Universidade Federal da Bahia
Instituto de Computação – DCC/UFBA

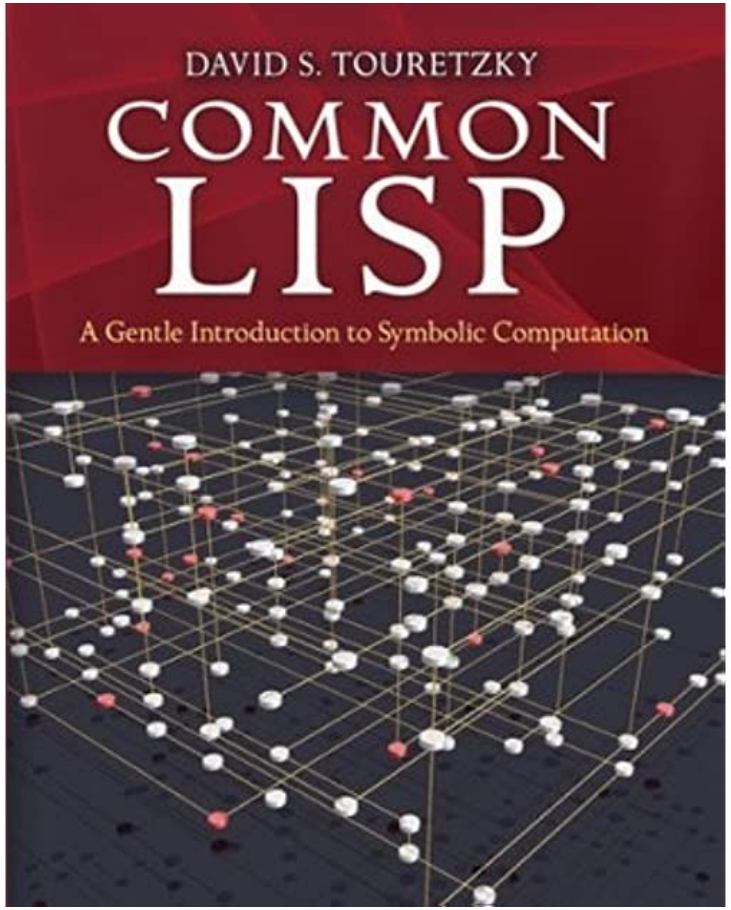
Programação Funcional
LISP

Prof.: Claudio Junior (claudiojns@ufba.br)

Paradigmas de Linguagem de Programação (MATA56)

2023.1

Referência


DAVID S. TOURETZKY
COMMON
LISP
A Gentle Introduction to Symbolic Computation

História

- LISt Processing;
- Criado em 1956 por John McCarthy;
- Influenciado pelo cálculo lambda, de Alonzo Church nos anos 40;
- Nasceu como uma ferramenta matemática;
- É a segunda linguagem de programação mais antiga que existe.

Objetivo

- É possível usar exclusivamente funções matemáticas como estrutura de dados elementares (McCarthy);
- 1958 – Primeira implementação do LISP em uma máquina: projeto de pesquisa para IA no MIT;
- Motivação:
  - Surgiu da ideia de desenvolver uma linguagem algébrica para processamento de listas para trabalho em IA.
- LISP 1.5, 1960-1965, primeiro dialeto do LISP.

Características do LISP

- Muitas vezes requerido como uma linguagem superior sobre as outras porque tem certas características que são únicas, bem integrada, ou de outra forma útil;

- Tipos de dados:
  - Átomo:
    - Números;
    - Strings;
    - Símbolos
      - ✓ Representam os identificadores da linguagem.
  - Lista:
    - Dados compostos;
    - Formadas a partir de pares (*cons cells*).

Características do LISP

- Versões e dialetos:
  - Franz, Mac, Inter, Common e Scheme;
- Fraca Tipagem:
  - As variáveis podem ser interpretadas de forma diferente, dependendo do contexto;
- Funções de ordem elevada:
  - Linguagens funcionais tipicamente suportam funções de ordem elevada;
- Concorrência (multitarefa):
  - É um paradigma de programação para a construção de programas de computador que fazem uso da execução concorrente (simultânea) de várias tarefas computacionais interativas, que podem ser implementadas como programas separadas ou como um conjunto de threads criadas por um único programa.

Características do LISP

- Common LISP em propósito geral, linguagem de programação multi-paradigma;
- Suporta uma combinação de procedimentos, funcional e orientada a objetos;
- Como programação dinâmica, facilita a evolução e desenvolvimento de *software* adicional, com interativa compilação em tempo de execução de eficientes programas.

Características do LISP

- LISP é frequentemente implementado por um interpretador:
  - Usuário entra com a expressão, o Interpretador avalia a expressão e imprime o resultado;
- LISP puro contem as seguintes primitivas:
  - Car: primeiro elemento de uma lista;
  - Cdr: o que sobra de uma lista com exceção do 1º elemento;
  - Cons: contrói uma lista dado um elemento e uma lista;
  - Eql: retorna T se os dois elementos que se seguem são iguais, NIL em caso contrário;
  - Atom: retorna T se elemento que o segue é atômico, NIL em caso contrário.

Chacota

- LISP = *Lots os Irritating Stupid Parentheses*
  - Monte de Parêntesis Estúpidos Irritantes, ou então
    - LISP = Linguagem Infernal Somente de Parêntesis.

Principais vantagens

- Alto nível de abstração, especialmente quando as funções são utilizadas, suprimindo muitos detalhes da programação e minimizando a probabilidade da ocorrência de muitas classes de erros;
- A não dependência das operações de atribuições permite aos programas avaliações nas mais diferentes ordens;
  - A característica de avaliação independente da ordem torna as linguagens funcionais mais indicadas para a programação de computadores maciçamente paralelos;
- A ausência de operações de atribuição torna os programas funcionais muito mais simples para provas de análises matemáticas do que os programas procedurais.

Principais vantagens

- Permite ilustrar com facilidade os comandos de recursão;
- Uma função frequentemente pode ser verificada visualmente;

Diferenças entre LISP e outras linguagens

- Programas e Dados tem a mesma forma;
- Não se trabalha com variáveis armazenadas:
  - Embora LISP suporte variáveis.

- Problemas que envolvam muitas variáveis (ex. contas de banco) ou muitas atividades sequenciais são muitas vezes mais fáceis de se trabalhar com programas procedurais ou programas orientados a objeto.

