

# LINGUAGEM LISP

Primeiros passos com Common LISP (CL)

## INCLUI:

- Tipos de dados
- Variáveis e constantes
- Formatação numérica
- Ações aritméticas
- Ações relacionais
- Ações lógicas
- Decisões e laços
- Programação estruturada
- Programação funcional
- Pacotes

Abrange  
Interpretadores  
**CLISP**  
&  
**SBCL**



José Augusto N. G. Manzano

## ATENÇÃO

Caso deseje obter uma cópia impressa deste material em formato livro poderá fazê-lo junto as plataformas de publicação:

Clube de autores: <https://clubedeautores.com.br/>

AgBook: <https://www.agbook.com.br>

Em ambas as plataformas efetue busca por "augusto manzano" e escolha o livro desejado.

José Augusto N. G. Manzano

# **Linguagem LISP**

**Primeiros passos com Common LISP (CL)**

**1ª Edição**

**São Paulo  
2019 – Propes Vivens**

**Todos os direitos reservados.** Proibida a reprodução total ou parcial, por qualquer meio ou processo, especialmente por sistemas gráficos, microfílmicos, fotográficos, reprográficos, fonográficos, videográficos, internet, e-books. Vedada a memorização e/ou recuperação total ou parcial em qualquer sistema de processamento de dados e a inclusão de qualquer parte da obra em qualquer programa juscibernético. Essas proibições aplicam-se também às características gráficas da obra e à sua editoração. A violação dos direitos autorais é punível como crime (art. 184 e parágrafos, do Código Penal, conforme Lei nº 10.695, de 07.01.2003) com pena de reclusão, de dois a quatro anos, e multa, conjuntamente com busca e apreensão e indenizações diversas (artigos 102, 103 parágrafo único, 104, 105, 106 e 107 itens 1, 2 e 3 da Lei nº 9.610, de 19.06.1998, Lei dos Direitos Autorais).

O Autor acredita que todas as informações aqui apresentadas estão corretas e podem ser utilizadas para qualquer fim legal. Entretanto, não existe qualquer garantia, explícita ou implícita, de que o uso de tais informações conduzirá sempre ao resultado desejado. Os nomes de sites e empresas, porventura mencionados, foram utilizados apenas para ilustrar os exemplos, não tendo vínculo nenhum com o livro, não garantindo a sua existência nem divulgação.

Conteúdo adaptado ao Novo Acordo Ortográfico da Língua Portuguesa, em execução desde 1º de janeiro de 2009.

**Dados Internacionais de Catalogação na Publicação (CIP)**  
**(Ficha catalográfica confeccionada pelo autor)**

M296l Manzano, José Augusto Navarro Garcia

Linguagem LISP: Primeiros passos com Common LISP (CL) / José Augusto N. G. Manzano -- 1. ed. -- São Paulo : Propes Vivens, 2019.  
232 p.

Bibliografia.

ISBN: 978-85-923720-6-4

1. LISP (Linguagem de programação para computadores)  
I. Título.

13-08576

CDD-005.133

Índices para catálogo sistemático:

1. LISP : Linguagem de programação : Computadores : Processamento de dados 005.133

# LINGUAGEM

Linguagem: **LISP**  
Autor: **John McCarthy**  
Sítio: **<https://lisp-lang.org/> (COMMON LISP Oficial Site)**

# INTERPRETADORES

Software: **CLISP**  
Sítios: **<http://clisp.org/>  
<https://sourceforge.net/projects/clisp/>**

Software: **SBCL**  
Sítios: **<http://sbcl.org/>  
<https://sourceforge.net/projects/sbcl/>**



## OUTROS INTERPRETADORES

### Gratuitos

Allegro CL Free Express Edition (<https://franz.com/downloads/clp/survey>)

Armed Bear Common Lisp (<https://abcl.org/>)

CL REPL (Google Play)

Clasp (<https://github.com/clasp-developers/clasp>)

