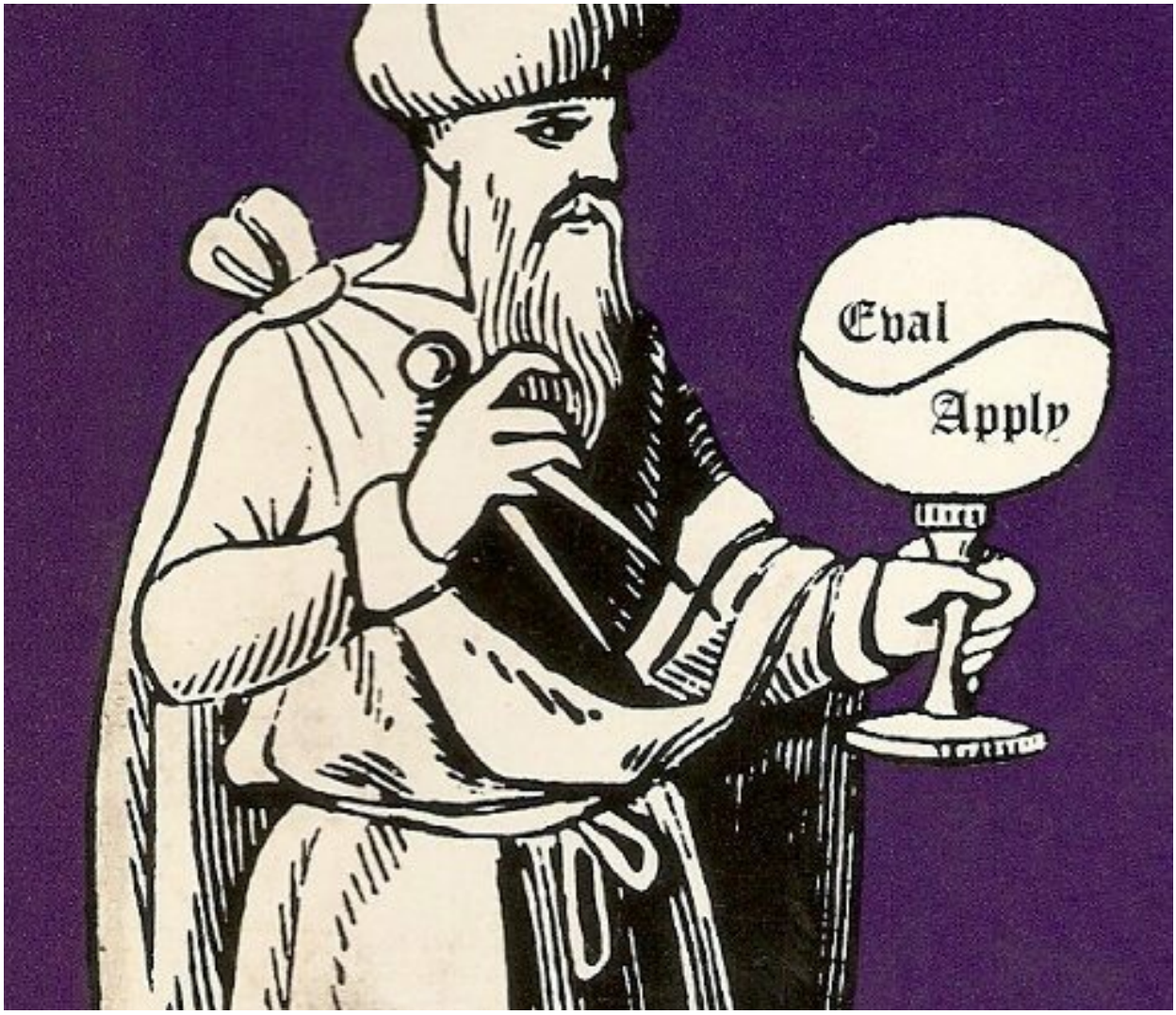
(((Scheme)))

