



Linguagem Funcional

Instituto Federal de Educação, Ciência e Tecnologia do Triângulo Mineiro
Prof. Edwar Saliba Júnior
Setembro de 2021



Introdução

- Em ciência da computação, programação funcional é um paradigma de programação que trata a computação como uma avaliação de funções matemáticas e que evita estados ou dados mutáveis;
- ela enfatiza a aplicação de funções, em contraste da programação imperativa, que enfatiza mudanças no estado do programa;
- enfatizando as expressões ao invés de comandos, as expressões são utilizadas para cálculo de valores com dados imutáveis.



Introdução

- Uma função pode ter ou não ter parâmetros e um simples valor de retorno;
 - > pi
3.1415926535897932385
- os parâmetros são os valores de entrada da função e o valor de retorno é o resultado da função;
 - > (+ 3 4)
7
- a definição de uma função descreve como a função será avaliada em termos de outras funções.



LISP - Curiosidades

- Linguagem de programação originalmente projetada por John McCarthy em 1958;
- seu nome “**LISP**” é derivado de “**LI**St **P**rocessor”;
- é a segunda linguagem de programação de alto nível, mais velha, ainda utilizada em larga escala pelo mercado. A primeira é Fortran;
- “listas ligadas” são uma das suas principais estruturas de dados e o código-fonte da linguagem é feito de listas.



Cálculo Lambda

- A linguagem Lisp foi inspirada pelo *Cálculo Lambda* (CL), que é um formalismo desenvolvido na década de 1930 por Alonso Church;
- o CL é parte de um sistema para lógicas de ordem superior e teoria das funções;
- o CL pode ser considerado uma linguagem de programação abstrata.



Cálculo Lambda

- **Notação:**

- considere a expressão matemática:

$$x + y$$

- esta expressão pode ser interpretada como uma função de x , como uma função de y ou como uma função de ambas:

$$f_1 : x \mapsto x - y$$

$$f_2 : y \mapsto x - y$$

$$f_3 : x, y \mapsto x - y$$

$$f_4 : y, x \mapsto x - y$$



Cálculo Lambda

- **Notação:**

- para eliminar a ambiguidade, em geral, são adotadas notações como, por exemplo, dar um nome arbitrário a cada função;
- no CL Church propõe uma maneira sistemática para apresentar uma função de uma ou mais variáveis, dada uma expressão matemática qualquer, sem a necessidade de se atribuir um nome a função. Isto é feito utilizando a letra grega λ :

$$f_1 : x \mapsto x - y \quad \Rightarrow \quad \lambda x. x - y$$

$$f_2 : y \mapsto x - y \quad \Rightarrow \quad \lambda y. x - y$$

$$f_3 : x, y \mapsto x - y \quad \Rightarrow \quad \lambda x y. x - y$$

$$f_4 : y, x \mapsto x - y \quad \Rightarrow \quad \lambda y x. x - y$$



Cálculo Lambda

- **Exemplos:**

- as equações:

$$f_1(3) = 3 - y \qquad f_3(5,7) = -2$$

- são escritas na notação lambda como:

$$(\lambda x.x - y)(3) = 3 - y \qquad (\lambda xy.x - y)(5,7) = -2$$

- usando o fato de serem permitidas funções cujos valores são outras funções, pode-se simplificar a notação utilizada para as funções f_3 e f_4 eliminando a necessidade de notações específicas para funções de duas ou mais variáveis:

$$f_3 = \lambda x.f_2 = \lambda x.(\lambda y.x - y) \qquad f_4 = \lambda y.f_1 = \lambda y.(\lambda x.x - y)$$

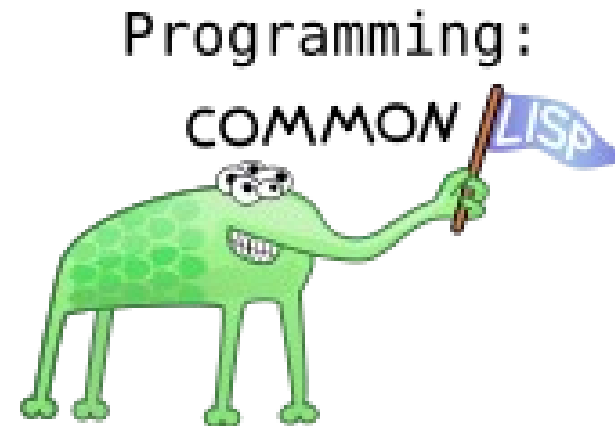
- a equação $f_3(5,7) = -2$ agora é escrita como:

$$(\lambda x.(\lambda y.x - y))(5)(7) = (\lambda y.5 - y)(7) = -2$$

- esta nova notação permite que apenas funções de uma variável sejam consideradas, simplificando a apresentação da linguagem.

Só para lembrar!

$$\begin{aligned} f_1 : x \mapsto x - y &\Rightarrow \lambda x.x - y \\ f_2 : y \mapsto x - y &\Rightarrow \lambda y.x - y \\ f_3 : x, y \mapsto x - y &\Rightarrow \lambda xy.x - y \\ f_4 : y, x \mapsto x - y &\Rightarrow \lambda yx.x - y \end{aligned}$$



Wikibooks. **Common Lisp**. Disponível em: <https://en.wikibooks.org/wiki/Common_Lisp>. Acesso em: 19 Ago. 2021.



Common Lisp

- Características:
 - é uma linguagem de programação de alto nível, *case-insensitive*, de notação prefixada, com gerenciamento de alocação de memória dinâmico (possui *Garbage Collection*);
 - é uma linguagem multiparadigma. Permite a escrita de programas nos paradigmas:
 - imperativo,
 - funcional e
 - orientado a objeto.



Common Lisp

- Características:
 - os paradigmas podem ser misturados livremente na programação com Common Lisp;
 - a linguagem permite a escolha da abordagem e do paradigma adequado, de acordo com o domínio do problema;
 - é uma linguagem “parentisada” e de notação prefixada. Significando que:
 - a criação de uma função “func” com argumento X e “hello” é escrita assim:

```
(func x "hello")
```
 - ao invés de:

```
func(x, "hello")
```
 - como seria feito numa linguagem imperativa;



Common Lisp

- Características:
 - possibilidade de extensão da sintaxe da linguagem por meio da criação de macros. Pode-se usar macros para a criação de novas “palavras reservadas” (comandos) na linguagem;
 - algumas implementações permitem que um programa seja digitado em um “prompt” ou carregado de um arquivo;
 - permite compilação incremental;
 - algumas implementações permitem compilar para código nativo ou para *bytecode*. Códigos compilados e interpretados podem ser misturados facilmente e
 - pode prover códigos eficientes e de alta performance.



Instalação

- Em GNU/Linux Debian:

```
sudo apt-get install clisp
```

- para Windows:

- <https://portacle.github.io/>

- ou

- <https://www.cs.utexas.edu/users/novak/gclwin.html>

- etc.



