Programação Funcional

PLP 2009/1
Profa. Heloisa

Programação Funcional

- Paradigma de programação baseado em Funções Matemáticas
- Essência de Programação: combinar funções para obter outras mais poderosas

- Programa Funcional (puro):
  - definições de funções
  - chamada de funções
- Versões mais atuais:
  - introduziram características imperativas
  - estenderam o conceito de função para procedimento
- Linguagens Funcionais:LISP, SCHEME, HOPE, ...

LISP

- **LIS**t **P**rocessing
- Linguagem de Programação Funcional
- LISP Original – proposto por John MacCarthy e um grupo em 1960 no Massachussets Institute of technology (MIT). Lisp puro, completamente funcional.

- Características:
  - linguagem funcional
  - linguagem simbólica
  - linguagem interpretada
  - linguagem declarativa e procedural

Conceitos Básicos

- Função: regra que associa elementos de um conjunto (domínio) com elementos de outro conjunto (codomínio).
- Definição de Função: especifica o domínio, codomínio e a regra de associação para a função

Ex: quadrado(x) = x*x

- Aplicação de Função: quando uma função é aplicada a um elemento do domínio fornece um resultado do codomínio. Na aplicação, um argumento substitui o parâmetro.

Ex: quadrado (2) = 4

- Forma Funcional: método de combinar funções para obter outra função como resultado.

Ex: Composição de funções

F = G°H F(x) = G(H(x))

Outras formas funcionais: condicional, recursão

Tipos de Dados (objetos) do LISP

- Átomos – elementos indivisíveis:
  A, F68, 27

  – Números (Átomos numéricos) –
  átomos só com números: 27, 3.14, -5.

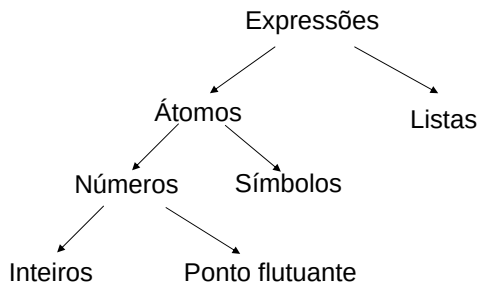
  – Símbolos (Átomos Simbólicos) –
  átomos que não são números: A NOME, X1

- Listas – série de átomos ou listas separadas por espaços e delimitadas por parêntesis:

(1 2 3 4)
(MARIA)
(JOAO MARIA)
( ) Lista vazia, também denotada por NIL
((v-1 valor-1) (v-2 valor-2) (v-3 valor-3))

S-expressões

- Átomos ou listas são chamados de S-expressões ou Expressões Simbólicas



Procedimentos e Funções

- **determinam o que deve ser feito em LISP**
- **são representados em forma de lista**

A mesma sintaxe representa tanto dados quanto programas:

(* 7 9)

(- (+ 3 4) 7)

Interpretador LISP

- Quando o interpretador LISP é ativado, o usuário passa a interagir com ele.
- Ciclo Lê-calcula-imprime:
  (executado pelo interpretador)
  - apresenta um sinal de *pronto*;
  - lê a entrada fornecida pelo usuário;
  - executa essa entrada;
  - se a execução obteve sucesso, imprime o resultado.

Exemplos

> (+ 2 1)
3
>

> (* 7 9)
63
>

>(- (+ 3 4) 7)
0
>

Avaliação de listas

- A notação (f x y) equivale a notação de função matemática f(x,y)
- O avaliador LISP quando recebe uma lista tenta interpretar o primeiro elemento da lista como o nome de uma função e e os restantes como seus argumentos
- Se o primeiro elemento da lista não for uma função definida ocorre um erro
  > (a b c)
  Error: invalid function: a
- Forma geral:
  (f a1 a2 a3 ....an)

- Regra de avaliação de expressões:
  - Avalia primeiro os argumentos
  - Aplica a função indicada pelo primeiro elemento da expressão
  - Se os argumentos forem expressões funcionais, aplica a regra recursivamente
- Exemplos:
  >(* (+ 2 5) (- 7 (/ 21 7)))
  28

  >(= (+ 2 3) 5)
  t

Convenções nas regras para avaliar expressões

- Por default, tudo é avaliado
- Números são avaliados como eles mesmos
- Símbolos como x podem ter um valor ligado. Se tiver, esse valor é retornado
- Se um símbolo for avaliado e não tiver um valor ligado, retorna um erro

Dados “versus” funções

- Para que a lista represente dados é necessário impedir sua avaliação pelo interpretador LISP
- Para isso é usada uma função especial de nome ***quote***, que também pode ser representada por aspas simples antes da expressão.
- Quote recebe um argumento e retorna esse argumento sem avaliar.

```
> (quote (a b c))
(a b c)
> (quote (+ 1 3))
(+ 1 3)
```