Desvantagens

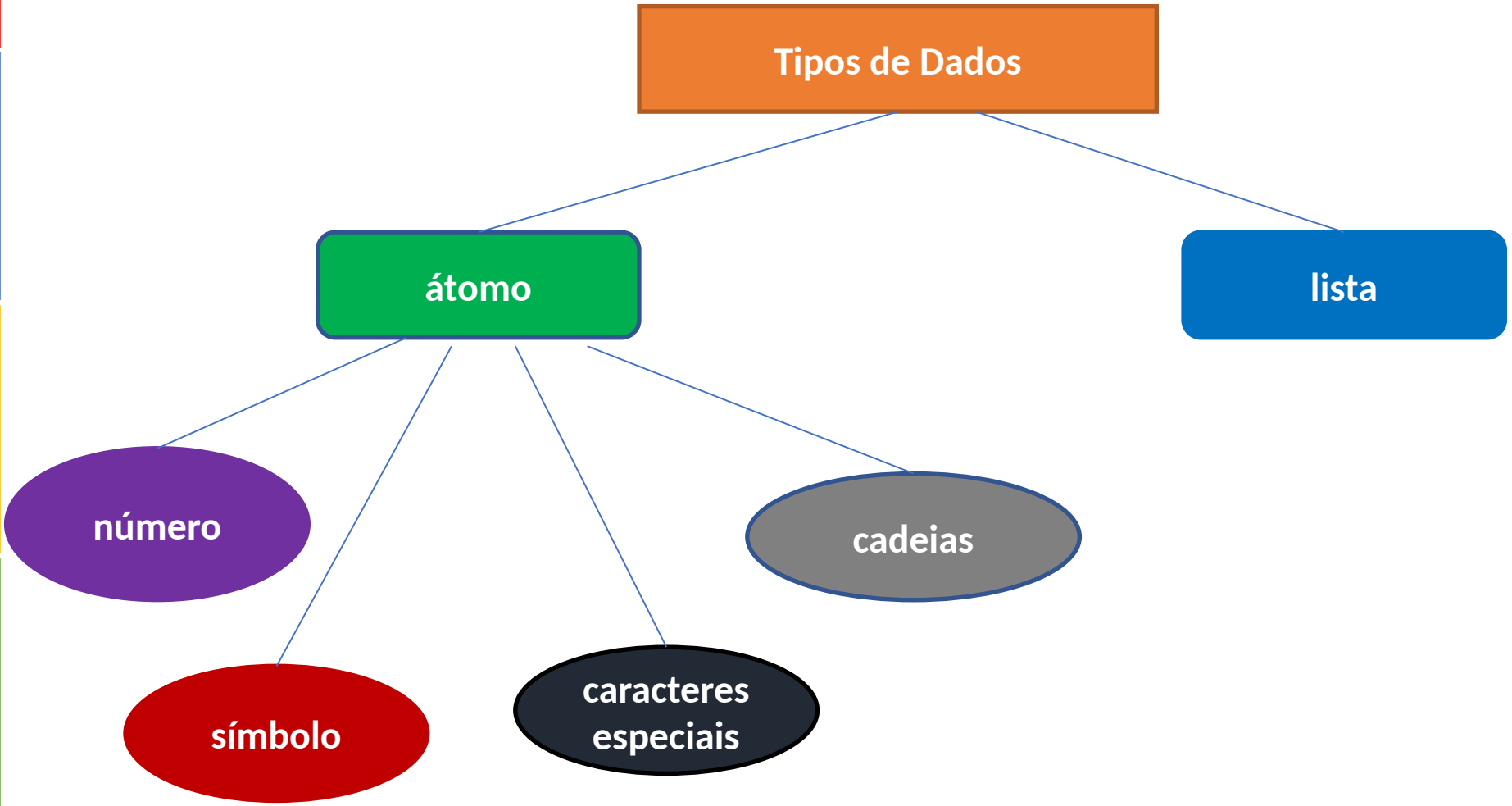
- Menor eficiência;

- Problemas que envolvam muitas variáveis (ex. contas de banco) ou muitas atividades sequenciais são muitas vezes mais fáceis de se trabalhar com programas procedurais ou programas orientados a objeto.

Aplicações reais de Common LISP

- Paul Graham – primeira versão do Yahoo! Store (então chamada Via Web);
- Primeira implementação de JavaScript no CVS do Mozilla;
- ITA Software, agora comprada pelo Google, mantém seus sistemas de grande porte em Common LISP;
- StumpWM, gerenciador de janelas para Unix;
- Desenvolvimento de jogos.

Tipos de Dados



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;

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.

LISP – Olá Mundo!

```
CL-USER> (format t "Hello, Word!")
Hello, Word!
NIL
```

E ainda...

```
(defun hello-word()
   (format t "Hello, World!"))
```

```
CL-USER> (hello-word)
Hello, World!
NIL
```

LISP

- Exemplo:
  - > (+ 3 4 5 6)

18

  - > (+ (+3 (+ (+ 4 5) 6)))

22

NIL e T - Exemplo

```
> (<= 2 3)
T

> (> 3 4)
NIL

> (and 2 t)
T

> (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
```

Conses (S-Expressions)

- Em LISP, se algo não é um átomo, então é um `cons` ou "*S-expression*";
- 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

- O que acontece se passarmos um átomo como segundo argumento de cons?

```
> (cons 'a 'b)
(A . B)
```

- De onde veio o ponto entre A e B?

- Resposta:
  - O ponto sempre existe num cons, mas nem sempre é impresso;

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)
```

Conses e Listas

- Podemos ver então que listas são conses que nunca são escritos com pontos:
  - Muitos autores preferem ignorar conses que não são listas

- Afinal, o que é então uma lista?

- Eis uma resposta formal:
  - Axioma: nil é uma lista;
  - Teorema: Se L é uma lista, e elem é um átomo ou uma lista, então (cons elem L) é uma lista.

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;

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: the function A is undefined
```

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 - Exemplos

```
> (car '(a b))
A

> (cdr '(a b))
(B)

> (car (cdr '(a b)))
B

> (cdr (cdr '(a b)))
NIL
```

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

Correspondência das Listas

- Uma Lista (m1 m2 ... mn) de elementos corresponde à seguinte expressão simbólica:
  - (m1. (m2. (... (mn . Nil) ...)))
- Onde:
  - (m) equivale a (m. Nil)
  - (m1 m2 ... mn) equivale a (m1. (m2. (... (mn. Nil) ...)))
  - (m1 m2 ... mn.x) equivale a (m1. (m2. (... (mn. x) ...)))
- ((AB . (C . NIL)) . (D . NIL)) equivale a:
  - ((AB C) D)
- ((A . (B . NIL)) . (C . (D . E))) equivale a:
  - ((A . B) C D . E)

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 expr1
  - Se nenhuma expressão lógica for avaliada como verdadeira, o cond retorna nil

If – then - else

- 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
```

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
```

