PARADIGMAS E LINGUAGENS DE PROGRAMAÇÃO

ENGENHARIA DA COMPUTAÇÃO – UFC/SOBRAL

Prof. Danilo Alves

danilo.alves@alu.ufc.br

ASPECTOS GERAIS

UNIVERSIDADE FEDERAL DO CEARÁ

- LISP é uma linguagem de programação funcional
- Inventado por J. McCarthy em 1959
- Muitos dialetos de LISP
- Padrão COMMON LISP
- CLISP, uma implementação de COMMON LISP quase completo (99%)
- São altamente portáveis entre máquinas e sistemas operacionais

ASPECTOS GERAIS

- processamento simbólico e de conhecimento (IA),
- processamento numérico (MACLISP),
- Programas muito difundidos como editores (EMACS) e CAD (AUTOCAD).

COMMON LISP

UNIVERSIDADE FEDERAL DO CEARÁ

- Sintaxe clara
- Muitos tipos de dados: numbers, strings, arrays, lists, characters, symbols, structures, streams etc.
- Tipagem em tempo de execução: No entanto recebe mensagens de erro caso haja violações de tipo (operações ilegais) .
- Funções genéricas: 88 funções aritméticas (integers, ratios, floating point numbers, complex numbers), 44 funções de pesquisa/filtragem/ordenação para listas, arrays e strings

COMMON LISP

UNIVERSIDADE FEDERAL DO CEARÁ

- Gerenciamento de memória automático (garbage collection)
- Pacoteamento (packaging) de programas em módulos
- Um sistema de objetos, funções genéricas com a possibilidade de combinação de métodos.
- Macros: todo programador pode realizar suas próprias extensões da linguagem

CLISP

UNIVERSIDADE FEDERAL DO CEARÁ

- Um interpretador.
- Um compilador para executáveis até 5 vezes mais rápidos.
- Todos os tipos de dados com tamanho ilimitado (a precisão e o tamanho de uma variável não necessita de ser declarado, o tamanho de listas e arrays altera-se dinamicamente)
- Integers de precisão arbitrária, precisão de ponto flutuante ilimitada.

UNIVERSIDADE FEDERAL DO CEARÁ

TUTORIAL DA LINGUAGEM LISP

COMMON LISP HINTS - GEOFFREY J. GORDON - FRIDAY, FEBRUARY 5, 1993

SÍMBOLOS

UNIVERSIDADE FEDERAL DO CEARÁ

- Um símbolo é somente um string de caracteres.
- a, b, c1, foo, bar, baaz-quux-garply

- Códigos para promp após “>”
- Comentários após “;”

SÍMBOLOS

UNIVERSIDADE FEDERAL DO CEARÁ

- Armazena 5 em a
  - > (setq a 5)
- Insere 6 em a temporariamente
  - >(let a 6)
- Soma a com 6
  - (+ a 6)

SÍMBOLOS

- símbolos especiais, t e nil
  - t e nil para representar verdadeiro e falso
  - Ex:
  - > (if t 5 6)
  - > (if nil 5 6)
  - > (if 4 5 6)

O último exemplo é estranho, mas está correto. nil significa falso e qualquer outra coisa verdadeiro

Símbolos como nil e t são chamados símbolos auto-avaliantes, porque avaliam para si mesmos.

SÍMBOLOS

UNIVERSIDADE FEDERAL DO CEARÁ

- Símbolos auto-avaliantes chamados palavras-chave.
- Qualquer símbolo cujo nome inicia com dois pontos é uma palavra-chave
- Ex:
  - > :this-is-a-keyword
  - > :so-is-this
  - > :me-too

SÍMBOLOS

UNIVERSIDADE FEDERAL DO CEARÁ

- Ex:
  - (defun show-members (&key a b c d) (write (list a b c d))) ->Define uma função
  - (show-members :a 1 :c 2 :d 3) -> Chama a função especificando as palavras-chave