Forma abreviada de quote

```
> ´(a b c)
(a b c)

> ´(+ 1 3)
(+ 1 3)

> ´(- (+ 3 4) 7)
(- (+ 3 4) 7)
```

Funções que operam sobre listas

- List – recebe qualquer número de argumentos e constrói uma lista a partir desses elementos

```
> (list 1 2 3 4 5)
(1 2 3 4 5)

> (list (+ 1 2) (+ 3 4))
(3 7)

>(list ´(+ 1 2) ´(+ 3 4))
((+ 1 2) (+ 3 4))
```

- nth – recebe como argumentos um número e uma lista e retorna o elemento da lista na posição indicada pelo número, começando a partir do zero

```
> (nth 2 ´(a b c d))
c

> (nth 1 (list 1 2 3 4 5))
2

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

- Length – retorna o número de elementos de uma lista

```
> (length ´(a b c d))
4
> (length ´(1 2 (3 4) 5 6))
5
```

- Member – recebe como argumento uma expressão e uma lista e verifica se a expressão é membro da lista

```
> (member 5 ´(1 2 3 4 5))
t
>(member ´a ´(1 2 3 4 5))
nil
```

- Nil – símbolo especial
- Representa o valor “falso” e a lista vazia
- É o único símbolo que é átomo e lista ao mesmo tempo

Listas como estruturas recursivas

- Algumas funções permitem tratar listas com número desconhecido de elementos, recursivamente
- Car – recebe um único argumento que deve ser lista e retorna o primeiro elemento dessa lista
- Cdr – recebe um único argumento que deve ser uma lista e retorna essa lista sem o primeiro elemento

```
> (car ´(a b c))
a
> (cdr ´(a b c))
(b c)
> (cdr ´((a b) (c d)))
((c d))
(car (cdr ´(a b c d)))
b
```

Composição de CAR e CDR

> (CAR (CDR '(A B C)))
B

Podemos substituir as funções CAR e CDR por uma primitiva composta como

CXXR ou CXXXR ou CXXXXR

onde X pode ser A (significando CAR)
ou D (significando CDR)

Exemplo

(CAR (CAR (CDR (CDR '(HOJE E (DIA DE) AULA)))))

DIA

Usando a forma combinada:

(CAADDR '(HOJE E (DIA DE) AULA))
DIA

Funções para construir listas

CONS **– recebe duas s-expressões como argumento, avalia e retorna uma lista que é o resultado de adicionar a primeira expressão no início da segunda, que é uma lista.**

Argumentos: 1) qualquer **S-expressão**
2) lista

> (CONS 'A (B C))
(A B C)

> (CONS '(A B C) '(A B C))
((A B C) A B C)

> (CONS 'NADA ( ))
(NADA)

> (CONS '((PRIM SEG) TER) '( ))
(((PRIM SEG) TER))

Cons é a operação inversa de car e cdr

LIST – constrói uma lista a partir dos seus argumentos

> **(LIST ‘A ‘B ‘C)**
(A B C)

> **(LIST ‘(A) ‘B ‘C)**
((A) B C)

> **(LIST ‘(A B) ‘((C) D))**
((A B) ((C) D))

APPEND – constrói uma lista com os elementos das listas dadas como argumentos. Argumentos devem ser listas.

> **(APPEND ‘(A B) ‘(C))**
(A B C)

> (APPEND ‘(A) ‘( ) ‘(B) ‘( ))
(A B)

> (LIST ‘(A) ‘( ) ‘(B) ‘( ))
((A) ( ) (B) ( ))

Atribuição de valores a átomos

SETQ – faz o primeiro argumento passar a ter o valor do segundo.

Argumentos: 1) símbolo
2) qualquer S-expressão

> (SETQ L ‘(A B))
(A B)

> (SETQ L1 ‘(A B) L2 ‘(C D) N 3)
3

O valor dessa função é o último valor atribuído mas o mais importante é o efeito colateral de L que fica com valor (A B).

Efeito Colateral: algo que o procedimento faz que persiste depois que seu valor é retornado.

Um Símbolo com valor pode aparecer como argumento de um procedimento.

> (SETQ L '(A B))
(A B)

> (CDR L)
(B)

> (SETQ L1 5 L2 2)
2

> (+ L1 L2)
7

Criar novas funções

- A programação em Lisp consiste em definir novas funções a partir de funções conhecidas
- Depois de definidas, as funções podem ser usadas da mesma forma que as funções embutidas na linguagem
- Defun – função para definir outras funções
- Argumentos:
  - Nome da função
  - Lista de parâmetros formais da função (átomos simbólicos)
  - Corpo da função (0 ou mais expressões s que definem o que a função faz)

