# The Metacircular Evaluator

João Victor Lopez Pereira

January 5, 2025

Rio de Janeiro - RJ

**Abstract**

This document provides a comprehensive explanation of a metacircular evaluator implemented for `Scheme`. Developing a metacircular evaluator is a fundamental step toward mastering the creation of domain-specific languages.

The implementation, detailed in the `metacircular-evaluator.scm` file, closely follows the evaluator described in *Structure and Interpretation of Computer Programs*[2]. It incorporates several enhancements and simplifications to improve both clarity and functionality.

Additionally, the discussion in Section 1.4 draws significant inspiration from the lecture "Metacircular Evaluator"[5], delivered by Professor Gerald Jay Sussman at MIT.

For a deeper understanding, it is strongly recommended to carefully read both the accompanying quotes and the text.

# Contents

# Chapter 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! [3]
>
> 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. 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.[4]".

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 Computer Algebra, but may not be appropriate for simply plotting a sin function. `Python`, on the other hand, can be used to plot a sin function in no more than 3 lines of code, but is not the best for Computer Algebra. By the same argument, this document was made in `LaTeX` instead of `C` or some other general purpose language.

In conclusion, domain-specific languages prove to be particularly effective in scenarios where specialized tasks cannot be efficiently addressed by general-purpose languages. To allow a programer to develop its own language to address a program is to allow the creation of an instrument capable of shaping the complexity of the problem to the specific needs of the domain.

## 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. [2]
>
> 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 that are written can be easily seen as a tiny extension of a existing programming language. For example, imagine the following `factorial` procedure:

In `Scheme`:

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

In `Python`:

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

In `C`:

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

The `factorial` program defined above is in fact a tiny 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].

[1]Note that in `C` and `Scheme`, indentation is only used for better visualization and understanding of the defini-

The `factorial` procedure is a simple program considering that it has a very strict input and a very strict output, exactly like a mathematical function $f : \mathbb{N} \to \mathbb{N}$ such that:

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

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 it is considered 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.

This perspective encourages us to view programming not just as writing instructions for a computer but as the creation of languages through which computations are described. 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 `Scheme` 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, characters, arrays and user-defined types. However, in programs like interpreters or compilers, the data being manipulated is code itself. Notice that `Scheme` code is itself a `Scheme` list. Manipulating this list (which is, in fact, code) isn't much different from manipulating other lists.

   See, for example, a variation of the symbolic differentiation procedure made by Gerald Jay Sussman and Harold Abelson in Structure and Interpretation of Computer Programs[2]:

```
(define (deriv exp var)
  (cond ((constant? exp var)   0)
        ((same-var? exp var)   1)
        ((sum? exp)            (sum-rule      exp var))
        ((sub? exp)            (sub-rule      exp var))
```

tions. However, these are not required.

```
((product? exp)        (product-rule  exp var))
((division? exp)       (quotient-rule exp var))
((exponentiation? exp) (power-rule    exp var))
((composite? exp)      (chain-rule    exp var))
((ln? exp)             (ln-rule       exp var))
((exp? exp)            (expt-rule     exp var))
((sin? exp)            (sin-rule      exp var))
((cos? exp)            (cos-rule      exp var))
(else
  (error "case not found!"))))
```

The procedure `deriv` takes as input an expression `exp` and a variable `var` with respect to which the expression will be differentiated. The argument `exp` is typically a `Scheme` list such as `(+ (* a (* x x)) (+ (* b x) c))`. If the variables `a`, `b`, `c`, and `x` were defined, this expression could be evaluated to yield a numeric result. In this sense, `deriv` operates on `exp` in a way analogous to how the evaluator processes expressions. Furthermore, both `deriv` and the evaluator accept structurally similar arguments. The procedures `deriv` and `eval` (the procedure responsible for evaluating expressions) both manipulate lists in a very similar way. Since `Scheme` code is itself a `Scheme` list, it is extremely simple for `eval` to interpret and manipulate expressions directly, as the problem can be reduced to simply manipulating `Scheme` lists (which, in fact, are not *any* different from `Scheme` code). This seamless integration between code and data allows procedures like `eval` and `deriv` to easily traverse, analyze, and transform expressions without the need for additional parsing or conversion steps. This property not only simplifies the implementation of such procedures but also highlights the elegance and power of `Scheme`'s design.

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

   For most non-lisp users, `Scheme` code can be seen as pretty strange, "why does one prefer to write `(+ 1 2)` instead of `(1 + 2)`?".

   A convenience of having `+` as the first argument of the expression is to facilitate the parsing. 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). This is, in fact, the eval-apply cycle that will be better discussed in section 1.3.

Try to imagine how are these following codes parsed in their respective languages:

In `Scheme`:

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

In `Python`:

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

In `C`:

```
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? How are the operations precedence implemented? How exactly does the parser work?".

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

`Scheme` code doesn't need to be heavily parsed and manipulated in order for it to be evaluated. Since `Scheme` code undergoes minimal parsing, the process of evaluation becomes much easier to implement. The code is directly represented as the same data structure that the evaluator manipulates, allowing for straightforward traversal, interpretation, and transformation of expressions without the need for complex intermediate representations (similar to the `deriv` procedure).

The process of evaluation of the `Scheme` code shown above goes as followed:

1. the special form `define` is applied to 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.

It is easily imaginable that the `Scheme` interpreter 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
```

Questions about how code is evaluated are resolved through an understanding of the eval-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. [5]
>
> 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` known as the eval-apply cycle.

