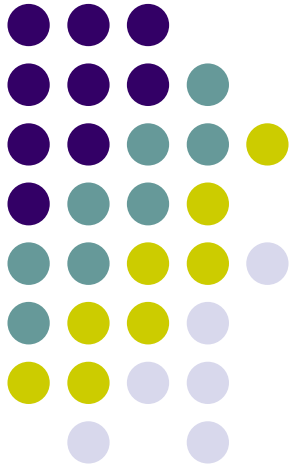
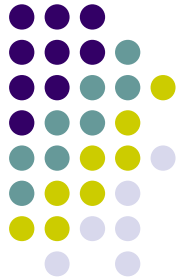
PLP
Programação Funcional

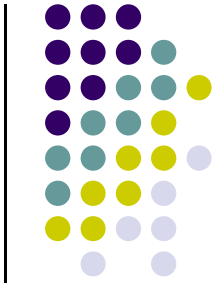
Prof. Salvador





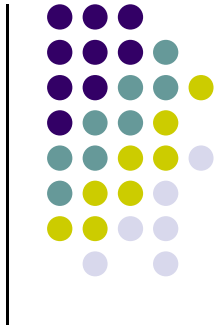
- **Programação Funcional**
- Emergiu como um paradigma distinto no início da década de 1960.
- Motivada pela necessidade dos pesquisadores de IA e seus subcampos: computação simbólica, prova de teoremas, sistemas baseados em regras e processamento de linguagem natural.
- Paradigma de programação baseado 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, Haskell, ...

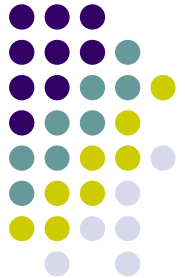
LISP



- Linguagem funcional original.

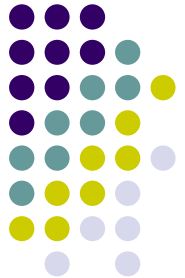
  Desenvolvida por John McCarthy e outros, em 1959.

- Palavras dos autores: “LISP tem sido usada para cálculos simbólicos em cálculo diferencial e integral, projeto de circuitos elétricos, lógica matemática, jogos e outros campos da inteligência artificial”.



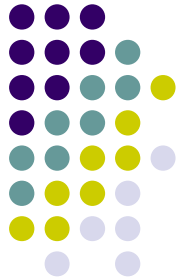
- Não há a noção de estado, logo, não há atribuição.
- É vista por alguns como um paradigma mais confiável para o desenvolvimento de software que a programação imperativa.
- É difícil de documentar: visões de solução da IA muitas vezes não são acessíveis a soluções imperativas.
- Baseada no cálculo lambda.

Por quê LISP?



- É uma linguagem cujos rudimentos podem ser explicados em poucas aulas
  - Poucas construções permitem fazer muito
- Permite ilustrar bem os princípios da recursão
  - Não vamos usar comandos de repetição
- Estruturas de dados complexas podem ser expressas de forma simples

Aplicações

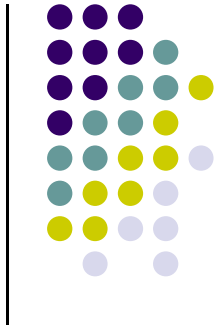


- Utilizado para desenvolver o primeiro sistema computacional de matématica simbólica, o MACSYMA.

- Linguagem de extensão do software do AutoCAD, desenvolvido pela AutoDesk; e do editor de textos Emacs

- Estudos de semântica formal das línguas naturais e de programação

Aplicações

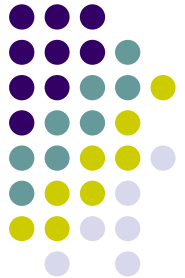


- Sistema de reserva de passagens Orbitz da ITA. Utilizado por diversas companhias aérias

- A Simbolics criou um sistema de modelagem 3D, adquirido pela IZWare e renomeado para Mirai, que foi utilizado nos efeitos de Senhor dos Anéis

- Sistema de e-commerce da viaweb (por Paul Graham) que posteriormente foi vendida para o Yahoo por US$ 40 milhões, na época da bolha da internet

Recursos



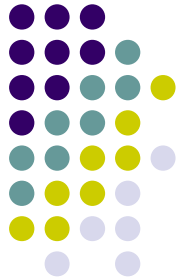
- Diversas implementações grátis
  - Recomendo a implementação de Common Lisp do Dr Bruno Haible
    - Ver http://www.clisp.org/
- Diversos livros, alguns grátis
  - Livros que pode ser consultados:

    Common LISP, an interactive approach

    por Stuart C. Shapiro

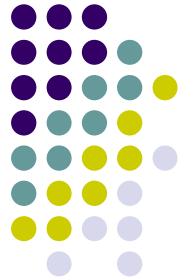
    ou Commom LISP the language. Guy L. Steele Jr.

Características do LISP



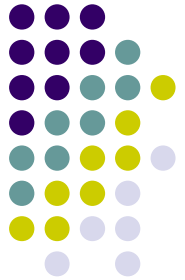
- Manipulação de informação simbólica
- Versão inicial do Lisp era pouco prática (sem iteração)
- Muitas versões e dialetos: Franz, Mac, Inter, Common (praticamente o padrão) e Scheme (variante “enxuta”)

Diferenças entre LISP e outras linguagens



- Programas e Dados têm a mesma forma
- Não se trabalha com variáveis armazenadas (embora LISP suporte variáveis)
  - Em “C++” podemos implementar uma função para inserir um elemento X numa variável L do “tipo” lista
  - Em LISP, escreve-se uma função que, dados um elemento X e uma lista L, retorna uma lista igual a L com X inserido

Usando LISP



- LISP é frequentemente implementada por um interpretador
  - Usuário entra com uma expressão
  - Interpretador avalia expressão e imprime o resultado
- Exemplo:

```
> (+ 3 4 )
7
```

