Programação Funcional

- funções como valores
  - argumentos
  - resultado
  - em variáveis
  - em estruturas de dados
- programação sem efeitos colaterais
- funcionais ou funções de ordem superior
  - recebem e retornam funções
- <u>Scheme e Sasl</u>

Scheme

- Lambda: principal diferença com Lisp é novo tipo de expressão:
  - (lambda (parFormal1 ... parFormalN) expr)
- esta expressão retorna função c/ parâmetros formais *parformal1,... parformalN* e “corpo” *expr*.
- ex: (lambda (x y) (+ (* x x) (* y y)))
  - retorna função que calcula soma dos quadrados
- uso:

->(**(lambda (x y) (+ (* x x) (* y y)))** 3 4)

25

Funções são valores de primeira ordem

- agora como funções são valores "normais" podemos guardá-los em variáveis, assim:

(define (f x1 ... xN) expr) ~ (define f (lambda (x1... xN) expr))

- podemos também passar funções como argumentos, e retornar funções como resultado, criand funções mais gerais.

Exemplo: função geral de ordenação

- para ordenar listas de números podemos utilizar a função primitiva “<“,porém para algoritmos mais gerais de ordenação queremos algo melhor:

```
->(define sort2 ( lambda (num1 num2 comparacao)
        (if (comparacao num1 num2)
            (list2 num1 num2)
            (list2 num2 num1))))
->(sort2 7 5 <)
(5 7)
->(sort2 7 5 >)
(7 5)
```

Exemplo - ordenação - 2

- podemos utilizar agora sort2 para ordenar pares de quaisquer elementos, desde que definamos uma ordem para os elementos:

```
->(define compara-pares (lambda (par1 par2)
     (if (< (car par1) (car par2))
          #t
          (if (< (car par2) (car par1))
               #f
               (< (cadr par1) (cadr par2))))))
->(sort2 ´(4 5) ‘(4 3)  compara-pares)
((4 3)(4 5))
```

Exemplo - ordenação - 2

- Podemos também generalizar sort2 para uma ordenação lexicográfica de duas listas quaisquer, onde quem define a ordem é o primeiro elemento diferente

```
(define ordem-lexicografica-numeros
 (lambda (lista1 lista2)
  (if (null? lista1)
      #t
      (if (null? lista2)
          #f
          (if (< (car lista1) (car lista2))
              #t
              (if (> (car lista1) (car lista2))
                  #f
                  (ordem-lexicografica-numeros (cdr lista1) (cdr lista2))))))))
(ordem-lexicografica ‘(1 2 3) ‘(1 2 3 4)) => #f
(ordem-lexicografica ‘(1 2 3 4) (1 2 4 4) => #t
```

Exemplo - ordenação
Podemos agora fazer um insertion-sort genérico

- Para isso basta passarmos a função de comparação como parâmetro:

```
->(define (insertion-sort  less-than l )
      (if  (null? l) l
           (insert less-than (car l)
           (insertion-sort  less-than (cdr l) ))))
  (define (insert less-than x l)
       (if (null? l) (list x)
           (if  (less-than x (car l)) (cons x l)
           (cons (car l) (insert less-than x (cdr l))))))
->(insertion-sort < '(3 2 1 4)) => '(1 2 3 4)
->(insertion-sort >  '(3 2 1 4)) => '(3 4 2 1))
->(insertion-sort  ordem-lexicografica-numeros '( (3 5) (4 3)) ==> '((4 3) (4 5))
```

NOTA: usando lambda podemos também inverter uma ordem

```
-> (define inverte-ordem (lambda (compara)(lambda (x y)  (compara y x)))))
```

Exemplo2 - função como resultado

- funções como parâmetro não são novidade, porem podemos ter funções como resultado:

```
->(define soma-fixa (lambda (incremento)
      (lambda (x) (+ x incremento))))
->(define soma-1 (soma-fixa 1))
->(soma-1 5)
6
```

- outro exemplo: derivada

```
(define deriva (lambda (f dx)
   (lambda (x) (/ (- (f (+ x dx))
                     (f x))
                  dx)))
```

Sintaxe

- input -> expression
- arglist -> (variable*)
- expression -> value
  - | variable
  - | (**if** expression expression expression)
  - | (**define** variable expression)
  - | (**begin** expression$^+$)
  - | (expression*)
  - | (**lambda** arglist expression)
- optr -> function | value-op
- value -> integer | quoted-const |
  closure| value-op
- value-op -> + | - | * | / | = | < | > |
  car | cdr | cons | number? | symbol? | list? |
  null? | procedure? | print

Semântica

- 2 tipos de funções:
  - value-ops (primitivas) e
  - lambdas (representando “fechamentos”)
- fechamentos (closure), notação:

  << lambda-expr, ambiente>>

- assim,

Ksoma-1 = << (lambda(x)(+ x incremento)), {incremento->1}>>

- uma expressão simbólica pode ser
  - símbolo
  - número
  - operação primitiva
  - fechamento
  - lista de expressões simbólicas

Calculando o valor de uma expr. simbólica

- ao calcular a expressão (lambda (x1 ... xN) e) no ambiente local *rho* temos como resultado o fechamento:

  <<(lambda(x1...xn)e),rho>>

  – ex: (deriva (soma1 0,000001)) retorna

  <<**(lambda(x)(/ (- (f (+x dx)) (f x)) dx))**, **{f->soma1, dx->0,000001}**>>

- agora, para calcularmos uma expressão simbólica qualquer (e0 e1...eN) devemos também calcular o valor de *e0* que deve ser:
  – operador de controle
  – operação primitiva
  – fechamento

Fechamentos

- fechamento é um par (lambda, ambiente|)
- Dado um fechamento <<(lambda(x1 ... xn) expr), rho>>
- algumas variáveis em *expr* podem não ser parâmetros (e.g. *x* em Kadd1) e constituem *variáveis livres*
- ambiente tem como função dar valor a estas variáveis
- Definição: o ambiente rho extendido de

$$\{\mathbf{x_1 -> v_1, ... x_N -> v_N}\},$$

denotado por

$$rho\{\mathbf{x_1 -> v_1, ... x_N -> v_N}\}$$

é o mesmo ambiente que rho, mas que associa a cada $\mathbf{x_i}$ o valor $\mathbf{v_i}$

Cálculo do valor de uma expressão simbólica

- para calcular o valor de

$$(\mathbf{e_0} \ \mathbf{...} \ \mathbf{e_N})$$

- quando e0 é o fechamento

$$\mathbf{<<(lambda(x_1...x_N)expr),rho>>}$$

- e v_1,...v_N forem os valores de e_1,...e_N,
- basta calcularemos o valor de *expr* no ambiente

$$\mathbf{rho\{x_1\text{->}v_1,...x_N\text{->}v_N)\}}$$

Exemplo: mapcar

- mapcar - recebe uma função *func* e uma lista *l* como argumentos, retorna uma lista com o resultado de aplicar *func* a cada elemento de *l*.
- (define mapcar (lambda (func l)

```
   (if (null? l)
         ‘()
         (cons (func (car l))
                 (mapcar func (cdr l))))))
->(mapcar soma1 ´(3 4 5)
(4 5 6)
->(mapcar number? ´(3 a (3 4) b))
(#t #f #f #f)
->(define soma1* (lambda (l) mapcar soma1 l))
->(soma1* ´(3 4 5))
(4 5 6)
```

Exemplo: curry

- a função curry cria funções que fixam parcialmente parâmetros de outras funções
- (define curry (lambda (f)
-         (lambda (x)
-                 (lambda (y) (f x y)))))

->((curry +) 3) 4)

(<<(lambda(f) (lambda(x)(lambda(y)(f x y)))),{}>> +)   3)   4)

((<<(lambda(x)(lambda(y)(f x y)),{f->+}>>   3)   4)

((<<(lambda(y)(f x y)), {f->+}{x ->3}>>   4)

(f x y) {f->+}, {x->3},{ y->4}

(+ 3 4)

7

Exemplo: curry

- a função curry cria funções que fixam parcialmente parâmetros de outras funções
- (define curry (lambda (f)
- (lambda (x)
- (lambda (y) (f x y)))))