Comandos Básicos do Ambiente

- Para entrar no interpretador: `clisp`
- para sair do interpretador: `(quit)`
- para usar o interpretador: digite as expressões e tecle `ENTER`
- para repetir a última expressão: `*`
- para repetir a penúltima expressão: `**`
- para repetir a antepenúltima expressão: `***`
- se ocorrer um erro: tecle `:rN` e volte ao nível inicial (onde `:rN` é a opção listada como `ABORT` no erro ocorrido)
- para usar com arquivo: `clisp < arquivo.lsp` (o `arquivo.lsp` deve ser um arquivo com expressões a serem avaliadas).



A linguagem

- Ao iniciar a **Common Lisp** você verá uma tela similar a esta:

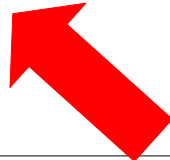
```
edwar@alpha: ~
File Edit View Search Terminal Help
edwar@alpha:~$ clisp
i i i i i i i      00000  0      0000000  00000  00000
I I I I I I I      8      8  8      8      8  0  8  8
I \ \ '+ ' / I      8      8  8      8  8      8  8
  \ \ -+ - /      8      8  8      00000  80000
   \ \ -+ - /      8      8  8      8  8
  \ \ -+ - /      8  0  8      8  0  8  8
   \ \ -+ - /      00000  8000000  0008000  00000  8
-----+-----

Welcome to GNU CLISP 2.49.92 (2018-02-18) <http://clisp.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992-1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2018

Type :h and hit Enter for context help.

[1]> █
```





Hello world!

- Common Lisp (CL)

The CL Hello World program reads as follows:

```
(format t "Hello World!")
```

Which outputs:

```
Hello World!
```

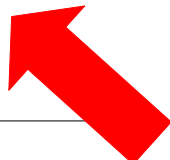
```
~/Documents/Private/Eddie/Empresas/IFTM/Disciplinas/Superior/Disc_CE/M... - □ ×
View Search Terminal Help
~/Documents/Private/Eddie/Empresas/IFTM/Disciplinas/Superior/Disc_CE
~/LISP$ clisp
  i i i      00000  0      0000000  00000  00000
 I I I      8 8 8      8 8 0 8 8
 / I      8 8 8      8 8 8 8
.' /      8 8 8      8 00000 80000
-' /      8 8 8      8 8 8
_-' /      8 0 8      8 0 8 8
-----  00000  8000000 0008000  00000  8

Welcome to GNU CLISP 2.49.92 (2018-02-18) <http://clisp.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992-1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2018

Type :h and hit Enter for context help.

[1]> (format t "Hello world")
Hello world
NIL
[2]> █
```





Compilando o Código-fonte

- Abra o editor de texto de sua preferência e digite o código-fonte a seguir:

```
(in-package :user)
```

```
(defun hello ()
```

```
  (write-string "Hello, World!")
```

```
)
```

- então salve o arquivo com o nome de `hello.lsp`
- abra o interpretador de Lisp.



Compilando o Código-fonte

- continuação:
 - execute o comando:

```
> (compile-file "/caminho/hello.lsp")
```
 - serão criados mais dois arquivos:
 - hello.lib e
 - hello.fas
 - no interpretador de Lisp digite:

```
> (load "/caminho/hello")  
T
```
 - agora você pode executar o comando:

```
> (hello)  
Hello, World!  
"Hello, World!"
```



Compilando o Código-fonte

- continuação:
 - Qual a diferença de executarmos a função (abaixo) diretamente no interpretador de Lisp ou a função compilada que carregamos no interpretador?

```
(in-package :user)
(defun hello ()
  (write-string "Hello, World!"))
)
```



Arquivo `.clinit.cl`

- O arquivo inicial de usuário, `.clinit.cl`, é procurado pela Lisp sempre que uma execução é iniciada;
- quando o arquivo é encontrado, então ele é carregado na memória e os comandos, nele contidos, são avaliados pelo interpretador Lisp;
- o arquivo “init” pode ser usado para definir bibliotecas de funções que são frequentemente utilizadas ou para controle de outros aspectos do ambiente da linguagem Lisp.
- Exemplo de arquivo “init”:

```
(in-package :user)

(defun load-project ()
  (load "~/lisp/part1")
  (load "~/lisp/part2")
)
```



Comentário

- Tudo que estiver após um “;” (ponto e vírgula).

- Exemplo:

```
> (setq a 5) ; armazena o valor em “a”  
5
```

```
> a ; mostra o valor armazenado em “a”  
5
```



Quebra de linha

- Para quebrar a linha na impressão de texto na tela do computador, basta utilizar `~%` dentro do texto a ser dividido.
- Exemplo:

```
> (format t "Hello world!~%meu nome é
Edwar")

Hello world!
meu nome é Edwar
```

```
[1]> (format t "Hello world!~%meu nome é Edwar")
Hello world!
meu nome é Edwar
NIL
[2]> █
```



Símbolos

- Um símbolo é somente uma *string*.

- Exemplos:

a

b

c1

faa

bar



Lendo Valores do Teclado

- Para lermos um valor do teclado usamos a função `(read)`.
- Exemplo:

```
> (princ "Digite um número:")
```

```
Digite um número:
```

```
"Digite um número:"
```

```
> (setq valor (read))
```

```
18
```

```
18
```

```
> valor
```

```
18
```

```
[7]> (princ "Digite um número:")  
Digite um número:  
"Digite um número:"  
[8]> (setq valor (read))  
18  
18  
[9]> valor  
18  
[10]> █
```




Valores Lógicos

- Lisp utiliza:

t = true e

NIL = false

Símbolos autoavaliantes.

- Exemplos:

```
> (if t 5 6)
```

```
5
```

```
> (if NIL 5 6)
```

```
6
```

```
> (if 4 5 6)
```

```
5
```

Este exemplo pode, inicialmente, parecer meio estranho. Mas, não é!

Na verdade o símbolo NIL significa “falso” e qualquer coisa diferente de NIL é verdadeiro, usa-se o “t” somente para clareza.



Números

- Um inteiro é um *string* de dígitos opcionalmente precedido do sinal de + ou -
- um número real é similar ao inteiro, porém possui um ponto decimal e, opcionalmente, pode ser escrito em notação científica;
- um número racional é escrito com uma barra “/” entre eles (simbolizando a divisão);
- LISP suporta números complexos, que são escritos assim $\#c(r\ i)$
- Exemplos:

-34 +6 18 2.59 3/4 1.95e-12 $\#c(2.15\ 0.69)$



Tipos

- Lisp é uma linguagem onde as variáveis não têm tipo, mas os dados sim.
- Exemplos:

```
array, atom, base-character, bignum, bit, bit-  
vector, character, compiled-function, complex, cons,  
double-float, extended-character, fixnum, float,  
function, hash-table, integer, keyword, list, long-  
float, nil, null, number, package, pathname,  
random-state, ratio, rational, readtable, sequence,  
short-float, signed-byte, simple-array, simple-bit-  
vector, simple-string, simple-vector, single-float,  
standard-char, stream, string, symbol, t, unsigned-  
byte e vector
```
- além destes, podem ser criados outros pelo usuário.