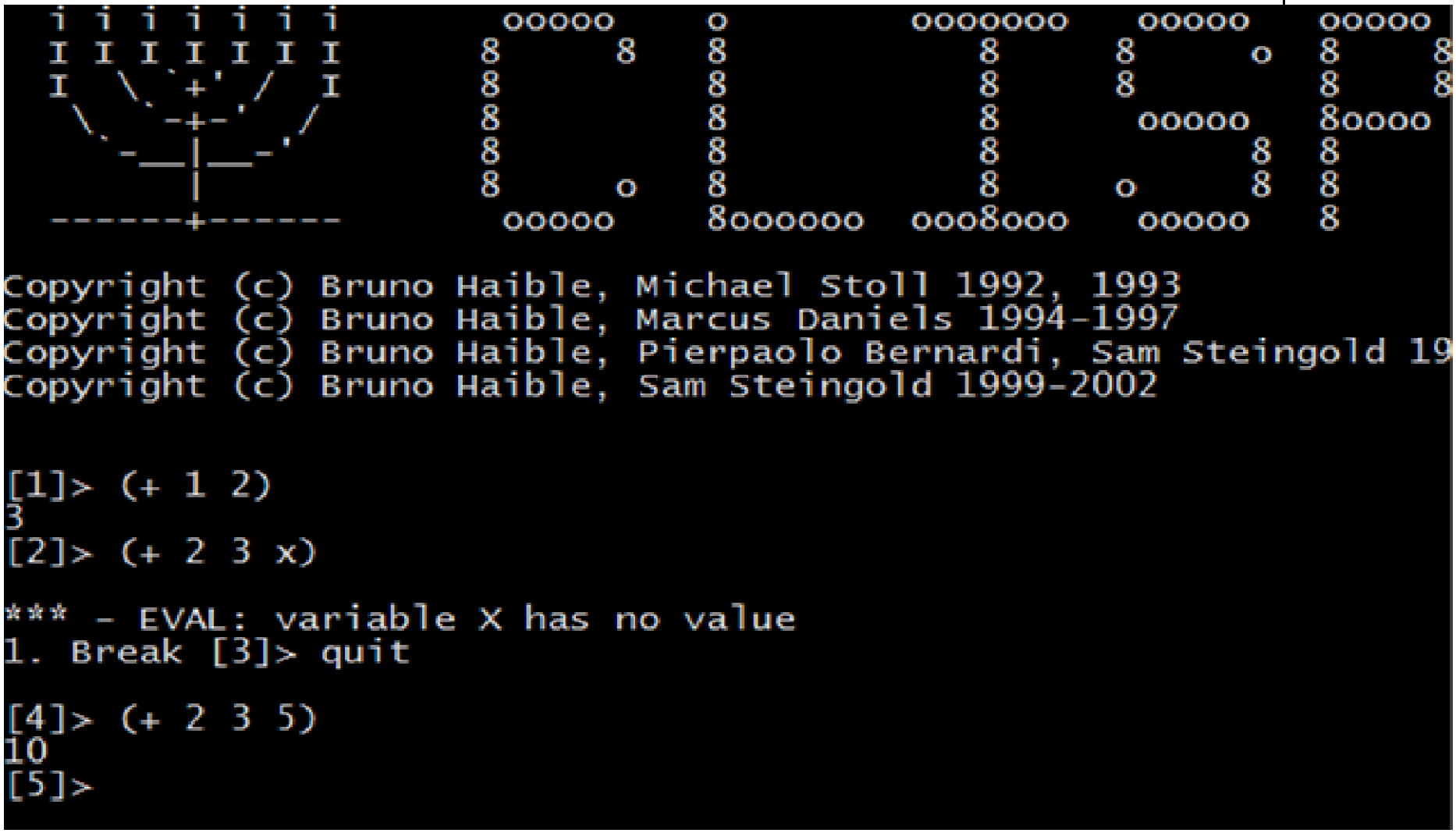
Usando o intepretador CLISP

- O interpretador numera os comandos
- Ao cometer um erro, o interpretador escreve uma mensagem de erro e entra num "*break loop*", isto é, num depurador (*debugger*)

- Se não for usar o depurador, digite `quit` para voltar ao modo de interação normal

- Para sair do programa, digite `(bye)`

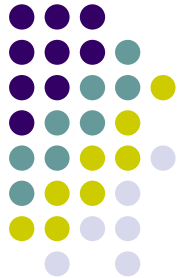
Exemplo de sessão com o CLISP



Tipos de dados do LISP

- **ÁTOMOS**
- São os elementos mais simples da linguagem
- Podem ser
  - Símbolos
    - a b c xxx x1 x-1
  - Constantes
    - Números: 1 2 1.33 -2.95
    - Cadeias: "abc de" "x y z"
- Um símbolo pode ser associado a um valor
  - Conceito semelhante ao de "variável" em linguagens imperativas

Listas

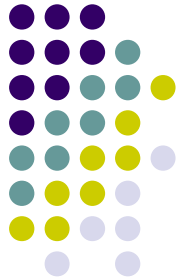


- Em LISP, se algo não é um átomo, então é uma lista
- Uma lista é uma sequência de átomos ou listas entre parênteses. Por exemplo:

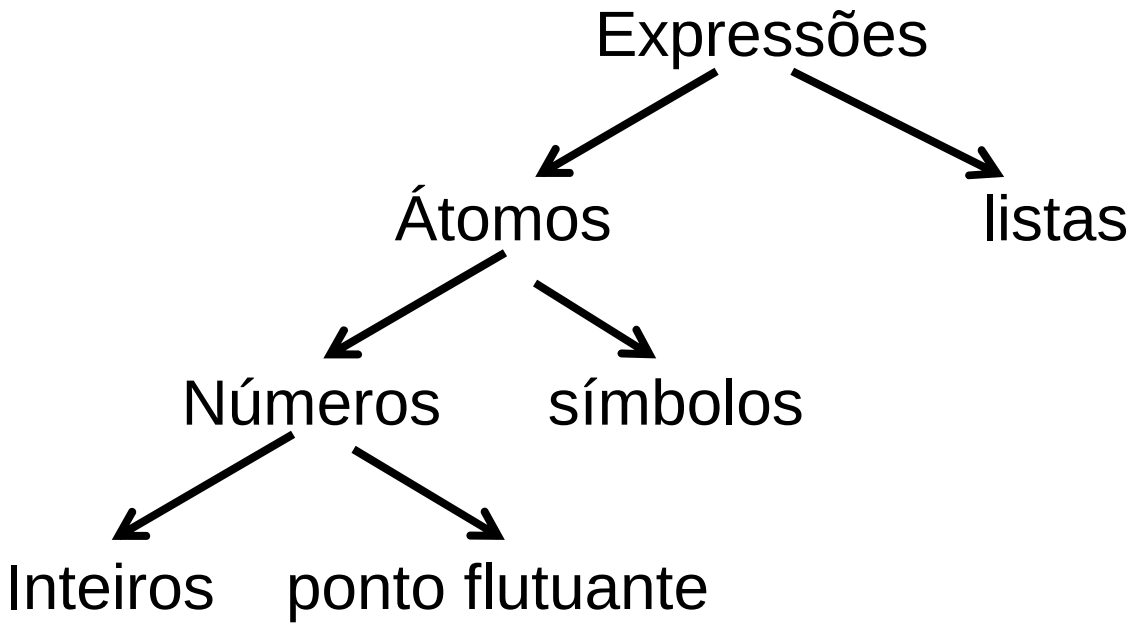
```
(a b c)       ; Lista com 3 elementos
(d (e f) g) ; Lista com 3 elementos
```

- Observe que os elementos das listas têm que estar separados por um ou mais espaços em branco
- Observe também que ponto-e-vírgula denota o início de um comentário em LISP

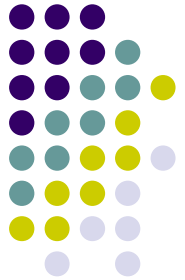
S-expressions



- Átomos ou listas são chamados de s-expressions, ou expressões simbólicas.

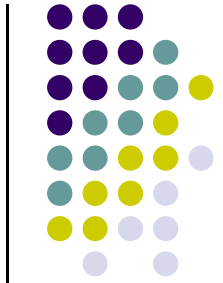


NIL e T



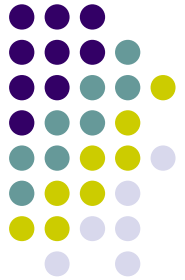
- Os símbolos nil e t são especiais pois seus valores são eles próprios
- Quando o LISP está interpretando uma expressão booleana, o átomo nil é usado para denotar o valor “falso”
- t denota o valor booleano “verdadeiro”, mas qualquer valor diferente de nil é entendido como verdadeiro

NIL e T - Exemplo



```
[1]> (<= 2 3)
T
[2]> (> 3 4)
NIL
[3]> (and 2 t)
T
[4]> (and 2 nil)
NIL
```

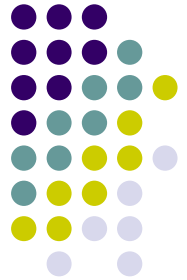
Avaliando Símbolos



- O interpretador sempre tenta avaliar símbolos a menos que sejam precedidos por um apóstrofo (*quote*)

```
> b
*** - EVAL: variable B has no value
> 'b
B
> nil
nil
> t
t
```

Avaliando Listas



- Assim como os símbolos, quando uma lista é apresentada ao interpretador, esta é entendida como uma função e avaliada, a menos que seja precedida por um apóstrofo

```
> (+ 1 2)
3
> '(+ 1 2)
(+ 1 2)
> (a b c)
*** - EVAL: undefined function A
```

Números

- Números inteiros têm uma particularidade em LISP: não há um limite de valor.

- Exemplo:

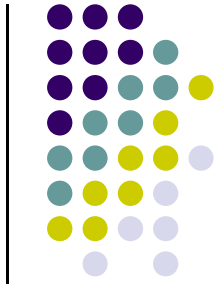
A multiplicação abaixo é feita normalmente.

> (* 5 99999888887777666665555544444333332222211111)

Resultando em:

499999444388888833332777772222166666111055555

Números



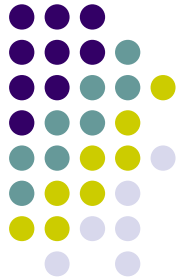
- Números reais podem ser especificados em LISP:
  - Em notação científica:

    0.27×10^{-5} pode ser escrito 0.27e-5
  - Usando como marcadores de expoente s, f, d, l (ou S, F, D, L), onde:
    - s = short-float
    - f = single-float
    - d = double-float
    - l = long-float

Exemplos

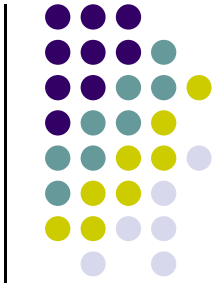
> Pi
3.14159265358979323851L0 (dado em long-float)
> (+ pi 0.004)
3.1455927
> (+ pi 4.0s-3)
3.1456s0
> (+ pi 4.0f-3)
3.1455927
> (+ pi 4.0d-3)
3.145592653589793d0
> (+ pi 4.0l-3)
3.14559265358979323861L0

Strings e caracteres



- Símbolos – os mais importantes tipos de objetos em LISP. Usados para:
  - variáveis de programas;
  - nomes de funções;
  - permitem programas LISP manipularem dados simbólicos.

Strings e caracteres



- Funções para manipular strings:
  - Tamanho: **length**.

    > (length “Isto é uma string”)

    17
  - Para obter um caractere particular da string: **char**

    Sintaxe: (char string índice)

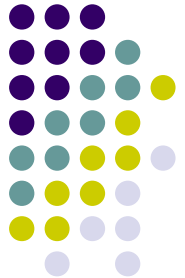
    Ex.: >(char “Isto e uma string” 1)

    #\s

    >(char “Isto e uma string” 5)

    #\e

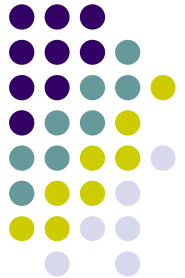
Strings e caracteres



- Para comparar dois caracteres, pode-se usar char=.
  - Exemplos:

> (char= (char “Uma string” 2) (char “outra” 4))
T
> (char= (char “Uma string” 2) #\a))
T
> (char= #\X #\X))
T
> (char= #\X #\x))
nil
> (char= “W” “W”)
Retorna um erro.

