The Metacircular Evaluator

João Victor Lopez Pereira

26 de dezembro de 2024

Rio de Janeiro - RJ

Sumário

Resumo

This document provides a detailed explanation of an evaluator designed for a `Scheme`-like language. The implementation, as presented in the `metacircular-evaluator.scm` file, closely follows the evaluator described in the book *Structure and Interpretation of Computer Programs*[1], incorporating several enhancements and simplifications to improve clarity and functionality.

The section 1.4 is heavily inspired by the lecture "7A Metacircular Evaluator[4]" taught by Gerald Jay Sussman in MIT.

Capítulo 1

The Evaluator

1.1 Motivation

> [...] It's in words that the magic is — Abracadabra, Open Sesame, and the rest — but the magic words in one story aren't magical in the next. The real magic is to understand which words work, and when, and for what; the trick is to learn the trick. [...] And those words are made from the letters of our alphabet: a couple-dozen squiggles we can draw with the pen. This is the key! And the treasure, too, if we can only get our hands on it! It's as if — as if the key to the treasure is the treasure! [2]
>
> John Barth,
> *Chimera*

A programmer may want to use different programming languages according to his desired-program needs. In programming a scheduler — for example — one may want to use a considerably amount of pointers and access to low-level instructions. A language suited for that could be `C`. In programming a machine learning algorithm, for instance, the same programmer may want to use a language that allows a huge load of abstraction, since it is not of his desire to be worried about memory or small details. By the same argument, the definition of low-level is arguable. According to Alan Jay Perlis, "A programming language is low level when its programs require attention to the irrelevant.[3]".

The point is that a language that is designed for a certain purpose is generally more appropriate for solving a certain issue. `Singular`, for example, may be a spectacular language for Algebraic Computing, but may not be appropriate for simply plotting a sin function. `Python`, on the other hand, can be used to plot this sin function in no more than 3 lines of code, but is not the best for Algebraic Computing.

In conclusion, domain-specific languages prove to be particularly effective in scenarios where

specialized tasks cannot be efficiently addressed by general-purpose languages.

1.2   The Evaluator as a Program

> It is no exaggeration to regard this as the most fundamental idea in programming: The evaluator, which determines the meaning of expressions in a programming language, is just another program. To appreciate this point is to change our images of ourselves as programmers. We come to see ourselves as designers of languages, rather than only users of languages designed by others. In fact, we can regard almost any program as the evaluator for some language. [1]
>
> Gerald Jay Sussman & Harold Abelson,
> *Structure and Interpretation of Computer Programs*

By implementing an interpreter for solving a specific problem, the developer is allowed to make decisions about the semantics and syntax of *his* programming language.

Note that most — maybe all — of the programs we write can be easily seen as a tiny programming language. For example, imagine the following `factorial` procedure:

In `Scheme`:

```scheme
(define (factorial n)
  (if (<= n 2)
      n
      (* n
         (factorial (- n
                      1)))))))
```

In `Python`:

```python
def factorial(n):
    if n <= 2:
        return n
    else:
        return n *
            factorial(n-1)
```

In `C`:

```c
int factorial(int n) {
  if (n <= 2) {
    return n;
  }
  else {
    return n *
        factorial(n-1);
  }
}
```

the `factorial` program defined above is in fact a programming language with its own semantics and syntax. All the three versions of `factorial` have the same semantics, they compute the factorial of a given number following the same mathematical definition. On the other hand, the three of them have different syntax. Note that the `Scheme` implementation require a lot of parentheses and the `C` implementation require some parentheses, curly brackets and semicolons. while the `Python` implementation relies mostly of its syntax on the indentation[1].

The `factorial` procedure is a simple program considering that it has a very strict input and

---

[1]Note that in `C` and `Scheme` we used indentation only for better visualization and understanding of the definition. However, these are not required.

a very strict output, exactly like a mathematical function $f : \mathbb{N} \to \mathbb{N}$ such that:

$$\begin{cases} f(n) = n & \text{if } n \leq 2 \\ f(n) = n \times f(n-1) & \text{if } n > 2 \end{cases}$$

The interpreter, on the other hand, will have far more expressiveness than the `factorial` procedure. In fact, the `factorial` procedure will be able to be passed as an argument to the interpreter, in what is known as a universal machine. This flexibility highlights the core concept of computation: the ability to evaluate any function, define new functions, and even create new languages and paradigms for computation. The interpreter serves as the bridge between the abstract, high-level description of a program and its execution on a physical machine.