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*

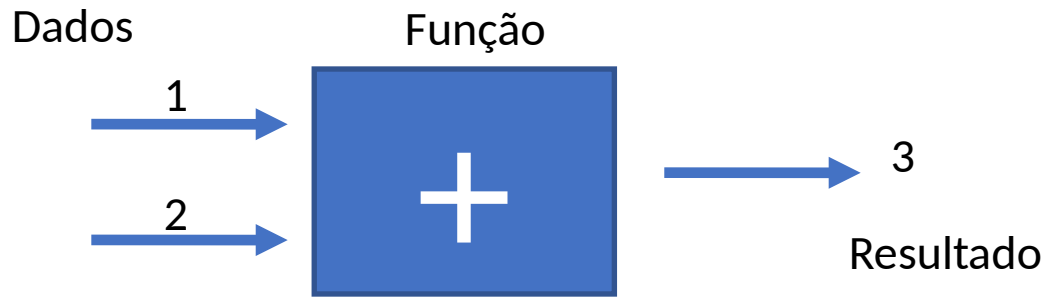
Usando o LISP

Dados e Funções

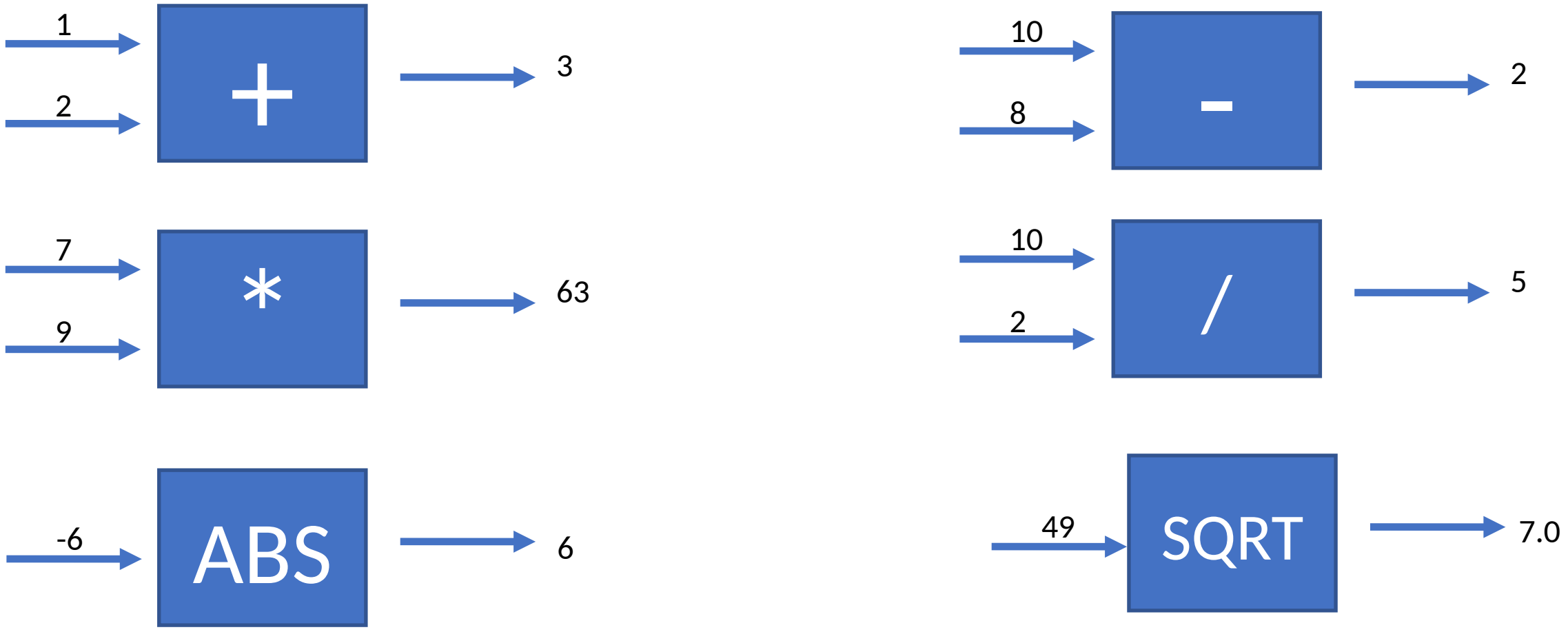
- Dados:
  - Informações (inputs, argumentos)
- Funções:
  - Processos pelos quais os dados passam;
  - Operam sobre os dados e retornam um resultado.

- Exemplo:

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



Funções numéricas em LISP



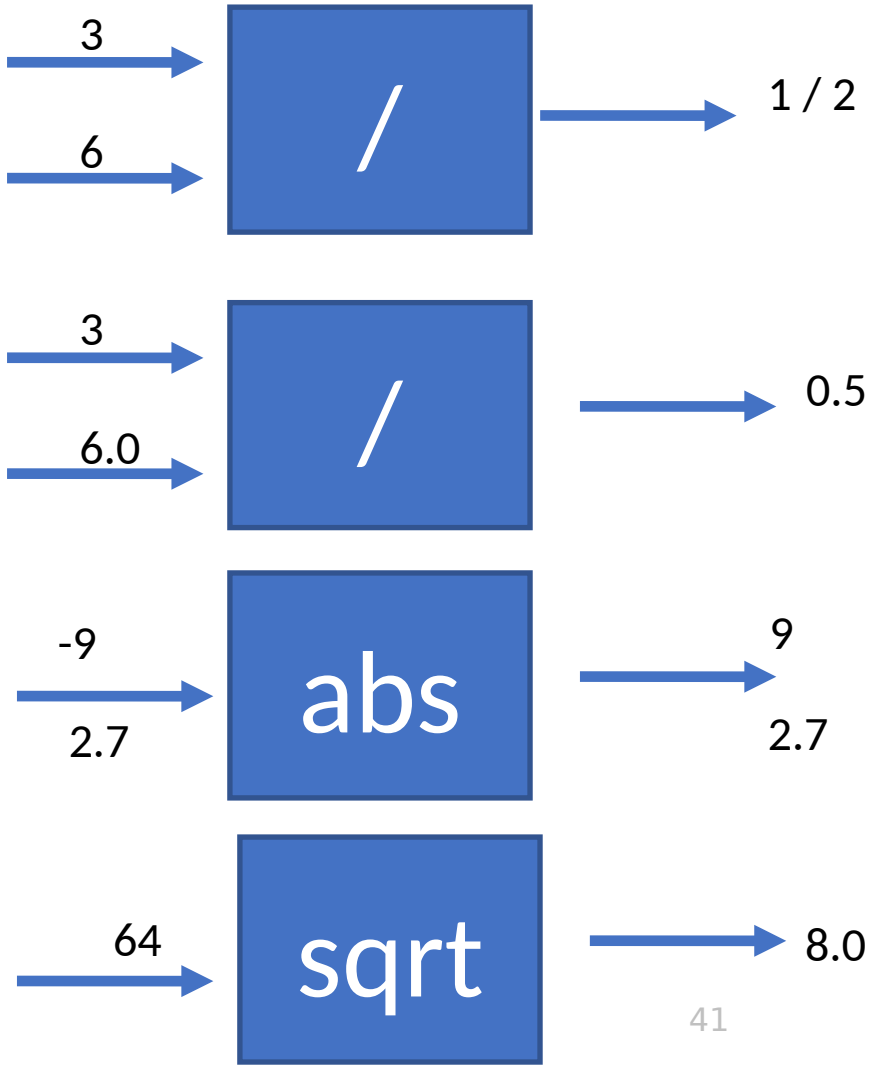
Funções numéricas em LISP

```
; SLIME 2013-11-17
CL-USER> (+ 1 2 )
3
CL-USER> (* 7 9 )
63
CL-USER> (ABS -6)
6
CL-USER> (- 10 8 )
2
CL-USER> (/ 10 2 )
5
CL-USER> (SQRT 49 )
7.0
CL-USER> (sqrt 49 )
7.0
CL-USER>
```

Tipos de Dados

- Numéricos
  - Integers: inteiros;
  - Ratios: razões, racionais;
  - Floating Ponts: ponto flutuante;
  - ...

- Observações:
  - Inteiro + Inteiro = Inteiro ou Razão
  - Inteiro + Ponto Flutuante = Ponto Flutuante



Tipos de Dados

```
; SLIME 2013-11-17
CL-USER> (/ 3 6 )
1/2
CL-USER> (/ 3 6.0)
0.5
CL-USER> (abs -9)
9
CL-USER> (abs 2.7)
2.7
CL-USER> (sqrt 64)
8.0
```

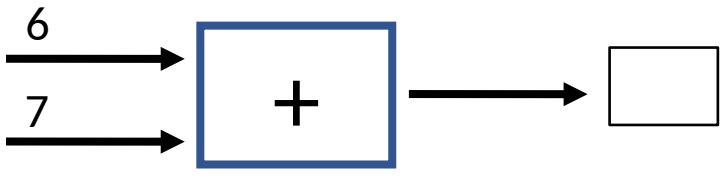
Ordem dos Inputs

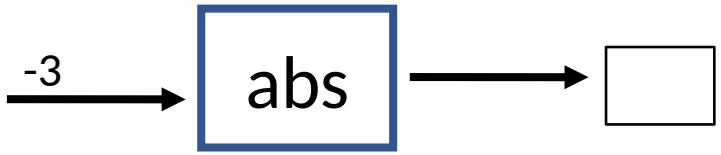
```
CL-USER> (/ 8 2)
4
CL-USER> (/ 2 8)
1/4
CL-USER> (/ 10 8)
5/4
CL-USER> (/ 8 10)
4/5
CL-USER> (/8 10)
; in: /8 10
;     (/8 10)
;
; caught STYLE-WARNING:
;   undefined function: /8
;
; compilation unit finished
;   Undefined function:
;     /8
;   caught 1 STYLE-WARNING condition
; Evaluation aborted on #<UNDEFINED-FUNCTION /8 {241F22A1}>.
```