Símbolos



- A comparação se dois símbolos são iguais pode ser feita com a função eql.
  - Exemplos:

> (eql ‘salvador ‘SALVADOR)

T

> (eql ‘um ‘ dois)

NIL

> (eql 50 50.0)

NIL

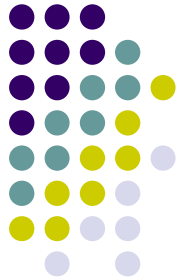
>(eql 5.0e1 50.0)

T

(eql #\a (char “uma” 2))

T

Pacotes



- Quando você está interagindo com o COMMOM-LISP, está em um pacote particular. Para saber qual é esse pacote, use *package*.

  >*package*

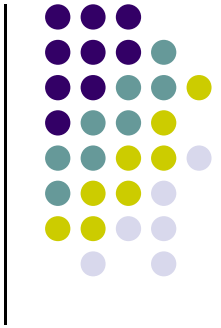
  #<PACKAGE COMMON-LISP-USER>

- Para ver a descrição de um símbolo j[a definido em seu pacote:

  > (describe ‘pi)

  3.14159265358979323 85L0 is a float with 64 bits of mantissa (ling-float).

Definindo Funções

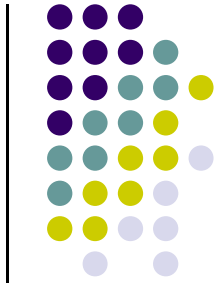


- A forma especial ***defun*** é usada para definir uma função

  (defun *nome lista-de-argumentos* doc-string *expressão*)

  - Define uma função chamada *nome* que avalia **expressão** substituindo os símbolos da **lista-de-argumentos** pelos valores passados quando a função for invocada.
  - Doc-string deve ser uma string.
- Quase não existe limitações para escrita de nomes. O nome de uma função, por exemplo, pode x+y-z.

Definindo Funções



- Exemplo

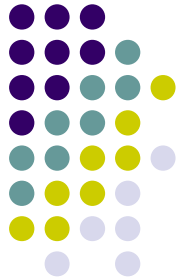
```
> (defun teste (a b)  "Duplica a soma dos valores informados"
                      (* 2 (+ a b)))
TESTE
```

Como se pode ver no exemplo acima, quando a função é escrita sem erro de sintaxe, o interpretador retorna o nome da função.

Outro exemplo, com nome não usual:

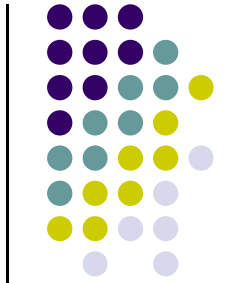
```
(defun a+b*c (a b c) (+  a  (*  b c)))
A+B*C
```

Funções



- O primeiro elemento de uma lista pode portanto denotar o nome de uma função
  - Nesse caso, os demais elementos são os argumentos da função
- Muitas funções são pré-definidas em LISP
- As seguintes são as que usaremos mais:
  - Aritmética: `+ - / *`
  - Relacionais: `> = < >= <=`
  - Lógicas: `and or not`
  - Manipulação de listas: `cons car cdr`
  - Condicionais: `if cond`

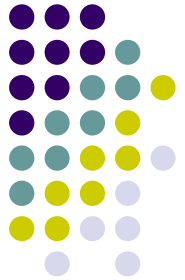
Funções básicas



- Exemplos:

```
>(+ 3 4)
7
>(+ 1 2 3 4 5)
10
>(-  8 5)
3
>(- 7 5 3 1)
-2
>(- -7 5 3 1)
-16
>(+ 3.1 2.7)
5.8
>(-  3.1  2.7)
0.39999986
>(*  4 3)
12
> (* 4 3.)
12
>(* 4  3.0)
12.0
>(* 3.1 7.0)
21.699999
>(* (+ 3 4) 8)
    56
```

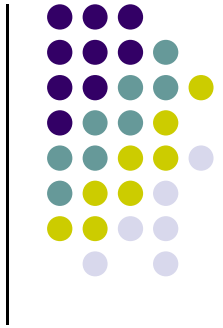
Funções básicas



- Exemplos:

```
> (/  8 2)
4

(/  3 2)
3/2

(/ 30  2  3)
5

(/  2) (inverso)
1/2

(* 3 (/  5  2))
15

(/ 3.0 2)
1.5
```

Funções básicas



- Raiz quadrada

  (sqrt 4) 2

  (sqrt 4.0) 2.0

  (sqrt -4) #C(0 2)

  Exponencial

  (expt 2 10) 1024

  (expt 2/3 3) 8/27

  (expt -4 ½) #C(0 2)

  Logaritmo

  (log 1) 0

  (log 10) 2.3025851

  Trunca

  (truncate 3.6) 3 ; 0.599999

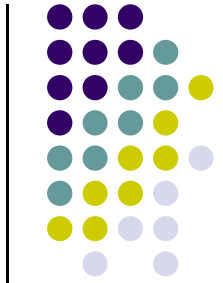
  (truncate -1.7) -1 ; -0.7000005

  Arredonda

  (round 3.7) 4 ; -0.2999995

  (round 3.2) 3 ; 0.20000005

Funções básicas



Valor absoluto

(abs 8) 8

(abs -8) 8

Resto

(rem 17 5) 2

(rem -18 5) -3

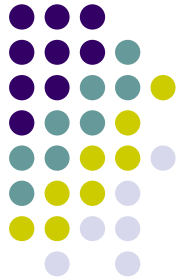
Float

(float 4) 4.0

pi

3.14159265358979323 85

Exercícios

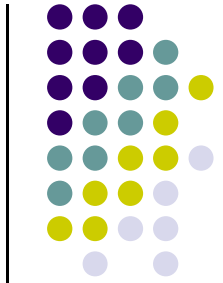


- Fazer uma expressão para calcular uma raiz da equação $2x^2 + 7x + 5 = 0$.