The role of the interpreter becomes clear when we consider that it can evaluate any expression or program written in a language. In a sense, an interpreter for a language is itself a program that reads another program, interprets its meaning, and produces an output. Thus, the interpreter is both a tool for executing programs and a means of creating new languages, shaping the computational universe.

This perspective encourages us to view programming not just as writing instructions for a computer but as the creation of languages through which we describe computations. By designing and implementing interpreters, we become language designers, defining the syntax and semantics that govern the execution of our programs. The ability to define new languages is powerful, enabling the creation of more expressive, efficient, or domain-specific solutions.

All these programming languages have different ways of parsing through the user's definitions and transcribing them to machine code language (if the language is compiled) or evaluating them (if the language is interpreted). Note that `Scheme`, despite having the most unusual syntax of the three languages shown above for the average programmer, actually makes parsing significantly easier for several reasons:

1. Homoiconicity: the fact that `Scheme` code is in fact a huge list makes it easier to manipulate its values.

   For an average program — such as the `factorial` function or a scheduler — the manipulated data typically consists of numbers or lists. However, in programs like interpreters or compilers, the data being manipulated is code itself. When this code is represented as a string-like structure, it often carries numerous restrictions and a specialized syntax tailored for specific tasks, making parsing both confusing and unnecessarily complex. Conversely, if the code is structured in a list-like format (or another straightforward data structure), the manipulation becomes significantly simpler. For instance, to retrieve the procedure `factorial` in a string-based representation, one must iterate character by character from `f` to `l`, checking for the string's end. In contrast, with a list-like representation, accessing the procedure involves merely retrieving the first element of the list. Nonetheless, the fact that `Scheme` relies solely on lists as its data structure makes their manipulation exceptionally convenient.

2. S-expressions: In `Scheme` all function calls has the function as the first argument of the expression.

3

This makes it very easily to apply a function to its arguments. In most cases, all the `Scheme` evaluator does is apply the first element of the list (the function) to the remaining evaluated elements of the list (the arguments).

Try to imagine how does these languages compilers or interpreters parse the following code:

In `Scheme`:

```
(define x 1)
(- (+ 2
      (* x
         (/ 10
            2)))
   3)
```

In `Python`:

```
x = 1
2 + x * 10 / 2 - 3
```

In `C`:

```
#include <stdio.h>
int main() {
    int x = 1;
    2 + x * 10 / 2 - 3;
    return 0;
}
```

Most programmers struggle to form an intuitive understanding or make good guesses about how `C` or `Python` code is parsed. How is the order of operations determined? Which operations take precedence? When is `x` substituted with its value, 1? How exactly does the parser work?

In contrast, the `Scheme` version of the code is remarkably simple and intuitive.

1. the special form `define` is applied on the arguments `x` and `1`. What `define` does is assign the value of the third element of the list (`1`) to the second element of the list (`x`).

2. The evaluation of `(- (+ 2 (* x (/ 10 2))) 3)` involves applying the procedure `-` to the results of evaluating `(+ 2 (* x (/ 10 2)))` and `3`. The evaluation of `(+ 2 (* x (/ 10 2)))` is done by applying the procedure `+` to the results of evaluating `2` and `(* x (/ 10 2))`. The evaluation of `(* x (/ 10 2))` applies the procedure `*` to the evaluation of `x` and `(/ 10 2)`. The evaluation of `x` yields the number `1`, while the evaluation of `(/ 10 2)` applies the function `/` to the evaluation of `10` and `2`. As expected, the evaluations of `3`, `2`, `10`, and `2` are simply the values themselves.

So, we can easily imagine that the interpreter of `Scheme` does something similar to:

```
(eval (define x 1))
...
(eval (- (+ 2 (* x (/ 10 2))) 3))
(apply (eval -) (eval (+ 2 (* x (/ 10 2)))) (eval 3))
(apply - (apply (eval +) (eval 2) (eval (* x (/ 10 2)))) 3)
(apply - (apply + 2 (apply (eval *) (eval x) (eval (/ 10 2)))) 3)
(apply - (apply + 2 (apply * 1 (apply (eval /) (eval 10) (eval 2)))) 3)
(apply - (apply + 2 (apply * 1 (apply / 10 2))) 3)
(apply - (apply + 2 (apply * 1 5)) 3)
(apply - (apply + 2 5) 3)
(apply - 7 3)
4
```

The questions we had before about how is the code evaluated are simply put to rest if we understand the `eval` and `apply` cycle.