Clozure Common Lisp (<https://ccl.clozure.com/>)

CMUCL (<https://www.cons.org/cmuc1/>)

Corman Lisp (<https://github.com/sharplispers/cormanlisp>)

Embeddable Common-Lisp (<https://common-lisp.net/project/ecl/>)

LispWorks Personal Edition (<http://www.lispworks.com/downloads/index.html>)

ManKai Common Lisp (<https://common-lisp.net/project/mkcl/>)

MicroLisp (<https://mr.gy/software/microlisp/microlisp-architektur.html>)

### Comerciais

Allegro CL (<https://franz.com/>): +/- a partir de US\$ 600.00

Golden Common LISP (<http://www.goldhill-inc.com/>): +/- US\$ 2,000.00

LispWorks (<http://www.lispworks.com/>): +/- US\$ 5,000.00

As ferramentas aqui indicadas são opções disponíveis para sistemas operacionais: Android, FreeBSD, Linux, Windows, UNIX, entre outros





## **AGRADECIMENTOS**

À minha esposa Sandra e à minha filha Audrey, motivos de inspiração ao meu trabalho.

Aos meus alunos por me incentivarem continuamente quando me questionam sobre temas que ainda desconheço e me levam a pesquisar muito mais.

Especialmente quero agradecer a pessoa de Jefferson de T. Pereira pela contribuição a melhora deste trabalho.

Vida longa e próspera!

Posso deitar-me, dormir e despertar,  
pois é o Senhor quem me ampara.

Sl 3, 6



# SUMÁRIO

## Capítulo 1 - Introdução

1.1 Linguagem LISP.....	15
1.2 Obtenção e instalação do CLISP .....	17
1.3 Obtenção e instalação do SBCL .....	23
1.4 Qual ambiente usar: CLISP ou SBCL? .....	29
1.5 Características básicas sobre LISP .....	30
1.6 Interação básica (primeiro contato).....	34

## Capítulo 2 - Recursos básicos

2.1 Ações interativas.....	39
2.2 Tipo de dado numérico .....	44
2.3 Identificação de tipos de dados.....	50
2.4 Funcionalidades matemáticas.....	51
2.5 Variáveis e constantes .....	57
2.6 Formatação numérica .....	69

## Capítulo 3 - Recursos intermediários

3.1 Tipo de dado caractere e cadeia.....	75
3.2 Conversão número texto e vice versa.....	85
3.3 Funcionalidades relacionais.....	88
3.4 Tipo de dado lista.....	93
3.5 Tipo de dado símbolo.....	110
3.6 Uso de decisões.....	113
3.7 Funcionalidades lógicas.....	119
3.8 Uso de laços .....	124
3.9 Tipo de dado função .....	139

## Capítulo 4 - Recursos avançados

4.1 Tipo de dado matriz .....	153
4.2 Entrada e saída de dados .....	170
4.3 Variáveis locais .....	172
4.4 Macros .....	175

4.5 Funções de tempo .....	181
4.6 Randomização.....	186
4.7 Função anônima (lambda) .....	188
4.8 Tipo de dado tabela de símbolo (hash) .....	190
4.9 Tipo de dado estrutura.....	199

## **Capítulo 5 - Programação CL**

5.1 Programação estruturada .....	203
5.2 Definição de comentários .....	207
5.3 Pacotes (package).....	210
5.4 Programação funcional.....	218

<b>Bibliografia .....</b>	<b>229</b>
---------------------------	------------

## PREFÁCIO

LISP designa uma grande família de linguagens de programação para computadores, como: Run, Cloujure, Racket, Scheme, AutoLISP, Emacs LISP, Common LISP, entre outras.

Nesta obra faz-se introdução ao dialeto **Common LISP** (LISP comum) referenciado neste texto como **CL**. Não é foco explorar a linguagem em sua totalidade, a intenção é fornecer um caminho de aprendizagem e orientação básica ao público brasileiro carente de obras neste contexto.