Exemplo: curry

- a função curry cria funções que fixam parcialmente parâmetros de outras funções
- (define curry (lambda (f)
- (lambda (x)
- (lambda (y) (f x y)))))

((curry +) 3) 4)

Exemplo: curry

- a função curry cria funções que fixam parcialmente parâmetros de outras funções
- (define curry (lambda (f)
- (lambda (x)
- (lambda (y) (f x y)))))

(((curry +) 3) 4)

Exemplo: curry

- a função curry cria funções que fixam parcialmente parâmetros de outras funções
- (define curry (lambda (f)
- (lambda (x)
- (lambda (y) (f x y)))))

(((curry +) 3) 4)

((<<(lambda(f) (lambda(x)(lambda(y)(f x y)))),{}>> +) 3) 4)

Exemplo: curry

- a função curry cria funções que fixam parcialmente parâmetros de outras funções
- (define curry (lambda (f)
- (lambda (x)
- (lambda (y) (f x y)))))

(((curry +) 3) 4)

((<<(lambda(f) (lambda(x)(lambda(y)(f x y)))),{}>> +) 3) 4)

Exemplo: curry

- a função curry cria funções que fixam parcialmente parâmetros de outras funções
- (define curry (lambda (f)
- (lambda (x)
- (lambda (y) (f x y)))))

((curry +) 3) 4)

(((<<(lambda(f) (lambda(x)(lambda(y)(f x y)))),{}>> +) 3) 4)

((<<(lambda(x)(lambda(y)(f x y),{f->+}>> 3) 4)

Exemplo: curry

- a função curry cria funções que fixam parcialmente parâmetros de outras funções
- (define curry (lambda (f)
- (lambda (x)
- (lambda (y) (f x y)))))

((curry +) 3) 4)

(((<<(lambda(f) (lambda(x)(lambda(y)(f x y)))),{}>> +) 3) 4)

((<<(lambda(x)(lambda(y)(f x y),{f->+}>> 3) 4)

Exemplo: curry

- a função curry cria funções que fixam parcialmente parâmetros de outras funções
- (define curry (lambda (f)
-     (lambda (x)
-         (lambda (y) (f x y)))))

((curry +) 3) 4)

```
((( <<(lambda(f) (lambda(x)(lambda(y)(f x y)))),{}>>  +)     3)     4)
( ( <<(lambda(x)(lambda(y)(f x y),{f->+}>>          3)       4)
( ( <<(lambda(y)(f x y), {f->+}{x ->3}>>                     4)
```

Exemplo: curry

- a função curry cria funções que fixam parcialmente parâmetros de outras funções
- (define curry (lambda (f)
-         (lambda (x)
-                 (lambda (y) (f x y)))))

((curry +) 3) 4)

(((<<(lambda(f) (lambda(x)(lambda(y)(f x y)))),{}>> +)   3)   4)

((<<(lambda(x)(lambda(y)(f x y),{f->+}>>   3)   4)

((<<(lambda(y)(f x y), {f->+}{x ->3}>>   4)

Exemplo: curry

- a função curry cria funções que fixam parcialmente parâmetros de outras funções
- (define curry (lambda (f)
- (lambda (x)
- (lambda (y) (f x y)))))

((curry +) 3) 4)

(((<<(lambda(f) (lambda(x)(lambda(y)(f x y)))),{}>> +) 3) 4)

((<<(lambda(x)(lambda(y)(f x y),{f->+}>> 3) 4)

((<<(lambda(y)(f x y), {f->+}{x ->3}>> 4)

(f x y) {f->+}, {x->3},{ y->4}

Exemplo: curry

- a função curry cria funções que fixam parcialmente parâmetros de outras funções
- (define curry (lambda (f)
-     (lambda (x)
-         (lambda (y) (f x y)))))

((curry +) 3) 4)

(((<<(lambda(f) (lambda(x)(lambda(y)(f x y)))),{}>> +)   3)   4)

((<<(lambda(x)(lambda(y)(f x y),{f->+}>>   3)   4)

((<<(lambda(y)(f x y), {f->+}{x ->3}>>   4)

(f x y) {f->+}, {x->3},{ y->4}

(+ 3 4)

Exemplo: curry

- a função curry cria funções que fixam parcialmente parâmetros de outras funções
- (define curry (lambda (f)
-         (lambda (x)
-                 (lambda (y) (f x y)))))

((curry +) 3) 4)

(((<<(lambda(f) (lambda(x)(lambda(y)(f x y)))),{}>> +)   3)   4)

((<<(lambda(x)(lambda(y)(f x y),{f->+}>>   3)   4)

((<<(lambda(y)(f x y), {f->+}{x ->3}>>   4)

(f x y) {f->+}, {x->3},{ y->4}

(+ 3 4)

Exemplo: curry

- a função curry cria funções que fixam parcialmente parâmetros de outras funções
- (define curry (lambda (f)
- (lambda (x)
- (lambda (y) (f x y)))))

((curry +) 3) 4)

(((<<(lambda(f) (lambda(x)(lambda(y)(f x y)))),{}>> +) 3) 4)

((<<(lambda(x)(lambda(y)(f x y),{f->+}>> 3) 4)

((<<(lambda(y)(f x y), {f->+}{x ->3}>> 4)

(f x y) {f->+}, {x->3},{ y->4}

(+ 3 4)

7

Exemplo: mapc

- (define mapc (curry mapcar))
- (define soma1* (mapc soma1))
- (define soma1** (mapc soma1*))
- Confuso? vejamos:

Ksoma1 = <<(lambda(y) (+ x y)), {x->1}>>

Kmapcar = << (lambda (f l) (if (null? l) ....)),{}>>

Kcurry = <<(lambda(f)(lambda(x)(lambda(y)(f x y)))),{}>>

Kmapc = <<(lambda(x) (lambda(y)(f x y))),{f->Kmapcar}>>

Ksoma1* = <<(lambda (y) (f x y)),{f->Kmapcar}{ x->Ksoma1}>>

Exemplo: usando mapc

->(soma1* ´(3 4 5))

(f x y) {f->Kmapcar}{x->Ksoma1}{ y->´(3 4 5)}

(Kmapcar Kadd1 ´(3 4 5))

calculamos (if (null? l) ....)

em {f->Kadd1,l->´(3 4 5)}

...

Exemplo: incrementa-matriz

- Dada uma matriz, soma 1 a todos os elementos
- representação de matriz:
  lista de listas
- Primeiro usaremos *mapcar*

```
(define incrementa-matriz  (lambda (m)
```

Exemplo: incrementa-matriz

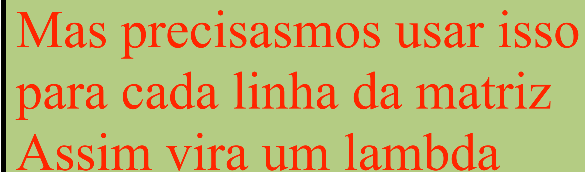
- Dada uma matriz, soma 1 a todos os elementos
- representação de matriz:
  lista de listas
- Primeiro usaremos *mapcar*

```
(define incrementa-matriz  (lambda (m)
                              (mapcar soma1 linha))
```

Exemplo: incrementa-matriz

- Dada uma matriz, soma 1 a todos os elementos
- representação de matriz:
  lista de listas
- Primeiro usaremos *mapcar*

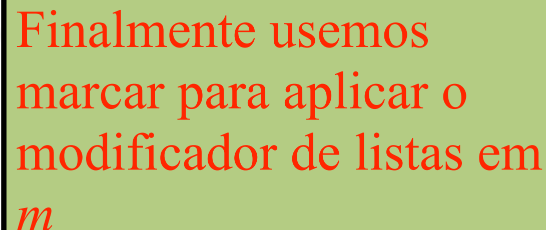
```
(define incrementa-matriz  (lambda (m)
        (lambda(linha) (mapcar soma1 linha))
```



Exemplo: incrementa-matriz

- Dada uma matriz, soma 1 a todos os elementos
- representação de matriz:
  lista de listas
- Primeiro usaremos *mapcar*

```
(define incrementa-matriz  (lambda (m)
    (mapcar (lambda(linha) (mapcar soma1 linha))
            m)))
```