1.3 Implementation

> We're going to understand what we mean by a program a little bit more profoundly than we have up till now. We've been thinking of programs as describing machines. [...] There's something very remarkable that can happen in the computational world which is that you can have something called a universal machine. [...] We'll see that among other things, it's extremely simple. Now, we are getting very close to the real spirit in the computer at this point. [...] There's a certain amount of mysticism that will appear here. [...] I wish to write for you the evaluator for Lisp. The evaluator isn't very complicated, it's very much like all the programs we've seen already: that's the amazing part of it. [4]
>
> Gerald Jay Sussman,
> *Lecture 7A: Metacircular Evaluator. Timestamp: 0:18*

The interpreter for `Scheme` is commonly referred to as the *Metacircular* Evaluator. It is called Meta because it is written in the language itself, meaning `Scheme` code is interpreting `Scheme`. It is called Circular because it — for the most part — consists of a recursive loop between the functions `Eval` and `Apply`.

The main function of the evaluator is `Eval` itself, which definition is:

```
(define (EVAL exp env)
  (cond
    ((self-value? exp)  exp)
    ((variable? exp)    (lookup-variable-value exp env))
    ((quoted? exp)      (text-of-quotation exp))
    ((assignment? exp)  (eval-assignment exp env))
    ((definition? exp)  (eval-definition exp env))
    ((if? exp)          (eval-if exp env))
    ((lambda? exp)      (make-procedure (lambda-ps exp) (lambda-body exp) env))
    ((begin? exp)       (eval-sequence (begin-actions exp) env))
    ((and? exp)         (eval-and exp env))
    ((or? exp)          (eval-or exp env))
    ((cond? exp)        (EVAL (cond->if exp) env))
    ((let? exp)         (EVAL (let->combination exp) env))
    ((let*? exp)        (EVAL (let*->nested-lets exp) env))
    ((application? exp) (APPLY (EVAL (operator exp) env)
                              (map (lambda (exp) (EVAL exp env)) (operands
                                  exp))))))
```

Something noticeable is that `Eval` is simply a case-analysis which determines what should be done to the given expression.

The fact that all expressions start with the function itself makes it extremely easy to implement the checking procedures since they are simply verifying if the first element of the expression is equal to something.

```
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      false))

(define self-value?            number?)
(define variable?              symbol?)
(define (quoted? exp)          (tagged-list? exp 'quote))
(define (assignment? exp)      (tagged-list? exp 'set!))
(define (definition? exp)      (tagged-list? exp 'define))
(define (if? exp)              (tagged-list? exp 'if))
(define (lambda? exp)          (tagged-list? exp 'lambda))
(define (begin? exp)           (tagged-list? exp 'begin))
(define (and? exp)             (tagged-list? exp 'and))
(define (or? exp)              (tagged-list? exp 'or))
(define (cond? exp)            (tagged-list? exp 'cond))
(define (let? exp)             (tagged-list? exp 'let))
(define (let*? exp)            (tagged-list? exp 'let*))
(define application?           pair?)
(define (compound-procedure? p) (tagged-list? p 'procedure))
(define (primitive-procedure? p) (tagged-list? p 'primitive))
```

Now, the procedures on the right side are obviously a bit harsher to be implemented. You can notice that we have 3 kind of procedures on this `eval` definition:

1. Special Forms: These constructs cannot be implemented as simple functions because their evaluation behavior differs from that of regular functions. Take, for example, the `if` special form. When evaluating (`if x y z`), it first checks whether `x` is true. If it is, `y` is evaluated and returned; otherwise, `z` is evaluated and returned. The distinction lies in how the expressions `y` and `z` are handled: if `if` were implemented as a regular function, both `y` and `z` would be evaluated before the function is called, even though only one of them is actually needed. This difference is particularly important when `y` or `z` involve side effects, as unnecessary evaluations could lead to unintended behavior.