O capítulo 1 apresenta informações sobre a linguagem, seu surgimento e características. Mostra como efetuar a obtenção e instalação dos interpretadores CLISP e SBCL, apesar de dar ênfase ao uso do interpretador SBCL.

No capítulo 2 é realizada a apresentação de diversos recursos fundamentais da linguagem. São mostrados os tipos de dados numéricos suportados, são apresentadas algumas funcionalidades matemáticas, além de demonstrar a definição de variáveis e constantes. Outro destaque do capítulo é a demonstração e uso dos recursos para apresentação formatada de valores numéricos.

O capítulo 3 aborda os tipos de dados caracteres, lista, símbolos e funções. São indicados o uso de operações de decisão e laços, além do uso de funcionalidades relacionais e lógicas. Na apresentação de funções é indicado o uso de recursividade simples e de cauda.

O capítulo 4 aborda os tipos de dados matrizes, tabelas de símbolos e estruturas. Demonstra o uso de ações de entrada e saídas de dados, funções anônimas e de tempo.

O capítulo 5 demonstra o uso dos conhecimentos indicados nos capítulos anteriores. São apresentados os efeitos de programação estruturada e funcional, além de demonstrar as regras de uso de linhas de comentários, desenvolvimento de pacotes e introdução a programação funcional.

Augusto Manzano



## **SOBRE O AUTOR**

**José Augusto Navarro Garcia Manzano** é brasileiro, nascido no estado de São Paulo, capital, em 26 de abril de 1965; professor e mestre com formação como Bacharel em Ciências Econômicas, Tecnólogo em Análise e Desenvolvimento de Sistemas e Licenciatura em Matemática. Atua na área de Tecnologia da Informação a partir do desenvolvimento de softwares comercial e de telecomunicações, ensino e treinamento desde o ano de 1986. Na carreira docente, iniciou suas atividades em cursos livres, trabalhando, posteriormente, em empresas de treinamento e nos ensinos técnico e superior. Trabalhou em empresas como Abak, Servimec, Montreal, CompuCenter, Cebel, SPCI, BEPE, Origin, OpenClass, entre outras.

Atualmente, é professor com dedicação exclusiva ao Instituto Federal de Educação, Ciência e Tecnologia de São Paulo (IFSP), antiga Escola Técnica Federal. Em sua carreira docente, possui condições de ministrar componentes curriculares de lógica de programação (algoritmos e técnicas de programação), estrutura de dados, microinformática, informática, linguagens de programação (estruturadas, orientadas a objetos e funcionais), engenharia de software, engenharia da Informação, arquitetura e organização de computadores, além de temas relacionados a tecnologias Web. Possui forte conhecimento das linguagens de programação Classic BASIC, COMAL, Logo, Object PASCAL, Assembly, FORTRAN, C, C++, D, Java, Modula-2, Structured BASIC, C#, Lua, HTML, XHTML, VBA, Ada, Rust, Hope, Python, Haskell, OCaml, Groovy, Julia, LISP e JavaScript/JScript. Possui mais de uma centena de obras publicadas, além de artigos no Brasil e no exterior.





## 1

# Introdução

---

*Este capítulo mostra detalhes e características básicas de operação da linguagem de programação LISP. São apresentadas informações sobre a obtenção, instalação e uso dos programas CLISP e SBCL, além de seus respectivos ambientes interativos de utilização. É demonstrado de forma simples o uso de algumas operações de cálculos matemáticos básicos e apresentação de cadeias de caracteres.*

## 1.1 LINGUAGEM LISP

---

A linguagem LISP sigla de **LIS**t **PRO**cessing (processamento de listas) foi desenvolvida pelo matemático e cientista da computação John McCarthy no verão de 1956 durante a realização do projeto de pesquisa sobre Inteligência Artificial no Dartmouth College, do qual era seu líder tendo sido inicialmente implementada em 1958.