Hierarquia Parcial de Tipos

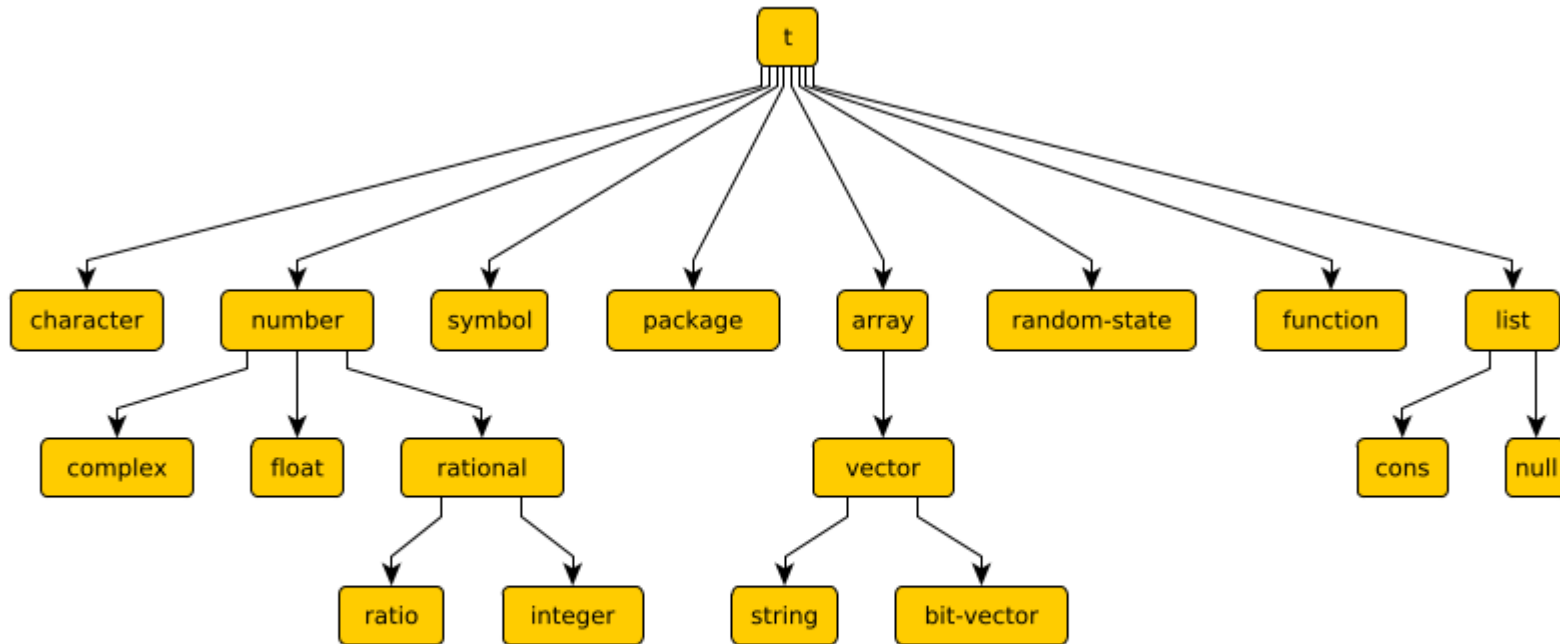


Figura 2.1: Hierarquia parcial de tipos em Lisp.



Múltiplos Argumentos

- As funções $+$ $-$ $/$ e $*$ aceitam múltiplos argumentos;
- Exemplos:

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

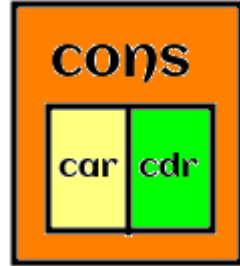
```
10
```

```
> (+ 5 4 3 (* 3 2))
```

```
18
```



cons(es) ou Pares-com-pontos ou Associações



- Um cons é somente **um registro de dois campos**;
- os campos são chamados de `car` e `cdr`, por razões históricas: na primeira máquina onde LISP foi implementada havia duas instruções *assembly* CAR (*Contents of Address Register*) e CDR (*Contents of Decrement Register*);
- os conses foram implementados utilizando-se estes dois registradores;
- Exemplos:

```
> (cons 4 5) ; aloca a car o valor 4 e a cdr o 5
```



cons(es) ou Pares-com-ponto ou Associações

- Exemplos:

```
edwar@delta: ~  
File Edit View Search Terminal Help  
  \  \--+ ' /      8      8      8      00000  80000  
   \  -+ | -'     8      8      8      8  8  
    |  | | |     8      0  8      8      0  8  8  
   -----+----- 00000  8000000  0008000  00000  8  
  
Welcome to GNU CLISP 2.49.92 (2018-02-18) <http://clisp.org/>  
  
Copyright (c) Bruno Haible, Michael Stoll 1992-1993  
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997  
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998  
Copyright (c) Bruno Haible, Sam Steingold 1999-2000  
Copyright (c) Sam Steingold, Bruno Haible 2001-2018  
  
Type :h and hit Enter for context help.  
  
[1]> (cons 4 5)  
(4 . 5)  
[2]> (cons (cons 4 5) 6)  
((4 . 5) . 6)  
[3]> (car (cons 4 5))  
4  
[4]> (cdr (cons 4 5))  
5  
[5]> □
```




Listas

- Você pode construir muitas estruturas de dados a partir de cons. A mais simples é a lista encadeada:
 - o `car` de cada cons aponta para um dos elementos da lista e
 - o `cdr` aponta para outro cons ou para `NIL`
- uma lista pode ser criada com a função de lista:

```
> (list 4 5 6)
(4 5 6)
```



Listas

- **Regras:**

- se o `cdr` de um `cons` é `nil`, Lisp não se preocupa em imprimir o ponto ou o `nil`:

```
> (cons 4 nil)
```

```
(4)
```