2. Derived Expressions: These constructs can easily be implemented as a list manipulation procedure that converts an expression to a special form. Take, for example, the `cond` expression. `Cond` can easily be implemented as a syntactic-sugar to nested `if` expressions. The same happens to `let`, which can be implemented as a `lambda` expression and `let*`, which can be implemented as nested `let` expressions.

3. Applications: These constructs are the remaining building blocks in the evaluation process. In an application, a function is applied to a sequence of arguments. The function is first evaluated to determine its value (a procedure), and then its arguments are evaluated in left-to-right order, unless explicitly specified otherwise (e.g., in the presence of lazy evaluation). The procedure is then invoked with the results of the arguments' evaluations. This process forms the backbone of computation in functional programming, as nearly every computation reduces to applying functions to arguments.

Apply is the procedure responsible for searching the procedure in the environment and applying it to the evaluated remaining arguments.

```
(define (APPLY procedure arguments)
  (cond
    ((primitive-procedure? procedure)
     (apply-primitive-procedure procedure arguments))
    ((compound-procedure? procedure)
     (eval-sequence (procedure-body procedure)
                    (extend-environment (procedure-parameters procedure)
                                        arguments
                                        (procedure-environment procedure)))))))
```

These are the most important definitions that are needed for understanding the evaluator.

If you examine the implementation of the metacircular evaluator in detail, you will notice that many definitions are simply aliases for basic list operations like car and cdr. This design choice creates a clear separation between syntax and semantics, making it easier to modify the language's behavior without changing its underlying semantics. For instance, consider the evaluator's final check:

```
((application? exp) (APPLY (EVAL (operator exp) env)
                          (list-of-values (operands exp) env)))
```

Here, the operator and operands procedures are defined as follows:

```
(define operator car)
```

```
(define operands cdr)
```

If we wanted to modify Scheme for educational purposes — such as placing the operator as the second element of the list — we could simply redefine these procedures:

```
(define operator cadr)
```

```
(define (operands x) (cons (car x) (cddr x)))
```

This change would allow expressions like (4 + 2) to evaluate correctly while preserving the original semantics. The ability to adapt the syntax so easily highlights the power and flexibility of this approach.

1.4 Lisp as a Fixed Point

There's an awful lot of strange nonsense here. After all, he purported to explain to me Lisp, and he wrote me a Lisp program on the blackboard. The Lisp program was intended to be interpreted for Lisp, but you need a Lisp interpreter in order to understand that program. How could that program have told me anything there is to be known about Lisp? [4]

Gerald Jay Sussman,
Lecture 7A: Metacicular Evaluator. Timestamp: 56:19

Given the set of equations

$$\begin{cases} x = 3 - y \\ y = -1 + x, \end{cases}$$

notice that x is defined in terms of y and y is defined in terms of x. This system has not only a solution but a unique one in x and y.

Given the set of equations

$$\begin{cases} 2x = 6 - 2y \\ y = 3 - x, \end{cases}$$

notice that x is once again defined in terms of y and y is defined in terms of x. Strangely enough, this system has no solution in x and y.

Given the set of equations

$$\begin{cases} x = 1 + y \\ x = 2 + y, \end{cases}$$

notice that the pattern of recursive definitions once again appears. However, this system has no solutions in x and y.

Given these three sets of equations, note that the number of solutions is not a consequence of their format, but of their content. The equation that we are interested in is the one that has a unique solution.

$$\begin{cases} x = 3 - y \\ y = -1 + x. \end{cases}$$

A way of seeing this set of equations is as a linear transformation T such that

$$\begin{bmatrix} x \\ y \end{bmatrix} = T \begin{bmatrix} x \\ y \end{bmatrix}.$$

Note that the solution of this equation is a fixed point of the transformation T^2.

Take a closer look at the `factorial` procedure we implemented in section 1.2. See that `factorial` is a kind of recursive equation that is somehow similar to the set of equations in x and y.

To find the fixed point of `factorial`, we need to first rewrite it in such a way that the transformation T becomes apparent.

```
(define (f g)
  (lambda (n)
    (if (<= n 2)
        n
        (* n (g (- n 1))))))
```

See that `f` is a procedure that if I had the solution `g`, than the result would be the `factorial` procedure. Suppose `g` is the `factorial` procedure, then `f` would return the exact `factorial` procedure defined in section 1.2.