LISP foi desenvolvida para ser usada como linguagem de processamento de listas algébricas para computação simbólica usada em sistemas de Inteligência Artificial. Um ano após seu lançamento é apresentada a primeira versão da linguagem chamada LISP 1, depois em 1962 é apresentada LISP 1.5 e em 1964 é apresentada LISP 2 que apesar de diversas melhorias sobre LISP 1.5 não foi amplamente usada. Aproximadamente 22 anos após o lançamento de LISP 1.5 é lançada em 1984 a versão COMMON LISP (ou CL tema desta obra) e em 1994 é lançada o padrão ANSI COMMON LISP identificado como ANSI X3.226-1994 e atualmente o padrão ANSI INCITS 226-1994, tendo sua documentação um total de 1153 páginas.

LISP é a segunda linguagem de programação de alto nível mais antiga em uso até os dias atuais, depois de FORTRAN (**FOR**mula **TRAN**slator) lançada em 1954.

LISP influenciou o desenvolvimento de diversas outras linguagens, tonando-se na verdade representante de uma família de diversas linguagens de programação, destacando-se: LISP 1.5 (1955); MacLISP (1965); LOGO (1968); InterLISP (1970); SmallTalk (1971); ZetaLISP, Scheme e NIL de **New Implementation of LISP** (1975); COMMON LISP (1980); CCL de **Clozure CL** (1984), não confundir **Clozure** com **Clojure**; Emacs LISP, AutoLISP e AutoLISP da série de programas AutoCAD (1985); CLOS de **Common Lisp Object System** (1989) incorporada a linguagem CL; EuLISP, Racket e ISLISP (1990); SBCL (1999); ACL2 (2005); Clojure (2005) e da brasileira Run (2017), entre outras aqui não citadas.

Entre os períodos de 1956 e 1984 ocorreram o lançamento de diversos dialetos da linguagem LISP que se tornaram incompatíveis entre si, apesar da versão LISP 1.5 ter sido a mais amplamente aceita. No início da década de 1970 dois eram os dialetos LISP mais populares: MACLISP desenvolvido pelo projeto MAC (**Mathematics and Computation**) do MIT (**Massachusetts Institute of Technology**) em 1965 e INTERLISP desenvolvida pela empresa BBN Technologies em 1967, posteriormente chamado de BBN LIST.

Tanto MACLISP como INTERLISP estenderam LISP 1.5 com funcionalidades que acabaram mais tarde influenciando o desenvolvimento do padrão CL.

O primeiro esforço de padronização da linguagem ocorreu em 1969, mas um efetivo trabalho só veio em 1984 com a publicação do documento "**Common Lisp: the Language**" de Guy L. Steele que formou a base para o padrão CL, tendo este trabalho sido agenciado pela agência DARPA (**Defense Advanced Research Projects Agenc**).

O passo seguinte ocorreu em 1990 quando do lançamento da CLOS (**Common Lisp Object System**) com suporte a orientação a objetos e em 1994 ocorre a publicação da segunda edição do documento "**Common Lisp: the Language**".

Embora os padrões ANSI COMMON LISP e COMMON LISP possuam diferenças a especificação ANSI COMMON LISP reconhece o padrão COMMON LISP publicado em 1990 e 1994.

Em 1987 é iniciado o trabalho de padronização ISO (**International Organization for Standardization**) formado pelo grupo SC22 WG16 (LISP WG) que estabeleceu um padrão mínimo e compacto para a linguagem LISP diferentemente do padrão adotado pelo CL.

