Programação Funcional – Parte 3

PLP 2019/1
Profa. Heloisa

Funções de ordem superior

- Funções que recebem outras funções como argumento
- Uma função pode ser passada como argumento para outra pelo seu nome, com o operador especial *function*

(function <nome_de_função>)

- Essa forma pode ser abreviada por

‘ <nome_de_função>

- Exemplos:
- (function quadrado) equivale a # ‘ quadrado

Funções de ordem superior

- Funções que recebem outras funções como argumento
- MAPCAR – aplica uma dada função repetidamente aos argumentos dados na forma de lista (uma para cada argumento)

(mapcar <nome da função> <lista de argumentos 1> <lista de argumentos 2> ...)

```
(defun soma-um (x)   (+ x 1))
SOMA-UM
> (mapcar #'soma-um '(1 4 7 3))
(2 5 8 4)
```


- Se a função tem mais de um argumento, mapcar deve ter tantas listas quantos forem os argumentos da função

```
> (mapcar # '+ '(1 2 3) '(3 4 1))
(4 6 4)

> (mapcar # 'equal  '(1 2 3) '(3 2 1))
(NIL T NIL)

> (mapcar # ' list '(a b c) '(3 4 5))
((A 3) (B 4) (C 5))
```

- Outros exemplos:

```
> (mapcar # ‘abs ‘(1 -2 3 -5 3 4 -1))
(1 2 3 5 3 4 1)

> (defun par-dobro (x) (list x (+ x x)))
PAR-DOBRO

> (mapcar # ‘par-dobro ‘(2 5 3))
((2 4) (5 10) (3 6))
```



```
>(defun monta-pares (L1 L2)
      (cond ((null L1) nil)
            (t (cons (list (car L1) (car L2))
                     (monta-pares (cdr L1) (cdr L2))))))

>(mapcar #  'monta-pares '((a b c) (1 2)) '((3 4 5) (3 4)))

(((A 3) (B 4) (C 5)) ((1 3) (2 4)))
```

- Apply – aplica uma função a uma lista de argumentos

(apply <nome da função> <lista de argumentos>)

```
> (apply # '+ (1 2 3 4))
10

> (defun CONTA-ATOMOS (L)
    (cond ((null L) 0)
          (( atom L) 1)
          (T (apply  # '+ (mapcar  #'CONTA-ATOMOS L))))))
```


Recursão X Iteração

- As repetições em um programa Lisp podem ocorrer por meio de recursão ou iteração.

- Alguns problemas são mais naturalmente resolvidos usando recursão, outros por iteração.
- Usando recursão, o programa fica mais declarativo
- Usando iteração, o programa fica mais imperativo

- As principais construções de Lisp para iteração são:
  - Do
  - Dolist
  - Dotimes
  - Mapcar

Exemplo: Recursão ou Iteração?

- A escolha da recursão ou da interação deve considerar o balanceamento entre a facilidade da construção da solução e a eficiência.

- Construir uma função Lisp que, calcula a função de Fibonacci, definida por:

f(0) = 1
f(1) = 1
f(n) = f(n-1) + f(n-2)

Fibonacci usando recursão

```
(defun FIBONACCI (N)
   (cond ((zerop N) 1)
         ((equal N 1) 1)
         (t (plus (FIBONACCI (- N 1))
                  (FIBONACCI (- N 2))))))
```

Essa função é naturalmente recursiva, mas a solução é bastante ineficiente, por exigir cálculos repetidos.

Fibonacci usando iteração

```
(defun fibonacci-iter (N)
   (cond ((equal N 0) 1)
         ((equal N 1) 1)
         (t (do ((N1 2 (+ N1 1)) (R-ant 1) (R 2))
                ((equal N1 N) R)
                (setq AUX (+ R-ant  R))
                (setq R-ant  R)
                (setq R AUX)))))
```

A função fibonacci pode ser implementada usando recursos de iteração (do, combinado com setq). Essa implementação é mais difícil de ser compreendida, pois não reflete a natureza recursiva da definição da função mas é mais eficiente em termos de tempo de execução.

Recursividade na cauda

- Recursividade na cauda: uma função é recursiva na cauda se os valores retornados por essa função não são alterados no nível anterior
- Recursividade na cauda é mais eficiente que recursividade geral
- Conta-atomos e Fibonacci NÃO são recursivas na cauda, portanto, são programas com custo computacional alto.

Versões recursivas X iterativas

- Problema: dados dois elementos e uma lista, sustituir todas as ocorrências do primeiro elemento pelo segundo, no primeiro nível da lista.