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. Notice that there are 3 kinds of procedures on the `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. 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.

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))
```

`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.

It's definitely uneven to think that a huge program can be evaluated by a program this small[2], but that's precisely the elegance of a metacircular evaluator: it leverages the power of the language it interprets to reflect its own semantics, demonstrating that even the most complex systems can be built upon simple, recursive principles.

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)
                           (map (lambda (exp) (EVAL exp env))
                                (operands exp))))
```

Here, the `operator` and `operands` procedures are defined as followed:

```
(define operator car)  ; first element of the expression

(define operands cdr)  ; all elements of the expression except for the first one
```

To modify `Scheme` for educational purposes — such as placing the operator as the second element of the list — it is possible to simply redefine these procedures:

```
(define operator cadr)      ; second element of the expression

(define (operands x)        ; all elements of the expression
  (cons (car x) (cddr x)))  ; except for the second one
```

This change would allow expressions like `(4 + 2)` and `(4 factorial)` to evaluate correctly while preserving the original semantics[3]. The ability to adapt the syntax so easily highlights the power and flexibility of this approach.

[2]The program includes more definitions than just `eval` and `apply` that, for the sake of simplicity and conciseness, will not be included here.

[3]With this modification some strange syntax would appear, like evaluating `(2 * 3 5 7)` in order to compute the product of the 4 first prime numbers.

Notice the similarities between `eval` and `deriv`. Both are recursive[4] procedures that traverse[5] a `Scheme` list and, based on the specific case encountered, invoke an appropriate procedure to operate on the given expression.

## 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 an interpreter 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? [...] The whole thing is sort of like these Escher's hands. [5]
>
> 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

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

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

[4]The recursive nature of `deriv` may not be immediately evident from its definition. However, consider the sum rule for differentiation, for instance, and the recursive structure of the procedure becomes apparent.

[5]Notice that not all functions within an expression passed to `eval` are evaluated. For instance, consider the `if` special form.

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 of interest 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 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 $T$[6].

Take a closer look at the `factorial` procedure 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`, it is first needed to rewrite it in such a way that the transformation $T$ becomes apparent.

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

Observe that `f` is a procedure such that, if the solution `g` were provided, the result would be the `factorial` procedure. Assuming `g` is the `factorial` procedure, `f` would produce precisely the same `factorial` procedure described 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 somehow the value of $x$ is needed to compute $x$ in $x = 4 - x$. As 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.

$$\begin{aligned} x_1 &= Tx_0 \\ x_2 &= Tx_1 \\ x_3 &= Tx_2 \\ &\vdots \\ x_k &= Tx_{k-1} \end{aligned}$$

Such that $x_i$, $\forall i \in \{1, \dots, k\}$, is an approximation of $x$. A similar approximation can be made to the factorial procedure:

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

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

; computes 2! with no errors
(define factorial-1 (f f-0))

; computes 3! with no errors
(define factorial-2 (f f-1))

; computes 4! with no errors
(define factorial-3 (f f-2))
```

and have `factorial-3` as an approximation of `factorial` that computes factorials up to 4 with no errors. It can concluded that the procedure `factorial` is equal to $\lim_{n\to\infty}$ `factorial-n`. It can also be said that `factorial` is equal to `(f (f (...(f factorial-0)...)))` 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)))))
      = (f (y f))
```

We can conclude 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 side of all those recursive equations, [...] Then the left hand side would also be Lisp". [5]
>
> 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. [6]
>
> 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.

> Once you have the interpreter in your hands, you have all this power to start playing with the language. [...] There’s this notion of metalinguistic abstraction, which says [...] that you can gain control of complexity by inventing new languages, sometimes. One way to think about computer programming is that it only incidentally has to do with getting a computer to do something. Primarily, what a computer program has to do with is a way of expressing ideas, of communicating ideas. Sometimes, when you want to communicate new kinds of ideas, you’d like to invent new modes of expressing them. [1]
>
> Lecture 8A: Logic Programming. Timestamp: 1:45,
> *Harold Abelson*

# Bibliography


[1] Harold Abelson. *Lecture 8A: Logic Programming, Part 1*. Accessed on 2024-12-28. URL: `https://youtu.be/rCqMiPk1BJE`.

[2] Harold Abelson and 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`.

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

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

[5] Gerald Jay Sussman. *Lecture 7A: Metacircular Evaluator, Part 1*. Accessed on 2024-12-24. URL: `https://youtu.be/aAlR3cezPJg`.

[6] Gerald Jay Sussman. *Lecture 7B: Metacircular Evaluator, Part 2*. Accessed on 2024-12-24. URL: `https://youtu.be/QVEOq5k6Xi0`.