

Paradigmas de Linguagens de Programação

Prof. Sergio D. Zorzo
Departamento de Computação - UFSCar
1º semestre / 2013
Aula 4

Material adaptado do Prof. Daniel Lucrédio

Programação Funcional

Um pouco de história

- 1936
 - Alonzo Church usou um sistema notacional denominado λ -cálculo para definir algoritmos
 - Alan Turing usou “máquinas” abstratas – autômatos – para definir algoritmos
- Conexão entre noção informal de algoritmo e definição precisa
 - ***Tese de Church-Turing***

Um pouco de história

- Máquinas de Turing Universais
 - Levaram ao surgimento do programa armazenado
 - Arquitetura de Von Neumann
 - Todos os computadores do mundo
 - Linguagens de programação imperativas
- Mecanismo de resolução de Robinson
 - Resolução SLD
 - Programação Lógica

Um pouco de história

- λ -cálculo de Alonzo Church
 - Funções matemáticas com poder computacional completo
 - Definição de funções anônimas
 - Funções para definir funções
 - Em teoria, deveria existir uma função capaz de calcular qualquer função
 - John McCarthy desenvolveu uma função universal: equivalente da Máquina de Turing universal em λ -cálculo
 - Deu origem ao primeiro interpretador LISP
 - Primeira das linguagens de programação funcionais

LISP vs Prolog

- Ambas:
 - Aplicações em IA
 - Processamento simbólico
 - Declarativa (*)
 - Obs: tanto Prolog como LISP podem ser “ajustados” para funcionar mais imperativamente
- Porém:
 - Prolog trabalha com relações
 - LISP trabalha com funções