- Qual a resposta do Lisp para:

  12/8 (/ 12 8) (/ 12 8.) (/ 12 8.0)

Predicados
Testando números



setf – atribui valor a um objeto.
Ex.: (setf a 5)
(setf b 32)
Considerando a e b acima:
(>) – ordem decrescente
(> 100 10 a 1.0 0.5) → T

(<) – ordem crescente
(< 1.0 b 50 200) → T

(=) – igualdade
(= a (/ 20 4) 5) → T
(= a b) → NIL

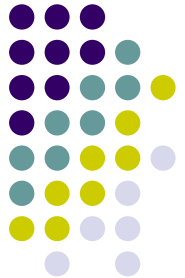
max – máximo
(max 1 -2 3 4 5) → 5

min – mínimo
(min 1 -2 3 4 5) → -2

evenp – par
(evenp (* 3 2)) → T
(evenp 7) → NIL

oddp – ímpar
(oddp (* 3 2)) → NIL
(oddp 7) → T

Testando números



minusp – negativo
(minusp 17) → NIL
(minusp -5) → T

zerop – zero
(zerop (*2 0)) → T
(zerop (+ 2 0)) → NIL

(numberp 5) → T

(integerp 5) → T

(floatp 5) → NIL

(characterp “a”) → NIL

(characterp #\a) → T

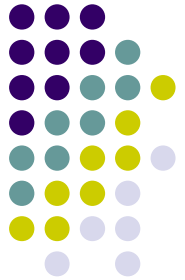
(stringp “a”) → T

(symbolp ‘\5) → T

(listp “ a list”) → NIL

(listp ‘(a list)) → T

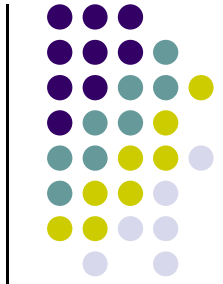
DOCUMENTATION



```
> (defun teste (a b)  "Duplica a soma dos valores
    informados"
                (* 2 (+ a b)))
TESTE

> (teste 3 4)
14
> (documentation 'teste 'function)
"Duplica a soma dos valores informados"
```

Google classroom: **vyltvg2**

Arquivos

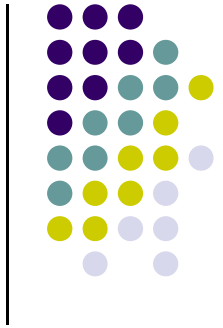


- No interpretador Common Lisp

  Um programa, ou uma coleção de funções podem ser armazenadas em um arquivo e processadas pelo interpretador.

  Um programa fonte pode ser armazenado em um arquivo com a extensão .lisp, outra extensão ou, até mesmo, sem extensão.

Arquivos



- Para carregar um arquivo:

(load “nomearq”)

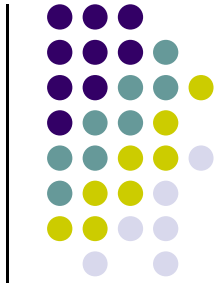
Caso “nomearq” possua a extensão .lisp, esta pode ou não ser digitada. O interpretador buscará o arquivo.

Ao carregar um arquivo o interpretador o lê e torna disponíveis todas as funções nele especificadas.

Expressões Lógicas

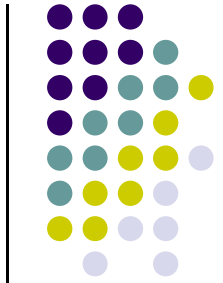
- São montadas com o auxílio das funções que implementam os predicados relacionais e lógicos tradicionais
  - Predicados lógicos: `and or not`
  - Predicados relacionais: `> = < >= <=`
    - Argumentos devem ser números
    - Para comparar símbolos usa-se o predicado *eq*
    - Para comparar conses estruturalmente usa-se o predicado *equal*

Expressões Lógicas - Exemplos



```
> (or (< 2 3) (> 2 3))
T
> (= 'a 'b)
*** - argument to = should be a number: A
> (eq 'a 'b)
NIL
> (eq 'a 'a)
T
> (eq '(a b) '(a b))
NIL
> (equal '(a b) '(a b))
T
```

Condicional simples



(if condição
    consequente
    alternativa)

Exemplo:

```
>  (if  (> 5 4)
          5
           4))
   5
```

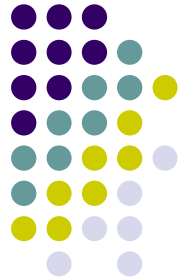
Cond -icionais

- A forma especial **cond** permite escrever funções que envolvem decisões
- Forma geral:

```
(cond       (bool1 expr1)
            (bool2 expr2)
            ...
            (boolN exprN)
)
```

- Funcionamento:
  - As expressões lógicas são avaliadas sucessivamente
  - Se **boolI** é verdadeira então o **cond** avalia e retorna **exprI**
  - Se nenhuma expressão lógica for avaliada como verdadeira, o **cond** retorna **nil**

cond - seleção múltipla



- O cond pode ser usado como um if-then-else:

```
(cond       (bool1   expr1)
            (bool2   expr2)
            (bool3   expr3)
            (t   expr4))
```

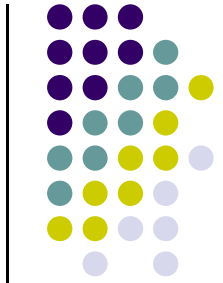
  - É equivalente à seguinte construção

```
if bool1   then   expr1
else if bool2   then   expr2
else if bool3   then   expr3
else   expr4
```

Exemplo

```
> (cond ((= 1 2) 'a)
        ((> 2 3) 'b)
        ((< 3 4) 'c)
)
C
> (defun f (lista elem)
         (cond ((eq lista nil) nil)
                  ((eq (car lista) elem) t)
                  (t (f (cdr lista) elem))
    )
)
F
> (f '(a b c) 'c)
T
> (f '(a b c) 'd)
NIL
```

Recursão



- Uma função recursiva terá a forma

(defun nome (lista_var) (cond condição))

Onde: condição deve especificar a parada da recursão.