Do básico ao intermediário



1. Introdução

> Computational processes are abstract beings that inhabit computers. As they evolve, processes manipulate other abstract things called data. The evolution of a process is directed by a pattern of rules called a program. People create programs to direct processes. In effect, we conjure the spirits of the computer with our spells.
>
> — Harold Abelson

Scheme é uma linguagem multiparadigma e um dos principais dialetos de LISP, uma família de linguagens com foco em programação funcional e cálculo lambda. Scheme foi criada na década de 70 pelos pesquisadores Guy L. Steele e Gerald Jay Sussman (MIT). Seguindo uma filosofia minimalista, Scheme pode ser considerada como uma linguagem *educacional*, e por um bom tempo foi a linguagem utilizada em disciplinas introdutórias de Ciência da Computação no MIT.

Uma linguagem de programação poderosa é mais do que simplesmente um meio de instruir o computador a executar determinada tarefa. A linguagem também serve para organizar e modelar nossas ideias sobre processos. Toda linguagem poderosa tem três mecanismos para realizar isso:

- **expressões primitivas**, que representam as mais simples entidades nas quais a linguagem está interessada;
- **meios de combinação**, pelo qual elementos compostos são construídos a partir de elementos mais simples;
- **meios de abstração**, pelo qual elementos compostos podem ser nomeados e manipulados como unidades.

Neste tutorial, irei exibir os principais elementos de Scheme -- uma linguagem simples e muitas vezes estranha. Não focarei nos elementos mais teóricos da linguagem, mas sim nos elementos práticos dela.

1.1 Instalação

Para programar em Scheme, antes é preciso instalar as dependências necessárias. Neste tutorial, utilizarei o MIT/GNU Scheme, uma implementação completa de Scheme com interpretador, compilador, debugger, etc. Os binários podem ser encontrados em https://www.gnu.org/software/mit-scheme/
ou no repositório de sua distribuição GNU/Linux.

Caso julgar a implementação MIT/GNU insuficientemente prática, você pode utilizar Racket com a DrRacket IDE e o pacote *sicp* (https://docs.racket-lang.org/sicp-manual/).

Os arquivos em Scheme têm a extensão *.scm*. Caso tiver instalado MIT/GNU Scheme, digite o comando `scheme < nome_do_arquivo.scm` para executar um programa pelo terminal.

2. Os básicos

2.1 Manipulação de expressões

Na maior parte deste tutorial, estaremos manipulando números, um tipo de *expressão primitiva*. Abra o REPL do Scheme digitando `scheme` no terminal.

Caso você digitar um número qualquer, como por exemplo 166, o interpretador retornará `166` como esperado. Naturalmente, você pode combinar expressões para gerar resultados mais interessantes. Como Scheme utiliza notação de prefixo ("notação polonesa"), o operador precede os operandos:

```
(+ 133 220)
; 353
(- 344 567)
; -233
(* 4 5)
; 20
(/ 30 5)
; 6
(+ 2.6 0.3)
; 2.9
(* 3 4 5 6 7 8)
; 20160
```

Uma vantagem da notação de prefixo é que você pode aninhar expressões de um modo bastante direto:

```
(+ (* 3 5) (- 10 6))
; 19
(/ 53.0 (+ 65 (* 2 2)))
; .7681159420289855
```

É uma boa prática alterar a forma do código quando necessário:

```
(+ (* 3
      (+ (* 2 4)
         (+ 3 5)))
   (+ (- 10 7)
      6))
; 57
```

2.2 A palavra-chave *define*

Em Scheme, utilizamos a palavra-chave *define* para nomear *coisas*. Pode-se dizer que o nome identifica uma *variável* cujo valor é o *objeto*. Por exemplo, ao digitarmos

```
(define soma-qualquer (+ 2 3))
```

Estamos definindo *soma-qualquer* como o valor da soma de 2 e 3.

```
soma-qualquer
; 5
```

2.3 *Definindo* procedimentos

Definição de procedimentos é uma técnica de abstração na qual pode ser dado um nome à uma operação composta, que por fim é referenciada como uma unidade qualquer.