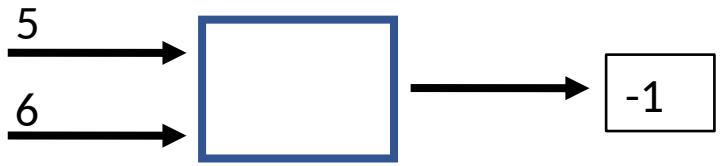
```
CL-USER> (- 1 1/3)
2/3
```

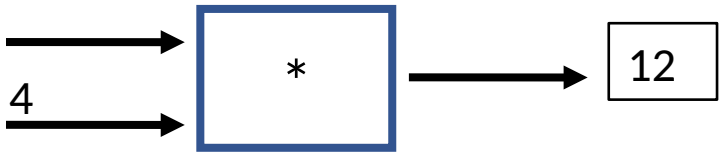
Exercícios – 18

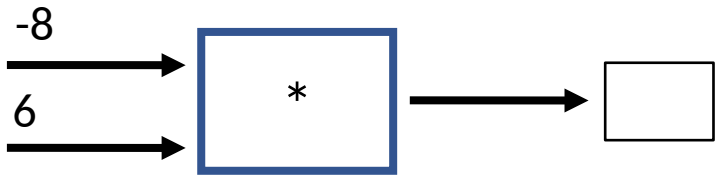
Complete as operações abaixo e em seguida execute-as no interpretador LISP para verificar o resultado.

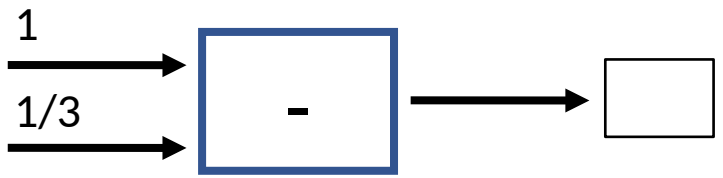


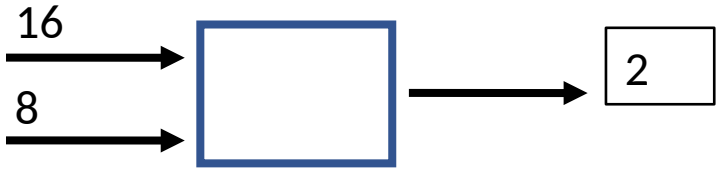


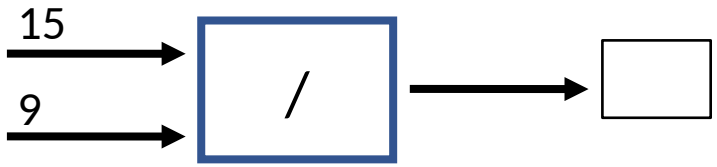


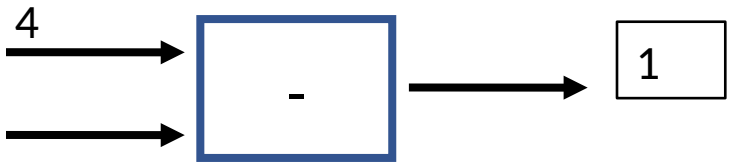


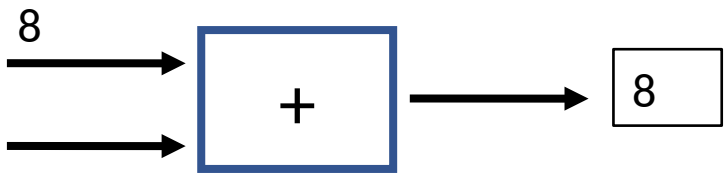






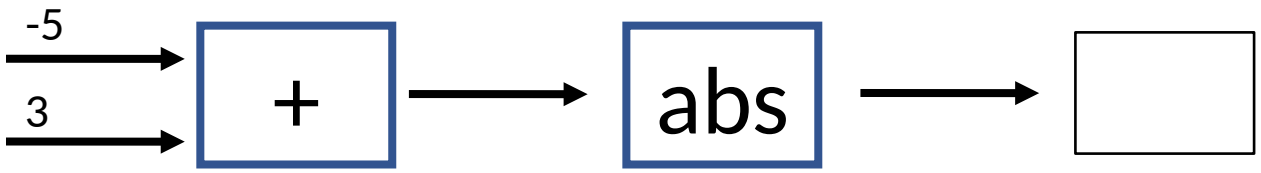






Exercícios – 18

Complete as operações abaixo e em seguida execute-as no interpretador LISP para verificar o resultado.



Na aula anterior

- LISt Processing:
  - LISP = Linguagem Infernal Somente de Parêntesis.
- Funções;
- Tudo em LISP que não for inteiro, ponto flutuante ou razão é um símbolo;

Tipos de Dados

- Symbols
  - Tipo de dado complexo
  - Representam identificadores, nomes de funções, variáveis e constantes;
  - Sequência de caracteres alfanuméricos;
  - Armazena:
    - name : próprio nome;
    - package: pacote ao qual pertence;
    - value: valor;
    - function: função;
    - plist: lista de propriedade.

name	“FOO”
package	
value	
function	
plist	

- Name:
  - LISP converte em maiúsculas;
  - Pode conter praticamente qualquer combinação de letras, números e caracteres especiais;
  - Caracteres especiais permitidos: +-*/@$%&_=,.~.
  - Caracteres geralmente não permitidos: #?[](){}|

Tipos de Dados

- Identificadores:
  - Símbolo que é usado para representar uma entidade (função, variável, constante);
- Como diferenciar um Symbol de um número:
  - Inteiro: sequência de dígitos (0 a 9), com/sem prefixo +/-;
  - Outros números: pontos flutuantes, razões;
  - Symbols: todo o resto.

Exercícios – 19

- Identifique o que é Symbol, Inteiro, Ratio ou Ponto Flutuante:

Tipo	Dado
S	AARDVARK
I	87
S	1-2-3-GO
I	1492
PF	3.14159265358979
R	22/7
S	ZEROP
S	ZERO
I	0
I	-12
S	SEVENTEEN

Tipo	Dado
S	UNWIND-PROTECT
S	+$
S	1+
I	+1
S	PASCAL_STYLE
S	B^2-4*A*C
S	FILE.REL.43
S	/USR/GAMES/ZORK

Símbolos especiais (T e NIL) / Predicados

- T:
  - Verdade, Veracidade, Sim.
- Nil:
  - Falso, Falsidade, Vazio, Não, Nulo.

- PREDICATES (Predicados):
  - São funções que retornam apenas T ou NIL;
- Funções Verdade:
  - São predicados cujos inputs são T ou NIL.