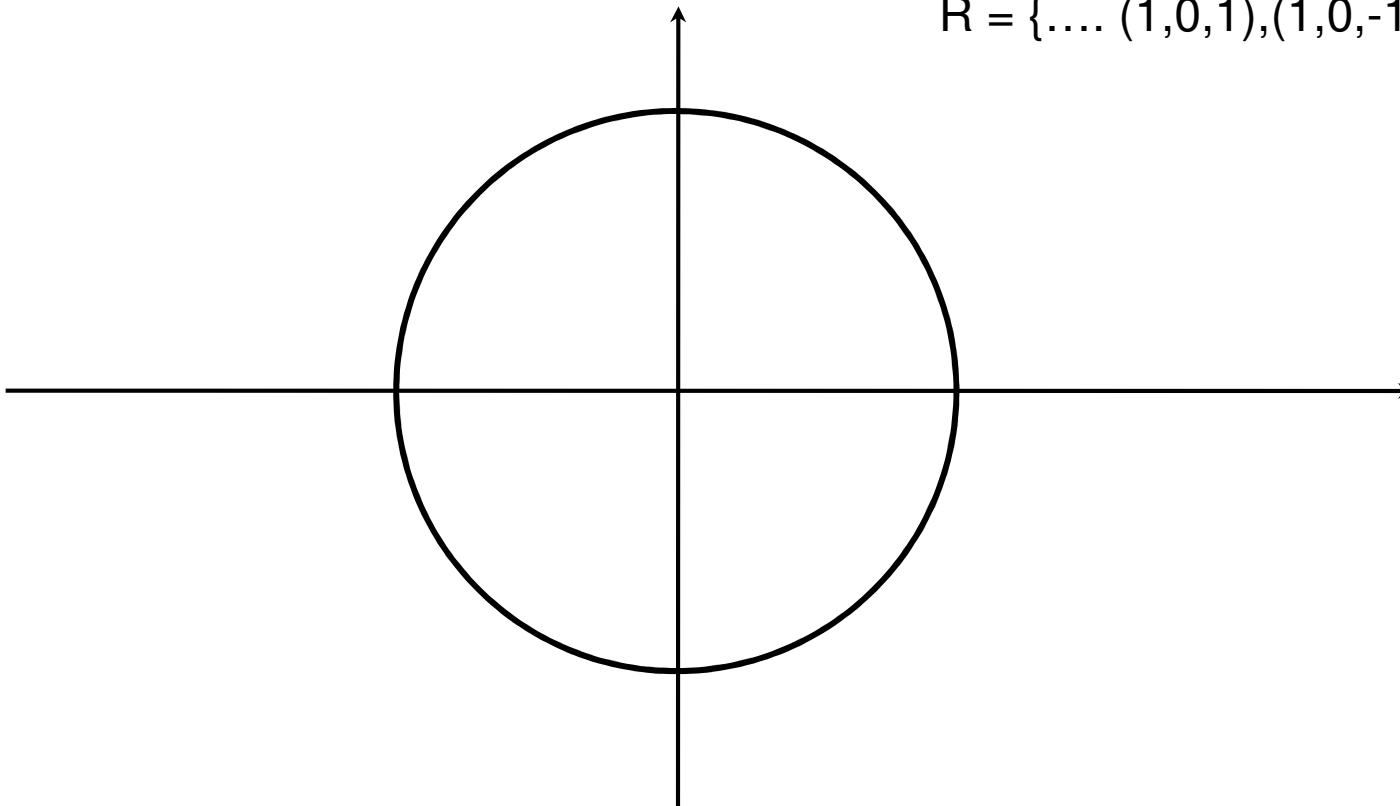
Relações vs Funções

- Uma **relação** é um mapeamento entre dois conjuntos:
 - Domínio (A) e contra-domínio (B)
 - Seja $A \times B$ o produto cartesiano entre A e B
 - Uma relação em $A \times B$ é qualquer subconjunto de $A \times B$
- Uma **função** é uma relação em $A \times B$, tal que:
 - Todo elemento de A deve ter correspondente em B
 - Cada elemento de A só poderá ter no máximo um correspondente em B

Relações vs Funções

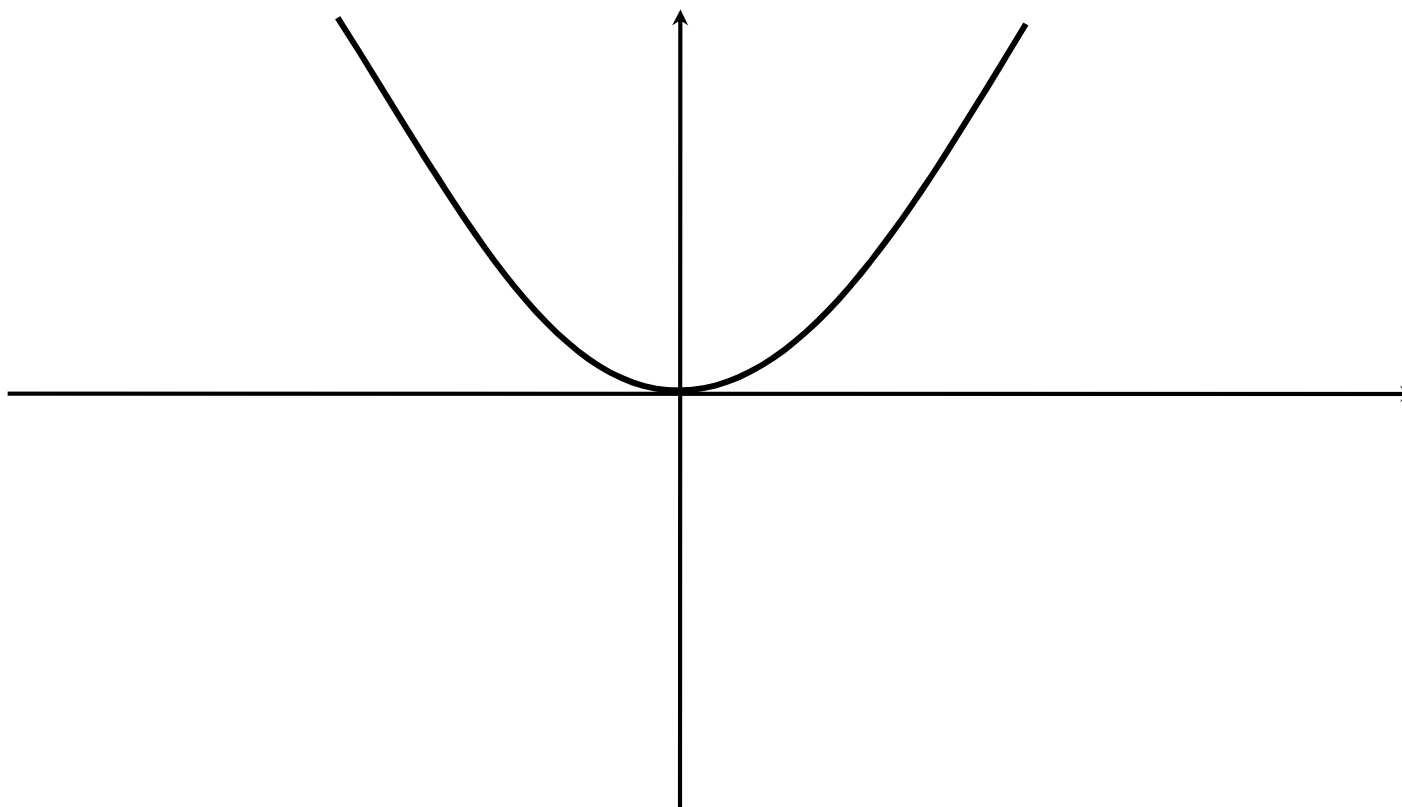
- Ex: no plano cartesiano
 - $R = \{(x,y) \in \mathbb{R}^2 : x^2 + y^2 = a^2\}$ é uma **relação**
 - mas não é uma função

