Programação Funcional – Parte 1

PLP 2019/1
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

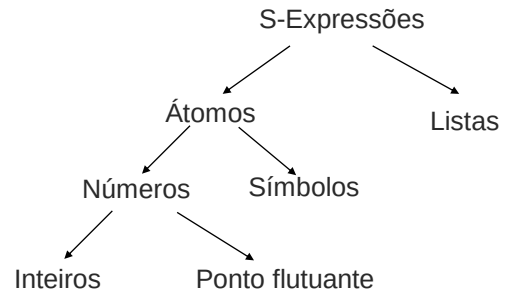
- Átomos – elementos indivisíveis:
  A, F68, 27
  - Números (Átomos numéricos) –
    átomos só com números: 27, 3.14, -5.
  - Símbolos (Átomos Simbólicos) –
    átomos que não são números: A NOME, X1

- 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



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.

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:

(a b c)

(* 7 9)

(- (+ 3 4) 7)

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 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

  >(= (+ 1 2) (- 6 (+ 1 1)))
  NIL

Valores true e false são representados pelos símbolos T e NIL

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

- Lisp tem muitas funções e procedimentos pré-definidos. Veremos alguns exemplos
  - 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))
(5)
>(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

Funções que constroem listas
list e append

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))

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

>(LIST ´(+ 1 2) ´(+ 3 4))
((+ 1 2) (+ 3 4))

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-definidos
- 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
> (setq L ´(a b))
(a b)
> (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)
> (numberp Digitos)
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
```


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
```