## 1.2 OBTENÇÃO E INSTALAÇÃO DO CLISP

O programa CLISP (de **COMMON LISP**) inclui para uso um interpretador, um compilador de **bytecode**, um depurador e os complementos CLOS e MOP (**Meta Object Protocol**), além de possibilitar o uso de expressões regulares POSIX e Perl entre outros recursos. CLISP pode ser executado em diversos sistemas operacionais, tais como: Linux, FreeBSD, NetBSD, OpenBSD, Solaris, Tru64, HP-UX, BeOS, IRIX, AIX, Mac OS X e Windows necessitando apenas de 4 MB (**megabyte**) de memória RAM (**Random Access Memory**), tendo sido criado por Bruno Haible e Michael Stoll inicialmente em 1987 na Alemanha. CLISP é um programa de uso livre sob a Licença Pública Geral GNU. O único inconveniente atualmente é o fato de parecer o projeto congelado, pois não a atualização da ferramenta desde 23 de abril de 2010.

Para obter informações diversas sobre o programa CLISP acesse o sítio oficial do projeto <http://clisp.org> como mostra a figura 1.1.

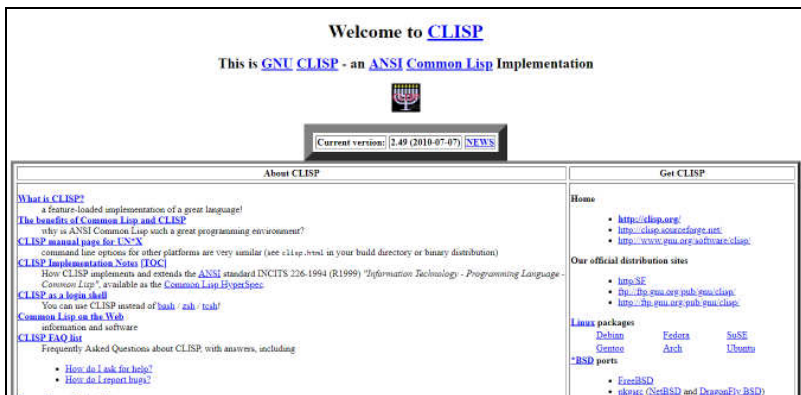


Figura 1.1 - Tela da página do projeto CLISP.

Para obter cópia do programa é necessário fazer acesso fora do sítio oficial. Assim sendo, acesse o sítio <https://sourceforge.net/projects/clisp/>, como apresenta a figura 1.2.

Na tela apresenta acione com um clique do ponteiro do **mouse** o botão verde denominado Download, aguarde o arquivo de programa `clisp-2.49-win32-mingw-big.exe` ser copiado em seu sistema, normalmente para a pasta de Downloads.

Após a cópia do programa `clisp-2.49-win32-mingw-big.exe` faça sua execução a partir de seu acionamento com um duplo clique do ponteiro do **mouse**. Se apresentada a caixa de diálogo de controle de acesso e usuário, confirme com o acionamento do botão **Sim**.

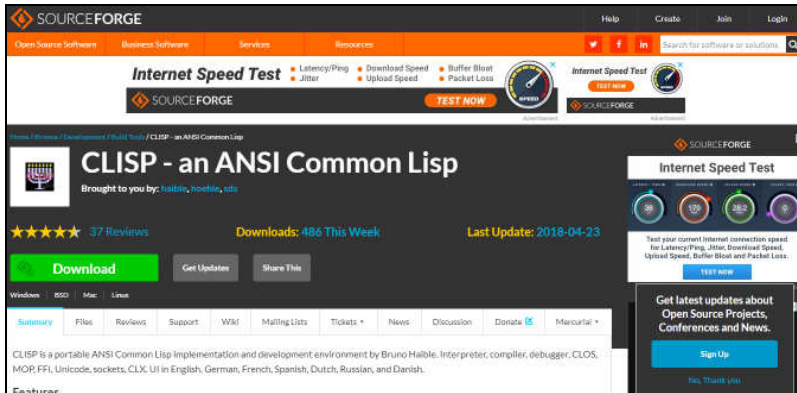


Figura 1.2 - Tela da página do projeto SourceForge.

Em seguida é apresentada a caixa de diálogo GNU CLISP 2.49 Setup, como mostra a figura 1.3.

