Programação Funcional – Parte 2

PLP 2019/1
Profa. Heloisa

Controle de fluxo em Lisp

- Os desvios em Lisp também são baseados em avaliações de funções, com o auxílio dos predicados
- Função **cond** – implementa desvio condicional
- Argumentos: pares do tipo condição-ação (qualquer número)

(**cond** (<condição1> <ação1>)
(<condição2> <ação2>)
....
(<condiçãon> <açãon>))

- Condições e ações podem ser s-expressões com cada par entre parênteses
- Cond não avalia todos os argumentos
- Avalia as condições até que uma retorne valor diferente de nil
- Quando isso acontece, avalia a expressão associada à condição e retorna esse resultado como valor da expressão cond
- Nenhuma das outras condições ou ações são avaliadas
- Se todas as condições são avaliadas como nil, cond retorna nil

Definição de funções usando cond

- Lisp tem uma função pré-definida para cálculo do valor absoluto de um número: abs
- Vamos redefinir essa função para ilustrar o uso de cond

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

valor-absoluto

> (valor-absoluto -5)
5
```

Ação default

- Definição alternativa para valor-absoluto:

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

- Nessa versão, a última condição (>= x 0) é substituída por t, pois é sempre verdadeira se a primeira for falsa
- Quando existem mais de dois pares de condição-ação, o uso de t na última condição serve par forçar a execução de alguma ação, quando todas as outras são falsas (valor nil)

Programas usando cond

```
(defun membro (elemento lista)
    (cond ((null lista)  nil)
              ((equal elemento (car lista)) lista)
              (t   (membro elemento (cdr lista)))))

> (membro 5 ´(1 4 2 8 5 3 6))
(5 3 6)
> (membro 5 ´(a b c d e))
nil
```

```
(defun comprimento (lista)
  (cond ((null lista) 0)
        (t  (+ (comprimento (cdr lista)) 1))))

(defun enesimo (n lista)
  (cond ((zerop n) (car lista))
        (t  (enesimo (- n 1) (cdr lista)))))
```


Função para eliminar os números negativos de uma lista de números

```
(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)))))

> (filtra-negativos ´(1 -1 3 4 -5 -2))
(1 3 4)
```

plusp – predicado que retorna t se seu argumento for positivo e nil se não for

Função para concatenar duas listas

```
(defun concatena (lista1 lista2)
  (cond ((null lista1) lista2)
        (t (cons (car lista1) (concatena (cdr lista1) lista2)))))

> (concatena ´(a b c) ´(1 2 3))
(a b c 1 2 3)

>(concatena ´(a (b) c d) ´((1 2 3) ((4))))
(a (b) c d (1 2 3) ((4)))
```


Função pra contar o número de átomos da lista, inclusive das sublistas

```
(defun conta-atomos (lista)
  (cond ((null lista) 0)
        ((atom lista) 1)
        (t (+ (conta-atomos (car lista))
              (conta-atomos (cdr lista))))))

> (conta-atomos ´((1 2) 3 (((4 5 (6))))))
6
```

Conectivos lógicos

- Not – recebe um argumento e retorna t se ele for nil e nil caso contrário
- And – recebe qualquer número de argumentos. Avalia da esquerda para direita, para quando encontrar um avaliado como *nil* ou quando avaliar todos. Retorna o valor do último avaliado.

```
> (and (member ´a ´(b c)) (+ 3 1))
nil
>(and (member ´a ´(a b c)) (+ 3 1))
4
```


- Or – recebe qualquer número de argumentos. Avalia da esquerda para direita somente até que um deles seja não *nil* e retorna esse valor.

```
> (or (member ´a ´(c a b)) (oddp 2))
(a b)

> (or (> 1 2) (< 3 4) (= 5 5))
t
```

oddp – predicado que retorna t se seu argumento for ímpar e nil se não for

Outras formas condicionais - IF

(if *condição expr-then* [*expr-else*])

A condição é calculada e:

se for não nula *expr-then* é calculada, seu valor retornado

se for nula *expr-else* é calculada, seu valor retornado
expr-else é opcional

tanto *expr-then* como *expr-else* devem ser expressões simples.

> (setq A 10 B 20)
20
> (if (> A B) A B)
20

Outras formas condicionais - IF

> (setq L1 ‘(x y z))
(x y z)

>(if (not (null L1)) (car L1) “lista nula”)
X

>(setq A 10)
10

>(if (> a 20) “A maior que 20” “A menor que 20”)
“A menor que 20”

PROGN

PROGN

- executa qualquer número de expressões na seqüência e retorna o valor da última.
- Combinado com IF, permite que mais de uma expressão seja executada

(PROGN <expr-1> <expr-2> ..... <expr-n>)

> (IF (listp L1) (car L1)
(PROGN (…….) (…….) (…….)))

> (IF (not *condição*) (PROGN *x* *y*))

Macros padrão: WHEN e UNLESS

- As macros em LISP permitem definir formas sintáticas que simplificam formas combinadas.
- As macros WHEN e UNLESS simplificam a combinação IF + PROGN.

- WHEN – calcula a condição e, se for não nula, executa as expressões
- (when *condição expr-1 expr-2 … expr-n*)