$$R = \{ \dots (1,0,1), (1,0,-1), \dots \}$$



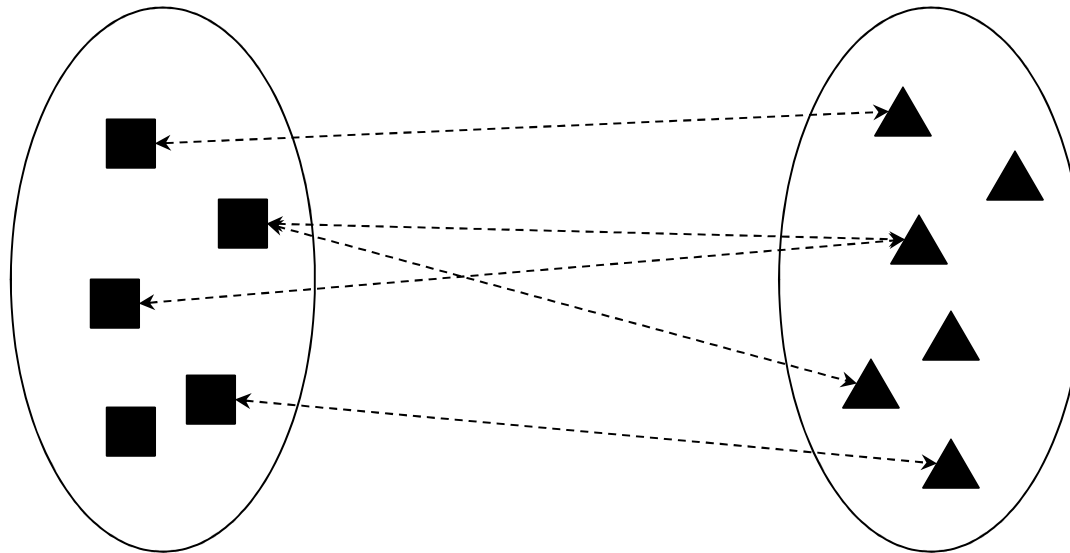
Relações vs Funções

- Ex: no plano cartesiano
 - $f(x)=x^2$ é uma **função**
 - Obviamente, também é uma relação