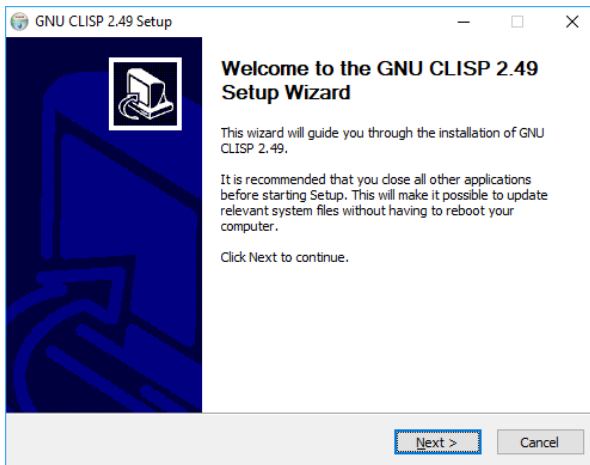


Figura 1.3 - Caixa de diálogo: GNU CLISP 2.49 Setup.

Acione o botão **Next** e será apresentada a tela **License Agreement**, como mostra a figura 1.4, após proceder a leitura e concordando com os termos acione o botão **I**

Agree para que o processo de instalação seja continuado. Caso selecione o botão Cancel o processo de instalação é interrompido.

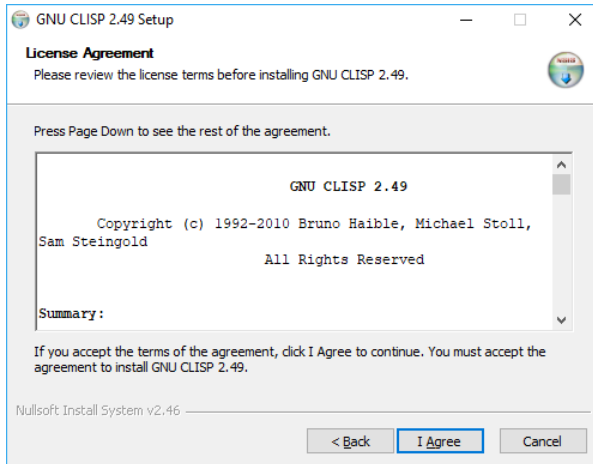


Figura 1.4 - Caixa de diálogo: License Agreement.

Após aceitar os termos do contrato e acionar o botão I Agree é apresentada a tela Choose Components como apresenta a figura 1.5. Mantenha as opções apresentadas e acione na sequência o botão Next para a próxima etapa.

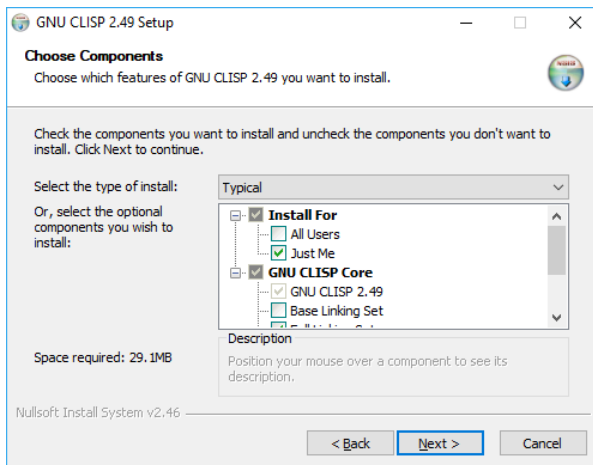


Figura 1.5 - Caixa de diálogo: Choose Components.

Na tela Choose Install Location, como indica a figura 1.6, mantenha a indicação de local apresentado e acione o botão Next.