Exemplo: incrementa-matriz

- Dada uma matriz, soma 1 a todos os elementos
- representação de matriz:
  lista de listas
- Primeiro usaremos *mapcar*

```
(define incrementa-matriz  (lambda (m)
    (mapcar (lambda(linha) (mapcar soma1 linha))
            m)))
```

- Poemos usar mapc para nos livrar de um lambda

Exemplo: incrementa-matriz

- Dada uma matriz, soma 1 a todos os elementos
- representação de matriz:
  lista de listas
- Primeiro usaremos *mapcar*

```
(define incrementa-matriz  (lambda (m)
    (mapcar (lambda(linha) (mapcar soma1 linha))
            m)))
```

- Poemos usar mapc para nos livrar de um lambda

```
(define incrementa-matriz (lambda (m)
    (mapcar (mapc soma1) m)
```

Exemplo: incrementa-matriz

- Dada uma matriz, soma 1 a todos os elementos
- representação de matriz:
  lista de listas
- Primeiro usaremos *mapcar*

```
(define incrementa-matriz  (lambda (m)
    (mapcar (lambda(linha) (mapcar soma1 linha))
            m)))
```

- Poemos usar mapc para nos livrar de um lambda

```
(define incrementa-matriz (lambda (m)
    (mapcar (mapc soma1) m)
```

- Ora, podemos então usar *,mapc* duas vezes

Exemplo: incrementa-matriz

- Dada uma matriz, soma 1 a todos os elementos
- representação de matriz:
  lista de listas
- Primeiro usaremos *mapcar*

```
(define incrementa-matriz  (lambda (m)
    (mapcar (lambda(linha) (mapcar soma1 linha))
            m)))
```

- Poemos usar mapc para nos livrar de um lambda

```
(define incrementa-matriz (lambda (m)
    (mapcar (mapc soma1) m)
```

- Ora, podemos então usar ,*mapc* duas vezes

```
(define incrementa-matriz  (mapc (mapc soma1)))
```

Exemplo: combine

- uma função muito útil é uma modificação de mapcar que aplica uma função a todos os elementos de uma lista e que combina os resultados utilizando uma função binária e um “zero”.
- assim (combine + id 0) seria o equivalente à somatória de uma lista.

Exemplo: combine

- uma função muito útil é uma modificação de mapcar que aplica uma função a todos os elementos de uma lista e que combina os resultados utilizando uma função binária e um “zero”.
- assim (combine + id 0) seria o equivalente à somatória de uma lista.
- Ex:

  ((combine + id 0) ‘(1 2 3 4 5 6)) => 21

  ((Combine * 2 1) ‘(1 2 3 4)) => ?

-

Exemplo: combine

- uma função muito útil é uma modificação de mapcar que aplica uma função a todos os elementos de uma lista e que combina os resultados utilizando uma função binária e um “zero”.
- assim (combine + id 0) seria o equivalente à somatória de uma lista.
- Ex:

  ((combine + id 0) ‘(1 2 3 4 5 6)) => 21

  ((Combine * 2 1) ‘(1 2 3 4)) => ?

- (define combine (lambda (combinador-binario   )

Exemplo: combine

- uma função muito útil é uma modificação de mapcar que aplica uma função a todos os elementos de uma lista e que combina os resultados utilizando uma função binária e um "zero".
- assim (combine + id 0) seria o equivalente à somatória de uma lista.
- Ex:

  ((combine + id 0) '(1 2 3 4 5 6)) => 21

  ((Combine * 2 1) '(1 2 3 4)) => ?

- (define combine (lambda (combinador-binario func-unaria)

Exemplo: combine

- uma função muito útil é uma modificação de mapcar que aplica uma função a todos os elementos de uma lista e que combina os resultados utilizando uma função binária e um "zero".
- assim (combine + id 0) seria o equivalente à somatória de uma lista.
- Ex:

  ((combine + id 0) '(1 2 3 4 5 6)) => 21

  ((Combine * 2 1) '(1 2 3 4)) => ?

- (define combine (lambda (combinador-binario func-unaria elemento-nulo)

Exemplo: combine

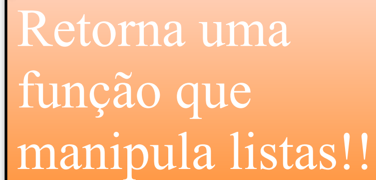
- uma função muito útil é uma modificação de mapcar que aplica uma função a todos os elementos de uma lista e que combina os resultados utilizando uma função binária e um "zero".
- assim (combine + id 0) seria o equivalente à somatória de uma lista.
- Ex:

  ((combine + id 0) '(1 2 3 4 5 6)) => 21

  ((Combine * 2 1) '(1 2 3 4)) => ?

- (define combine (lambda (combinador-binario func-unaria elemento-nulo)

  (lambda (lista)



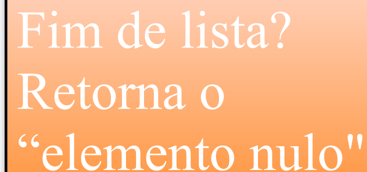
Exemplo: combine

- uma função muito útil é uma modificação de mapcar que aplica uma função a todos os elementos de uma lista e que combina os resultados utilizando uma função binária e um “zero”.
- assim (combine + id 0) seria o equivalente à somatória de uma lista.
- Ex:

  ((combine + id 0) ‘(1 2 3 4 5 6)) => 21

  ((Combine * 2 1) ‘(1 2 3 4)) => ?

```
- (define combine (lambda (combinador-binario func-unaria elemento-nulo)
      (lambda (lista)
          (if (null? lista) elemento-nulo
```



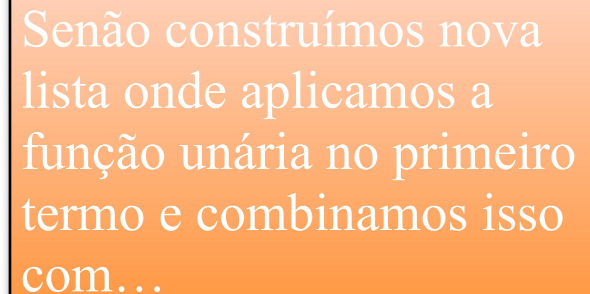
Exemplo: combine

- uma função muito útil é uma modificação de mapcar que aplica uma função a todos os elementos de uma lista e que combina os resultados utilizando uma função binária e um "zero".
- assim (combine + id 0) seria o equivalente à somatória de uma lista.
- Ex:

  ((combine + id 0) '(1 2 3 4 5 6)) => 21

  ((Combine * 2 1) '(1 2 3 4)) => ?

```
- (define combine (lambda (combinador-binario func-unaria elemento-nulo)
      (lambda (lista)
          (if (null? lista) elemento-nulo
              (combinador-binario  (func-unaria (car lista))
```



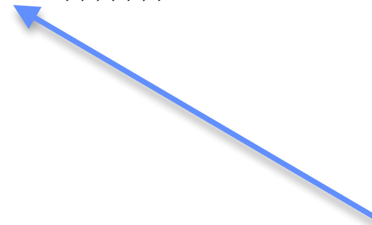
Exemplo: combine

- uma função muito útil é uma modificação de mapcar que aplica uma função a todos os elementos de uma lista e que combina os resultados utilizando uma função binária e um “zero”.
- assim (combine + id 0) seria o equivalente à somatória de uma lista.
- Ex:

  ((combine + id 0) ‘(1 2 3 4 5 6)) => 21

  ((Combine * 2 1) ‘(1 2 3 4)) => ?

```
- (define combine (lambda (combinador-binario func-unaria elemento-nulo)
      (lambda (lista)
          (if (null? lista) elemento-nulo
              (combinador-binario  (func-unaria (car lista))
                                  ((combine combinador-binario func-unaria elemento-nuloz)
                                    (cdr lista)))))))
```



Senão construímos nova lista onde aplicamos a função unária no primeiro termo e combinamos isso com…

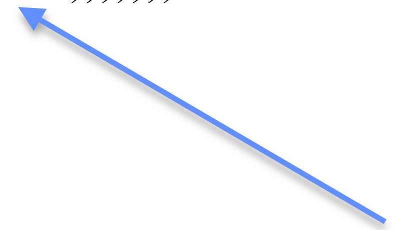
Exemplo: combine

- uma função muito útil é uma modificação de mapcar que aplica uma função a todos os elementos de uma lista e que combina os resultados utilizando uma função binária e um “zero”.
- assim (combine + id 0) seria o equivalente à somatória de uma lista.
- Ex:

  ((combine + id 0) ‘(1 2 3 4 5 6)) => 21

  ((Combine * 2 1) ‘(1 2 3 4)) => ?

```
- (define combine (lambda (combinador-binario func-unaria elemento-nulo)
      (lambda (lista)
          (if (null? lista) elemento-nulo
              (combinador-binario  (func-unaria (car lista))
                                   ((combine combinador-binario func-unaria elemento-nuloz)
                                    (cdr lista)))))))
```



CUIDADO!,
Como geramos uma função sem colocarmos seu valor em nenhum lugar, precisamos aplicar combine novamente!!!!