NÚMEROS

UNIVERSIDADE FEDERAL DO CEARÁ

- Um inteiro é um string de dígitos opcionalmente precedido de um + ou -
- Um real parece com um inteiro, só que possui um ponto decimal e pode opcionalmente ser escrito em notação científica
- Um racional se parece com dois inteiros com um / entre eles
- LISP suporta números complexos que são escritos #c(r i)

NÚMEROS

UNIVERSIDADE FEDERAL DO CEARÁ

- Exemplos:
  - 17
  - -34
  - +6
  - 3.1415
  - 1.722e-15
  - #c(1.722e-15 0.75)

NÚMEROS

UNIVERSIDADE FEDERAL DO CEARÁ

- As funções aritméticas padrão são todas avaliáveis: +, -, *, /, floor, ceiling, mod, sin, cos, tan, sqrt, exp, expt e etc.
- Todas elas aceitam qualquer número como argumento:
  - > (+ 3 3/4)
  - > (exp 1)
  - > (exp 3)
  - > (+ 5 6 7 (* 8 9 10))
  - > (/ 7 3.0)
- Não existe limite para o valor absoluto de um inteiro exceto a memória do computador.
- Evidentemente cálculos com inteiros ou racionais imensos podem ser muito lentos.

CONSES - ASSOCIAÇÕES

UNIVERSIDADE FEDERAL DO CEARÁ

- Cons é somente um registro de dois campos
- Os campos são chamados de "car" e "cdr" por razões históricas
  - "contents of address register" e "contents of decrement register“

CONSES - ASSOCIAÇÕES

UNIVERSIDADE FEDERAL DO CEARÁ

- Ex:
  - > (cons 4 5) -> (4 . 5)
  - > (cons (cons 4 5) 6) -> ((4 . 5) . 6)
  - > (car (cons 4 5)) -> 4
  - > (cdr (cons 4 5)) -> 5

LISTAS

UNIVERSIDADE FEDERAL DO CEARÁ

- Estruturas de dados a partir de conses
- A mais simples com certeza é a lista encadeada:
  - O car de cada cons aponta para um dos elementos da lista
  - O cdr aponta ou para outro cons ou para nil.

LISTAS

- Ex:
  - (list 4 5 6) -> (4 5 6)
  - (nth 2 '(a b c d)) -> elemento da lista na posição 2
  - (nthcdr 2 '(a b c d)) -> lista restante após a 2ª posição
  - Regras:
  - Se o cdr de um cons é nil, lisp não se preocupa em imprimir o ponto ou o nil
  - Se o cdr de cons A é cons B, então lisp não se preocupa em imprimir o ponto para A nem o parênteses para B
  - > (cons 4 nil) -> (4)
  - > (cons 4 (cons 5 6)) -> (4 5 . 6)
  - > (cons 4 (cons 5 (cons 6 nil))) -> (4 5 6)

Note que nil corresponde à lista vazia

- O car e cdr de nil são definidos como nil.
- O car de um átomo é o próprio átomo.
- O cdr de um átomo é nil.

LISTAS

- armazena uma lista em uma variável, pode fazê-la funcionar como uma pilha:
- Ex:
  - > (setq a nil) -> nil
  - > (push 4 a) -> (4)
  - > (push 5 a) -> (5 4)
  - > (pop a) -> 5
  - >a -> (4)
  - > (pop a) -> 4
  - >a -> nil

EXERCÍCIOS

UNIVERSIDADE FEDERAL DO CEARÁ

1) Desenhe as representações internas de dados para as listas seguintes:

(A 17 -3)

((A 5 C) %)

(NIL 6 A)

((A B))

(* (+ 15 (- 6 4)) -3)

2) Qual é o CAR de cada uma das listas do exercício anterior?

3) Qual é o CDR de cada uma das listas do exercício anterior 1?