- 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 (cons 5 6))
```

```
(4 5 . 6)
```

```
> (cons 4 (cons 5 (cons 6 nil)))
```

```
(4 5 6)
```

- este último exemplo corresponde ao `(list 4 5 6)` anterior. Note que `nil` corresponde à lista vazia.



Listas

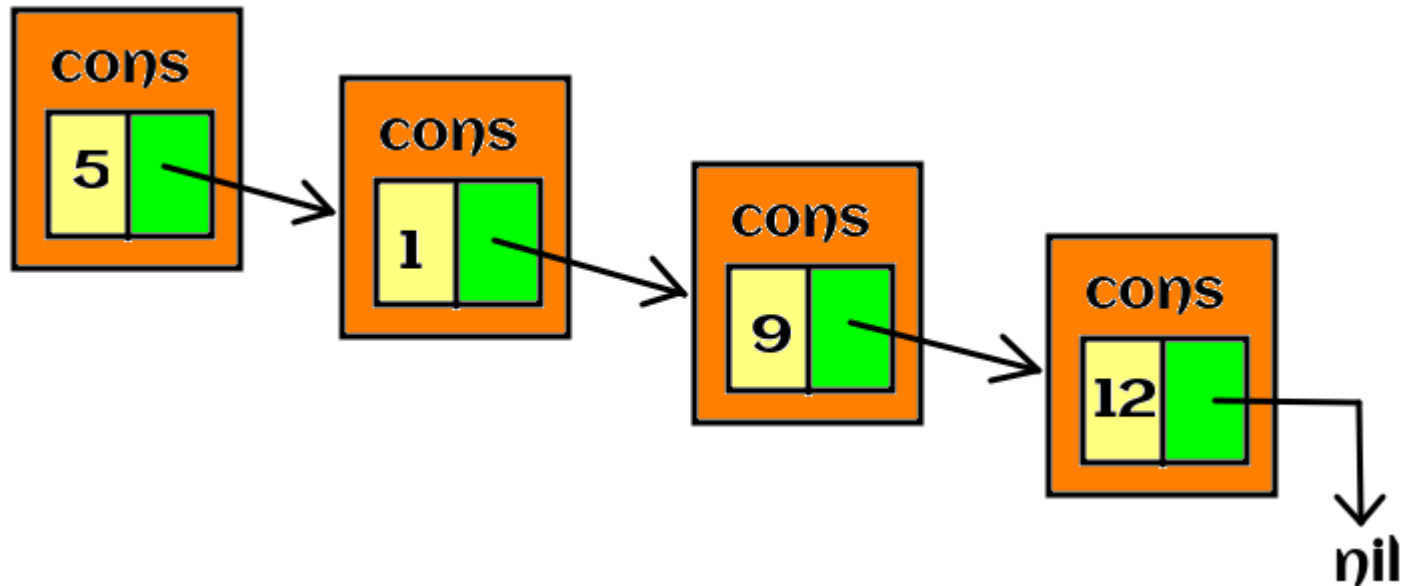
- **Exemplo:**

```
> (cons 5 (cons 1 (cons 9 (cons 12 nil))))
```

```
(5 1 9 12)
```

ou

```
> (list 5 1 9 12)
```





Pilha

- Se você armazena uma lista em uma variável, pode fazê-la funcionar como uma pilha:

```
> (setq a nil)
```

```
NIL
```

```
> (push 4 a)
```

```
(4)
```

```
> (push 5 a)
```

```
(5 4)
```

```
> (pop a)
```

```
5
```



Pilha

- continuação:

```
> a
```

```
(4)
```

```
> (pop a)
```

```
4
```

```
> (pop a)
```

```
NIL
```

```
> a
```

```
NIL
```



Funções

- Um programa em Lisp define uma ou mais funções;
- existem dois tipos de funções:
 - puras e
 - com efeitos colaterais
 - uma função tem efeitos colaterais quando modifica seus argumentos ou modifica variáveis globais.



Definindo Funções

- A macro DEFUN é usada par definir funções em Lisp.

- Sintaxe:

```
defun nome lista-de-argumentos corpo
```

- exemplos:

- para calcular o quadrado de um número:

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

- para o discriminante da equação ax^2+bx+c

```
(defun discr (a b c) (- (* b b) (* 4 a c)))
```



Funções Recursivas

- Exemplo:
 - cálculo do fatorial:

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




Condicionais

- Em Lisp temos dois tipos de estruturas condicionais:
 - a `IF` e a `COND`
- o `IF` recebe 3 argumentos e, caso o valor do primeiro argumento (condição) seja diferente de `NIL`, então:
 - o `IF` devolve o segundo argumento
- caso o resultado do primeiro argumento seja igual a `NIL`, então:
 - o `IF` devolve o terceiro argumento;
- exemplo:

```
> (if (> 2 3) 4 5)  
5
```



CondicionaI IF

- Em Lisp a estrutura condicional `IF` se assemelha muito à encontrada nas planilhas eletrônicas, como: Calc (do LibreOffice e Excel da Microsoft Office);
- o `IF` recebe 3 argumentos e, caso o valor do primeiro argumento (condição) seja diferente de `NIL`, então:
 - o `IF` devolve o segundo argumento
- caso o resultado do primeiro argumento seja igual a `NIL`, então:
 - o `IF` devolve o terceiro argumento;
- exemplo:

```
> (if (> 2 3) 4 5)
```

```
5
```



CondicionaI IF

- outros exemplos:
 - uma função para determinar o maior de dois números:

```
(defun maior (x y) (if (> x y) x y))
```

- e outra função para determinar o maior de três números:

```
(defun maior3 (x y z)
  (if (> x y)
    (if (> x z) x z)
    (if (> y z) y z)
  )
)
```



Exclusão Múltipla COND

- A forma especial `COND` tem um número qualquer de argumentos:
 - cada um dos argumentos é uma lista e, em cada um deles, o primeiro elemento da lista é uma condição ou teste;
 - se a condição for verdadeira (diferente de `NIL`) então o segundo elemento da lista é retornado;
 - os testes são avaliados em ordem e o primeiro que der certo determina a resposta e
 - se nenhum dos testes for verdadeiro, então o resultado produzido por `COND` será `NIL`.
- Para as pessoas acostumadas com programação imperativa, pode-se dizer que a `COND` é semelhante ao `switch ... case` das linguagens C e Java.



Exclusão Múltipla COND

- exemplo:

```
(defun maior (x y)
  (cond
    ((> x y) x)
    (t y)
  )
)

(defun maior3 (x y z)
  (cond
    ((and (>= x y) (>= x z)) x)
    ((and (>= y x) (>= y z)) y)
    ((and (>= z x) (>= z y)) z)
  )
)
```



Variáveis Locais

- Define-se a amarra ou atamento de valores às variáveis locais em LISP através do comando `let`;
- a criação de uma variável local com o comando `let` sobrepõe outra variável que, por ventura, tenha o mesmo nome da variável definida por `let`, temporariamente.
- Exemplo:

```
Copyright (c) Bruno Haible, Sam Steingold 1999-20
Copyright (c) Sam Steingold, Bruno Haible 2001-20

Type :h and hit Enter for context help.

[1]> (setq x 15)
15
[2]> (let x 3)
3
[3]> x
15
[4]> █
```



Variáveis Locais

- Outro exemplo:

Criamos a função **discr** para calcular o discriminante da fórmula de Bhaskara na equação $ax^2 + bx + c$

```
Copyright (c) Bruno Haible, Sam Steingold 1999-2000  
Copyright (c) Sam Steingold, Bruno Haible 2001-2018
```

```
Type :h and hit Enter for context help.
```

```
[1]> (defun discr (a b c) (- (* b b) (* 4 a c)))
```

```
DISCR
```

```
[2]> (defun raiz (a b c)  
      (let ((valor (discr a b c)))  
        (/ (+ (- b) (sqrt valor)) (* 2 a))  
      )  
    )
```

```
RAIZ
```

```
[3]> (raiz 3 0 2)
```

```
#C(0 0.8164966)
```

```
[4]> (raiz 9 2 0)
```

```
0
```

```
[5]> (raiz 9 3 1)
```

```
#C(-1/6 0.28867513)
```

```
[6]> █
```

Criamos a função **raiz** que recebe os coeficientes de uma equação de segundo grau e retorna uma de suas raízes.



Variáveis Locais

- Exemplo com erro:

Criamos a função **discr** para calcular o discriminante da fórmula de Bhaskara na equação $ax^2 + bx + c$

```
[1]> (defun discr (a b c) (- (* b b) (* 4 a c)))  
DISCR  
[2]> (defun raizes (a b c)  
      (let ((valor (discr a b c))  
            (r1 (/ (+ (- b) (sqrt valor)) (* 2 a)))  
            (r2 (/ (- (- b) (sqrt valor)) (* 2 a))))  
        (list r1 r2))  
      )  
    )  
RAIZES
```

Criamos a função **raizes** que recebe os coeficientes de uma equação de segundo grau e retorna ambas as raízes.