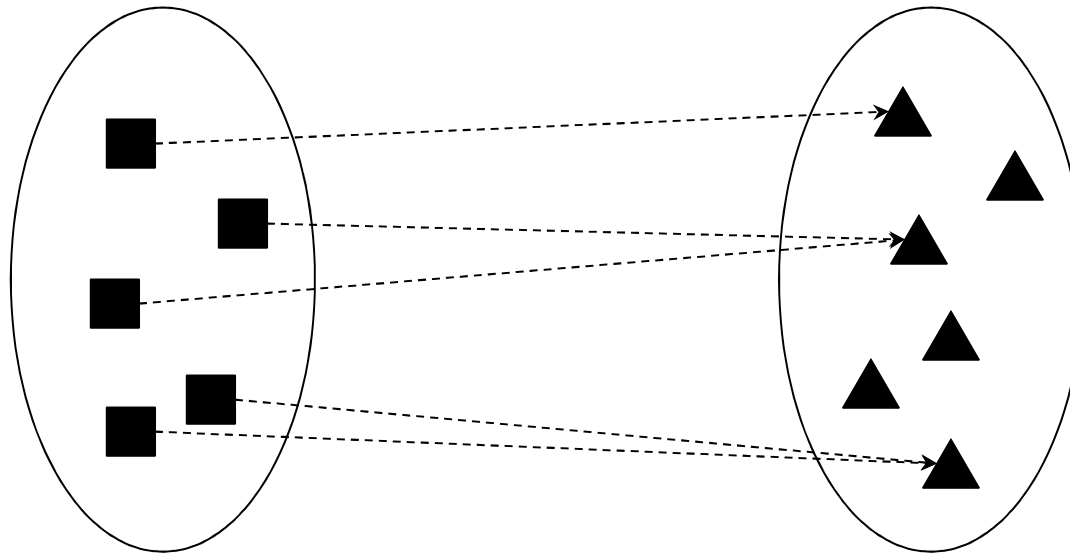
Relações (Prolog)

- Consultas diversas
 - Mecanismo de resolução



Funções (Prolog)

- Consultas de avaliação de expressões
 - Aplicação de funções



LISP vs Prolog

- Mecanismo de inferência do Prolog é mais poderoso
 - É possível programar funcionalmente

`f(X,Y) :- Y is X*3+4`

- Mecanismo do LISP não permite fazer inferências diretamente
 - Mas é possível programar logicamente

`(RULE IDENTIFY16`

`(IF ((> ANIMAL) IS A (> TYPE))`

`((< ANIMAL) IS A PARENT OF (> CHILD)))`

`(THEN ((< CHILD) IS A (< TYPE))))`

LISP vs linguagens imperativas

- Programação Funcional Pura
 - O objetivo é imitar as funções matemáticas
- Em uma linguagem imperativa
 - Baseada em estados/instruções
 - Baseada em atribuição
 - Baseada em armazenamento
 - Ou seja, o programador ensina como o computador deve computar
 - Ex: Calcular $(x+y)/(a-b)$ em assembly
 - Linguagens de alto nível fazem a mesma coisa

LISP vs linguagens imperativas

- Mesmo com o conceito de subrotinas com valor de retorno (funções)
 - Existem variáveis globais
 - Passagem por referência
 - Efeitos colaterais
 - Expressões mais complexas exigem comandos imperativos
 - Preocupação **explícita** com manipulação da memória
- Mas o objetivo é claro: eficiência na arquitetura

LISP vs linguagens imperativas

- Em linguagens puramente funcionais
 - Não existem variáveis
 - Não existe atribuição
 - O valor de uma expressão depende somente dos valores das suas subexpressões (se houverem)
 - Programar funcionalmente é programar sem atribuição
 - Sem efeitos colaterais, uma expressão tem sempre o mesmo valor, sempre que avaliada com os mesmos parâmetros

LISP vs linguagens imperativas

- Em linguagens puramente funcionais
 - Não existe repetição explícita
 - Não existe o conceito de instrução
 - Que é um reflexo da arquitetura de Von Neumann
 - Funções são elementos de primeira classe
 - Podem ser passadas como parâmetro
 - Armazenadas em listas
 - Usadas como valores em expressões
- 4 componentes:
 - Conjunto de funções primitivas
 - Conjunto de formas funcionais
 - Operação de aplicação / avaliação
 - Conjunto de dados

Linguagens funcionais

- Funções primitivas
 - Funções predefinidas pela linguagem
 - Podem ser aplicadas diretamente
 - Ex: soma, comparação, etc...
- Formas funcionais
 - Maneiras de combinar funções para criar novas funções
 - Composição

$$f(x) \equiv x + 2$$

$$g(x) \equiv 3 * x$$

$$h \equiv f \circ g$$

$$h(x) \equiv f(g(x)) \equiv (3 * x) + 2$$

Linguagens funcionais

- Formas funcionais
 - Maneiras de combinar funções para criar novas funções
 - Construção: lista de funções como parâmetro, todas são aplicadas, retornando uma lista de valores
 - Apply-to-all: uma função aplicada a uma lista, retornando uma lista de valores
- Operação de aplicação
 - Mecanismo embutido para aplicar uma função a seus argumentos e produzir um valor