O usuário pode definir seus próprios procedimentos utilizando a palavra-chave *define* vista anteriormente. Vamos definir, por exemplo, o procedimento *cubo*, que retorna um número *x* elevado à terceira potência.

```
(define (cubo x) (* x x x))

(cubo 3)
; 9

(cubo (+ 3 3))
; 216

(cubo (cubo (+ 3 3)))
; 10077696

; Note como x pode ser um procedimento. Em Scheme,
; você pode manipular procedimentos como se fossem
; valores quaisquer.

; Esqueci de informar: podemos usar ponto e vírgula
; para escrever comentários.
```

Como pode ser observado acima, a forma geral de um procedimento é a seguinte:

```
(define (<nome> <parâmetros formais>)
  <corpo>)
```

Podemos utilizar *cubo* como um bloco de construção para procedimentos mais complexos:

```
(define (soma-de-cubos x y)
  (+ (cubo x) (cubo y)))

(soma-de-cubos 5 5)
; 250
```

Ou, ainda, podemos definir *cubo* localmente:

```
(define (soma-de-cubos x y)
  (define (cubo x) (* x x x))
  (+ (cubo x) (cubo y)))

(soma-de-cubos 34534 2353456)
; 13035257711177228120
```

2.4 Expressões condicionais

Scheme possui quatro principais palavras-chave usadas em controle de fluxo: *if*, *cond*, *and* e *or*. Para análises de caso, a expressão *cond* é comumente utilizada.

Vamos escrever, por exemplo, um procedimento que retorna o valor absoluto de um número *x*:

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

Observando acima, podemos observar que a forma geral de *cond* é a seguinte:

```
(cond (<predicado> <expressão consequente>)
      (<predicado> <expressão consequente>)
      ...
```

```
      (<predicado> <expressão consequente>))
```

Se quisermos reescrever *abs* usando *if*:

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

Observa-se que a forma de *if* é:

```
(if <predicado>
    <consequente>
    <alternativo>)
```

Como foi indicado anteriormente, em Scheme também podemos utilizar operadores lógicos como *not*, *or* e *and*. Como exemplo, podemos definir um predicado para testar se um número *x* é maior ou igual a outro número *y*:

```
(define (>= x y) (or (> x y) (= x y)))

(>= 5 3)
; #t
(>= 3 5)
; #f
```

Exemplo: Raiz quadrada pelo método de Newton

Neste código, utilizamos um caso específico do método de Newton (aproximações sucessivas) para calcular a raiz quadrada de um número. Este método diz que, quando tivermos um palpite *y* para o valor da raiz quadrada de um número *x*, nós podemos obter um palpite melhor fazendo a média de *y* com *x/y*. Repetindo esse processo, nós obtemos aproximações cada vez melhores da raiz quadrada.

```
(define (raiz-quadrada x)
  (define (raiz-quadrada-iter palpite x) ; iteração
    (if (suficientemente-bom? palpite x)
        palpite
```

```
        (raiz-quadrada-iter (melhore palpite x) x)))
  (raiz-quadrada-iter 1.0 x)) ; 1.0 é o nosso palpite inicial

(define (suficientemente-bom? palpite x)
  (< (abs (- (square palpite) x)) 0.001))

; obs.: em Scheme, é uma boa prática colocar pontos de interrogação
; em predicados.

(define (melhore palpite x)
  (media palpite (/ x palpite)))

(define (media x y)
  (/ (+ x y) 2))

(raiz-quadrada 4.0)
; 2.0000000929222947
(raiz-quadrada 2.0)
; 1.4142156862745097
```

2.5 Pares e listas

Em Scheme, *pares* de expressões são usados como base para listas. Um par combina dois valores, e os procedimentos *car* e *cdr* (pronuncia-se "'kü'dEr") são usados para acessar o primeiro e o segundo valor do par, respectivamente.

```
; Usamos 'cons' para gerar um par de dois valores quaisquer.
(define par (cons 1 2))

par
; (1 . 2)

(car par)
; 1
(cdr par)
; 2

; Podemos criar pares de pares. Notou algo interessante no
; resultado de 'novo-par'?
(define novo-par (cons par 3))

novo-par
; ((1 . 2) . 3)

(car novo-par)
; (1 . 2)
(cdr novo-par)
; 3
```

