Avaliador Metacircular de Lisp

João Victor Lopez Pereira

Instituto de Computação – UFRJ

3 de abril de 2025

Um pouco sobre a linguagem

- Linguagem funcional criada em 1958 para manipulação simbólica de listas;

Um pouco sobre a linguagem

- Linguagem funcional criada em 1958 para manipulação simbólica de listas;
- Apresenta listas como estrutura de dados fundamental;

Um pouco sobre a linguagem

- Linguagem funcional criada em 1958 para manipulação simbólica de listas;
- Apresenta listas como estrutura de dados fundamental;
- Os códigos da linguagem também são listas.

Exemplo de Código Lisp

```
(define x 10)
```

Exemplo de Código Lisp

```
(define x 10)

(define fib
  (lambda (n)
    (if (<= n 2)
        n
        (+ (fib (- n 1))
           (fib (- n 2))))))
```

Exemplo de Código Lisp

```
(define x 10)

(define fib
  (lambda (n)
    (if (<= n 2)
        n
        (+ (fib (- n 1))
           (fib (- n 2))))))
```

```
(define fact
  (lambda (x)
    (if (= x 1)
        1
        (* x
           (fact (- x 1))))))
```

Facilidade de Manipulação

É fácil manipular estruturas de dados! (pois todas as estruturas são listas!)

Facilidade de Manipulação

É fácil manipular estruturas de dados! (pois todas as estruturas são listas!)

```
(define map
  (lambda (l f)
    (if (null? l)
        '()
        (cons (f (car l))
              (map (cdr l) f)))))
```

Facilidade de Manipulação

Mas o próprio código de Lisp é formado por listas...

Facilidade de Manipulação

Mas o próprio código de Lisp é formado por listas...

Isso significa que é fácil manipular código Lisp?

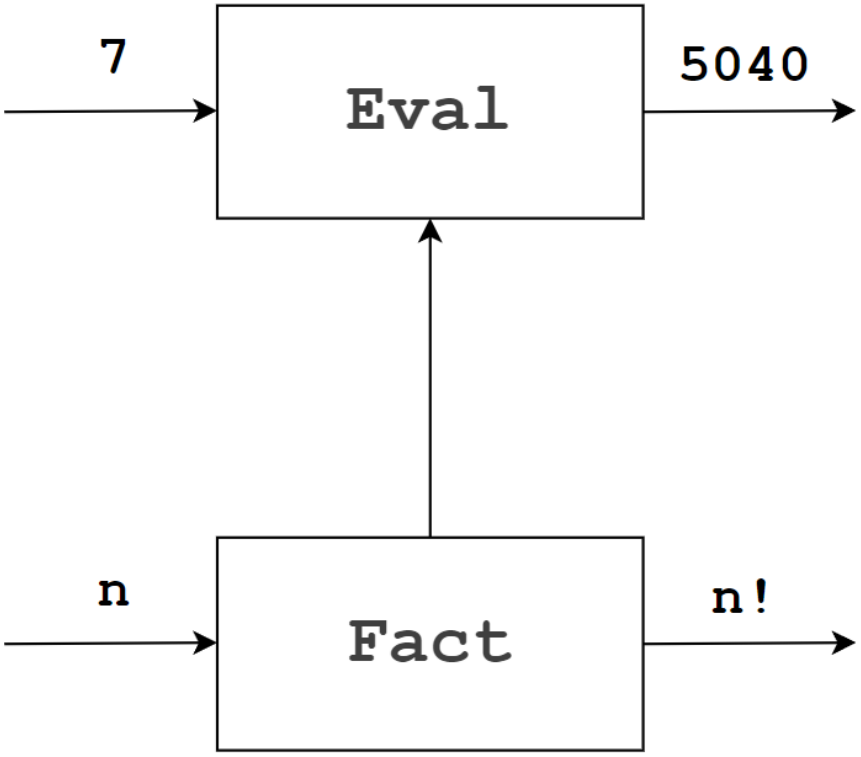
Universal Machine

Linguagens de programação interpretadas/avaliadas apresentam uma "máquina" chamada `eval`.

Universal Machine

Universal Machine

Universal Machine



Universal Machine

De maneira que essa construção, conhecida como *Universal Machine*, seja um simulador (*Describing Machine*) para a máquina `fact`, que, inclusive, é infinita.

O Avaliador

The evaluator isn't very complicated, it's very much like all the programs we have seen already: that's the amazing part of it.
— Gerald Jay Sussman.