```
[3]> (raizes 2 12 -14)
```

Observe que ao chamarmos a função **raizes** e passarmos os coeficientes 2, 12 e -14, ela nos retorna um erro e não calcula as raízes.

```
*** - LET: variable VALOR has no value  
The following restarts are available:  
USE-VALUE      :R1      Input a value to be used instead of VALOR.  
STORE-VALUE    :R2      Input a new value for VALOR.  
ABORT          :R3      Abort main loop
```


Variáveis Locais

- Exemplo com `let*`

```
[1]> (defun discr (a b c) (- (* b b) (* 4 a c)))
```

```
DISCR
```

```
[2]> (defun raizes (a b c)
      (let ((valor (discr a b c))
            (r1 (/ (+ (- b) (sqrt valor)) (* 2 a)))
            (r2 (/ (- (- b) (sqrt valor)) (* 2 a))))
        (list r1 r2))
      )
      )
```

```
RAIZES
```

```
[3]> (raizes 2 12 -14)
```

```
*** - LET: variable VALOR has no value
```

```
The following restarts are available:
```

```
USE-VALUE      :R1      Input a value to be used instead of VALOR.
```

```
STORE-VALUE    :R2      Input a new value for VALOR.
```

```
ABORT          :R3      Abort main loop
```

```
Break 1 [4]> (defun raizes (a b c)
              (let* ((valor (discr a b c))
                     (r1 (/ (+ (- b) (sqrt valor)) (* 2 a)))
                     (r2 (/ (- (- b) (sqrt valor)) (* 2 a))))
                    (list r1 r2))
              )
              )
```

```
RAIZES
```

```
Break 1 [4]> (raizes 2 12 -14)
```

```
(1 -7)
```

```
Break 1 [4]> █
```

Recriamos a função **raizes**, que recebe os coeficientes de uma equação de segundo grau e retorna ambas as raízes.

Observe que ao chamarmos a função **raizes** e passarmos os coeficientes 2, 12 e -14, ela nos retorna o resultado das duas raízes calculadas.

Mudamos o comando `let` para `let*` e agora, antes do cálculo de `r1` e `r2` serem efetivados, primeiramente será feito o cálculo de `valor`. Pois, tanto `r1` quanto `r2` dependem do resultado que será apurado para `valor`.



Particularidade do `let*`

- Para que o `let*` funcione, a expressão que deve sofrer o atraso no cálculo deve estar no mesmo nível de “profundidade” do `let*`.
- Exemplos:

```
Copyright (c) Sam Steingold, Bruno Haible 2001-2018
```

```
Type :h and hit Enter for context help.
```

```
[1]> (let (
      (x 1)
      (y 2)
      (z (+ x 2))
    )
  z ; Imprime o valor contido em "y"
)
```

```
*** - LET: variable X has no value
```

```
The following restarts are available:
```

```
USE-VALUE      :R1      Input a value to be used instead of X.
STORE-VALUE    :R2      Input a new value for X.
ABORT          :R3      Abort main loop
```

```
Break 1 [2]>
```

```
Copyright (c) Bruno Haible, Pierpaolo Berna
Copyright (c) Bruno Haible, Sam Steingold 19
Copyright (c) Sam Steingold, Bruno Haible 20
```

```
Type :h and hit Enter for context help.
```

```
[1]> (let* (
      (x 1)
      (y 2)
      (z (+ x 2))
    )
  z ; Imprime o valor contido em "z"
)
```

```
3
```

```
[2]> █
```



Inibindo a Avaliação

- O interpretador de Lisp avalia expressões. No entanto, é possível inibir a avaliação utilizando o comando `QUOTE` ou `'` (apóstrofo).
- Exemplo:

```
Copyright (c) Sam Steingold, Bruno Haible 2001-2018
```

```
Type :h and hit Enter for context help.
```

```
[1]> (setq a 10)
10
[2]> a
10
[3]> (quote a)
A
[4]> 'a
A
[5]> a
10
[6]> █
```



Inibindo a Avaliação Parcialmente

- Pode haver a necessidade de se avaliar parte de uma expressão e outra parte não. Para tanto, basta trocarmos o apóstrofo (') pela crase (`) e colocarmos uma vírgula (,) antecedente à parte que deva ser avaliada.
- Exemplo:

```
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2018

Type :h and hit Enter for context help.

[1]> (list (+ 5 4) (* 3 9) (/ 4 3))
(9 27 4/3)
[2]> '(list (+ 5 4) (* 3 9) (/ 4 3))
(LIST (+ 5 4) (* 3 9) (/ 4 3))
[3]> `(list (+ 5 4) ,( * 3 9) ,( / 4 3))
(LIST (+ 5 4) 27 4/3)
[4]> █
```



Funções Aritméticas

- Básicas:

+ - * /

- Arredondamento:

`floor` - arredonda para baixo,

`ceiling` - arredonda para cima,

`truncate` - arredonda em direção ao zero e

`round` - arredonda para o inteiro mais próximo.

- Incremento e decremento:

`1+`

`1-`



Funções Aritméticas

- Máximo Divisor Comum (*Greatest Common Divisor*):

`gcd`

- Mínimo Múltiplo Comum (*Least Common Multiple*):

`lcm`

- Valor Absoluto:

`abs`

- Quadrado:

`sqr`



Funções Aritméticas

- Raiz Quadrada:

`sqrt`

- Exponenciais:

- que calcula e^x : (`exp x`)

- que calcula x^y : (`expt x y`)

- Função Logarítmica $\log_y x$:

(`log x y`) o segundo argumento é facultativo e, caso omitido a base padrão é $e = 2.7182817\dots$



Funções Trigonométricas

- Funções:

`sin`

`cos`

`tan`

`asin`

`acos`

`atan`



Funções Hiperbólicas

- Funções:

`sinh`

`cosh`

`tanh`

`asinh`

`acosh`

`atanh`



Funções Especiais

- Funções para números complexos:

`conjugate`

`realpart`

`imagpart`

- Constante:

`pi = 3.1415926...`



Funções - Argumentos Opcionais

- Há a possibilidade da criação de funções que possuam argumentos opcionais;
- esse tipo de argumento é “prefixado” pela chave `&optional`
- Exemplo:

```
(defun func (a b &optional c)
  (if c
    (+ a b c)
    (+ a b)
  )
)
```

```
[1]> (defun func (a b &optional c)
      (if c
        (+ a b c)
        (+ a b)
      )
      )
FUNC
[2]> (func 5 9)
14
[3]> (func 5 9 11)
25
[4]> █
```



Funções - Argumentos Opcionais

- Em casos onde hajam argumentos opcionais, é possível definir um valor *default* para estes argumentos.
- Exemplo:

```
(defun func (a b &optional (c 0))  
  (+ a b c)  
)
```

```
[4]> (defun func (a b &optional (c 0))  
      (+ a b c)  
      )  
FUNC  
[5]> (func 3 4)  
7  
[6]> (func 3 4 5)  
12  
[7]> █
```



Funções - Argumentos Restantes

- É possível especificar que todos os argumentos de função, a partir de um ponto, sejam colocados em uma lista;
- para isto basta usar a chave `&rest`
- Exemplo:

```
(defun func (a b &rest c)
  (list 'Recebi a b 'e 'mais (length c) 'argumentos)
)
```

```
Type :h and hit Enter for context help.
[1]> (defun func (a b &rest c)
      (list 'Recebi a b 'e 'mais (length c) 'argumentos)
      )
FUNC
[2]> (func 7 8)
(RECEBI 7 8 E MAIS 0 ARGUMENTOS)
[3]> (func 7 8 9)
(RECEBI 7 8 E MAIS 1 ARGUMENTOS)
[4]> (func 7 8 9 10 11 12)
(RECEBI 7 8 E MAIS 4 ARGUMENTOS)
[5]> █
```



Funções - Argumentos Restantes

- Criando sua própria lista.
- Exemplo:

```
(defun my-list (&rest all)
  all
)
```

Copyright (c) Sam Steingold, Bruno Haible 2001-2018

Type :h and hit Enter for context help.

```
[1]> (defun my-list (&rest all)
      all
      )
MY-LIST
[2]> (my-list 1 2 3 4 5)
(1 2 3 4 5)
[3]> (setq lista (my-list 1 2 3 4 5 6 7 8 9 10))
(1 2 3 4 5 6 7 8 9 10)
[4]> lista
(1 2 3 4 5 6 7 8 9 10)
[5]> █
```



Funções - Argumentos Restantes

- Recriando função +
- Exemplo:

```
(defun my+ (&rest args)
  (soma-tudo args)
)

(defun soma-tudo (args)
  (if (null args)
      0
      (+ (car args) (soma-tudo (cdr args))))
)
)
```

```
[5]> (defun my+ (&rest args)
      (soma-tudo args)
      )
MY+
[6]> (defun soma-tudo (args)
      (if (null args)
          0
          (+ (car args) (soma-tudo (cdr args))))
      )
      )
SOMA-TUDO
[7]> (my+ 1 2 3 4 5)
15
[8]> █
```



Funções - Args Por Palavra-chave

- Normalmente a ordem dos argumentos de uma função deve ser respeitada quando da chamada da função. Porém, em Lisp, se feito da maneira correta esta ordem poderá ser mudada.

- Exemplo:

- para calcular o volume do cone temos a seguinte fórmula:

$$\text{volume} = (\text{pi} * \text{raio}^2 * \text{altura}) / 3$$

- assim sendo, uma função em Lisp seria:

```
(defun vcone (&key raio altura)
  (/ (* pi raio raio altura) 3)
)
```




Funções - Args Por Palavra-chave

- continuação:
 - a chave `&key` nos proporciona uma maneira de podermos passar os argumentos da função na ordem que quisermos, desde que saibamos os nomes dos argumentos. Assim:

```
> (vcone :raio 3 :altura 5)
```

```
47.123889...
```

```
> (vcone :altura 5 :raio 3)
```

```
47.123889...
```

- Obs.: uma vez definida a chave, a função não mais poderá ser chamada sem o nome dos argumentos precedidos de dois pontos (:).

```
[8]> (defun vcone (&key raio altura)
      (/ (* pi raio raio altura) 3)
      )
```

```
VCONE
```

```
[9]> (vcone :raio 3 :altura 5)
```

```
47.123889803846898582L0
```

```
[10]> (vcone :altura 5 :raio 3)
```

```
47.123889803846898582L0
```

```
[11]> (vcone 3 5)
```

```
*** - VCONE: &KEY marker 3 is not a symbol
```

```
The following restarts are available:
```

```
ABORT          :R1      Abort main loop
```

```
Break 1 [12]> █
```



Funções de Comparação

- Também chamados de “predicados de igualdade”, existem três que causam muita confusão entre os iniciantes em Lisp: `eq`, `eql` e `equal`;
- o predicado `equal` é o mais confiável e também o mais demorado. Ele compara as duas expressões, que lhes são passadas, recursivamente até chegar em átomos, que são comparados com `eql`. E `eq` apenas compara os endereços dos argumentos, produzindo resultados inesperados, como:

```
> (eq (cons 'a nil) (cons 'a nil))
```

```
NIL
```



Funções de Comparação

- continuação:
- `eq` não deve ser usado para comparar pares-com-ponto. Mesmo com átomos a veracidade do resultado de uma expressão como:
$$> (eq\ 3\ 3)$$
- vai depender da implementação Common Lisp que estiver sendo utilizada e
- existem implementações que adotam a convenção de que constantes com o mesmo valor devem compartilhar o mesmo endereço, por motivos de economia de memória. Nestas implementações o valor da expressão acima é `T`. Mas esta convenção não é universal entre as implementações de Common Lisp.



Funções de Comparação

- Continuação:
- já o predicado `eq1` é como `eq`, exceto pelo fato de garantir que constantes de mesmo valor e mesmo tipo serão consideradas iguais. Ou seja:
 - > `(eq1 3 3)`
`T`
- porém, veja como o tipo é relevante:
 - > `(eq1 3 3.0)`
`NIL`
- assim sendo, para os novatos as seguintes diretrizes podem ser usadas para garantir sucesso em grande parte das situações:
 - na comparação de expressões arbitrárias, use `equal`,
 - se seus argumentos são átomos, use `eq1` e
 - evite usar `eq`.



Funções para Listas

- **Atenção!** Nos exemplos a seguir será colocado o prefixo `my-` nas funções. Isto para que não corramos o risco de sobrepor as funções originais provenientes da linguagem. O que é possível de ser feito em algumas implementações.
- Existem as funções: `first`, `second`, `third`, `fourth` e até `tenth`.

```
[1]> (first (list 1 2 3 4 5 6 7 8 9 10 11))  
1  
[2]> (second (list 1 2 3 4 5 6 7 8 9 10 11))  
2  
[3]> (third (list 1 2 3 4 5 6 7 8 9 10 11))  
3  
[4]> (fourth (list 1 2 3 4 5 6 7 8 9 10 11))  
4  
[5]> (tenth (list 1 2 3 4 5 6 7 8 9 10 11))  
10  
[6]> █
```



Funções para Listas

- vamos criar agora a função `nth` que retorna o enésimo item da lista, os índices começam de zero:

```
(defun my-nth(indice lista)
  (if (= indice 0) ;início em zero
      (car lista)
      (my-nth (1- indice) (cdr lista)))
  )
)
```

- a função `(elt lista indice)` faz exatamente o que a função `my-nth` faz e a função `(last lista)` retorna uma lista com o último elemento da lista para ela passada, se a lista passada estiver vazia, retorna `NIL`.



Funções para Listas

- `(length lista)` retorna a quantidade de itens na lista;
- `(member item lista)` retorna o próprio `item` testado, caso o mesmo seja membro da lista. Caso contrário retorna `NIL`;
- `(reverse lista)` retorna a lista na ordem invertida;
- `(append lista outralista)` retorna a concatenação de `lista` com `outralista`;
- `(list args)` retorna uma lista com os `args` que foram passados;
- `(subst novo velho lista)` retorna uma lista substituindo o `velho` pelo `novo`. Caso o `velho` não exista em `lista`, retorna a própria `lista`;
- `(position item lista)` retorna o índice do primeiro `item` encontrado na lista (começa do zero), caso não encontre o `item` retorna `NIL`.



Funções para Listas

- `(count item lista)` conta quantos “item” existem na lista;
- `(subseq lista início fim)` retorna um trecho da `lista`, começando pelo índice `início` e opcionalmente indo até o índice `fim`. Caso o índice `fim` seja omitido, então, a função retornará todos os itens existentes a partir do índice `início` até o final da lista;
- `(remove item lista)` retorna uma nova lista obtida de `lista` já com os elementos iguais a `item`, removidos;
- `(mapcar função lista)` retorna uma lista composta do resultado de aplicar a função dada a cada elemento da `lista` dada. Exemplos:

```
> (mapcar #'1+ '(1 2 3))
```

```
(2 3 4)
```

```
> (mapcar #'+ '(1 2 3) '(4 5))
```

```
(5 7)
```




Funções para Conjuntos

- Pode-se usar listas para representar conjuntos em Lisp;
- a linguagem oferece suporte com algumas funções que possibilitam operações comuns entre conjuntos, como:
 - união,
 - interseção e
 - diferença.



Funções para Conjuntos

- `(union lista1 lista2)` retorna uma nova lista que é a fusão de `lista1` e `lista2` sem elementos repetidos;
- `(intersection lista1 lista2)` retorna uma nova lista com os elementos que são a interseção de `lista1` e `lista2`;
- `(set-difference lista1 lista2)` retorna uma lista com os elementos `lista1` que não estão em `lista2`;
- `(subsetp lista1 lista2)` retorna verdadeiro quando cada elemento de `lista1` aparece em `lista2`.



Arrays

- Para a criação de um *array* utiliza-se a função `(make-array args)`:
 - > `(make-array 4)`
`#(NIL NIL NIL NIL)`
 - > `(make-array '(3 3))`
`#2A((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))`
- para acessar os elementos de um *array* utiliza-se a função `(aref args)`:
 - > `(aref * 1 1)`
`NIL`



Arrays

- continuação:
- para colocar um elemento em um *array* utiliza-se a função `(setf (aref args) args)`:

```
> (setq mat (make-array '(2 2)))
```

```
#2A((NIL NIL) (NIL NIL))
```

```
> (setf (aref mat 0 0) 5)
```

```
5
```

```
> mat
```

```
#2A((5 NIL) (NIL NIL))
```

	0	1
0	5	
1		



Array - Exemplo 1

```
> (setq a (make-array 3))
```

```
#(NIL NIL NIL)
```

```
> (aref a 1)
```

```
NIL
```

```
> (setf (aref a 1) 3)
```

```
3
```

```
> a
```

```
#(NIL 3 NIL)
```

```
> (aref a 1)
```

```
3
```





Array - Exemplo 2

```
> (setq mat (make-array '(3 3)))  
#2A((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))  
  
> (setf (aref mat 1 1) 9)  
9  
  
> mat  
#2A((NIL NIL NIL) (NIL 9 NIL) (NIL NIL NIL))  
  
> (setf (aref mat 0 0) 8)  
8  
  
> (setf (aref mat 2 2) 10)  
10  
  
> mat  
#2A((8 NIL NIL) (NIL 9 NIL) (NIL NIL 10))
```

8		
	9	
		10



Strings

- Um *string* é uma sequência de caracteres entre aspas duplas. Lisp representa um *string* como um *array* de tamanho variável de caracteres;
- pode-se escrever um *string* que contém a aspa dupla precedendo a aspa por uma barra invertida \
- e pode-se escrever uma barra invertida colocando-se duas \, mas aparecerá apenas uma \
 - "abcd" ;tem 4 caracteres
 - "\" ;tem 1 caractere
 - "\\ " ;tem 1 caractere



Funções para *Strings*

- Para concatenação:

```
> (concatenate 'string "abcd" "efg")  
"abcdefg"
```

- extração de caracteres de um *string*:

```
(char "abc" 1)
```

```
#\b ;LISP escreve caracteres precedidos por  
#\
```

- como *strings* também são vetores:

```
(aref "abc" 1)
```

```
#\b ;temos o mesmo resultado
```




Funções para *Strings*

- Ainda na concatenação:

```
> (concatenate 'string' (#\a #\b) '#\c')  
"abc"
```

```
> (concatenate 'list' "abc" "de")  
(#\a #\b #\c #\d #\e)
```

```
> (concatenate 'vector' #(3 3 3) '#(3 3 3)')  
#(3 3 3 3 3 3)
```



Estruturas de Repetição

- **loop**
 - é a estrutura de repetição mais simples de Lisp. Um `loop` se repete até encontrar um comando `return`.

- **Exemplo 1:**

```
> (setq a 4)
4
> (loop
  (setq a (+ a 1))
  (when (> a 7) (return a))
  )
8
```



Estruturas de Repetição

- **loop**

- **Exemplo 2:**

```
> (setq b 8)
```

```
8
```

```
> (loop
```

```
    (setq b (- b 1))
```

```
    (when (< b 3) (return))
```

```
)
```

```
NIL
```



Estruturas de Repetição

- **dolist**

- ata uma variável aos elementos de uma lista na sua ordem e termina quando encontra o fim da lista.

- Exemplo:

```
> (dolist (x '(a b c)) (print x))
```

```
A
```

```
B
```

```
C
```

```
NIL
```

- `dolist` sempre retorna `NIL` como valor. Observe que o valor de `x` no exemplo acima nunca é `NIL`. O valor `NIL` abaixo do `c` é o retorno do comando `dolist`.



Estruturas de Repetição

- **do**
 - a primitiva de iteração mais complexa é o comando `do`.
 - Explicando o `do`:
 - 1) a primeira parte do `do` (em azul) especifica quais variáveis que devem ser atadas, quais são os seus valores iniciais e como eles devem ser atualizados;
 - 2) a segunda parte especifica uma condição de término (em verde) e um valor de retorno e
 - 3) a última parte (em preto) é o corpo.

- A estrutura:

```
(do ( (x 1 (+ x 1)) ;variável x, com valor inicial 1
      (y 1 (* y 2)) ;variável y, com valor inicial 1
      )
      ((> x 5) y) ;retorna valor de y quando x > 5
      (print y) ;corpo
      (print 'working) ;corpo
      )
```



Estruturas de Repetição

- Exemplo:

```
> (do ( (x 1 (+ x 1)) ;variável x, com valor inicial 1
      (y 1 (* y 2)) ;variável y, com valor inicial 1
      )
    ((> x 5) y) ;retorna valor de y quando x > 5
    (print y) ;corpo
    (print 'working) ;corpo
  )
1
WORKING
2
WORKING
4
WORKING
8
WORKING
16
WORKING
32
```



Funccall, Apply e Mapcar

- Em Lisp funções podem ser argumentos de outras funções. E se assim for, um destes comandos deverá ser utilizado para executar a função passada como argumento.
- Explicando:
 - `funccall` chama seu primeiro argumento com os argumentos restantes como argumentos deste;
 - `apply` é semelhante a `funccall`, exceto que seu argumento final deverá ser uma lista. Os elementos desta lista são tratados como se fossem argumentos adicionais ao `funccall` e
 - `mapcar` deve receber, como argumento, uma função de um argumento. `mapcar` aplicará a função a cada elemento da lista dada e coletará os resultados em outra lista.



Funcall, Apply e Mapcar

- Exemplos:

```
> (funcall #' + 3 4)
```

```
7
```

```
> (apply #' + 3 4 '(3 4))
```

```
14
```

```
> (apply #' + 3 4 '(5 6))
```

```
18
```

```
> (mapcar #' not '(t nil t nil t nil))
```

```
(NIL T NIL T NIL T)
```




Ordenação

- Lisp provê duas primitivas para ordenação:
 - **sort**

```
> (sort '(2 4 1 5 4 6) #'<)
(1 2 4 4 5 6)
> (sort '(2 1 5 4 6) #'>)
(6 5 4 2 1)
```
 - o primeiro argumento para `sort` é uma lista e o segundo é a função de comparação;
 - Atenção! `sort` tem permissão para destruir o seu argumento. Por isso, se você deseja manter a lista original, então copie-a com `copy-list` ou `copy-seq`.



Ordenação

- **stable-sort**

```
> (stable-sort '(2 4 1 5 4 6) #'<)
```

```
(1 2 4 4 5 6)
```

```
> (stable-sort '(2 1 5 4 6) #'>)
```

```
(6 5 4 2 1)
```

- O `stable-sort` é bem semelhante ao `sort`. Porém alguns autores retratam que o comando `sort`, em algumas situações bem específicas, pode falhar. O que não aconteceria com o `stable-sort`. Mas, este por sua vez é mais lento que o `sort`.



Packages

- De acordo com Cooper Jr. (2021), pacotes podem ser pensados como “áreas de código” para símbolos;
- um símbolo existe em um determinado pacote;
- ao iniciar o ambiente, provavelmente você estará no pacote `COMMON-LISP-USER`. Para verificar basta utilizar o comando `*package*` e
- se precisar referenciar um símbolo que foi definido em outro pacote, então use:
 - `nomeDoPacote:nomeDoSímbolo` (caso este tenha sido exportado previamente) ou
 - `nomeDoPacote::nomeDoSímbolo` (caso este NÃO tenha sido exportado previamente).



Packages

- Um símbolo existe em um determinado pacote:

```
[1]> (make-package 'teste)
#<PACKAGE TESTE>
[2]> (in-package teste)
#<PACKAGE TESTE>
TESTE[3]> (setq valor 100)
100
TESTE[4]> valor
100
TESTE[5]> (in-package :user)
#<PACKAGE COMMON-LISP-USER>
[6]> valor

*** - SYSTEM::READ-EVAL-PRINT: variable VALOR has no value
The following restarts are available:
USE-VALUE      :R1      Input a value to be used instead of VALOR.
STORE-VALUE    :R2      Input a new value for VALOR.
ABORT          :R3      Abort main loop
Break 1 [7]> :r3
[8]> teste::valor
100
[9]> █
```



***Packages* - Comandos Úteis**

- `(make-package 'nomeDoPacote)` - cria um pacote;
- `(delete-package 'nomeDoPacote)` - apaga um pacote, mesmo que este contenha variáveis, funções e etc.;
- `(in-package nomeDoPacote)` - muda o ambiente do pacote atual para o pacote `nomeDoPacote`.



Packages - Dicas

- Aplicações pequenas devem ficar no pacote de usuários (predefinido);
- já as aplicações maiores devem ficar em pacotes próprios, até mesmo pela questão de modularização do sistema;
- o pacote padrão geralmente é o `COMMON-LISP-USER`, que pode ser acessado pelo seu apelido. Ou seja `(in-package :user)`



Bibliografia

- BITTENCOURT, Guilherme. **Inteligência Artificial - Ferramentas e Teorias**. 2. ed. Florianópolis: Ed. da UFSC, 2001.
- COOPER JR. David J. **Basic Lisp Techniques**. Disponível em: <https://franz.com/resources/educational_resources/cooper.book.pdf>. Acesso em: 07 Set. 2021.
- MEDEIROS, Adelardo A. Dantas. **Linguagem LISP**. Disponível em: <<https://www.dca.ufrn.br/~adelardo/lisp/>>. Acesso em: 21 Ago. 2021.
- MEIDANIS, João. **MC346 - Paradigmas de programação Lisp**. Disponível em: <<https://www.ic.unicamp.br/~meidanis/courses/mc346/2015s2/funcional/apostila-lisp.pdf>>. Acesso em: 21 Ago. 2021.
- SEIBER, Seibel. **Practical Common Lisp**. Disponível em: <<http://www.e-booksdirectory.com/details.php?ebook=1323>>. Acesso em 19 Ago. 2021.
- SERGERAERT, Francis. **Packages in Common Lisp, a tutorial**. Disponível em: <<https://www-fourier.ujf-grenoble.fr/~sergerar/Papers/Packaging.pdf>>. Acesso em: 08 Set. 2021.
- SITE INOVAÇÃO TECNOLÓGICA. **Glóbulo branco robótico navega contra corrente sanguínea**. 01/06/2020. Online. Disponível em <<https://www.inovacaotecnologica.com.br/noticias/noticia.php?artigo=globulo-branco-robotico>>. Acesso em: 18 Out. 2020.



Bibliografia

- WANGENHEIM, Aldo Von. **Métodos e Técnicas de Programação Funcional: Linguagem LISP**. Disponível em: <<http://www.inf.ufsc.br/~aldo.vw/func/lisp1.html>>. Acesso em: 23 Ago. 2021.
- Wikibooks. **Common Lisp**. Disponível em: <https://en.wikibooks.org/wiki/Common_Lisp>. Acesso em: 19 Ago. 2021.
- Wikipedia. **Lisp (programming language)**. Disponível em: <[https://en.wikipedia.org/wiki/Lisp_\(programming_language\)#cite_note-ArtOfLisp-9](https://en.wikipedia.org/wiki/Lisp_(programming_language)#cite_note-ArtOfLisp-9)>. Acesso em: 17 Ago. 2021.
- Wikipedia. **Programação Funcional**. Disponível em: <https://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_funcional>. Acesso em: 17 Ago. 2021.
- X2 Inteligência Digital. **Inteligência Artificial**. Disponível em: <<https://x2inteligencia.digital/2020/03/03/filmes-sobre-a-inteligencia-artificial/>>. Acesso em: 16 Out. 2020.
- ZAMBIASI, Saulo Popov.. **Tutorial de Programação LISP**. Disponível em: <https://saulo.arisa.com.br/wiki/index.php/Tutorial_de_Programa%C3%A7%C3%A3o_Lisp>. Acesso em: 03 Set. 2021.