Versão recursiva:

```
>(defun SUBSTITUI (E1 E2 LISTA)
   (cond ((null Lista) NIL)
         ((equal (car Lista) E1) (cons E2 (SUBSTITUI E1 E2 (cdr Lista))))
         (t (cons (car Lista) (SUBSTITUI E1 E2 (cdr Lista))))))
SUBSTITUI
>(substitui 'a 'b '(d a c (b a) a x))
(D B C (B A) B X)
```


Versões recursivas X iterativas

- Problema: dados dois elementos e uma lista, sustituir todas as ocorrências do primeiro elemento pelo segundo, no primeiro nível da lista.

Versão iterativa:

```
>(defun SUBST-ITER (E1 E2 LISTA)
     (DO ((R NIL) (L LISTA (cdr L))
          ((null L) R)
          (IF (equal (car L) E1)
                    (setq R (append R (list E2)))
                    (setq R (append R (list (car L)))))
       ))
```

Versões recursivas X iterativas

- Problema: dados um elemento e uma lista, retirar todas as ocorrências do elemento no primeiro nível da lista.

Versão recursiva:

```
>(defun APAGA (ELEM Lista)
          (cond ((null Lista) nil)
                    ((equal ELEM (car Lista)) (APAGA ELEM (cdr Lista)))
                     (t (cons (car Lista) (APAGA ELEM (cdr Lista))))))
APAGA
>(apaga 'a  '(d a c (b a) a x))
(D C (B A) X)
```


Versões recursivas X iterativas

- Problema: dados um elemento e uma lista, retira todas as ocorrências do elemento, no primeiro nível da lista.

Versão iterativa:

```
(defun tira_elem (lista e)
   (do ((l-aux lista (cdr l-aux)) (res ( )))
         ((null l-aux) res)
          (if (not (equal (car l-aux) e))
             (setq res (append res (list((car l-aux)) )))))
TIRA_ELEM
>(tira_elem  '(d a c (b a) a x) 'a)
(D C (B A) X)
```

Escopo léxico X escopo dinâmico

- Common Lisp tem dois tipos de variáveis:
  léxicas e especiais
- As variáveis léxicas são definidas:
  - Por alguma construção sintática (let ou defun) e podem ser referenciadas pelo código que aparece dentro dessa construção.
  - Por setq e podem ser referenciadas em qualquer lugar, desde que não tenham sido redefinidas (dependendo da implementação)
- Variáveis léxicas tem escopo léxico – definido de acordo com o texto do código do programa

Exemplo – variável léxica definida por let

```
(let ((x 1))          ; Define uma ligação para x
    (print x)
    (let ((x 2))      ; Define uma segunda ligação para x
        (print x)
        (setq x 3) ; Muda o valor da segunda ligação
        (print x))
     (print x))       ; O valor da primeira ligação não se altera
```

A saída será: 1 2 3 1

O SETQ muda o valor da segunda ligação (interna) de x, sem alterar a primeira (externa)

Exemplo – variável léxica definida por setq

```
> (setq var-lex 5)
5
> (defun verifica-lex ( ) var-lex)
VERIFICA-LEX

> (verifica-lex)
5
> (let ((var-lex 6)) (verifica-lex))
5
> (let ((var-lex 6)) (print var-lex) (verifica-lex))
6
5
```


Arquivo variaveis-lexicas.lisp

```
(defun foo( )
(let ((x 1) (y 2)) ; define ligações para x e y
      (frotz x y) ;    referências as ligações
      (let ((y 5) (z 3)) ; define outra ligação para y e uma para z
            (baz x y z)) ; referências a x (primeira) y (segunda) e z.
                         ; a primeira ligação de y não pode ser
                           ; acessada aqui.
                           ; ela foi "sombreada" pela segunda ligação
            (print "Fora de baz o valor de z eh ") z)) ;       referência a
      variável livre z
(defun frotz (x y)
  (print "Estou no escopo de frotz")
  (print "x= ") (prin1 x) (prin1 " y= ") (prin1 y)
)
(defun baz (x y z)
  (print "Aqui em baz o valor de x eh = ") (prin1 x) (terpri)
  (print "Aqui em baz o valor de y eh = ") (prin1 y) (terpri)
  (print "Aqui em baz o valor de z eh = ") (prin1 z) (terpri)
)
```

Como carregar arquivos com programas lisp

- Se o código anterior estiver no arquivo chamado variaveis-lexicas.lisp no diretório D:\\Projetos Lisp

- Usar a função load:

> (load "D:/Projetos Lisp/variaveis-lexicas.lisp")


```
[1]> (load "D:/Projetos Lisp/variaveis-lexicas.lisp")

;; Loading file D:\Projetos Lisp\variaveis-lexicas.lisp ...
;; Loaded file D:\Projetos Lisp\variaveis-lexicas.lisp
T
[2]> (setq z 100)

100
[3]> (foo)

"Estou no escopo de frotz"
"x= " 1" y= "2
"Aqui em baz o valor de x eh = " 1

"Aqui em baz o valor de y eh = " 5

"Aqui em baz o valor de z eh = " 3

"Fora de baz o valor de z eh "
100
[4]>
```

```
(defun foo( )
(let ((x 1) (y 2)) ; define ligações para x e y
     (frotz x y) ;    referências as ligações
     (let ((y 5) (z 3)) ; define outra ligação para y e uma para z
          (baz x y z)) ; referências a x (primeira) y (segunda) e z.
                       ; a primeira ligação de y não pode ser
                         ; acessada aqui.
                         ; ela foi "sombreada" pela segunda ligação
          (print "Fora de baz o valor de z eh ") z)) ;       referência a
variável livre z
```

```
[2]> (setq z 100)

100
[3]> (foo)

"Estou no escopo de frotz"
"x= " 1" y= "2
"Aqui em baz o valor de x eh = " 1

"Aqui em baz o valor de y eh = " 5

"Aqui em baz o valor de z eh = " 3

"Fora de baz o valor de z eh "
100
[4]>
```


Variáveis especiais

- As variáveis especiais são definidas por DEFVAR
- Por convenção, são escritas entre *

> (defvar *x*)

- Variáveis especiais tem escopo dinâmico – definido pela sequência de chamadas, ou da execução do programa

Exemplo – variável especial definida por defvar

```
> (defvar var-lex 5)
VAR-LEX

> (defun verifica-lex ( ) var-lex)
VERIFICA-LEX

> (verifica-lex)
5
> (let ((var-lex 6)) (verifica-lex))
6
> (verifica-lex)
5
(let ((var-lex 6)) (print var-lex) (verifica-lex))
6
6
```



```
>(defvar x 1)
x

>(defun baz () x)
baz

> (defun foo ()
        (let ((x 2))
             (baz)))
foo

> (foo)
2

> (baz)
1
```

Função symbol-value

- A função symbol-value retorna o valor de um símbolo
- Não permite acessar o valor da ligação local de variáveis léxicas.
- Permite acessar o valor da ligação dinâmica local de uma variável especial

```
> (setq a `global-a)
GLOBAL-A
> (defvar *b* `global-b)
*B*
(defun fn ( ) *b*)
FN
> (let ((a `local-a) (*b* `local-b))
   (list a *b* (fn) (symbol-value `a) (symbol-value `*b*)))
(local-a local-b local-b global-a local-b)
```



```
>(setq a 1)
1
>(defvar  *b* 1)
*B*
(symbol-value 'a)
 1
;; SYMBOL-VALUE cannot see lexical variables.
(let ((a 2)) (symbol-value 'a))
1
>(let ((a 2)) (setq a 3) (symbol-value 'a))
 1
 ;; SYMBOL-VALUE can see dynamic variables.
>(let ((*b* 2)) (symbol-value '*b*))
2
>(symbol-value *b*)
1
```

Paradigma Funcional – Conclusão
Revisão dos principais conceitos

- Usa funções matemáticas
- Processamento Simbólico
  usa símbolos e conceitos ao invés de números e expressões

  Números: 3 5.1 -4.67

  Símbolos: A X Nome Substitui

  Listas: (A B C)

Paradigma Funcional – Conclusão
Revisão dos principais conceitos

- Declarativo
  - Foco da programação: especificar O QUE deve ser feito, sem detalhes de operações da arquitetura da máquina
  - Programas são tipicamente especificações de funções

```
(defun filtra-negativos (lista-num)
 (cond ((null lista-num) nil)
       ((plusp (car lista-num))
            (cons (car lista-num) (filtra-negativos (cdr lista-num))))
       (t (filtra-negativos (cdr lista-num)))))
```

Paradigma Funcional – Conclusão
Revisão dos principais conceitos

- Estrutura principal: listas

(a b c)
(X Y 25 (3 Z Y))

- São baseadas em funções

```
(defun valor-absoluto (x)
  (cond ((< x 0) (-x))
        ((>= x 0) x)))
```


Paradigma Funcional – Conclusão
Revisão dos principais conceitos

- A ordem de especificação das funções não é fundamental
- A ordem de execução é determinada na própria execução

```
(defun INTER (L1 L2)
        (cond ((null L1) NIL)
              ((null L2) NIL)
               ((member (car L1) L2) (cons (car L1) (INTER (cdr L1) L2)))))
               ( t (INTER (cdr L1) L2))))
(defun ELE-COMUM (L1 L2)
              (NOT (NULL (inter L1 L2))))
```

```
(ele-comum '(a b c d) '(f b t a x))

T
```

Paradigma Funcional

- Originalmente interpretado

  Permite interação direta do usuário com o interpretador e permite que funções sejam passadas como argumentos de outras funções

- Área de aplicação: IA

  - Processamento simbólico é adequado para representar conhecimento