Exemplo: combine

- uma função muito útil é uma modificação de mapcar que aplica uma função a todos os elementos de uma lista e que combina os resultados utilizando uma função binária e um "zero".
- assim (combine + id 0) seria o equivalente à somatória de uma lista.
- Ex:

  ((combine + id 0) '(1 2 3 4 5 6)) => 21

  ((Combine * 2 1) '(1 2 3 4)) => ?

```
- (define combine (lambda (combinador-binario func-unaria elemento-nulo)
      (lambda (lista)
          (if (null? lista) elemento-nulo
              (combinador-binario  (func-unaria (car lista))
                                   ((combine combinador-binario func-unaria elemento-nuloz)
                                    (cdr lista)))))))
->(define id (lambda (x) x))
->(define produtoria (combine * id 1))
->(define somatoria (combine + id 0))
```

Exemplo: combine

- uma função muito útil é uma modificação de mapcar que aplica uma função a todos os elementos de uma lista e que combina os resultados utilizando uma função binária e um "zero".
- assim (combine + id 0) seria o equivalente à somatória de uma lista.
- Ex:

```
((combine + id 0) '(1 2 3 4 5 6))  => 21
((Combine * 2 1) '(1 2 3 4 )) => ?
```

```
- (define combine (lambda (combinador-binario func-unaria elemento-nulo)
      (lambda (lista)
          (if (null? lista) elemento-nulo
              (combinador-binario  (func-unaria (car lista))
                                   ((combine combinador-binario func-unaria elemento-nuloz)
                                    (cdr lista)))))))
->(define id (lambda (x) x))
->(define produtoria (combine * id 1))
->(define somatoria (combine + id 0))
->(define mapcar (lambda (f l) ( (combine cons f´()) l )
```

Um exemplo menos óbvio

Exemplo: find

- outra função útil, que dado um predicado *pred* acha se **algum** elemento de uma lista satisfaz a pred:

```
->(define find (lambda (pred lista)
     (if (null? lista) #f
          (if (pred (car lista)) #t
               (find pred (cdr lista))))))
```

Exemplo: ordens - 1

```
->(define compara-pares-de-pares (lambda (t1 t2)
     (if compara-pares (car t1) (car t2) #t
        (if (compara-pares (car t2) (car t1) #f
           (compara-pares (cadr t1) (cadr t2))))))
```

- função acima é o mesmo que estender o “<“ de compara-pares:

```
->(define ordem-lexicografica-pares  (lambda (<1 <2)
     (lambda (par1 par2)
        (if (<1 (car p1) (car p2)) #t
           (if (<1 (car p2) (car p1)) #f
              (<2 (cadr p1) (cadr p2)))))))
```

Exemplo: ordens -2

```
->(define compara-pares (ordem-lexicografica < <))
->(define compara-pares-de-pares
      (ordem-lexicografica compara-pares compara-pares)
```

- para que dois argumentos?
  - ordens diferentes para clientes diferentes:

```
->(define ordem-estudantes (ordem-lexicografica < >))
->(sort2 '(85 1005) '(95 20010) ordem-estudante)
  '((85 1005) (95 200010))
 ->(sort2 '(97 100) '(97 200) ordem-estudante)
  '((97 200) (95 100))
```

```
->(define inverte-ordem (lambda (<)
        (lambda (x y) (< y x))))
```

Exemplo: ordens - 3

- Vamos agora calcular ordens em registros mais complexos, baseados em apenas 2 campos (registros representados como listas):
- primeiro função para selecionar campos:

```
(define seleciona-2-colunas (lambda (num-col-1 num-col-2)
        (lambda (l)
            (list2 (n-esimo num-col-1 l) (n-esimo num-col-2 l)))))
```

- Agora uma função para composição

```
(define compoe-2-1 (lambda (f g)
        (lambda (x y) (f (g x) (g y)))))
```

- Finalmente a forma final

```
(define compara-colunas (lambda (num-col1 num-col2)
            (compoe-2-1 (ordem-lexicografica < <)
                            (seleciona-colunas num-col1
                                                num-col2))))
```

Exemplo: ordens - 4

- podemos fazer ordens mais gerais ainda, construindo novas funções e compondo com as antigas. Agora ordenaremos alunos baseados na melhoria nas notas:

```
(define compoe-1-1 (lambda (f g) (lambda (x) (f (g x)))))

(define aplica-em-dupla (lambda (f)
        (lambda (l) (f (car l) (cadr l)))))

(define melhora (compoe-1-1 (aplica-em-dupla -)
                                        (seleciona-colunas 3 2) ))

(define compara-melhora (compoe-2-1 > melhora))

(sort2 ´(Alan 1005 9 10) ´(Mitchell 1008 4 9) compara-melhora)
((Mitchell 1008 4 9) (Alan 1005 9 9))
```

Exemplo: conjuntos

- Podemos usar funções vistas agora para uma representar conjuntos de maneira mais concisa do que em Lisp:
- Conj-vazio não é alterado

->(define conj-vazio (lambda () ´()))

- A função “member?” poderia usar find….(sugestões?)

->

Exemplo: conjuntos

- Podemos usar funções vistas agora para uma representar conjuntos de maneira mais concisa do que em Lisp:
- Conj-vazio não é alterado

```
->(define conj-vazio (lambda () ´()))
```

- A função “member?” poderia usar find….

```
->(define member? (lambda (elem conj)
      (find                                  conj)))
```

Exemplo: conjuntos

- Podemos usar funções vistas agora para uma representar conjuntos de maneira mais concisa do que em Lisp:
- Conj-vazio não é alterado

```
->(define conj-vazio (lambda () ´()))
```

- A função “member?” poderia usar find….

```
->(define member? (lambda (elem conj)
        (find (lambda (x) (= x elem)) conj))) ;direto um fechamento
```

Exemplo: conjuntos

- Podemos usar funções vistas agora para uma representar conjuntos de maneira mais concisa do que em Lisp:
- Conj-vazio não é alterado

->(define conj-vazio (lambda () ´()))

- A função "member?" poderia usar find….

->(define member? (lambda (elem conj)

(find conj))); ou usando curry…

Exemplo: conjuntos

- Podemos usar funções vistas agora para uma representar conjuntos de maneira mais concisa do que em Lisp:
- Conj-vazio não é alterado

```
->(define conj-vazio (lambda () ´()))
```

- A função “member?” poderia usar find….

```
->(define member? (lambda (elem conj)
        (find ((curry =) elem) conj)));  ou usando curry…
```

Exemplo: conjuntos

- Podemos usar funções vistas agora para uma representar conjuntos de maneira mais concisa do que em Lisp:
- Conj-vazio não é alterado

```
->(define conj-vazio (lambda () ´()))
```

- A função "member?" poderia usar find….

```
->(define member? (lambda (elem conj)
        (find ((curry =) elem) conj))); ou usando curry…
```

- adiciona-elemento poderia usar combine, mas ficaria muito ineficiente (você consegue adivinhar porquê?)

```
->(define adiciona-elemento (lambda (elem conjunto)
        (if (member? elem conjunto) conjunto
           ((combine cons id (list elem)) conjunto ))))
```

Exemplo: conjuntos

- Podemos usar funções vistas agora para uma representar conjuntos de maneira mais concisa do que em Lisp:
- Conj-vazio não é alterado

```
->(define conj-vazio (lambda () ´()))
```