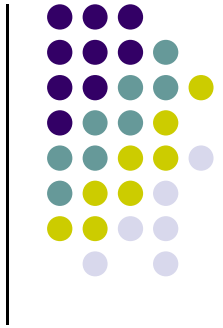
Ou a forma:

(defun nome (lista_var) (if teste se_verdade se_falso))

onde: se_verdade deve expressar a parada da recursão

se_falso deve ser uma expressão que contém a chamada recursiva.

Recursão



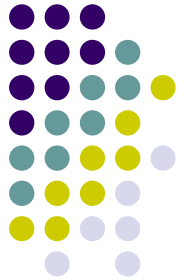
- Fatorial recursivo:

  fat(n)
  se n = 0 retorne 1
  senão
  retorne n * fat(n -1)

  Em LISP:

```
 (defun fat (n)
   (if (= n 0) 1 (* n (fat (- n 1)))))
```

Recursão



- Uma definição recorrente para a soma de dois inteiros:

  1. $s(0) = m$
  2. $s(n) = s(n-1) + 1$, para $n \geq 1$.

  Para realizar a soma de $m = 3$ com $n = 4$:

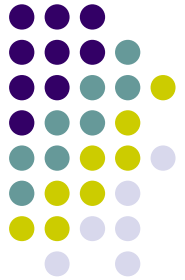
  $s(0) = 3$

  $s(1) = s(0) + 1 = 3 + 1 = 4$

  $s(2) = s(1) + 1 = 4 + 1 = 5$

  $s(3) = s(2) + 1 = 5 + 1 = 6$

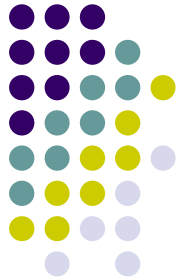
  $s(4) = s(3) + 1 = 6 + 1 = 7$



Definindo uma função recursiva para solucionar o problema:

```
(defun soma (m n)
        (if (= n 0) m
                (+ 1 (soma m (1-  n)))))
```

Trace



- É possível realizar o acompanhamento da execução de funções através da forma especial ***trace***.

  Ela recebe o nome da função que se pretende analisar e as altera de forma a mostrar a evolução da execução.

  Sintaxe: (trace f1 f2 .. fn)

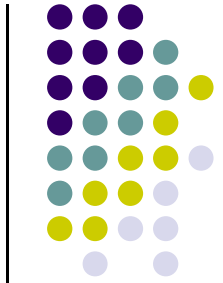
  Para encerrar a depuração de funções, usa-se a forma especial ***untrace***.

  Sintaxe: (untrace f1, f2, ...fn)

  (untrace) – sem argumentos – desativa todos os traces ativos.

  Obs.: *trace*, sem argumentos retorna NIL

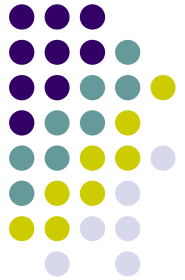
Operadores



- (1- n) é equivalente a (- n 1)
- (1+ n)é equivalente a (+ n 1)
- Porém as formas:

  (1 – n), (1 + n), (3- n), (3+ n), não são válidas.

Entendendo a recursão

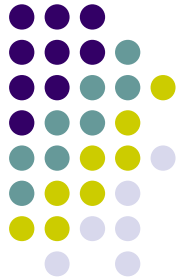


- Executar a função **fat**, para cálculo do fatorial, com o *trace* ativo.

- Fazer uma função (recursiva) em LISP, para calcular o somatório abaixo, dados ***a*** e ***b***:

$$\sum_{i=a}^{b} i^2$$

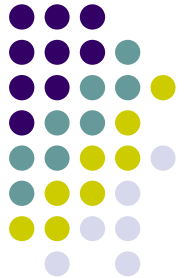
Entendendo a recursão



- Solução:

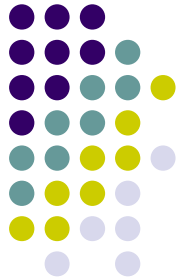
```
(defun quadrado (a) (* a a))

(defun soma-quadrados (a b)
        (if (> a b)
        0
        (+ (quadrado a) (soma-quadrados (1+ a) b))))
```

Exercício



- Fazer uma função ***potencia***, em LISP, para receber a base e o expoente e retornar o valor da base elevada ao expoente.

setq



- Uma forma de atribuir um valor não numérico a um átomo LISP:

  (setq argu1 valor1 arg2 valor2 ...)

  Ex.:

  >(setq l1 ‘(a b c))

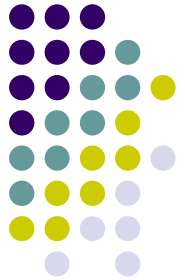
  (A B C)

  >(setq l2 ‘(d e) l3 5)

  5

  Entretanto, l2 está associado a ‘(d e).

Entrada



- A função de leitura do teclado e ***read***.

  Apenas recebe dados. Não avalia a entrada.

  Geralmente, vem combinada com outras funções.

  Digitar um valor e, a seguir, *ENTER*.

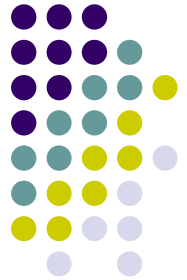
Exemplo:

```
(defun soma ()
   (setf x (read)) (setf y (read))
   (+ x y))
```

Execução:

```
> (soma)
5     (1º val digitado)
6     (2º val digitado)
11
```

Entrada

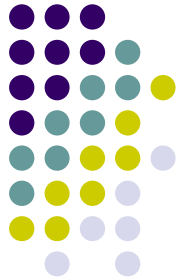


- Outro exemplo:

```
>(defun media ()
        (/   (+ (read) (read) (read) (read)) 4))
```

Após chamar a função (media), digita-se 4 valores. Retornará a média aritmética.

Saída



- Para saída de dados, existem algumas funções. Uma delas é ***print***.
  - Sintaxe: (print argumento)
- Imprime o argumento e o retorna. Observar que o argumento é obrigatório.