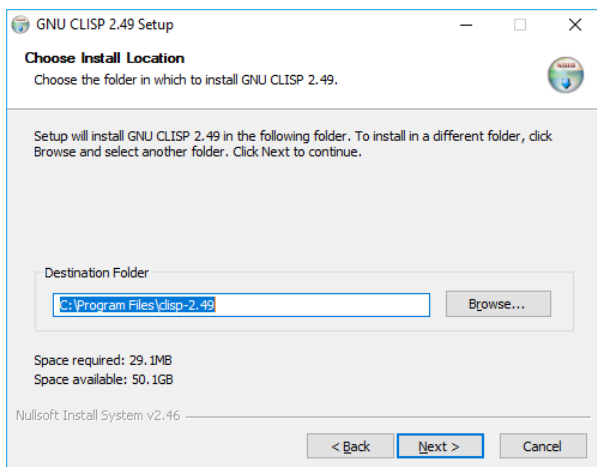


Figura 1.6 - Caixa de diálogo: tela Choose Install Location.

Será então apresentada a tela Choose Start Menu Folder onde poderá ser selecionado a pasta de gravação do programa, como mostra a figura 1.7. Nesta etapa mantenha a informação apresentada e acione simplesmente o botão Install para que o processo de instalação seja iniciado.

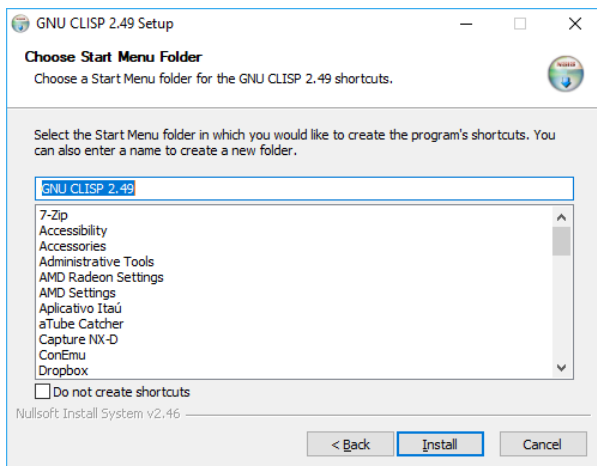


Figura 1.7 - Caixa de diálogo: tela Choose Start Menu Folder.

Aguarde o término da instalação e ao final deste processo acione o botão **Close** da tela **Installation Complete** como indica a figura 1.8.

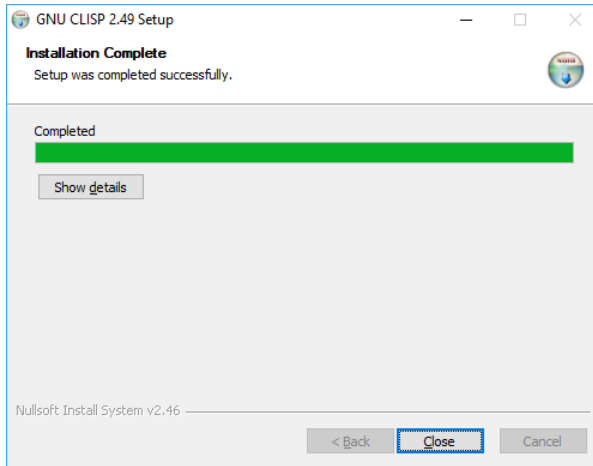


Figura 1.8 - Caixa de diálogo: tela *Installation Complete*.

Após a conclusão da instalação você notará a criação de um ícone de atalho para acesso chamado **GNU CLISP 2.49** na área de trabalho do sistema. Assim sendo, acione com um duplo clique do ponteiro do **mouse** o ícone indicado e veja a tela do ambiente de programação LISP apresentado como mostra a figura 1.9.