```
> (defun quadrado (x)
        (* x x))
quadrado
```

- Defun retorna como resultado o nome da função
- Não avalia os argumentos
- Efeito colateral: cria uma nova função e adiciona ao ambiente Lisp
- Forma geral:

```
(defun <nome da função> (<parâmetros formais>)
        <corpo da função>)
```

Chamada de funções

- A função definida por defun deve ser chamada com o mesmo número de argumentos (parâmetros reais) especificados na definição
- Os parâmetros reais são ligados aos parâmetros formais
- O corpo da função é avaliado com essas ligações
- A chamada:

```
> (quadrado 5)
```

- Faz com que 5 seja ligado ao parâmetro
- Na avaliação de (* x x), a avaliação de x resulta em 5, causando a avaliação de (* 5 5)

Uso de funções

- Uma função definida por defun pode ser usada da mesma forma que as funções embutidas (pré-definidas na linguagem)
- Definir uma função que calcula o comprimento da hipotenusa de um triângulo reto dados os outros dois lados do triângulo

```
(defun hipotenusa (x y)
      (sqrt (+ (quadrado x)
               (quadrado y)))))
```

Sqrt – função embutida que calcula a raiz quadrada

Exemplos

```
> (defun F-to-C (temp)
     (/ (- temp 32)  1.8))
F-to-C

> (F-to-C 100)
37.77

> (defun TROCA (par)
      (list (cadr par) (car par)))
TROCA

>(TROCA ´(a b))
(b a)
```

Predicados

- Um predicado é uma função que retorna T (verdadeiro) ou Nil (falso)
- T e NIL são átomos especiais com valores pré-definido
- ATOM – verifica se seu argumento é um átomo

```
> (atom 5)
T
> (atom ´L)
T
> (setq L ´(a b c))
(a b c)
> (atom L)
Nil
```

- LISTP – verifica se seu argumento é uma lista

```
> (listp ´(a b c))
T
> (setq L ´(a b c))
(a b c)
> (listp L)
T
> (listp ´L)
nil
```

- Equal – recebe dois argumentos e retorna T se eles são iguais e NIL se não são.

> (equal `a `a)
T
> (equal L L)
T
> (equal L ´(a b))
T
> (equal L ´L)
nil

- Null – verifica se seu argumento é uma lista vazia

> (null ´( ))
T

(null L)
Nil

(null ´L)
Nil

- NUMBERP – verifica se seu argumento é um número

>(numberp 5)
T
> (setq N 5)
5
> (numberp N)
T
> (setq Digitos ´(1 2 3 4 5))
(1 2 3 4 5)
> (number Digitos)
Nil

- Greaterp – argumentos devem ser números. Verifica se estão em ordem decrescente
- > (greaterp 5 4 3 2 1)
- T
- > (greaterp 3 1 4)
- Nil

- LESSP – argumentos devem ser números. Verifica se estão em ordem crescente
- > (lessp 1 2 3 4 5)
- T
- >(lessp 3 6 2)
- nil

- ZEROP – argumento deve ser número. Verifica se é zero

```
> (setq zero 0)
0
> (zerop zero)
T
```

- MINUSP – argumento deve ser número. Verifica se é negativo

```
> (setq N -5)
-5
> (minusp N)
T
```

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

(**cond** (<condição1> <ação1>)
(<condição1> <ação1>)
....
(<condição1> <ação1>))

- Condições e ações podem ser s-expressões com cada par entre parênteses
- Cond não avalia todos os argumentos
- Avalia as as condições até que uma retorna 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)

Valores nil e “não nil”

- Os predicados em Lisp são definidos de forma que qualquer expressão diferente de NIL é considerada verdadeira
- Member – recebe dois argumentos, o segundo deve ser lista
- Se o primeiro for membro do segundo retorna o sufixo do segundo argumento que tem o primeiro argumento como elemento inicial

```
> (member 3 ´(1 2 3 4 5))
(3 4 5)
> (member 6 ´(1 2 3 4 5))
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 list-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)
```

Função para concatenar duas listas

```
(defun concatena (lista1 lista2)
   (cond ((null lista1) lista2)
         (t (cons (car lista 1) (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 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

Outras formas condicionais

(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.

> (if (> A B) A B)

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

Variáveis livres e ligadas

```
(defun incremento (parametro)
     (setq parametro (plus 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
```

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
```

PROGN – executa qualquer número de expressões na seqüência e retorna o valor da última.

(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*)

> (when (> I 0) (setq R (+ R Parcela)) (setq I (+ I 1)) (print R)

- 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)

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

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))**
1
2
3
nil

Entrada e Saída

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

> (setq L1 (read) L2 (read))
(e f g)

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

- **PRINT – função que recebe um argumento, o avalia e imprime esse resultado na saída padrão.**

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

> (concatena L1 L2)
A lista resultante e
(a b c d e f g)
```