Predicados

Predicados	Retorno	Condição
NUMBERP	T	Se número
SIMBOLP	T	Se símbolo
ZEROP	T	Se zero
ODDP	T	Se ímpar
EVENP	T	Se par
<	T	Se i1 < i2
<=	T	Se i1 <= i2
>	T	Se i1 > i2
>=	T	Se i1 >= i2
EQUAL	T	Se i1 e i2 são expressões simbólicas idênticas
EQ	T	Se i1 e i2 são átomos e idênticos

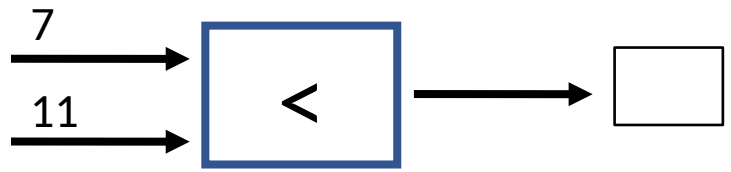
Predicados

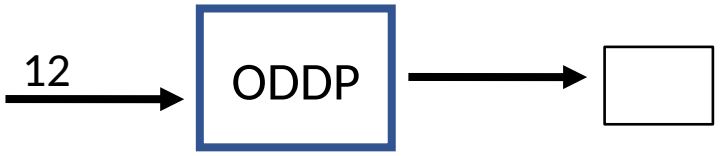
```
CL-USER> (numberp 42)
T
CL-USER> (numberp 'casa)
NIL
CL-USER> (symbolp 3.14)
NIL
CL-USER> (symbolp 'rua)
T
CL-USER> (zerop 12)
NIL
CL-USER> (oddp 13)
T
CL-USER> (evenp 7)
NIL
CL-USER> (> 12 5)
T
```

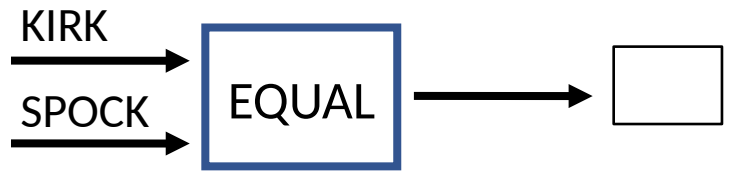
```
CL-USER> (>= 11 10)
T
CL-USER> (< 5 7)
T
CL-USER> (< 8 6)
NIL
CL-USER> (EQUAL 'GATO 'RATO)
NIL
CL-USER> (EQUAL 'GATO 'GATO)
T
CL-USER> (EQUAL 3.12 3.12)
T
CL-USER> (EQUAL 3.120 3.12)
T
CL-USER> (EQUAL 1/2 0.5)
NIL
```

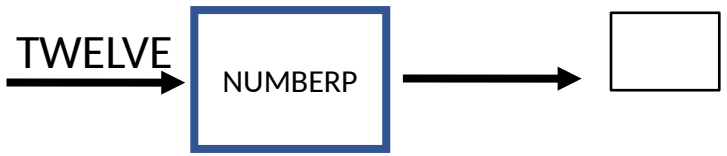
Exercícios – 20

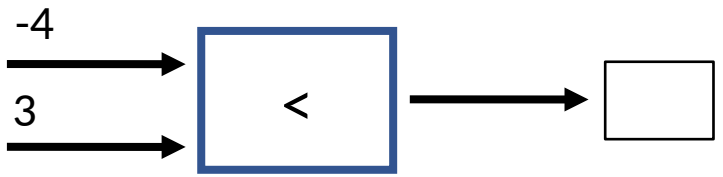
Complete as operações abaixo e em seguida execute-as no interpretador LISP para verificar o resultado.



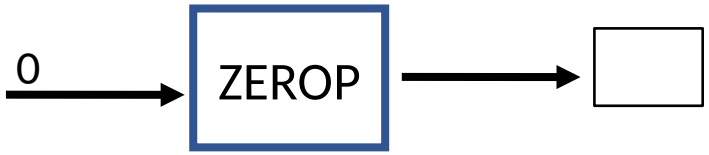


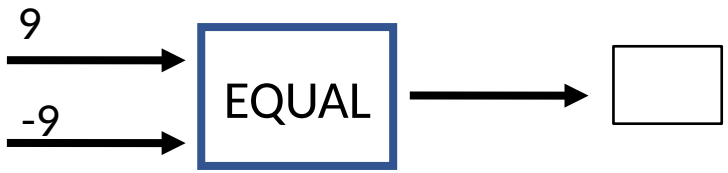






1





LISP - Avaliação

- Avaliação refere-se ao processo de calcular o valor de uma expressão ou de executar um programa;
- As expressões são compostas por listas, onde o primeiro elemento é interpretado como uma função e os elementos subsequentes são seus argumentos;
- Sequência:
  - Verificar se a expressão é um valor imediato, como um número ou uma string. O valor é retornado diretamente;
  - Verificar se a expressão é uma variável e, neste caso, o valor associado é retornado;
  - Verificar se a expressão é uma lista:
    - O primeiro elemento da lista é avaliado;
    - Os argumentos subsequentes são avaliados;
    - O valor do primeiro elemento (função ou operador) é aplicado aos argumentos avaliados.

LISP - Avaliação

- (+ 2 5)
  - Avaliação da função/operador +;
  - Os argumentos 2 e 3 são avaliados;
  - + é aplicado aos argumentos 2 e 5, resultando em 7;
  - O valor 5 é retornado como resultado da avaliação da expressão.

LISP - Avaliação

- E se for: '(+ 2 5)
  - O ' indica que deve ser tratada como uma lista literal e não ser avaliada;
  - A lista é retornada exatamente como está, sem realizar avaliação dos seus elementos;

```
'(+2 3)

(print '(+ 2 3))
```

```
Output:


(+ 2 3)
```

Funções

- Funções Primitivas
  - São funções que já estão prontas na linguagem (funções “built-in”);
  - Incorporadas diretamente na linguagem;
  - Implementadas e chamadas pelo interpretador;
  - Escritas em código de máquina ou em linguagem de baixo nível.

- Funções do Usuário:
  - Funções que o usuário cria a partir das funções primitivas e/ou de outras funções do usuário previamente definidas;
  - São definidas usando a sintaxe especial de definição de função de LISP;
  - Definidas pelo programador para realizar tarefas específicas dentro de um programa.

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:
  - Aritméticas: `+ - / *`
  - Relacionais: `> = < >= <=`
  - Lógicas: `and or not`
  - Manipulação de listas: `cons car cdr`
  - Condicionais: `if cond`

Funções

Aritméticas	Comparação	Listas	Fluxo de Controle
(+ 2 3); Soma	(> 7 1); Maior que	(cons 1 '(2 3 4): cria nova lista	(if (> 5 3) 'maior 'menor)
(- 5 1); Subtração	(< 1 4); Menor que	(car '(1 2 3)): 1º elemento	(cond ((> 5 3) 'maior)
(* 3 12); Multiplicação	(<= 7 7); Menor ou igual	(cdr '(1 2 3)): lista sem 1º elemento	(let ((x 5) (y 3)) (+ x y))
(/ 12 4); Divisão	(= 4 4); Igual a	(length '(1 2 3)): tamanho	
	(>= 9 1); Maior ou igual	(nth 2 '(1 2 3)): elemento na posição	
	(/= 6 1); Diferente		

Definindo Funções

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

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

  - Define uma função chamada *nome* que avalia expressão substituindo os símbolos da lista-de-argumentos pelos valores passados quando a função for invocada;

- Exemplo:

```
> (defun test (a b) (* 2 (+ a b)))
TEST
> (test 3 4)
14
```

Entendendo Funções em LISP: área do retângulo 01

- Crie uma função em LISP para calcular a área de um retângulo;
- Area = base * altura

```
(defun calcular_area_retangulo (base altura)
  (* base altura))
```

```
(print (calcular_area_retangulo 5 8 ))
```

```
40
```

Entendendo Funções em LISP: área do retângulo 02

- Crie uma função em LISP para calcular a área de um retângulo;
- Area = base * altura;
- Apresente uma mensagem para informar a área

```
(defun calcular_area_retangulo (base altura)
  (* base altura))
```

```
(format t "a area do retangulo e: ~d" (calcular_area_retangulo 5 8))
```

```
a area do retangulo e: 40
```

Entendendo Funções em LISP: format

```
(format t "a area do retangulo e: ~d" (calcular_area_retangulo 5 8))
```

- Usada para formatar e imprimir dados em um fluxo de saída;
- Permite a criação de strings formatadas combinando texto fixo com valores variveis:
  - (format destino controle &rest argumentos)
- Destino: o local onde o resultado será enviado (t para console);
- Controle: string de controle que define a formatação do resultado. Contém o texto fixo e diretrizes para formatação (~a para string; ~d para número decimal, ~f para número com ponto flutuante;
- Argumentos são os valores que se deseja inserir na string formatada e são passados como uma lista separada por espaços.