EXERCÍCIOS

UNIVERSIDADE FEDERAL DO CEARÁ

4) Escreva as declarações necessárias, usando CAR e CDR, para obter os valores seguintes das listas do exercício 1:

(-3)	(5 (%))
(-3 -3)	(6 (6))
(C %)	(5 %)
(A C %)	((B) A)
(6 (6) 6)	(A ((B) B))

ENTRADA E SAÍDA

UNIVERSIDADE FEDERAL DO CEARÁ

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

>(setq L1 (read) L2 (read))

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

> (print "Digite um numero")

FUNÇÕES

- Exemplos de funções:

```
> (+ 3 4 5 6) ; Essa função pode ter vários argumentos
18
> (+ (+ 3 4) (+ (+ 4 5) 6)) ; A saída de uma função já pode servir para entrada em outra
22
```

FUNÇÕES

- Código **defun**
  - Sintáxe (defun nome(params) (retorno))
- Definindo uma função

```
> (defun foo (x y) (+ x y 5))
> (foo 5 0) ; chamando a função
```

- Função Recursiva

```
> (defun fatorial (x)
    (if (> x 0)
       (* x (fatorial (- x 1)))
        1
  ))
```

FUNÇÕES

- Funções Mutuamente Recursivas
- Funções chamam uma a outra e vice-versa em recursão

```
> (defun a (x) (if (= x 0) t (b (- x))))
> (defun b (x) (if (> x 0) (a (- x 1)) (a (+ x 1))))
```

- Função com múltiplos comandos em seu corpo

```
> (defun bar (x)
    (setq x (* x 3))
    (setq x (/ x 2))
    (+ x 4)
  )
```

FUNÇÕES

- Escopo de variáveis
  - Variáveis são colocadas no escopo de forma léxica
  - Se **foo** chama **bar** e **bar** tenta referenciar x, **bar** não obterá os valor de x de **foo**.
  - Associar um valor a um símbolo durante um certo escopo léxico é chamado em LISP de ateamento

  > (defun foo (x y) (+ x y 5))

  - x está atado ao escopo de foo

FUNÇÕES

- Número Variável de Argumentos para Funções
  - Qualquer argumento após o símbolo **&optional** é opcional

> (defun bar (x &optional y) (if y x 0))

  - Dois argumentos opcionais

> (defun baaz (&optional (x 3) (z 10)) (+ x z))

  - Os dois parâmetros são opcionais, mas possuem valor default para cada

FUNÇÕES

- Número Indefinido de Parâmetros
  - Terminando a sua lista de parâmetros com o parâmetro **&rest**
  - Todos os argumentos que não sejam contabilizados para algum argumento formal são armazenados em uma lista
  - (defun foo (x &rest y) y)
- Passagem de Parâmetros por Nome
  - Faz referencia a cada valor por uma chave

> (defun foo (&key x y) (cons x y))

> (foo :x 5 :y 3)

IMPRESSÃO

- Saídas mais complexas

  > (format t "An atom: ~S~%and a list: ~S~%and an integer:~D~%" nil (list 5) 6)

  - O primeiro argumento a format é ou t, ou NIL ou um arquivo;
  - T especifica que a saída deve ser dirigida para o terminal
  - NIL especifica que não deve ser impresso nada
  - Padrão de formatação semelhante a C
  - ~S = aceita qualquer objeto Lisp
  - ~D = só aceita inteiros
  - ~% = Quebra de linha
  - ~A = String
  - ~F = Float
  - ~X = hexadecimal

FORMS E O LAÇO TOP-LEVEL

- As coisas que você digita para o interpretador LISP são chamadas forms. O interpretador repetidamente lê um form, o avalia e imprime o resultado.
- Este procedimento é chamado **read-eval-print loop**.
- Alguns forms vão provocar erros. Após um erro, LISP vai pô-lo/la no ambiente do debugger.

FORMS E O LAÇO TOP-LEVEL

- Em geral, um form é ou um átomo (p.ex.: um símbolo, um inteiro ou um string) ou uma lista.
- Se o form for um átomo, LISP o avalia imediatamente. Símbolos avaliam para seu valor, inteiros e strings avaliam para si mesmos.
- Se o form for uma lista, LISP trata o seu primeiro elemento como o nome da função, avaliando os elementos restantes de forma recursiva. Então chama a função com os valores dos elementos restantes como argumentos.
- Por exemplo, se LISP vê o form (+ 3 4):
- Irá tratar + como o nome da função.
- Ele então avaliará 3 para obter 3, avaliará 4 para obter 4 e finalmente chamará + com 3 e 4 como argumentos.
- A função + retornará 7, que LISP então imprime.