It's strange that the `g` procedure is needed for `g` to be the result of this function, by the same way that we somehow need the value of x to compute x in $x = 4 - x$. A shown in linear algebra, a way to approximate the solution x of this equation in simply by having a start guess x_0 and apply the transformation T to x multiple times.

$$x_1 = Tx_0$$
$$x_2 = Tx_1$$
$$x_3 = Tx_2$$
$$\vdots$$
$$x_{k+1} = Tx_k$$

x_3 is an approximation of x. We can make a similar approximation to the factorial procedure:

```
(define factorial-0 ; computes 1! with no errors
  (lambda (n)
    1))
```

```
(define factorial-1 ; computes 2! with no errors
  (lambda (n)
    (if (<= n 2)
        n
```

---

[2] In this specific case, T is not a matrix due to the constants 3 and -1.

9

```
            (* n (factorial-0 (- n 1))))))

(define factorial-2 ; computes 3! with no errors
  (lambda (n)
    (if (<= n 2)
        n
        (* n (factorial-1 (- n 1))))))

(define factorial-3 ; computes 4! with no errors
  (lambda (n)
    (if (<= n 2)
        n
        (* n (factorial-2 (- n 1))))))
```

and have `factorial-3` as an approximation of `factorial` that computes factorials up to 4 with no errors. We can conclude that `factorial` $= \lim_{n \to \infty}$ `factorial-n`. We can also say that `factorial = (f (f (f (f ...  x0))))` or that `factorial` is a fixed-point of the function `f`.

A way of making an infinite loop is by the Curry's Paradoxical Combinator of Y:

```
y = (lambda (f)
      ((lambda (x) (f (x x)))
       (lambda (x) (f (x x)))))
```

By applying `y` to `f`

```
(y f) = ((lambda (x) (f (x x)))
         (lambda (x) (f (x x))))
      = (f ((lambda (x) (f (x x)))
            (lambda (x) (f (x x)))))
```

we can conclude than that `(y f) = (f (y f))` which is exactly what we wanted (An infinite loop).

What Lisp is, is the fixed point of the process which says "If I knew what Lisp was and substituted it in for eval and apply, and so on, on the right hand sides of all those recursive equations, [...] Then the left hand side would also be Lisp". [4]

Gerald Jay Sussman,
Lecture 7A: Metacircular Evaluator. Timestamp: 1:16:37

1.5 Conclusion

> There are many languages that have made a mess of themselves
> by adding huge numbers of features. [...] I like to think of
> it is that many systems suffer from what is called "creeping
> featurism". [...] After a while, the thing has a manual 500 pages
> long that no one can understand. [...] In computer languages,
> I think it's a disaster to have too much stuff in them. [5]
>
> Gerald Jay Sussman,
> *Lecture 7B: Metacircular Evaluator. Timestamp: 2:48*

The fact that `Scheme` code is inherently represented as lists, with functions appearing as the first element of these lists, makes it exceptionally simple, convenient, and elegant to implement a metacircular evaluator. `Scheme`'s minimalistic design, combined with its remarkable abstraction capabilities, allows it to serve as an ideal foundation for creating domain-specific languages.

Understanding how to implement such an easy evaluator for a language like `Scheme`, is the first step towards learning how to create programming languages ourselves.

Bibliografia

[1] Harold Abelson e Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* 2nd. Cambridge, MA: MIT Press, 1996. ISBN: 978-0262510875. URL: `https://mitpress.mit.edu/sites/default/files/sicp/index.html`.

[2] John Barth. *Chimera.* Random House, 1972.

[3] Alan Perlis. *Epigrams in Programming.* online. Accessed on 2024-12-24. URL: `https://www.cs.yale.edu/homes/perlis-alan/quotes.html`.

[4] Gerald Jay Sussman. *Lecture 7A: Metacircular Evaluator, Part 1.* Accessed on 2024-12-24. URL: `https://youtu.be/aAlR3cezPJg?si=i4y2_VHeh16LAx6V&t`.

[5] Gerald Jay Sussman. *Lecture 7B: Metacircular Evaluator, Part 2.* Accessed on 2024-12-24. URL: `https://youtu.be/QVEOq5k6XiO?si=z_Mkb5oqF-BJo7Lp&t`.