Entendendo Funções em LISP: área do retângulo 03

```
(defun calcular-area-retangulo (base altura)
  (* base altura))

(format t "Digite a base do retangulo: ") ; Solicitar entrada do usuário para base e altura
(let ((base (read)))
   (terpri)    ;imprimir uma linha em branco

  (format t "Digite a altura do retangulo: ")
  (let ((altura (read)))

     (terpri) ;imprimir uma linha em branco

    (format t "A area do retangulo e: ~d m2" (calcular-area-retangulo base altura))))
```

Entendendo Funções em LISP: área do retângulo 03

```
(defun calcular-area-retangulo (base altura)
  (* base altura))

; Solicitar entrada do usuário para base e altura
(format t "Digite a base do retangulo: ")
(let ((base (read)))
  ;imprimir uma linha em branco
  (terpri)
  (format t "Digite a altura do retangulo: ")
  (let ((altura (read)))
    ;imprimir uma linha em branco
    (terpri)
    ; Chamar a função calcular-area-retangulo com
    (format t "A area do retangulo e: ~d m2"
            (calcular-area-retangulo base altura))))
```

STDIN

```
13
7
```

Output:

```
Digite a base do retangulo:
Digite a altura do retangulo:
A area do retangulo e: 91 m2
```

Entendendo Funções em LISP: let

(let ((base (read))) ... (let ((altura (read)))

- Usada para criar um escopo local onde se pode definir variáveis locais e atribuir valores a ela. Limita a visibilidade das va

```
(let ( (variavel1 valor1)
       (variavel2 valor2)
       ...
       (variavelN valorN)
 corpo)
```

- Pode-se definir várias variáveis e atribuir valores a elas agrupando-as em pares (variável valor). O valor de cada variável é calculado uma única vez, no momento em que o let é avaliado;
- Em seguida pode-se executar o corpo, sequência em LISP, que serão avaliadas dentro do escopo do let;
- Variáveis definidas dentro do let estarão disponíveis apenas dentro desse escopo e não afetarão variáveis com o mesmo nome e fora dele.

Entendendo Funções em LISP: let x let*

- As funções let e let* são usadas para criar variáveis locais;
- Diferem na maneira como são definidas e vinculadas;
- O let permite que várias variáveis possam ser definidas simultaneamente, onde a expressão que define o valor da variável pode fazer referência a variáveis previamente definidas no mesmo escopo;
- Por outro lado, o let* apesar de permitir a definição de várias variáveis, difere no fato que cada variável é definida e vinculada uma de cada vez, na ordem em que são especificadas:
  - Ou seja, vpcê só pode fazer referência apenas às variáveis anteriores já definidas.

Entendendo Funções em LISP: let x let*

```
(defun exemplo-let ()
   (let
     ((x 1))
     (let
       ((y (+ x 2)))
       (+ x y)
     )
   )
)

(exemplo-let) ; Resultado: 4
```

```
(defun exemplo-let* ()
   (let*
     ( (x 1)
       (y (let
            ((x 2))
            (+ x 1)
          )
       )
     )
     (+ y x)
   )
)

(exemplo-let*) ; Resultado: 4
```

Entendendo Funções em LISP: setq

- Em LISP, tanto setq quanto let são usados para atribuir valores a variáveis, mas a diferença está em relação ao escopo das variáveis;
- A função setq atribui um valor a uma variável globalmente. Ou seja, a variável terá o mesmo valor em todo o escopo do da posteriormente em outra

```
(setq x 10)
(setq y 5)
(setq z (+ x y))

(print (format t "O valor de z e: ~a" z))
```

Output:

```
O valor de z e: 15
NIL
```

Entendendo Funções em LISP: read

```
(let ((base (read)))    ...  (let ((altura (read)))
```

- Usada para ler dados de entrada do usuário

```
(read)
```

- Espera que o usuário insira um valor no formato adequado e retorna esse valor como um objeto LISP;
- Frequentemente usada em conjunto com a função format para criar interaçes com o usuário, permitindo a entrada de dados durante a execução de um programa LISP;
- Neste caso, utilizamos a função read para receber a entrada do usuário para a base e altura do retângulo;
- A função read espera que o usuário insira valores numéricos válidos, e esses valores são então atribuídos às variáveis base e altura usando a função let.

Entendendo Funções em LISP: terpri

Output:

```
Digite a base do retangulo: Digite
```

```
...
  (terpri)    ;imprimir uma linha em branco
...
```

Output:

```
Digite a base do retangulo:
Digite a altura do retangulo:
A area do retangulo e: 91 m2
```

Entendendo Funções em LISP: Verificar nº positivo

```
(defun verifica-positivo (num)
  (if (> num 0)
      (format t "O numero ~a e positivo.~%" num)
      (format t "O numero ~a nao e positivo.~%" num)))

(print (verifica-positivo 5)) ; O número 5 é positivo.
(print (verifica-positivo -2)) ; O número -2 não é positivo.
```

```
Output:

O numero 5 e positivo.

NIL O numero -2 nao e positivo.

NIL
```

Entendendo Funções em LISP: if

```
(if (> num 0)
    (format t "O numero ~a e positivo.~%" num)
    (format t "O numero ~a nao e positivo.~%" num)))
```

- Forma condicional que permite executar diferentes blocos de código com base em um condição:
  - (if (condicao) (entao) (senão))
- Condição é uma expressão que é avaliada como verdadeira ou falsa;
- Se a (condição) for verdadeira (ou seja, um valor diferente de nil), o (então) é executado;
- Se a (condição) for falsa (ou seja, nil), o (senão) é executado caso exista.