Linguagens funcionais

- Conjunto de dados
 - Elementos permitidos nos domínios e contra-domínios
 - Em LISP, são as listas
- Quinto elemento
 - Amarração de nomes a funções
 - Para permitir sua reutilização

Linguagens funcionais

- Como programar?
 - Definindo funções
 - Chamando funções
- Na prática
 - Muitos conceitos de linguagens imperativas
 - Linguagens funcionais “impuras”
 - Mas aqui vamos nos concentrar nos aspectos puramente funcionais

LISP

LISP

- LISt Processing
 - Linguagem funcional
 - Originalmente puramente interpretada (e pura)
 - Atualmente é compilada (e impura)
- Por que LISP?
 - Primeira
 - Referência
 - Sintaxe simplificada
 - Padronizada
- Existem outras: Scheme (LISP simplificada), ML, Haskell

Tipos de dados (objetos)

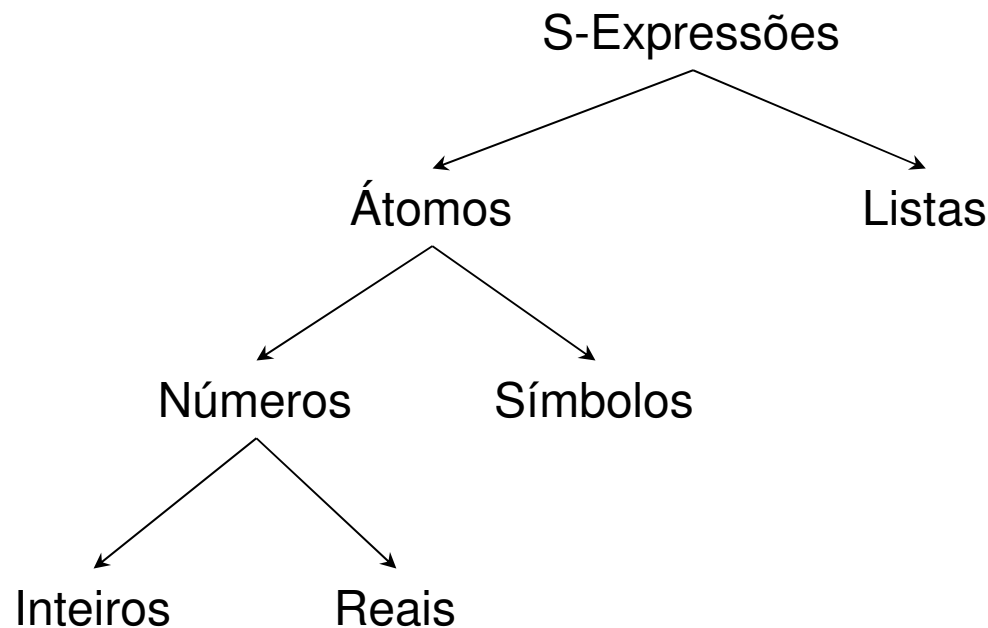
- Átomos – elementos indivisíveis:
 - A, F68, 27
- Números (Átomos numéricos)
 - átomos só com números: 27, 3.14, -5.
- Símbolos (Átomos Simbólicos)
 - átomos que não são números: A, NOME, X1

Listas

- Sequência 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
 - Ou simplesmente “expressões”



Procedimentos e funções

- Determinam o que deve ser feito em LISP
 - São representados em forma de lista
- Programas são conjuntos de procedimentos
- A mesma sintaxe é utilizada para representar:
 - Dados (a b c)
 - Aplicação de funções (- (+ 3 4) 7)
 - Definição de funções (defun quadrado (x) (* x x))

Utilização de LISP

- Ciclo Lê-Calcula-Imprime
 - Interpretador LISP
- Demonstração

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
- 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
- Demonstração

