *Circular*
  - The language is implemented in itself (self-hosting).
- *Evaluator*

So what is a MCE anyway? Quite simply, it's a program written in Source (or any language for that matter) that executes a program written in Source just like Source would. In other words, it is a Source interpreter written in Source! As you might know Source is written in JavaScript, so the interpreter in the browser is not a MCE. However once we have such a thing, we could write a Source program to mimic this interpreter.

Programming

- Leaving the abstract: how does a computer work?
- They operate on a pre-defined language — machine code.
  - Normal desktops and laptops: `x86`
  - Mobile devices: `ARM`
  - eg.: 893C2500000000
  - Can be read with tools eg. `xxd`, `hexdump`.
- Not very friendly for humans.
  - Assembly: macros
  - eg.: `mov $0x1,%edi`
  - Can be read with tools eg. `gdb`.
- High level programming languages:
  - `return 1;`
  - Can be read with tools eg. your eyes

In most fields progress is made by building up on abstractions. For instance in the creation of computers. The earliest computers were unable to be programmed. The stored program computer was a revolution that first appeared in theoretical computer science as "universal machines".

The history of programming is a lot older than what is on this slide. For example, punch cards and tape. However the stepping stone towards our modern idea of programming language were the gradual abstraction of "macros" for CPU instructions. From binary to assembly then to low level languages like C, followed by huge strides of abstraction in operating systems, virtual machines, etc., which allowed for more high level languages.

Parsing

- Parsing is the way to comprehend a language.
- The programming language is defined with strict rules.
- The language can be decomposed into elements with these rules.
- Can be seen with `parse(str)` in Source.

In the execution of a program there are two main steps. The first step is parsing. A language first is built up of very strict rules. For most programming languages this is specified as a kind of *context-free grammar*. This is specified on the language spec sheet. This serves as a solid foundation for the execution of our language because it should leave no room for doubt as to what the computer understands by a certain program. Languages that are too strict are too hard to program in, and languages that are too loose (human languages, for example) have too much freedom of interpretation.

We will not go into how parsing is done.

Evaluation

- Irreducible things: done.
- Reducible things: evaluate statement by statement.
- See lecture slides for step-by-step walkthrough.

For evaluation we just follow the same evaluation rules we have been using all this while. There are a few pre-defined things that are irreducible. For anything that is reducible, we try to reduce it along the reduction rules. For example functions are called, arithmetic is calculated, etc.

S12 Q1

Implement function definition hoisting in the MCE.

```
function reorder_statements(stmts) {
    function split_statements(stmts) {
        if (is_null(stmts)) {
            return pair(null, null);
        } else {
            const first_statement = head(stmts);
            const split_rest = split_statements(tail(stmts));
            return is_function_declaration(first_statement)
                   ? pair(pair(first_statement, head(split_rest)),
                          tail(split_rest))
                   : pair(head(split_rest),
                          pair(first_statement, tail(split_rest)));
        }
    }
    const split = split_statements(stmts);
    return append(head(split), tail(split));
}
```

We want to move all function declarations in a block to the top. We can do this anywhere before `evaluate` but certainly the most convenient place to do it is right at `evaluate`. The line in question is the line `eval_sequence(sequence_statements(component), env)`.

Here is a simple way of doing this. Keep a pair of lists. Go through the list of statements, and every time we meet a function declaration, put it into the head list. Otherwise, put it into the tail list. Then at the end, merge these two lists, and we are done.

S12 Q2

Make the MCE detect undeclared names.

```
function evaluate(component, env) {
    return is_literal(component)
        ? literal_value(component)
        : is_name(component)
        ? lookup_symbol_value(
        symbol_of_name(component),
        env)
        : is_application(component)
        ? apply(
        evaluate(
            function_expression(component),
            env),
        list_of_values(
            arg_expressions(component),
            env))
        : is_operator_combination(component)
        ? evaluate(
            operator_combination_to_application(component),
            env)
```

```
function check_names(component, env) {
    is_literal(component)
        ? "ok"
        : is_name(component)
        ? lookup_symbol_value(symbol_of_name(component), env)
        : is_application(component)
        ? check_names(
            make_sequence(
                pair(function_expression(component),
                arg_expressions(component))),
            env)
        : is_operator_combination(component)
        ? check_names(
            operator_combination_to_application(component),
            env)
```

This is actually quite tedious. This is because there can be many ways a name might appear. For example we not only have to check for simple things like a+2, but also undeclared function name, undeclared function parameters, inside return expressions, etc.

Fortunately, the evaluate function already separates out all the possible cases. All we have to do, is to extend the function to check the names.