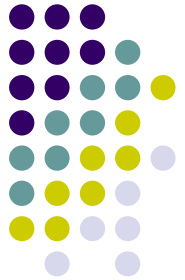
  Exemplo:

  (print 3)

  3

  3

Saída



- Outra função para saída de dados é:

(format arg1 arg2, ... argn)

Onde:

arg1 – o primeiro argumento pode ser:

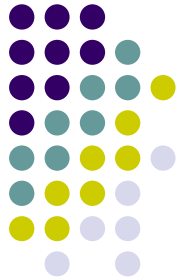
t, NIL ou uma referência a arquivo.

t – especifica a saída para o terminal

NIL – não deve haver saída, ou

referência ao arquivo de saída.

Saída



(format arg1 arg2, ... argn)

Onde:

arg2 – o segundo argumento é uma string, um template de formatação, contendo cadeias de caracteres e diretivas de formatação.

Algumas diretivas de formatação

~D – dá saida em valores numéricos em sistema decimal

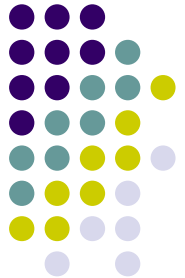
~S – para imprimir um átomo qualquer

~% – quebra de linha

Existem diretivas para saída em binário, octal, hexadecimal, ponto flutuante.

Além de várias outras.

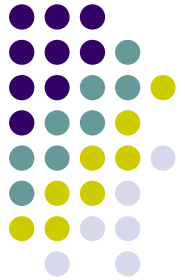
Saída



- Exemplo de saída usando format:

```
>(setq a 20)
>(format t “Um valor numérico ~D~%Um átomo ~S~%Base
    octal ~O” a ´( a b) a)
Um valor numerico 10
Um atomo ´(A B)
Base octal 12
```

Cons- truindo Listas



- (cons elem lista) retorna uma cópia de lista com elem inserido como seu primeiro elemento
- Exemplo:

  > (cons 'a '(b c))

  (A B C)

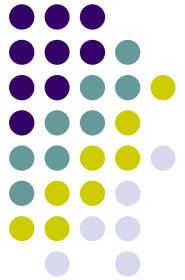
  > (cons 'a '(b))

  (A B)

  > (cons 'a nil)

  (A)
- Teorema: (cons (car L) (cdr L)) = L

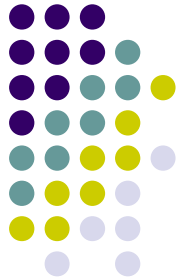
Cons



Exemplos:

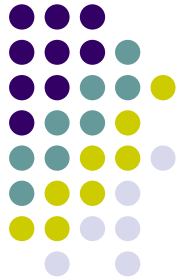
```
>(cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)

>(cons 1 (cons 2 (cons 3 (cons 4 ´(5)))))
(1 2 3 4 5)
```

Cons



- Um `cons` nada mais é que um registro com dois campos, o primeiro é chamado de `car` e o segundo de `cdr`
- A regra do ponto:
  - O `cons` é escrito com o valor dos dois campos entre parênteses ou separados por um ponto
  - *Entretanto*, se o campo `cdr` é `nil` ou um `cons`, o ponto pode ser omitido

Cons



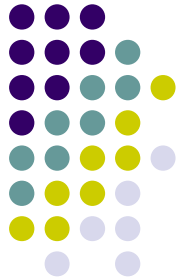
- Quando passamos um átomo como segundo argumento de `cons`,

  `> (cons 'a 'b)`

  o sistema responde com:

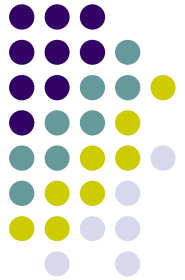
  `(A . B)`

Conses – Exemplos



```
> (cons 'a 'b)
(A . B)
> '(a . b)
(A . B)
> '(a . nil)
(A)
> '(a . (b . (c . nil)))
(A B C)
> '(a . (b . c))
(A B . C)
> '((a . b) . c)
((A . B) . C)
> '((a . b) . (b . c))
((A . B) B . C)
```

Outros construtores de listas



- list

  >(list 1 2 3 4 5)

  (1 2 3 4 5)

- make-list (make-list quant : initial-elem elem)

  onde:

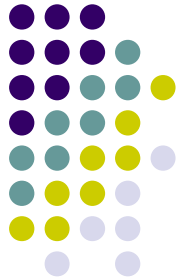
  quant = quantidade de elementos da lista

  elem = elemento que será repetido na lista

  >(make-list 7 : initial-elem 4)

  (4 4 4 4 4 4 4)

Examinando Listas

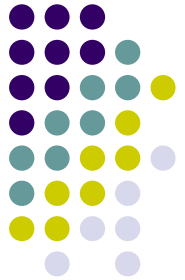


- (car lista) retorna o primeiro elemento de lista
  - Um sinônimo de car é first
  - CAR = Contents of Address Register
- (cdr lista) retorna a lista sem o seu primeiro elemento
  - Um sinônimo de cdr é rest
  - CDR= Contents of Decrement Register
- Uma lista vazia () também pode ser escrita como nil
  - nil é tanto um átomo como uma lista!

Examinando Listas - Exemplo

```
> (car '(a b))
A
> (cdr '(a b))
(B)
> (car (cdr '(a b)))
B
> (cdr (cdr '(a b)))
NIL
> (first (rest '(1 2 3)))
(2 )
```

Funções úteis



- Para testar se duas listas são iguais:

  > (equal ‘(a (b c) d) ‘(a (b c) d))

  T

  > (equal ‘(a (b c) d) ‘(a b c d))

  NIL

- Tamanho da lista:

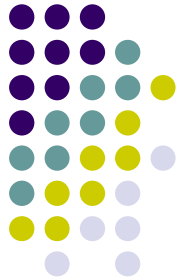
  > (length ‘(a (b c) d))

  3

  > (length ‘(a b c d))

  4

Funções úteis



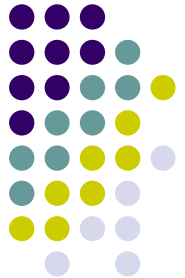
- Para obter o elemento da enésima posição em uma lista, pode-se usar a função nth.

  (nth posição lista)

  Obs.: o primeiro elemento está na posição 0.

  >(nth 3 ‘(a b c d e f))
  D

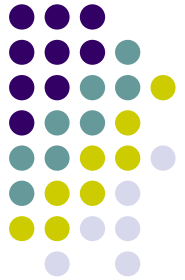
Exercícios



- Construir uma função ***tamanho***, que retorna o tamanho de uma lista dada.