Manipulação de listas

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

Dados vs 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
 - Demonstração

Manipulação de listas

- List
 - recebe qualquer número de argumentos e constrói uma lista a partir desses elementos
- 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
- Length
 - retorna o número de elementos de uma lista
- Member
 - recebe como argumento uma expressão e uma lista e verifica se a expressão é membro da lista
 - Retorna uma lista começando pelo elemento encontrado
- Demonstração

Manipulação de listas

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

Recursividade em listas

- 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
 - Obs: o nome das funções deriva de instruções da máquina IBM 704, onde o primeiro interpretador LISP foi construído
 - CAR = Contents of Address Register
 - CDR = Contents of Decrement Register

Composição de CAR e CDR

- 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)
- Demonstração

Construção de 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 é a operação inversa de car e cdr
- Demonstração

Composição de listas

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

Outras funções de listas

Butlast

Sintaxe : (butlast <list> [<n>])

Retorna uma cópia dos elementos da lista, removendo desta lista os <n> elementos finais.

Exemplos:

> (butlast '(a b c d e) 2)
(A B C)

> (butlast '(a b c d e))
(A B C D)

Outras funções de listas

Length

Sintaxe : (length <expr>)

Retorna o tamanho de uma lista.

Exemplos:

> (length '(a b c d e))
5

> (length '((a b c) (d e)))
2

Atribuição

- SETQ – faz o primeiro argumento passar a ter o valor do segundo.
 - Argumentos:
 - 1) símbolo
 - 2) qualquer S-expressão
 - Também funciona para o terceiro e quarto, quinto e sexto, etc.
- Demonstração

Atribuição

- 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).
- Mas tem um efeito colateral
 - Área global de memória é afetada
- Efeito Colateral: algo que o procedimento faz que persiste depois que seu valor é retornado.
- Faz parte das “impurezas” do LISP
 - Mas necessário para aplicações práticas
- Demonstração

Criação de 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)
- Forma geral:

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

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
- Demonstração

Exemplos

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

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

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

Predicados

- Um predicado é uma função que retorna T (verdadeiro) ou Nil (falso)
 - T e NIL são átomos especiais com valores pré-definido
- Alguns exemplos:
 - ATOM – verifica se seu argumento é um átomo
 - LISTP – verifica se seu argumento é uma lista
 - Equal – recebe dois argumentos e retorna T se eles são iguais e NIL se não são.
 - Null – verifica se seu argumento é uma lista vazia
 - NUMBERP – verifica se seu argumento é um número
 - ZEROP – argumento deve ser número. Verifica se é zero
 - MINUSP – argumento deve ser número. Verifica se é negativo
- Demonstração

Condicional

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

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

Condicional

- 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
- Demonstração

If

- Função If
- Teste simples, com else opcional

```
(if <condição> <ação-then> [<ação-else>])
```

- Demonstração

Verdadeiro / falso

- Os predicados em Lisp são definidos de forma que qualquer expressão diferente de NIL é considerada verdadeira
- Ex: 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
```

Igualdade

- Vários predicados para teste de igualdade dos argumentos
- eq
 - Testa se são o mesmo objeto
- eql
 - Testa se são eq
 - Ou se são números do mesmo tipo
 - Ou caracteres representando o mesmo objeto
- equal
 - Testa se são estruturalmente idênticos (mesma árvore)
- equalp
 - Testa números de tipos diferentes e caracteres sem considerar o caso
- Demonstração

Igualdade aritmética

- Sempre que o objetivo é testar o valor de expressões, utilizar a função =

- Exs:

```
> (= 10 10)
```

```
T
```

```
> (= 1.0 1)
```

```
T
```

```
> (= 1 (sin (/ pi 2)))
```

```
T
```

Conectivos lógicos

- Not – recebe um argumento e retorna t se ele for nil e nil caso contrário
- And – recebe qualquer número de argumentos, e os avalia da esquerda para a direita
 - Quando encontrar um nil, retorna nil e não avalia mais
 - Se não encontrar nil, retorna o valor do último avaliado
- Or – recebe qualquer número de argumentos, e os avalia da esquerda para a direita
 - Quando encontrar um não-nil, retorna esse valor e não avalia mais
 - Se só encontrar nil, retorna nil
- Demonstração

Variáveis livres e ligadas

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

- Uma variável **ligada** em relação a um procedimento é um símbolo que aparece na lista de parâmetros
- Uma variável **livre** em relação a um procedimento é um símbolo que não aparece na lista de parâmetros

Variáveis livres e ligadas

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

- Neste exemplo:
 - “parametro” é ligada ao procedimento incremento
 - “livre” é livre em relação ao procedimento incremento
 - “saída” é livre em relação ao procedimento incremento

Variáveis livres e ligadas

- Variáveis ligadas
 - Recebe um valor na chamada
 - Se existia antes da chamada, seu valor é restaurado
 - Se não existia antes da chamada, continua não existindo depois
- Variáveis livres
 - Precisa existir antes da chamada
 - Continua existindo após a chamada
- Demonstração

Chamada por valor

- LISP trabalha com passagem de parâmetros por valor
 - Ou seja, apenas o valor de uma variável é passado para dentro de um procedimento
 - Pode-se modificar o valor dentro do procedimento, sem alterá-lo fora
- Demonstração

Variáveis locais com let

- Let controla:
 - a atribuição de valores a variáveis
 - a criação de variáveis (ligadas ao procedimento let)

```
(let ((parametro1 valorInicial1)
      (parametro2 valorInicial2)
      ...
      (parametron valorInicialn))
  <expressões>)
```

Variáveis locais com let

- Se uma variável já existir, a mesma é associada a um novo valor
 - Mas após a chamada, volta a ter o valor anterior
- Se uma variável não existir, a mesma é criada
 - Mas após a chamada, é como se não existisse
- Demonstração

Variáveis locais com let

- Let faz as atribuições todas em paralelo

```
(let ((parametro1 valorInicial1)
      (parametro2 valorInicial2)
      ...
      (parametron valorInicialn))
  <expressões>)
```

- Neste exemplo, parametro1, parametro2, ... parametron são inicializadas ao mesmo tempo
 - Ou seja, parametro2 não conhece o valor de parametro1, por exemplo

Variáveis locais com let

- Para fazer atribuições sequencialmente

```
(let* ((parametro1 valorInicial1)
      (parametro2 valorInicial2)
      ...
      (parametron valorInicialn))
  <expressões>)
```

- Neste exemplo, parametro1, parametro2, ... parametron são inicializadas em sequência
 - Ou seja, parametro2 pode usar o valor de parametro1, por exemplo
- Demonstração

Atribuição paralela com psetq

- Atribuição com setq é sequencial
 - Ex: (setq x 1 y (+ x 1))
 - Atribui 1 a x
 - Atribui $x + 1 = 2$ a y
- Para fazer atribuição paralela: psetq
 - Ex: (psetq x 1 y (+ x 1))
 - Erro
- Demonstração

Passagem de funções como parâmetro

- Em programação funcional, funções são elementos de primeira classe
 - Podem ser passados como parâmetros
 - Podem ser armazenados em listas
- Demonstração

Macros

Macros

- Facilitam algumas tarefas comuns
- Macros são funções especiais que servem de “apelidos” para construções sintáticas mais complexas
 - A escrita do programa fica mais simplificada
 - O programa fica mais legível
- Ex: When e Unless
- Demonstração

Do

```
(do ((var1 ini1 inc1)
    (var2 ini2 inc2)
    ...
    (varn inin incn))
  (teste-saída resultado)
  corpo ... )
```

- **Funcionamento:**
 - Inicializa var1, var2, ..., varn com ini1, ini2, ..., inin
 - A cada iteração:
 - Verifica o teste de saída: se diferente de nil, resultado é retornado
 - Executa o corpo
 - Incrementa var1, var2, ..., varn com inc1, inc2, ..., incn

Do

- Iteração estilo imperativo
- Repetição genérica
 - Qualquer número de variáveis
 - Qualquer inicialização
 - Qualquer tipo de incremento
- Demonstração

Do

- É um exemplo de macro, equivale a

```
(block nil
  (let ((var1 init1)
        (var2 init2)
        ...
        (varn initn)))
  declarations
  (loop
    (when end-test (return (progn . result)))
    (tagbody . tagbody)
    (psetq var1 step1
           var2 step2
           ...
           varn stepn)))
)
```

Do*

- Versão “sequencial”