- A função "member?" poderia usar find….

```
->(define member? (lambda (elem conj)
        (find ((curry =) elem) conj)));  ou usando curry…
```

- Assim usaremos a definição original

```
->(define adiciona-elemento (lambda (elem conjunto)
        (if (member? elem conjunto) conjunto
            (cons elem conjunto))))
```

Exemplo: conjuntos

- Podemos usar funções vistas agora para uma representar conjuntos de maneira mais concisa do que em Lisp:
- Conj-vazio não é alterado

```
->(define conj-vazio (lambda () ´()))
```

- A função "member?" poderia usar find….

```
->(define member? (lambda (elem conj)
        (find ((curry =) elem) conj)));  ou usando curry…
```

- Assim usaremos a definição original

```
->(define adiciona-elemento (lambda (elem conjunto)
        (if (member? elem conjunto) conjunto
            (cons elem list))))
```

- Finalmente, podemos usar combine para implementar união

```
->(define uniao (lambda (conj1 conj2)
        ((combine                          ) conj1)))
```

Exemplo: conjuntos

- Podemos usar funções vistas agora para uma representar conjuntos de maneira mais concisa do que em Lisp:
- Conj-vazio não é alterado

```
->(define conj-vazio (lambda () ´()))
```

- A função "member?" poderia usar find….

```
->(define member? (lambda (elem conj)
        (find ((curry =) elem) conj))); ou usando curry…
```

- Assim usaremos a definição original

```
->(define adiciona-elemento (lambda (elem conjunto)
        (if (member? elem conjunto) conjunto
            (cons elem list))))
```

- Finalmente, podemos usar combine para implementar união

```
->(define uniao (lambda (conj1 conj2)
        ((combine  adiciona-elem id conj2) conj1)))
```

Exemplo: polimorfismo, 3 abordagens

- conjuntos são bons exemplos para pensarmos na questão do polimorfismo e de sua implementação
- podemos pensar em conjuntos com quaisquer tipos de elementos, porém nem sempre podemos pensar na função *equal?* como suficiente para identificar identidade
- um bom exemplo: conjuntos de listas de associações, se definirmos que duas listas são iguais quando associam as mesmas chaves aos mesmos valores independente da ordem em que foram adicionadas.
- veremos a seguir três maneiras de implementar polimorfismo em conjuntos

Exemplo: polimorfismo 1- Listas de associação

- uma lista de associação lista1é sub-lista de lista2 se não conseguimos encontrar um elemento de lista1 que não tenha o mesmo valor associado em lista1 e lista2. Em Scheme:

- duas listas de associação são iguais quando são sub-listas uma da outra

Exemplo: polimorfismo 1- Listas de associação

- uma lista de associação lista1é sub-lista de lista2 se não conseguimos encontrar um elemento de lista1 que não tenha o mesmo valor associado em lista1 e lista2. Em Scheme:

```
->( set sub-lista-assoc (lambda (lista1 lista2)
            (not (find


                 lista1))))
```

- duas listas de associação são iguais quando são sub-listas uma da outra

Exemplo: polimorfismo 1- Listas de associação

- uma lista de associação lista1é sub-lista de lista2 se não conseguimos encontrar um elemento de lista1 que não tenha o mesmo valor associado em lista1 e lista2. Em Scheme:

```
->( set sub-lista-assoc (lambda (lista1 lista2)
            (not (find  (lambda (par)
                    (not (equal?                    )))
                  lista1))))
```

- duas listas de associação são iguais quando são sub-listas uma da outra

Exemplo: polimorfismo 1- Listas de associação

- uma lista de associação lista1é sub-lista de lista2 se não conseguimos encontrar um elemento de lista1 que não tenha o mesmo valor associado em lista1 e lista2. Em Scheme:

```
->( set sub-lista-assoc (lambda (lista1 lista2)
            (not (find  (lambda (par)
                          (not (equal? (cadr par) (assoc (car par) lista2))))
                        lista1))))
```

- duas listas de associação são iguais quando são sub-listas uma da outra

Exemplo: polimorfismo 1- Listas de associação

- uma lista de associação lista1é sub-lista de lista2 se não conseguimos encontrar um elemento de lista1 que não tenha o mesmo valor associado em lista1 e lista2. Em Scheme:

```
->( set sub-lista-assoc (lambda (lista1 lista2)
            (not (find  (lambda (par)
                           (not (equal? (cadr par) (assoc (car par) lista2))))
                        lista1))))
```

- duas listas de associação são iguais quando são sub-listas uma da outra

```
->(define =lista-assoc (lambda (lista1 lista2)
        (and (sub-lista-assoc lista1 lista2)
             (sub-lista-assoc lista2 lista1) )))
```

Exemplo: polimorfismo 1 - Listas de associação -2

- Poderíamos reescrever a função *membro?* para utilizar *=lista-assoc*, porém isto implicaria em reescrever esta função para cada novo tipo de conjunto.
- Podemos em vez disso parametrizar as operações de conjunto pela operação de comparação

```
->(define membro? (lambda (elemento conjunto =)
  (if (null? conjunto) #f
     (if (= elemento (car conjunto)) #t
        (membro? elemento (cdr conjunto) =)))))
->(define adiciona
         (lambda (elemento conjunto =)
            (if (membro? elemento conjunto =) conjunto
               (cons elemento conjunto))))
```

Exemplo: polimorfismo 2

- a solução anterior tem a desvantagem de obrigar o usuário a sempre passar mais um argumento
- uma outra solução seria incluir a função de igualdade na representação:

->

Exemplo: polimorfismo 2

- a solução anterior tem a desvantagem de obrigar o usuário a sempre passar mais um argumento
- uma outra solução seria incluir a função de igualdade na representação:

```
->(define conj-nulo (lambda (=) (list = ´())))
```

Exemplo: polimorfismo 2

- a solução anterior tem a desvantagem de obrigar o usuário a sempre passar mais um argumento
- uma outra solução seria incluir a função de igualdade na representação:

```
->(define conj-nulo (lambda (=) (list = ´())))
->(define membro? (lambda (elemento conjunto)
      (find ( (curry (car conjunto)) elemento)
            (cadr conjunto))))
->
```

Exemplo: polimorfismo 2

- a solução anterior tem a desvantagem de obrigar o usuário a sempre passar mais um argumento
- uma outra solução seria incluir a função de igualdade na representação:

```
->(define conj-nulo (lambda (=) (list = ´())))
->(define membro? (lambda (elemento conjunto)
      (find ( (curry (car conjunto)) elemento)
            (cadr conjunto))))
-> define adiciona (lambda (elemento conjunto)
      (if (membro? elemento conjunto) conjunto
         (list2 (car conjunto)
                (cons elemento (cadr conjunto))))))
```

Exemplo: polimorfismo 2

- a solução anterior tem a desvantagem de obrigar o usuário a sempre passar mais um argumento
- uma outra solução seria incluir a função de igualdade na representação:

```
->(define conj-nulo (lambda (=) (list = ´())))
->(define membro? (lambda (elemento conjunto)
      (find ( (curry (car conjunto)) elemento)
            (cadr conjunto))))
-> define adiciona (lambda (elemento conjunto)
      (if (membro? elemento conjunto) conjunto
         (list2 (car conjunto)
                (cons elemento (cadr conjunto))))))
```

- Como ficaria a uniao? (exercício)

Exemplo: polimorfismo 3

- a versão anterior é mais fácil de utilizar, mas sobrecarrega a representação de conjuntos com um elemento a mais por conjunto
- uma solução intermediária é fazer com que o próprio Scheme produza as funções de manipulação de conjunto para cada tipo necessário

->

Exemplo: polimorfismo 3

- a versão anterior é mais fácil de utilizar, mas sobrecarrega a representação de conjuntos com um elemento a mais por conjunto
- uma solução intermediária é fazer com que o próprio Scheme produza as funções de manipulação de conjunto para cada tipo necessário

```
->(define cria-operacoes-de-conjuntos (lambda (=)
```

Exemplo: polimorfismo 3

- a versão anterior é mais fácil de utilizar, mas sobrecarrega a representação de conjuntos com um elemento a mais por conjunto
- uma solução intermediária é fazer com que o próprio Scheme produza as funções de manipulação de conjunto para cada tipo necessário

```
->(define cria-operacoes-de-conjuntos (lambda (=)
        (list (lambda () '()) ; ====>conjunto nulo
```