FORMS E O LAÇO TOP-LEVEL

- O top-level loop provê algumas outras conveniências.
- Uma particularmente interessante é a habilidade de falar a respeito dos resultados de forms previamente digitados: LISP sempre salva os seus três resultados mais recentes. Ele os armazena sob os símbolos *, ** e ***.

FORMS ESPECIAIS

- Há um número de forms especiais que se parecem com chamadas a funções mas não o são. Um form muito útil é o form aspas. As aspas prevêm um argumento de ser avaliado.

> (setq a 3)

> (quote a)

- Outro form especial similar é o form function.
- Provoca que seu argumento seja interpretado como uma função ao invés de ser avaliado:

> (setq + 3)

> (function +)

> #'+

O form especial function é útil quando você deseja passar uma função como parâmetro para outra função.

BINDING - ATAMENTO/AMARRAÇÃO

- Binding é uma atribuição escopada lexicamente. Ela ocorre com as variáveis de uma lista de parâmetros de uma função sempre que a função é chamada: os parâmetros formais são atados aos parâmetros reais pela duração da chamada à função.
- Você pode também amarrar variáveis em qualquer parte de um programa com o form especial let:

```
(let ((var1 val1)
        (var2 val2)
        ...
       )
   body
  )
```

- Let ata var1 a val1, var2 a val2, e assim por diante; então executa os comandos de seu corpo.

BINDING - ATAMENTO/AMARRAÇÃO

- O corpo de um let segue as mesmas regras de um corpo de função:

```
> (let ((a 3)) (+ a 1))
> (let ((a 2)
        (b 3)
        (c 0))
    (setq c (+ a b))
    c
  )
> (setq c 4)
> (let ((c 5)) c)
```

A invés de (let ((a nil) (b nil)) ...)você pode escrever (let (a b) ...).

BINDING - ATAMENTO/AMARRAÇÃO

- Os valores val1, val2, etc. dentro de um let não podem referenciar as variáveis var1, var2, etc. que o let está atando:

```
> (let ((x 1)
        (y (+ x 1)))
    y)
```

Se o símbolo x já possui um valor?

- O form especial let* é semelhante, só que permite que sejam referenciadas variáveis não definidas anteriormente:

```
(let* ((x 1)
        (y (+ x 1)))
    y
 )
```

BINDING - ATAMENTO/AMARRAÇÃO

O form

```
(let* ((x a)
       (y b))
  ...
)
```

é equivalente a:

```
(let ((x a))
  (let ((y b))
    ...
) )
```

ARRAYS

- A função make-array faz um array.
- A função aref acessa seus elementos. Todos os elementos de um array são inicialmente setados para nil:

> (make-array '(3 3))

> (aref * 1 1)

> (make-array 4)

Índices de um array sempre começam em 0

> (setf (aref my-array 0) 25)

ARRAYS

- Percorrendo uma matriz

```
(dotimes (i 3)
        (dotimes (j 4)
                (setf (aref mat i j) 1)
        )
    )
```

STRINGS

- Um string é uma seqüência de caracteres entre aspas duplas. LISP representa um string como um array de tamanho variável de caracteres.
- Algumas funções para manipulação de strings:

```
> (concatenate 'string "abcd" "efg")
> (char "abc" 1)
> (aref "abc" 1)
```

ESTRUTURAS

- Estruturas LISP são análogas a structs em "C“

```
(defstruct book
   title
   author
   subject
   book-id
)
```

- Criando elemento pela estrutura

```
( setq book1 (make-book :title "C Programming"
   :author "Nuha Ali"
   :subject "C-Programming Tutorial"
   :book-id "478")
)
```

- Acessando

```
(book-title book1)
```

REMOVER UM ELEMENTO ESPECÍFICO

- (defun remove-nth (n list)

```
(declare
    (type (integer 0) n)
    (type list list))
  (if (or (zerop n) (null list))
    (cdr list)
    (cons (car list) (remove-nth (1- n) (cdr list)))))
```