Entendendo Funções em LISP: Temperatura

```
(defun verificar-temperatura (temperatura)
  (cond ((< temperatura 0) "Está muito frio!")
        ((< temperatura 10) "Está frio.")
        ((< temperatura 20) "Está agradável.")
        ((< temperatura 30) "Está quente.")
        (t "Está muito quente!")))

; Exemplos de uso:
(verificar-temperatura -5) ; retorna "Está muito frio!"
(verificar-temperatura 15) ; retorna "Está agradável."
(verificar-temperatura 35) ; retorna "Está muito quente!"
```

Output:

```
"Esta muito frio!"
"Esta agradavel."
"Esta muito quente!"
```

Entendendo Funções em LISP: cond

- Construção condicional em LISP que permite executar diferentes ações com base em uma série de condições;

```
(cond
  (condição-1 expressão-1)
  (condição-2 expressão-2)
  ...
  (condição-N expressão-N)
  (t expressão-padrão) )
```

- Avalia cada par de condição-expressão sequencialmente, da esquerda para a direita;
- Cada condição é avaliada em ordem e, quando uma condição é verdadeira, a expressão correspondente é avaliada e seu valor retornado;
- Se nenhuma das condições anteriores for verdadeira, o par '(t expressão-padrão) ' é opcionalmente fornecido como uma condição padrão. Se todas as condições anteriores falharem, a expressão-padrão é avaliada e seu valor retornado;

Entendendo Funções em LISP

- Crie uma função em LISP para calcular o Indice de Massa Corporal (IMC) com base no peso e na altura fornecidos;
- IMC = peso / altura^2

```
(defun  calc_imc  (peso altura)
    ( let  ( (imc     (/ peso  (* altura altura ) )  )
             imc)
)
```

```
(print (calc_imc 75 1.75))
```

```
24.489796
```

Entendendo Funções em LISP

```
(defun encontrar-maior-inteiro (lista)   ; a função encontrar-maior-inteiro recebe uma lista
  (if (null lista)                       ; a função null list verifica se a lista é vazia. Se for, retorna nil
      nil
      (let ((maior (car lista)))         ; caso contrário, a função let armazena o primeiro elemento da lista em maior
        (dolist (elemento (cdr lista))   ; a função dolist percorre todos os demais elementos da lista
          (when (and (integerp elemento) ; no loop, verifica se elemento é um inteiro e se é maior
                     (> elemento maior) ) ; do que o valor armazenado
            (setq maior elemento)))      ; se for maior, atualiza a variável com o valor maior
        maior)))                         ; retorna o valor de maior
```

```
CL-USER> (encontrar-maior-inteiro '(100 2000 3 44 99 1))
2000
```

Entendendo Funções em LISP - dolist

- Construção comumente usada em LISP para percorrer uma lista e executar ações para cada elemento;
- É uma forma de laço de repetição que simplifica o processo de iterar sobre os elementos de uma lista;
  - (dolist (variavel1 lista & optional resultado) corpo)
- Variável é o nome da variável que será associada a cada elemento da lista durante a iteração;
- Lista é a lista que será percorrida;
- Corpo é uma sequência de expressões que serão executadas para cada elemento;

Entendendo Funções em LISP - dolist

```
(setq lista '(1 2 3 4 5))
(dolist (elemento lista)
  (print elemento)
```

```
(setq lista '(1 2 3 4 5))
(setq soma 0)
(dolist (elemento lista soma)
  (setq soma (+soma elemento)))
```

- Construção comumente usada em LISP para percorrer uma lista e executar ações para cada elemento;
- É uma forma de laço de repetição que simplifica o processo de iterar sobre os elementos de uma lista;
  - (dolist (variavel1 lista & optional resultado) corpo)
- Variável é o nome da variável que será associada a cada elemento da lista durante a iteração;
- Lista é a lista que será percorrida;
- Corpo é uma sequência de expressões que serão executadas para cada elemento;

Entendendo Funções em LISP - when

- Forma condicional que permite executar uma sequência de expressões se uma condição for verdadeira;
- A condição é avaliada e, se for verdadeira, as expressões subsequentes são executadas em ordem;
- Se a condição for falsa, o corpo do "when" é simplesmente ignorado e a execução continua após o "when";

Entendendo Funções em LISP - when

```
(defun verifica-maior-que-zero (valor)
   (when (> valor 0)
                (format t "O valor é maior que zero.~%")
                (format t "O dobro do valor é ~d.~%" (* 2 valor))
   )
 )

(verifica-maior-que-zero 5) ; Exibe as mensagens, pois 5 é maior que zero

(verifica-maior-que-zero -3) ; Não exibe as mensagens, pois -3 não é maior que zero
```

Entendendo Funções em LISP - when

```
(defun verifica-numero (valor)
  (cond
    ((when (= valor 0)
       (format t "O valor é igual a zero.~%")))
    ((when (> valor 0)
       (format t "O valor é maior que zero.~%")))
    ((when (< valor 0)
       (format t "O valor é menor que zero.~%")))))

(verifica-numero 0)   ; Exibe "O valor é igual a zero."
(verifica-numero 5)   ; Exibe "O valor é maior que zero."
(verifica-numero -3)  ; Exibe "O valor é menor que zero."
(verifica-numero 10)  ; Não exibe nenhuma mensagem, nenhuma condição é verdadeira.
```

Entendendo Funções em LISP

```
(defun percorrer-lista (lista)
  (dolist (elemento lista)
      (if (listp elemento)
          (percorrer-lista elemento)
          (format t "~a~%" elemento)
      )
  )
)

(percorrer-lista '(1 2 (3 4 5) 6 7))
```

```
Output:

1
2
3
4
5
6
7
```

Funções

```
CL-USER> (defun fatorial (n)
  (if (<= n 1)
      1
      (* n (fatorial (- n 1)))))

(format t "Digite um número: ")
(let ((numero (read)))
  (format t "O fatorial de ~a é: ~a" numero (fatorial numero)))

Digite um número: 100
O fatorial de 100 é: 933262154439441526816992388562667004907159682
NIL
CL-USER>
```

```
O fatorial de 100 é: 933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000
NIL
```

Tipos de dados

- Símbolos:
  - São utilizados para representar identificadores, nomes de variáveis, nomes de funções e outros elementos semelhantes.
  - Eles são representados por sequências de caracteres alfanuméricos, começando com uma letra ou um caractere especial, seguido por zero ou mais letras, dígitos ou caracteres especiais.
  - São usados para associar valores a variáveis ou para referenciar funções e outras estruturas de dados.
  - Eles são geralmente utilizados para fins de programação, como controle de fluxo, atribuição de valores e definição de funções.
- Strings:
  - Sequências de caracteres delimitadas por aspas duplas ("...") ou aspas simples ('...').
  - São usadas para representar dados textuais, como mensagens de texto, nomes de arquivos, conteúdo de documentos, entre outros.
  - Diferentemente dos símbolos, as strings são tratadas como dados e não podem ser avaliadas diretamente como expressões Lisp.
  - As strings podem ser concatenadas, divididas em substrings, comparadas e manipuladas de várias formas usando funções Lisp.

Tipos de dados

- Números podem ser inteiros ou de ponto flutuante e suportam operações matemáticas comuns;
- Listas são estruturas de dados compostas que podem conter elementos de qualquer tipo, inclusive outras listas;
-

Tipos de dados - Símbolos

- (defparameter nome "Alice")
  - nome é um símbolo que representa uma variável;
- (defun saudacao (nome)
  - saudacao é um símbolo que represenya uma função;
- (defparameter lista '(a b c))
  - Neste caso a, b e c são símbolos que representam elementos de uma lista;
- (if condicao)
  - Neste caso, condição é um símbolo que representa uma variável ou expressão cujo valor será avaliado por if

01 – Avaliar as expressões básicas

```
(+ 2 3)
(-5 -1)
(* 3 12)
(/ 12 4)
(/ 18 5)
(> 7 1)
(< 1 4)
(<= 7 7)
(= 4 4 )
(>= 9 1)
(/= 6 1)
(or (< 2 3) (> 2 3))
(= 'a 'b)
(eq 'a 'b)
(eq 'a 'a)
(eq '(a b) '(a b))
(equal '(a b) '(a b))
```