Se você percebeu que o resultado de *novo-par* se parece com uma lista, parabéns: listas, em Scheme, não passam de pares em cascata. Veja como podemos gerar uma pequena lista apenas com pares:

```
(define lista (cons 0
                    (cons 1
                          (cons 2
                                (cons 3 '())))) ; '() -> nil (nulo)
lista
; (0 1 2 3) ; Ora ora, uma lista!

(pair? lista)
; #t
(car lista)
; 0
(cdr lista)
; (1 2 3)
```

Naturalmente, seria complicado se tivéssemos que escrever pares em cascata toda vez que precisássemos criar uma lista. Usando o procedimento *list*, podemos criar listas facilmente.

```
(define lista1 (list 1 2 3))

lista1
; (1 2 3)

; Você também pode definir listas aninhadas:
(define lista2 (list 1 (list 5 6 (list 7 2 5)) 10 11 12 lista1))

lista2
; (1 (5 6 (7 2 5)) 10 11 12 (1 2 3))

; Podemos utilizar cadr e suas variações para acessar elementos
; específicos da lista:
(cadr lista2) ; retorna o segundo elemento da lista1
(caddr lista2) ; retorna o terceiro
(cadddr lista2) ; retorna o quarto e assim por diante

; Veja mais em:
;https://franz.com/support/documentation/8.1/ansicl/dictentr/carcdrca.htm
```

Certo, listas são bem úteis. Mas como posso percorrê-la em um "loop", como estamos acostumados? Podemos acessar os diferentes elementos em uma lista "cdrando", ou seja, aplicando *cdr* sucessivamente na lista.

Vamos implementar uma função que retorne o valor de uma lista indicado pelo argumento de referência (ou índice):

```
(define (lista-ref lista i)
  (if (zero? i)
      (car lista)
      (lista-ref (cdr lista) (- i 1))))

(lista-ref (list "s" "c" "h" "e" "m" "e") 2)
; "h"
```

Como geralmente percorremos uma lista com muita frequência, Scheme nos fornece o predicado primitivo *null?*, que checa se o argumento é a lista vazia. O procedimento abaixo calcula o tamanho de uma lista:

```
(define (tamanho lista)
  (if (null? lista)
      0
      (+ 1 (tamanho (cdr lista)))))

(tamanho (list 1 2 3 4 5 (list 1 2 3)))
; 6
```

2.6 Map

Em Scheme, *map* é um procedimento que *faz algo* com cada elemento de uma lista. *Map* aceita como argumentos um procedimento e uma lista, e executará esse procedimento sobre a lista.

```
(define (duplicar x) (+ x x))
(map duplicar (list 1 2 3 4 5))
; (2 4 6 8 10)
```

Veja outros procedimentos sobre listas em https://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/Mapping-of-Lists.html

3. O intermediário

3.1 Funções de ordem superior

Dos conceitos de programação funcional, uma função de ordem superior caracteriza-se por ser uma função que tem como argumento uma função ou que retorne uma função.

Veja um exemplo de procedimentos de ordem superior sendo utilizados nas duas maneiras descritas:

```
(define (adicione-um)
  (define (inc x) (+ x 1))
  inc) ; retorna um procedimento

adicione-um
; #[compound-procedure 13 adicione-um]

(define f (adicione-um)) ; tem um procedimento como argumento
(f 5)
; 6
```

Funções de ordem superior são usadas muito comumente em Scheme. E, na verdade, já havíamos usado esse tipo de função na seção 2.X. Volte para ela e veja se consegue achar.

3.2 Lambda

Em Scheme, podemos criar funções anônimas (sem nome) quando quisermos. O uso de lambda é extremamente útil caso precisarmos criar um procedimento específico que será usado em apenas um lugar.

Na verdade, toda definição de procedimento em Scheme faz uso de lambda. Quando fazemos *(define f 5)*, por exemplo, estamos utilizando lambda por baixo.