```
(block nil
  (let* ((var1 init1)
          (var2 init2)
          ...
          (varn initn))
    declarations
    (loop
      (when end-test (return (progn . result)))
      (tagbody . tagbody)
      (setq var1 step1
            var2 step2
            ...
            varn stepn)))
)
```

Dolist

- Macro que simplifica o uso de “do”
 - Específico para percorrer listas

```
(dolist (var lista [resultado])  
  corpo ... )
```

- Demonstração

Dotimes

- Macro que simplifica o uso de “do”
 - Específico para iterar um determinado número de vezes

```
(dotimes (var número [resultado])  
  corpo ... )
```

- “var” começa em 0 e é incrementado até número - 1
- Demonstração

Mapcar

- Aplica uma função repetidamente aos argumentos fornecidos

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

- Demonstração

Entrada e saída

Entrada e saída

- READ – função que não tem parâmetros
 - Quando é avaliada, retorna a próxima expressão inserida no teclado
- PRINT – função que recebe um argumento
 - O avalia e imprime seu resultado na saída padrão
- TERPRI – função sem argumentos que insere um caracter newline na saída padrão
- FRESH-LINE – semelhante a TERPRI, mas só insere o newline se a linha atual não estiver no início
- Demonstração

Escopo léxico vs escopo dinâmico

Escopo léxico

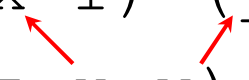
- Common Lisp tem dois tipos de variáveis: léxicas e especiais
- As variáveis léxicas são definidas:
 - Por alguma construção sintática (let ou defun) e podem ser referenciadas pelo código que aparece dentro dessa construção.
 - Por setq e podem ser referenciadas em qualquer lugar, desde que não tenham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

Escopo léxico

```
(defun foo( )  
  (let ((x 1) (y 2))  
    (frotz x y)  
    (let ((y 2) (z 3))  
      (baz x y z))  
    z) )
```

Escopo léxico

```
(defun foo( )  
  (let ((x 1) (y 2))  
    (frotz x y)  
    (let ((y 2) (z 3))  
      (baz x y z)  
      z) )  
  )
```




The diagram illustrates lexical scoping in the provided code. Two red arrows originate from the variable 'y' in the expression '(frotz x y)'. One arrow points to the 'y' in the first 'let' binding '(y 2)', indicating that this is the environment in which 'frotz' is executed. The second arrow points to the 'y' in the second 'let' binding '(y 2)', indicating that this is the environment in which 'baz' is executed.

Escopo léxico

```
(defun foo( )  
  (let ((x 1) (y 2))  
    (frotz x y)  
    (let ((y 2) (z 3))  
      (baz x y z))  
    z) )
```

The diagram illustrates lexical scope resolution. Red arrows point from the 'y' in the 'baz' call to the 'y' in the inner 'let' block, and from the 'x' in the 'baz' call to the 'x' in the outer 'let' block. The 'y' in the outer 'let' block is highlighted with a red circle.

Escopo léxico



```
(defun foo( )  
  (let ((x 1) (y 2))  
    (frotz x y)  
    (let ((y 2) (z 3))  
      (baz x y z))  
    z) )
```

Escopo léxico

- Demonstração

Escopo dinâmico

- Variáveis ditas “especiais” operam segundo escopo dinâmico
- As variáveis especiais são definidas por DEFVAR ou SPECIAL
 - Por convenção, são escritas entre *
 - > (defvar *x*)
 - > (declare (special *x*))
- Regras de escopo são definidas pela sequência de chamadas, ou pela execução do programa
- Demonstração

Bibliografia

- Nesta aula, utilizamos conceitos dos seguintes livros:
 - Sebesta, R.W. Conceitos de Linguagens de Programação – 5ª edição. Bookman, 2003.
 - Ghezzi, C. & Jazayeri, M. Conceitos de Linguagens de Programação. Campus, 1985.
 - Sethi, R. Programming Languages. Addison-Wesley, 1989.
 - Winston, P.H. & Horn, B.K.P. LISP – Second Edition. Addison-Wesley, 1984.

Manual

- Consulte também o manual do Common Lisp
- <http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node1.html>

Fim