= É usado para testar igualdade numérica;

eq É usado para testar a identidade dos objetos. Verifica se os dois argumentos se referem ao mesmo objeto na memória. Usado para comparar símbolos e outras estruturas imutáveis. Não deve ser usado para strings;

equal É usado para testar a igualdade de valores de objetos. Verifica se os argumentos tem o mesmo valor, mesmo que não sejam o mesmo objeto na memória. Usado para strings, listas.

```
(cons 1 '(2 3 4)
(car '(1 2 3))
(cdr '(1 2 3))
(car (cdr '(1 2 3)))
(cdr (cdr '(1 2 3)))
(length '(1 2 3))
(nth 2 '(1 2 3))
(cond ((= 1 2) 'a)
      ((> 2 3) 'b)
      ((<3 4) ‘c))
(if (> 5 3) 'maior 'menor)
(cond ((> 5 3) 'maior)
(let ((x 5) (y 3)) (+ x y))
```

Tipos de dados - Strings

- (setq mensagem "Olá, mundo!")
  - A string "Olá, mundo" é atribuída à variável mensagem;
- (setq vazio "")
  - Aqui atribuímos uma string vazia a uma variável;
- ((setq nome "Claudio")
  (setq sobrenome "Junior")
  (setq nome_completo (concatenate 'string nome " " sobrenome))

02 – Análise da área do retângulo

```
(
  defun calcular-area-retangulo (base altura)
  (* base altura)
)

; Solicitar entrada do usuário para base e altura
(format t "Digite a base do retangulo: ")

(let
  (
    (base (read))
  )
  ;imprimir uma linha em branco
  (terpri)
  (format t "Digite a altura do retangulo: ")

  (let
    (
      (altura (read))
    )
    ;imprimir uma linha em branco
    (terpri)
    ; Chamar a função calcular-area-retangulo com os valores digitados pelo usuário
    (format t "A area do retangulo e: ~d m2"
      (calcular-area-retangulo base altura))
    )
)
```

STDIN

```
12
10
```

Output:

```
Digite a base do retangulo:
Digite a altura do retangulo:
A area do retangulo e: 120 m2
```

03 – Verificar um valor

```
1  (defun verifica-numero (valor)
2    (when (= valor 0) (format t "O valor ~d e igual a zero.~%" valor) )
3    (when (> valor 0) (format t "O valor ~d e maior que zero.~%" valor))
4    (when (< valor 0) (format t "O valor ~d e menor que zero.~%" valor))
5   )
6
7  (verifica-numero 0) ; Exibe "O valor é igual a zero."
8  (verifica-numero 5) ; Exibe "O valor é maior que zero."
9  (verifica-numero -3) ; Exibe "O valor é menor que zero."
10 (verifica-numero 10) ;
11
12
13 (let
14     (
15       (val (read))
16     )
17     ;imprimir uma linha em branco
18     (terpri)
19     ; Chamar a função varefica-numero com os valores digitados pelo usuário
20     (verifica-numero val)
21 )
```

04 – Percorrer uma lista

```
1  (defun percorrer-lista (lista)
2     (dolist (elemento lista)
3       (if (listp elemento)
4          (percorrer-lista elemento)
5          (if (oddp elemento)
6             (format t "O elemento ~a da lista eh impar~%" elemento)
7             (format t "O elemento ~a da lista eh par~%" elemento)
8          )
9       )
10    )
11 )
12
13 (percorrer-lista '(1 2 (3 4 5) 6 7 ( 9 11 13)))
```

Output:

```
O elemento 1 da lista eh impar
O elemento 2 da lista eh par
O elemento 3 da lista eh impar
O elemento 4 da lista eh par
O elemento 5 da lista eh impar
O elemento 6 da lista eh par
O elemento 7 da lista eh impar
O elemento 9 da lista eh impar
O elemento 11 da lista eh impar
O elemento 13 da lista eh impar
```

05 – Verificar a idade das pessoas

```
(defun verifica-idade (idade)
   (terpri)  (format t "Uso do cond:")  (terpri)
   (cond
        ((< idade 18) (format t "Voce tem ~a anos. Voce eh menor de idade~%" idade ))
        ((< idade 64) (format t "Voce tem ~a anos. Voce eh um adulto~%" idade ))
        (t (format t "Voce tem ~a anos. Voce eh idoso~%" idade )))
   (terpri) (format t "Uso do if:") (terpri)
   (if (< idade 18)
       (format t "Voce tem ~a anos. Voce eh menor de idade~%" idade )
       (if (< idade 64)
           (format t "Voce tem ~a anos. Voce eh adulto~%" idade )
           (format t "Voce tem ~a anos. Voce eh idoso~%" idade )
       )
   )
   (terpri) (format t "Uso do when:") (terpri)
   (when (< idade 18) (format t "Voce tem ~a anos. Voce eh menor de idade~%" idade ))
   (when (and (< idade 64)(>= idade 18)) (format t "Voce tem ~a anos.Voce eh adulto~%" idade))
   (when (>= idade 64)  (format t "Voce tem ~a anos. Voce eh idoso~%" idade ))
)
(
  let (
       (idade (read))
      )
      (verifica-idade idade)
```

Output:

```
Uso do cond:
Voce tem 11 anos. Voce eh menor de idade

Uso do if:
Voce tem 11 anos. Voce eh menor de idade

Uso do when:
Voce tem 11 anos. Voce eh menor de idade
```

06 – Cálculo do IMC

```lisp
(defun calcular-imc (peso altura)
  (/ peso (* altura altura))
)


(defun determinar-categoria (imc)
  (cond ((< imc 18.5) "Abaixo do peso")
        ((< imc 24.9) "Peso ideal")
        ((< imc 29.9) "Sobrepeso")
        ((< imc 34.9) "Obesidade grau 1")
        (t "Obesidade grau 2")
  )
)



(main) ; programa principal

```

```lisp
(defun main ()
  (format t "Calculadora de IMC~%")
  (terpri)
  (format t "Digite o peso (em kg): ")

  (let ((peso (read)))
    (terpri)
    (format t "Digite a altura (em metros): ")
    (let
      ((altura (read)))
      (let
        (
          (imc (calcular-imc peso altura))
        )
        (terpri)
        (format t "Seu IMC e ~2,2f~%" imc)
        (terpri)
        (format t "Categoria: ~a~%" (determinar-categoria imc))
      )
    )
  )
)
```

Output:

```
Calculadora de IMC

Digite o peso (em kg):
Digite a altura (em metros):
Seu IMC e 29.73

Categoria: Sobrepeso
```

Listas – Exemplo intercalação

```lisp
(defun intercalar-listas (lista1 lista2)
  (if (or (null lista1) (null lista2))
      (append lista1 lista2)
      (cons (car lista1)
            (cons (car lista2)
                  (intercalar-listas (cdr lista1) (cdr lista2))
            )
      )
  )
)

;; Exemplo de uso:
(let ((lista1 '(1 3 5))
      (lista2 '(2 4 6)))
  (print (intercalar-listas lista1 lista2)))
```

Output:

(1 2 3 4 5 6)

Caractere

```
(let ((my-string "Ola Mundo!"))
  (let ((char1 (char my-string 0)))
    (format t "Primeiro caractere da string: ~c~%" char1)))


(setq abc "Ola Mundo!")
(print (char abc 0))
(terpri)
(terpri)

(setq abc "Ola Mundo!")
(format t "~c~%" (char abc 0))


(let ((sstring "Ola Mundo!"))
  (let ((char-list (coerce sstring 'list))) ; converte a string em lista
    (let ((char1 (nth 0 char-list))) ; aqui já vimos
      (format t "Primeiro caractere da string: ~c~%" char1)))) ;
```

Output:

```
Primeiro caractere da string: 0

#\0

0
Primeiro caractere da string: 0
```