Lembra-se do código utilizado na seção *2.6 Map*? Podemos reescrevê-lo utilizando lambda da seguinte forma:

```
(map (lambda (x) (+ x x)) (list 1 2 3 4 5))
; (2 4 6 8 10)
```

3.3 *Closure* (clausura)

Uma clausura é um procedimento que "grava" o ambiente no qual ela foi criada. Quando você o chama, esse ambiente é restaurado antes do código ser executado.

Na verdade, usamos clausura na seção *3.1*, quando definimos `(define f (adicione-um))`. Note que *adicione-um* retorna uma clausura, que contém uma referência à *(inc x)* e uma cópia do ambiente ao redor dele. Quando definimos *f*, fazemos que ele retorne um procedimento. Assim, ao invocarmos *f* com algum valor, será retornado esse valor somado a 1.

Veja outro exemplo de clausura, dessa vez usando lambda:

```
(define (somador a)
    (lambda (b) (+ a b)))

(define mais-5 (somador 5))

mais-5
; #[compound-procedure 13]

(mais-5 4)
; 9
```

3.4 Recursão

Scheme faz uso constante de recursão (procedimentos que chamam a si mesmos), em especial de *recursão de cauda*, onde utilizamos recursão ao invés de iteração. São executados os cálculos primeiro e, em seguida, é executada a chamada recursiva, passando os resultados de sua etapa atual para o próximo passo recursivo. Na recursão tradicional, o que ocorre é o oposto.

Vamos ver como o clássico fatorial é implementado em Scheme:

```
(define (fact n)
  (if (= 0 n)
      1
      (* n (fact (- n 1)))))

(fact 6)
; 720
```

Mais interessante ainda: implementaremos uma versão simples de map utilizando recursão.

```
(define (mapeie f lista)
  (if (null? lista)
```

```
      '()
      (cons (f (car lista))
            (mapeie f (cdr lista)))))

(mapeie square (list 1 2 3 4 5))
; (1 4 9 16 25)
```

Naturalmente, operações recursivas são bastante custosas. Podemos optar por soluções iterativas quando for conveninente:

```
(define (fact-iter produto contador max-cont)
  (if (> contador max-cont)
      produto
      (fact-iter (* contador produto)
                 (+ contador 1)
                 max-cont)))

; Ou, de outro modo:

(define (fact n)
  (define (fact-iter acumulador contador)
    (if (= 0 contador)
        acumulador
        (fact-iter (* contador acumulador) (- contador 1))))
  (fact-iter 1 n))

(fact 6)
; 720
```

Comparando as formas das versões recursiva e iterativa de fatorial, temos o seguinte:

```
--> Versão recursiva:
(fact 6)
(* 6 (fact 5))
(* 6 (* 5 (fact 4)))
(* 6 (* 5 (* 4 (fact 3))))
(* 6 (* 5 (* 4 (* 3 (fact 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (fact 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720

--> Versão iterativa (1a):
(fact 6)
```

```
(fact-iter 1 1 6)
(fact-iter 1 2 6)
(fact-iter 2 3 6)
(fact-iter 6 4 6)
(fact iter 24 5 6)
(fact-iter 120 6 6)
(fact-iter 720 7 6)
720
```

Outros conceitos

Chegamos ao fim deste guia, que abrange o básico e um pouco do intermediário em Scheme. Na verdade, com os conceitos que vimos até agora, já podemos implementar uma boa parcela de problemas em Scheme, especialmente matemáticos.

Fica a cargo do leitor fazer uma busca por outros conceitos. Se quiser, comece pelas referências abaixo.

Referências

Structure and Interpretation of Computer Programs, 2nd ed., Harold Abelson and Gerald Jay Sussman with Julie Sussman

The Little Schemer, 4th ed., Daniel P. Friedman & Matthias Felleisen

https://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/

ftp://ftp.cs.utexas.edu/pub/garbage/cs345/schintro-v13/schintro_122.html

https://people.eecs.berkeley.edu/~bh/ssch8/higher.html

ftp://ftp.cs.utexas.edu/pub/garbage/cs345/schintro-v13/schintro_127.html