O Avaliador

```
(eval 5) -> 5
```

O Avaliador

```
(eval 5) -> 5

(eval (define x 17)) -> "variavel definida"
```

O Avaliador

```
(eval 5) -> 5

(eval (define x 17)) -> "variavel definida"

(eval x) -> 17
```

O Avaliador

```
(eval 5) -> 5

(eval (define x 17)) -> "variavel definida"

(eval x) -> 17

(eval (if (> 5 3) 1 0)) -> 1
```

O Avaliador

```
(eval 5) -> 5

(eval (define x 17)) -> "variavel definida"

(eval x) -> 17

(eval (if (> 5 3) 1 0)) -> 1

(eval ((lambda (x) (* x 5)) 7)) -> 35
```

O Avaliador

```
(eval 5) -> 5

(eval (define x 17)) -> "variavel definida"

(eval x) -> 17

(eval (if (> 5 3) 1 0)) -> 1

(eval ((lambda (x) (* x 5)) 7)) -> 35

(eval (+ 1 2 3)) -> 6
```

O Avaliador

```
(define (EVAL exp env)
  (cond
    ((number? exp)       exp)
    ((variable? exp)     (lookup-variable-value exp env))
    ((definition? exp)   (eval-definition exp env))
    ((if? exp)           (eval-if exp env))
    ((lambda? exp)       (make-procedure (cadr exp)
                                         (cddr exp) env))
    ((application? exp) (apply (EVAL (car exp) env)
                               (map (lambda (exp)
                                            (EVAL exp env))
                                    (cdr exp))))))
```

Lisp como Ponto Fixo

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?

— Gerald Jay Sussman.

Convergência a Depender do Conteúdo

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

Solução única em x e y.

Convergência a Depender do Conteúdo

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

Nenhuma solução em x e y.

Convergência a Depender do Conteúdo

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

Definição de Fact

```
(define fact
  (lambda (x)
    (if (= x 1)
        1
        (* x (fact (- x 1))))))
```

Fact como Ponto Fixo

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

Fact como Ponto Fixo

```
(define fact-0 (lambda (n) 1))
(define fact-1 (t (fact-0)))
(define fact-2 (t (fact-1)))
(define fact-3 (t (fact-2)))
...
```

Fact como Ponto Fixo

```
(define fact-0 (lambda (n) 1))
(define fact-1 (t (fact-0)))
(define fact-2 (t (fact-1)))
(define fact-3 (t (fact-2)))
...
```

$$\texttt{factorial} = \lim_{k \to \infty} \texttt{t}^k \texttt{fact-0}$$

Eval como Ponto Fixo

```
(define (EVAL exp env)
  (cond
    ((number? exp)      exp)
    ((variable? exp)    (lookup-variable-value exp env))
    ((definition? exp)  (eval-definition exp env))
    ((if? exp)          (eval-if exp env))
    ((lambda? exp)      (make-procedure (cadr exp)
                                        (cddr exp) env))
    ((application? exp) (apply (EVAL (car exp) env)
                               (map (lambda (exp)
                                      (EVAL exp env))
                                    (cdr exp))))))
```

Eval como Ponto Fixo

```
(define t (lambda (ev) (lambda (exp env)
  (cond
    ((number? exp)       exp)
    ((variable? exp)     (lookup-variable-value exp env))
    ((definition? exp)   (eval-definition exp env))
    ((if? exp)           (eval-if exp env))
    ((lambda? exp)       (make-procedure (cadr exp)
                                         (cddr exp) env))
    ((application? exp) (apply (ev (car exp) env)
                               (map (lambda (exp)
                                      (ev exp env))
                                    (cdr exp)))))))))
```

Eval como Ponto Fixo

```
(define eval-0 (lambda (n) 1))
(define eval-1 (t (eval-0)))
(define eval-2 (t (eval-1)))
(define eval-3 (t (eval-2)))
...
```

Eval como Ponto Fixo

```
(define eval-0 (lambda (n) 1))
(define eval-1 (t (eval-0)))
(define eval-2 (t (eval-1)))
(define eval-3 (t (eval-2)))
...
```

$$\texttt{eval} = \lim_{k \to \infty} \texttt{t}^k \texttt{eval-0}$$

Eval como Ponto Fixo

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

— Gerald Jay Sussman.

Por que um Interpretador?

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.

— Gerald Jay Sussman & Harold Abelson.

Por que um Interpretador?

- Desmistificação do interpretador;

Por que um Interpretador?

- Desmistificação do interpretador;
- Flexibilidade e extensão;

Por que um Interpretador?

- Desmistificação do interpretador;
- Flexibilidade e extensão;
- Criação de *Domain Specific Languages*.

Muito obrigado!

Perguntas?

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.

— Harold Abelson.

Referências

[1] Harold Abelson. *Lecture 8A: Logic Programming, Part 1*. Accessed on 2024-12-28.

[2] Harold Abelson e Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. Cambridge, Massachusetts: The MIT Press, jul. de 1984. ISBN: 9780262010771.

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