```
>(setq I 3 R 10 Parcela 2)
2
> (when (> I 0) (setq R (+ R Parcela)) (setq I (+ I 1)) (print R))
12
12
```


- UNLESS – calcula a condição e, se for nula, executa as expressões
- (unless *condição expr-1 expr-2 … expr-n*)

```
>(unless (= I 0) (setq R (+ R Parcela)) (setq I (+ I 1)) (print R))

14
14
```

Variáveis livres e ligadas

```
(defun incremento (parametro)
      (setq parametro (+ parametro livre))
      (setq saida parametro))
incremento
```

Parametro é ligada com relação à função porque aparece na lista de parâmetros. Recebe um valor na entrada mas seu valor anterior é restaurado na saída da função

Livre é uma variável livre com relação à função porque não aparece na lista de parâmetros


```
>(setq parametro 15)
15
>(setq livre 10)
10
>(setq saida 10)
10
>(setq argumento 10)
10
>(incremento argumento)
20
>saida
20
>parametro
15
>argumento
10
```

```
(defun incremento (parametro)
      (setq parametro (+ parametro livre))
      (setq saida parametro))
incremento
```

Variáveis locais com let

- Let controla a ligação de variáveis
  (let (<variáveis-locais>) <expressões>)

Variáveis-locais são átomos simbólicos ou pares da forma
(<símbolo> <expressão>)

```
>(setq a 0)
0
>(let ((a 3) b)
       (setq b 4)
        (+ a b))
7
>a
0
>b
Error – b is not bound at top level
```


Entrada e Saída

- READ – função que não tem parâmetros. Quando é avaliada, retorna a próxima expressão inserida no teclado.

```
>(setq L1 (read) L2 (read))
(a b c d)                ;digitado
(e f g)                  ;digitado
(e f g)                  ;valor retornado

> (append L1 L2)
(a b c d e f g)
```

- PRINT – função que recebe um argumento, avalia e imprime esse resultado na saída padrão. Começa uma nova linha antes de imprimir e termina com um espaço branco
- PRIN1 – o mesmo que PRINT, sem começar uma nova linha
- PPRINT – o mesmo que PRINT, omitindo a duplicação de valores na saída.


```
> (progn (print 'a) (print 'b))

A
B
B
> (progn (prin1 'a) (prin1 'b))
AB
B
> (progn (print 'a) (prin1 'b))

A B
B
```

> (progn (pprint 'a) (pprint 'b))

A
B
> (print 5.1)

5.1
5.1
> (pprint 5.1)

5.1


```
> (defun concatena ( L1 L2)
        (print “A lista resultante e “)
        (append L1 L2))
concatena

> (concatena L1 L2)

A lista resultante e
(a b c d e f g)
```

- TERPRI – função sem argumentos que insere um caracter newline na saída padrão

```
> (defun concatena ( L1 L2)
(print “A lista resultante e “) (terpri)
(append L1 L2))
concatena

>(concatena L1 L2)

A lista resultante e

(a b c d e f g)
```


Iteração - Loop

- Permite executar algumas expressões repetidamente até que um return seja encontrado

```
>(setq a 10)
10
>(loop
    (setq a (+ a 1))
    (write a)
    (terpri)
    (when (> a 17) (return a))
)
```

Saída:
11
12
13
14
15
16
17
18
18

Macros padrão: DOLIST e DOTIMES

- São formas especiais para repetição adequadas a situações mais simples, nas quais não é necessário usar todos os recursos da forma DO.

Dolist

(DOLIST (*variável lista resultado-opcional*) *corpo*)

O *corpo* do loop é executado uma vez para cada valor de *variável*, que assume valores de *lista*.

No final DOLIST retorna o valor da expressão resultado-opcional, caso ela apareça, senão retorna NIL.

```
>(dolist (x '(1  2  3)) (print x))

1
2
3
nil
```

Dolist

```
>(dolist (x '(1  2  3)) (print x) (if (evenp x) (return)))

1
2
nil

>(setq L '(1 2 3))
(1 2 3)

>(dolist (x L 'fim ) (print x) )

1
2
3
FIM
```


Exemplo
(função sem argumentos, dolist e prin1)

```
(defun test ()
   (dolist (truth-value '(t nil 1 (a b c)))
     (if truth-value (print 'true) (print 'false))
     (prin1 truth-value)))

> (test)
TRUE T
FALSE NIL
TRUE 1
TRUE (A B C)
NIL                    ;resultado final retornado por Dolist
```

Dotimes

(DOTIMES (*variável numero resultado-opcional*) *corpo*)

O *corpo* do loop é executado uma vez para cada valor de *variável*, que assume valor inicial 0 e é incrementada até o valor de *número - 1* .

(dotimes (i 4) (print i))

0
1
2
3
nil

Dotimes

>(dotimes (i 4 ‘acabou) (print i))

0
1
2
3
ACABOU

Do

Forma repetitiva geral

(DO ((*variável v-inicial próximo) ...)*
(teste-saída resultado)
corpo ....)

- Cada variável é inicialmente ligada ao valor inicial.
- Se o teste de saída é verdadeiro, resultado é retornado.
- Senão, o corpo é executado e cada variável é atualizada para o próximo valor.
- Se o próximo valor é omitido, a variável não é alterada.


```
(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 ´(1 4 6 V (d 4) 4 (1 3)) 4)
  (1 6 V (d 4) (1 3))
```