Exemplo: polimorfismo 3

- a versão anterior é mais fácil de utilizar, mas sobrecarrega a representação de conjuntos com um elemento a mais por conjunto
- uma solução intermediária é fazer com que o próprio Scheme produza as funções de manipulação de conjunto para cada tipo necessário

```
->(define cria-operacoes-de-conjuntos (lambda (=)
        (list (lambda () '()) ; ====>conjunto nulo
              (lambda (elemento conjunto)
                      (find ((curry =) elemento) conjunto));====>membro?
```

Exemplo: polimorfismo 3

- a versão anterior é mais fácil de utilizar, mas sobrecarrega a representação de conjuntos com um elemento a mais por conjunto
- uma solução intermediária é fazer com que o próprio Scheme produza as funções de manipulação de conjunto para cada tipo necessário

```
->(define cria-operacoes-de-conjuntos (lambda (=)
        (list (lambda () '()) ; ====>conjunto nulo
              (lambda (elemento conjunto)
                    (find ((curry =) elemento) conjunto));====>membro?
              (lambda (elemento conjunto)
                 (if (find ((curry =)elemento)conjunto) conjunto
                    (cons elemento conjunto)));===========>adiciona
 ... ))
```

Exemplo: polimorfismo 3 - 2

- Assim, para obtermos as operações de manipulação de um tipo basta utilizarmos a função acima, e criar um nome para cada nova operação:

->

Exemplo: polimorfismo 3 - 2

- Assim, para obtermos as operações de manipulação de um tipo basta utilizarmos a função acima, e criar um nome para cada nova operação:

```
->(define op-conjunto-lista-assoc
        (cria-operacoes-de-conjuntos =lista-assoc))
```

Exemplo: polimorfismo 3 - 2

- Assim, para obtermos as operações de manipulação de um tipo basta utilizarmos a função acima, e criar um nome para cada nova operação:

```
->(define op-conjunto-lista-assoc
        (cria-operacoes-de-conjuntos =lista-assoc))
->(define la-conj-vazio (car op-conjunto-lista-
   assoc))
```

Exemplo: polimorfismo 3 - 2

- Assim, para obtermos as operações de manipulação de um tipo basta utilizarmos a função acima, e criar um nome para cada nova operação:

```
->(define op-conjunto-lista-assoc
            (cria-operacoes-de-conjuntos =lista-assoc))
->(define la-conj-vazio (car op-conjunto-lista-
   assoc))
->(define la-membro? (cadr op-conjunto-lista-
   assoc))
```

Exemplo: polimorfismo 3 - 2

- Assim, para obtermos as operações de manipulação de um tipo basta utilizarmos a função acima, e criar um nome para cada nova operação:

```
->(define op-conjunto-lista-assoc
        (cria-operacoes-de-conjuntos =lista-assoc))
->(define la-conj-vazio (car op-conjunto-lista-
  assoc))
->(define la-membro? (cadr op-conjunto-lista-
  assoc))
->(define la-adiciona (caddr op-conjunto-lista-
  assoc
```

“Own variables” - variáveis próprias

- podemos utilizar a expressão lambda para criar variáveis locais que mantém estado de uma chamada de função para outra, como variáveis “static” em C.
- um bom exemplo: gerador de números pseudo-aleatórios:
  - (define rand (lambda (semente) (.....semente...)))
- neste formato qualquer função que utilizar *rand* precisa guardar o valor anterior para poder gerar o próximo. Isto implicaria potencialmente em adicionar um parâmetro a uma série de funções, violando o princípio de modularidade
- alternativa (ruim) seria utilizar variáveis globais

Variáveis próprias: implementação

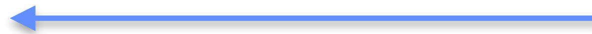
- podemos criar uma função geradora de funções *rand* que recebe como parâmetro o valor-inicial.
- criando um nível a mais de "lambda" podemos guardar o último valor utilizado em um argumento.
- Vejamos a implementação:

Variáveis próprias: implementação

- podemos criar uma função geradora de funções *rand* que recebe como parâmetro o valor-inicial.
- criando um nível a mais de "lambda" podemos guardar o último valor utilizado em um argumento.
- Vejamos a implementação:

```
(define init-rand (lambda (valor-inicial)
   (lambda () (


                    )))
```



Neste lambda temos
Um escopo com a variável
Valor-inicial

Variáveis próprias: implementação

- podemos criar uma função geradora de funções *rand* que recebe como parâmetro o valor-inicial.
- criando um nível a mais de "lambda" podemos guardar o último valor utilizado em um argumento.
- Vejamos a implementação:

```
(define init-rand (lambda (valor-inicial)
   (lambda () (begin
                 (set! valor-inicial
                     (remainder (+ (* valor-inicial 9) 5) 1024))
                 valor-inicial)))
```

Basta agora, a cada passo modificá-la para conter o novo valor

Variáveis próprias: implementação

- podemos criar uma função geradora de funções *rand* que recebe como parâmetro o valor-inicial.
- criando um nível a mais de “lambda” podemos guardar o último valor utilizado em um argumento.
- Vejamos a implementação:

```
(define init-rand (lambda (valor-inicial)
   (lambda () (begin
                 (set! valor-inicial
                    (remainder (+ (* valor-inicial 9) 5) 1024))
                 valor-inicial)))
->(define rand (init-rand 1))
->(rand)
14
->(rand)
```

Let, let*, letrec

- o scheme verdadeiro oferece uma série de recursos para definição de variáveis locais.
- são três construções sintáticas, uma para cada tipo de definição: let, let* e letrec.

Let

- formato: (let ((x_1 e_1) (x_2 e_2)...(x_N e_N)) exp)
- primeiramente calcula valores de e1,...eN no ambiente local rho, obtendo $v_1,...,v_N$
- em seguida calcula *exp* no ambiente:

  rho$\{x_1 \rightarrow v_1,...,x_N \rightarrow lv_N\}$
- assim temos que a expressão acima é equivalente a:

((lambda (x_1...x_N) exp) e_1 ... e_N)

Operador argumentos

Let*

- similar ao anterior, mas adiciona as associações sequencialmente:
- (let* ((x_1 e_1) (x_2 e_2)...(x_N e_N)) exp)
- calcula e_1 em rho obtendo v_1, cria rho_1 = rho{$x_1 \rightarrow v_1$}
- calcula e_2 em rho_1 obtendo v_2, cria rho_2 = rho1{$x_2 \rightarrow v_2$}
- ...
- calcula e_n em rho_{N-1} obtendo v_n, cria rho_N = rho_{N-1}{$x_N \rightarrow v_N$}
- cacula *exp* em rho_N
- i.é., é equivalente a:

((lambda(x_1)((lambda (x_2)(...<u>(lambda (x_N) exp)e_N)</u>...)e_2)e_1)

Let*

- similar ao anterior, mas adiciona as associações sequencialmente:
- (let* ((x_1 e_1) (x_2 e_2)...(x_N e_N)) exp)
- calcula e_1 em rho obtendo v_1, cria $rho_1 = rho\{x_1 \rightarrow v_1\}$
- calcula e_2 em rho_1 obtendo v_2, cria $rho_2 = rho1\{x_2 \rightarrow v_2\}$
- ...
- calcula e_n em rho_{N-1} obtendo v_n, cria $rho_N = rho_{N-1}\{x_N \rightarrow v_N\}$
- cacula *exp* em rho_N
- i.é., é equivalente a:

  ((lambda(x_1)((lambda (x_2)(...(lambda (x_N) exp)e_N)...)e_2)e_1)

chamadas encaixadas

Let*

- similar ao anterior, mas adiciona as associações sequencialmente:
- (let* ((x_1 e_1) (x_2 e_2)...(x_N e_N)) exp)
- calcula e_1 em rho obtendo v_1, cria rho_1 = rho{$x_1 \rightarrow v_1$}
- calcula e_2 em rho_1 obtendo v_2, cria rho_2 = rho1{$x_2 \rightarrow v_2$}
- ...
- calcula e_n em rho_{N-1} obtendo v_n, cria rho_N = rho_{N-1}{$x_N \rightarrow v_N$}
- cacula *exp* em rho_N
- i.é., é equivalente a:

((lambda(x_1)((lambda (x_2)(...(lambda (x_N) exp)e_N)...)e_2)e_1)

Cada uma com um argumento: a associação do parâmetro mais externo fica disponível às aplicações mais internas

Letrec

- utilizado para definir funções recursivas locais
- exemplo

```
->(letrec ( (contauns (lambda (lista)
                              (if (null? lista) 0
                                 (if (= (car lista) 1)
                                     (+ 1 (contauns (cdr lista)))
                                      (contauns (cdr lista))))))
          )
          (contauns ´(1 2 3 1 0 1 3 1 5)
 )
4
```

Letrec

- utilizado para definir funções recursivas localmente
- exemplo

```
->(letrec ( (contauns (lambda (lista)
                          (if (null? lista) 0
                              (if (= (car lista) 1)
                                  (+ 1 (contauns (cdr lista)))
                                  (contauns (cdr lista))))))
            )
            (contauns ´(1 2 3 1 0 1 3 1 5))
  )
4
```

Note que, com letrec, conseguimos fazer uma função local recursiva, o que não é possível com let ou let*

Letrec - 2

- Com letrec, podemos fazer nosso combine criar apenas um fechamento, e não um fechamento por aplicação recursiva:

Letrec - 2

- Com letrec, podemos fazer nosso combine criar apenas um fechamento, e não um fechamento por aplicação recursiva:
- forma antiga

```
->(define combine (lambda (soma f zero)
      (lambda (lista)
         (if (null? lista) zero
            (soma  (f (car lista)) ((combine soma f zero)(cdr lista)))))))
```

Letrec - 2

- Com letrec, podemos fazer nosso combine criar apenas um fechamento, e não um fechamento por aplicação recursiva:
- forma antiga

```
->(define combine (lambda (soma f zero)
      (lambda (lista)
         (if (null? lista) zero
            (soma  (f (car lista)) ((combine soma f zero)(cdr lista)))))))
```

- forma nova

```
->(define combine (lambda (soma f zero)
     (letrec ( (loop (lambda (lista)
                   (if (null? lista) zero
                      (soma (f (car lista)) (loop (cdr lista))))))
        loop))))
```

Cria função local recursiva: *loop*

Letrec - 2

- Com letrec, podemos fazer nosso combine criar apenas um fechamento, e não um fechamento por aplicação recursiva:
- forma antiga

```
->(define combine (lambda (soma f zero)
      (lambda (lista)
         (if (null? lista) zero
            (soma  (f (car lista)) ((combine soma f zero)(cdr lista)))))))
```

- forma nova

```
->(define combine (lambda (soma f zero)
     (letrec ( (loop (lambda (lista)
                    (if (null? lista) zero
                       (soma (f (car lista)) (loop (cdr lista))))))
          loop))))
```

Espressão de letrec é referência à variável local *loop*, cujo valor é um fechamento

Letrec - 2

- Com letrec, podemos fazer nosso combine criar apenas um fechamento, e não um fechamento por aplicação recursiva:
- forma antiga

```
->(define combine (lambda (soma f zero)
     (lambda (lista)
        (if (null? lista) zero
           (soma  (f (car lista)) ((combine soma f zero)(cdr lista)))))))
```

- forma nova

```
->(define combine (lambda (soma f zero)
     (letrec ( (loop (lambda (lista)
                    (if (null? lista) zero
                       (soma (f (car lista)) (loop (cdr lista))))))
           loop)))
```

Qual é o valor de *loop?*

Letrec - 2

- Com letrec, podemos fazer nosso combine criar apenas um fechamento, e não um fechamento por aplicação recursiva:
- forma antiga

```
->(define combine (lambda (soma f zero)
      (lambda (lista)
         (if (null? lista) zero
            (soma  (f (car lista)) ((combine soma f zero)(cdr lista)))))))
```

- forma nova

```
->(define combine (lambda (soma f zero)
     (letrec ( (loop (lambda (lista)
                     (if (null? lista) zero
                        (soma (f (car lista)) (loop (cdr lista))))))
          loop)))
```

Qual é o valor de *loop?*

<<(lambda (lista) ….); {loop-> }>>

É um fechamento com um ambiente local onde *loop* aponta para o próprio fechamento

Letrec - 3

- Na verdade letrec também pode ser açúcar sintático, veja como implementá-lo com *begin* e *let:*
- (letrec ((f e) exp))~ (let ((f ´())

```
                              (begin (set! f e)
                                     exp))
```

- ambiente para *(begin (define f e) exp))*?

Letrec - 4

- no nosso caso, *e* era um lambda, assim vira um fechamento:
- <<e, {f→'()}   >>

- o comando *(set! f e)* faz com que tenhamos:

Ke = <<e, {f→ Ke} >>

ou seja, o ambiente do fechamento aponta para o próprio fechamento

Continuações

- podemos utilizar o fato de que funções são valores para tratamento de situações que requerem controle de fluxo não usual
- ex: função *mdc**, que calcula mdc de uma lista de inteiros estritamente positivos, potencialmente longa, com probabilidade de conter o número 1 uma ou mais vezes.

Mdc* - versão 1

- Versão simples (supomos existência da função *mdc)*:

```
(define mdc* (lambda (lista)
    (if (= (car lista) 1) 1
        (if (null? (cdr lista)) (car lista)
            (mdc (car lista) (mdc* (cdr lista)))))))
```

Mdc* - versão 1

- Versão simples (supomos existência da função *mdc)*:

```
(define mdc* (lambda (lista)
    (if (= (car lista) 1) 1
        (if (null? (cdr lista)) (car lista)
            (mdc (car lista) (mdc* (cdr lista)))))))
```

- embora termine de percorrer a lista ao achar o primeiro número 1, ainda faz as comparações com os elementos anteriores da lista:

  ->(mdc* ´(3 7 9 15 1 3 8))

  chama *mdc* 4 vezes

Mdc* - versão2

- podemos tentar utilizar uma função auxiliar acumulando resultados parciais:

```
(define mdc*-aux (lambda (res-parcial lista)
      (if (null? lista) res-parcial
          (if (= (car lista) 1) 1
              (mdc*-aux  (mdc res-parcial (car lista))
                             (cdr lista))))))
```

Mdc* - versão2

- podemos tentar utilizar uma função auxiliar acumulando resultados parciais:

```
(define mdc-melhor* (lambda (lista )
    (if (=  (car lista) 1) 1
        (mdc*-aux (car lista) (cdr lista)))))
(define mdc*-aux (lambda (res-parcial lista)
    (if (null? lista) res-parcial
        (if (= (car lista) 1) 1
            (mdc*-aux  (mdc res-parcial (car lista))
                        (cdr lista))))))
```

Mdc* - versão2

- podemos tentar utilizar uma função auxiliar acumulando resultados parciais:

```
(define mdc-melhor* (lambda (lista )
    (if (=  (car lista) 1) 1
        (mdc*-aux (car lista) (cdr lista)))))
(define mdc*-aux (lambda (res-parcial lista)
    (if (null? lista) res-parcial
        (if (= (car lista) 1) 1
            (mdc*-aux  (mdc res-parcial (car lista))
                       (cdr lista))))))
```

- versão termina quando acha o primeiro um mas, novamente, pode ter computado vários mdc's

Mdc - versão 3 - 1

- gostaríamos de fazer o equivalente a:

```
function mdcestrela (lista: LISTPTR): integer;
label 99;
    function recmdcestrela(lista: LISTPTR): integer;
    begin
        if lista^.head = 1
          then goto 99
          else if lista^.tail = nil
                then recmdcestrela := lista^.head
                else recmdcestrela :=
                        mdc(lista^.head, recmdcestrela(lista^.tail))
    end;/*recmdcestrela*/
begin /*gcdestrela*/
   gcdestrela := 1;
   gcdestrela := recgcdestrela(lista);
99:
end;/*gcdestrela*/
```