- Construir uma função ***posição***, para receber um elemento e uma lista que contém esse elemento e retornar a posição do elemento na lista.

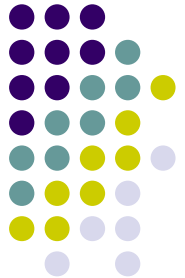
Funções úteis



- Append – concatena duas ou mais listas

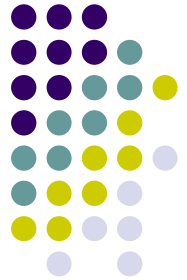
```
>(append ‘(1 2 3)  ‘(a b c) ‘(X Y))
(1 2 3 A B C X Y)
```

Exercício



- Fazer um programa em LISP para receber duas listas como argumentos e retornar uma terceira lista que representa a concatenação das listas dadas.

Funções úteis



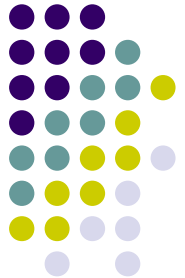
- Para inverter os elementos de uma lista, pode-se usar a função ***reverse***.

Ex.:

>(reverse ‘(1 2 3 4))

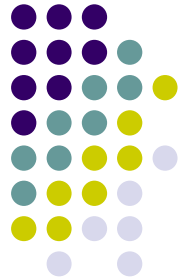
(4 3 2 1)

Exercício



- Fazer uma função ***inverte***, em LISP, que recebe uma lista e devolve a lista recebida invertida.

Funções úteis



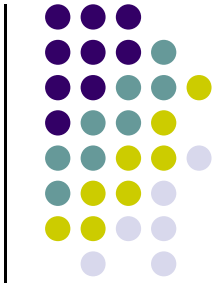
- Dados uma lista e dois elementos, a função ***subst***, ao encontrar o segundo elemento substitui pelo primeiro.

Exemplo:

> (subst ‘d ‘c ‘(a b c a c)

(A B D A D)

Funções úteis



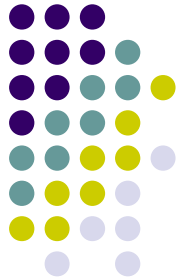
- A função remove-duplicates, remove todos os elementos duplicados em uma lista.

Ex.:

>(remove-duplicates ‘(a b c a b d a d e))

(C B A D E)

Assert



Uma forma de fazer uma crítica de valores de entrada em uma função é usar ***assert***.

Ex.: uma função requer como parâmetro de entrada uma lista, ou o valor de uma variável deve ser maior que o de outra, etc.

Sintaxe:

(*assert* asserção (variável(is)_a ser(em) mudada(s))
string)

Exemplo:

```
(defun t5(e f) (assert (< e f)(e f) "e deve ser menor que f”)
                (- f  e))
```

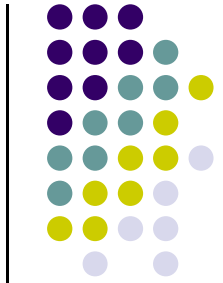
Dotimes

(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
```

Dolist



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

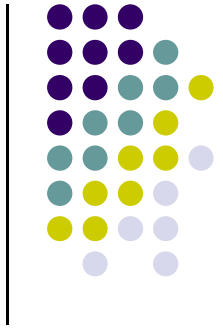
O corpo do loop é executado uma vez para cada valor de

variável, sendo que esta 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 (x '(1 2 3)) (print x) (if (evenp x) (return)))
1
2
NIL
```

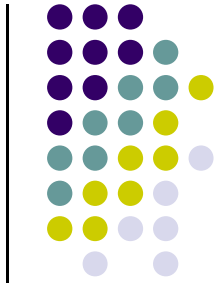
Loop



- LISP permite o uso do loop

```
> (setq a 4)
4
> (loop (setq a (+ a 1))
            (when (> a 7) (return a))
   )
8
> (loop (setq a (- a 1))
            (when (< a 3) (return))
   )
NIL
```

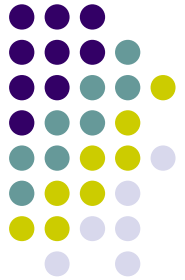
Um ambiente de desenvolvimento



- EMACS – Um editor de textos

- SLIME – The Superior Lisp Interaction Mode for Emacs

IDE Lispbox



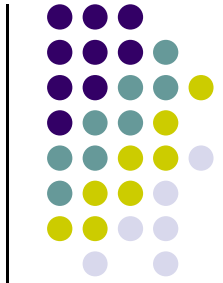
- IDE Lispbox - baixar de:
  https://common-lisp.net/project/lispbox/

Para executar:

Descompactar o pacote

Executar lispbox.bat

Lispbox



Ao executar o arquivo lispbox, abrirá uma janela maior, onde o SLIME está disponível e uma janelinha inferior.

Na janela do SLIME, pode-se executar o LISP.

Clicando na janelinha inferior, abrirá outra janela maior.

Posicionando nesse janela que se abre, pode-se abrir ou criar arquivos.

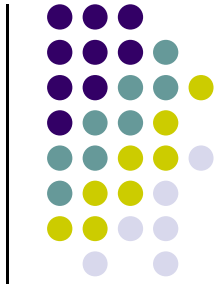
Após digitar seu programa no arquivo, acessar no menu superior:

SLIME >compilation>compile/load file

Na outra janela, executar as funções do arquivo.

Caso abra o arquivo na janela do SLIME, poderá abri-lo na outra janela com: ALT x slime

Exercícios



- Escreva as funções
  - (apaga L X)
    - Dada uma lista L e um elemento X , retorna L sem X. Se L não contém elem inicialmente, retorna uma cópia exata de L

```
> (apaga '(a b c d a) 'a)
(b c d)
```

  - (acresc L X Y)
    - Dada uma lista L, um elemento X e um elemento Y, retorna uma cópia de L onde um Y é inserido depois de cada X

```
> (acresc '(a b c d a) 'a 'k)
(a k b c d a k)
```