Mdc - versão 3 - 1

- gostaríamos de fazer o equivalente a:

```
function mdcestrela (lista: LISTPTR): integer;
label 99;
    function recmdcestrela(lista: LISTPTR): integer;
    begin
        if lista^.head = 1
          then goto 99
          else if lista^.tail = nil
               then recmdcestrela := lista^.head
               else recmdcestrela :=
                        mdc(lista^.head, recmdcestrela(lista^.tail))
    end;/*recmdcestrela*/
begin /*gcdestrela*/
   gcdestrela := 1;
   gcdestrela := recgcdestrela(lista);
99:
end;/*gcdestrela*/
```

Chamada recursiva: nada é feito até acharmos a base da recursão

Mdc - versão 3 - 1

- gostaríamos de fazer o equivalente a:

```
function mdcestrela (lista: LISTPTR): integer;
label 99;
     function recmdcestrela(lista: LISTPTR): integer;
     begin
          if lista^.head = 1
            then goto 99
            else if lista^.tail = nil
                    then recmdcestrela := lista^.head
                    else recmdcestrela :=
                              mdc(lista^.head, recmdcestrela(lista^.tail))
     end;/*recmdcestrela*/
begin /*gcdestrela*/
    gcdestrela := 1;
    gcdestrela := recgcdestrela(lista);
99:
end;/*gcdestrela*/
```

Se acharmos um 1, o goto, no pascal, garante que as chamadas são todas canceladas e retornamos o valor 1 diretamente

O rótulo do *goto* fica após a chamada inicial

Mdc -versão 3 - 2

- ao invés de calcularmos o mdc, passamos uma função que “lembra” de calculá-lo: todos os cáculos são adiados até atingirmos o fim da lista
- idéia principal: sempre que chamamos mdc*-aux, se aplicarmos *resto-da-conta* ao mdc* de *lista*, obtemos o mdc da lista original
- assim, primeiro valor de *resto-da-conta* é a função identidade, já que mdc*-aux será aplicada a toda a lista original

Mdc-versão 3 - 3

- podemos fazer isso utilizando fechamentos, basta "adiarmos" o calculo dos mdc construindo um fechamento para executar este cálculo mais tarde:

```
(define mdc-otimo* (lambda (lista)
      (mdc*-aux2 (lista   id   )))
(define mdc*-aux2 (lambda (lista resto-da-conta)
     (if (= (car lista) 1) 1
         (if (null? (cdr lista)) ; acabou a lista,calculemos
             (resto-da-conta (car lista))
             (mdc*-aux2 (cdr lista)
                        (lambda (n) (resto-da-conta (mdc (car lista)n)))
)))))                   ;^^^^^^^^^^^->novo resto-da-conta
```

- note que os mdc só vão ser calculados quando encontramos o final da lista

Continuações

- o argumento *resto-da-conta* é chamado de continuação, pois encapsula o "futuro" do processo, o o que deve ser feito em seguida (daí seu nome)
- utilizando continuações podemos cirar vários "futuros" alternativos, assim, qualquer tipo de fluxo de controle pode ser implementado de maneira "elegante"

Exemplo2: mdc-s

- vamos tentar utilizar continuações para fazer algo ainda mais geral, um mdc para qualquer árvore, calculando o mdc dos átomos
- o difícil é associar a continuação à dupla recursáo (para o car e para o cdr)
- vamos facilitar, primeiro a versão "ineficiente"

```
(define mdc-s-inef (lambda (s-expr)
   (if (number? s-expr) s-expr
      (if (null? (cdr s-expr))
          (mdc-s (car s-expr))
         (mdc (mdc-s (car-expr)) (mdc-s-inef (cdr s-epr)))))))
```

```
(define mdc-s (lambda (s-expr) (mdc-s-aux s-expr id)))
(define mdc-s-aux (lambda (s-expr continuacao)
  (if (number? s-expr)
     (if (= s-expr 1) 1 (continuacao s-expr))
     (if (null? (cdr s-expr)) ; só tem car
        (mdc-s-aux  (car s-expr) continuacao)
        ;agora vem a parte difícil
        (mdc-s-aux
               (car s-expr)
               (lambda (n) (mdc-s-aux (cdr s-expr)
                                      (lambda (p) (continuacao (mdc p n))))

        ))))))
```

Call/cc

- (call with current continuation)
- scheme possui uma função primitiva especial que permite ao programador utilizar a continuação atual do interpretador
- continuação atual:
  - o que o "eval" pretende fazer com o valor da expressão que está sendo calculada
  - a continuação é uma função de 1 argumento que usa o valor da expressão atual para fornecer uma resposta final
- exemplo: (+ 3 4)
  - "+","3" e "4" são expressões
  - + : continuação é (lambda (f) (f 3 4)) pois eval pretende aplicar + a 3 e 4.
  - 3 : continuação é (lambda (x) (+ x 4)) pois eval vai somar valor de 3 a 4
  - 4 : continuação é (lambda (x) (+ 3 x)) pois eveal vai somar valor de 4 a 3

Call/cc - 2

- call/cc é uma função de um argumento a qual, por sua vez também deve ser uma função
- call/cc aplica esta função passando como argumento a continuação atual
- ex:

```
->(define f (lambda (k) (k 5)))
->(+ (call/cc f) 4)
9
```

- razão:
  - continuação de call/cc era (lambda (x) (+ x 4))
  - (call/cc f) fica (f (lambda (x) (+ x 4)))

Call/cc -final

- com call/cc podemos “limpar” nossa versão de mdc-*, veja:

```
(define mdc-callcc* (lambda (lista)
  (call/cc (lambda (exit) ; argumento aqui é  o ponto de saída de mdc*
    (letrec ( (mdc*-aux
                (lambda (lista)
                  (if (= (car lista) 1) (exit 1)
                  (if (null? (cdr lista))  (car lista))
                  (mdc (car lista) (mdc*-aux (cdr lista)))))))
              (mdc*-aux lista)))))
```

Um pouco de mundo real: efeitos colaterais

- No modelo funcional puro não existem operações de modificação, apenas de construção
- Assim, se queremos modificar uma lista, precisamos replicar o seu início, criar uma nova célula com a posição modificada, e reutilizar o prefixo inalterado da lista antiga na lista nova
- Isso potencialmente aumenta muito o uso de memória de um programa
- Além disso as vezes gostaríamos de modelar com precisão estruturas mutantes.
  - Quanto mais nos afastamos de uma programação com fundamento matemático sólido, mas tendemos a ver nosso programa como um conjunto de dados que é manipulado e não como um conjunto de funções que transformam valores

Um pouco de mundo real: efeitos colaterais

- Na verdade, como sabemos, o computador em sí é uma máquina com estado
- Nele as posições de memória são alteráveis.
- No mundo real todas as linguagens oferecem operações de alteração de memória
- A grande questão é tentar reduzir estas ao máximo, utilizando apenas os casos onde a eficiência é fundamental ou onde isso torna o programa mais claro, ou encapsular estas mudanças em uma interface de funções que discipline os efeitos colaterais, garantindo a integridade dos dados

Modelando efeitos colaterais

- Como veremos em nosso interpretador, existem dois tipos de mudanças que podemos modelar
  - Mudança em *campos* de um registro
  - Mudança em valor de uma variável
- Esta noção é importante para entendermos o texto do interpretador.
- Utilizaremos dois conceitos, um para cada um dos tipos de efeitos colaterais
  - *Box* para modelarmos *campos*
  - *Storage* (ou memória) para modelarmos mudança de valores.
- IMPORTANTE: com efeitos colaterais, valores associados a identificadores mudam durante a execução de uma função. Assim é necessário que possamos definir ordem de execução de um conjunto de expressões
- Precisamos de um BEGIN
- Ele pode ser apenas açúcar sintático utilizando um let

Modelando efeitos colaterais: *boxes*

- Para simplificar, no caso de estruturas, vamos modelar estruturas com apenas um campo (mais campos é apenas uma generalização)
- Assim introduzimos um novo tipo, *box* com as seguintes operações
  - Box: criação de um campo mutável armazenando um valor determinado
  - Unbox: recuperação do valor encapsulado no campo
  - Set-box!: mudança no valor encapsulado no box.
- Assim em nosso interpretador teremos expressões equivalentes
  - boxC
  - unboxC
  - setboxC
  - seqC (não podemos esquecer sequências de operações)