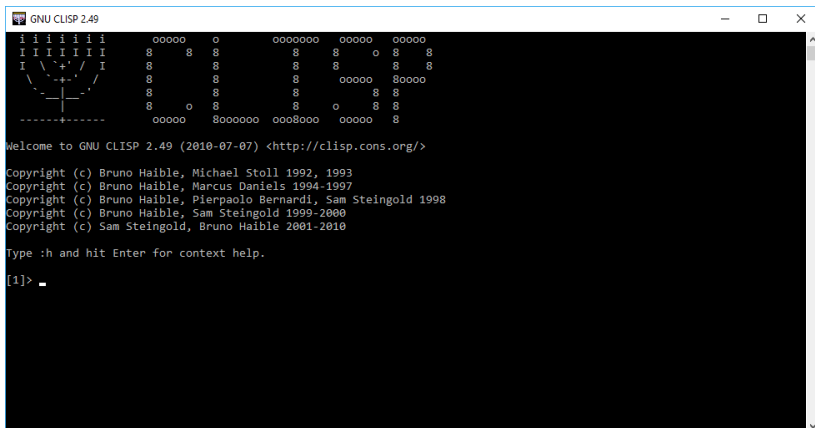


Figura 1.9 - Tela do ambiente *CLISP* a partir do ícone da área de trabalho.







O programa SBCL passa por constantes revisões e atualizados lançamentos, sendo descente do código do programa CMU CL, criado na universidade Carnegie Mellon para sistemas operacionais destinados a plataforma UNIX.

Para obter informações diversas sobre o programa SBCL acesse o sítio oficial do projeto <http://sbcl.org/> como mostra a figura 1.13.

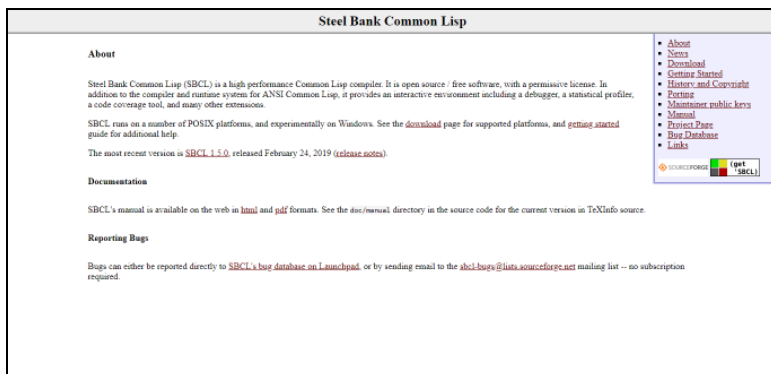


Figura 1.13 - Tela da página do projeto CLISP.

A obtenção de uma cópia para instalação pode ser obtida a partir do sítio do projeto a partir da seleção da opção de menu Download. No quadro apresentado na seção Binaries selecione na linha que indica o sistema operacional Windows a opção da coluna x86 para Windows de 32 bit ou AMD64 para Windows de 64 bits.

Na sequência é apresentada a página do projeto SourceForge indicando o nome do arquivo selecionada e poucos segundos depois apresenta uma caixa de diálogo mostrando o arquivo de programa selecionado para cópia. Neste instante, acione o botão Salvar arquivo para copiar o programa de instalação para seu computador.

Se selecionada a opção da coluna x86 será indicado a cópia do arquivo de instalação `sbcl-999-x86-windows-binaries.msi`, mas, se selecionada a opção da coluna AMD64 será indicado a cópia do arquivo de instalação `sbcl-999-x86-64-windows-binaries.msi`, onde 999 é o número de identificação da versão mais recente disponibilizada para aquisição.

Após a cópia do arquivo de instalação vá ao local copiado em com um duplo clique do ponteiro do mouse selecione o programa para instalação. Em seguida se apresentada a caixa de diálogo de aviso de segurança solicitando autorização para iniciar a instalação proceda com a seleção do botão Executar.

Em seguida é apresentada a caixa de diálogo de início de instalação Steel Bank Common Lisp 1.4.14 (X86) Setup, semelhante a imagem indicada na figura 1.14.

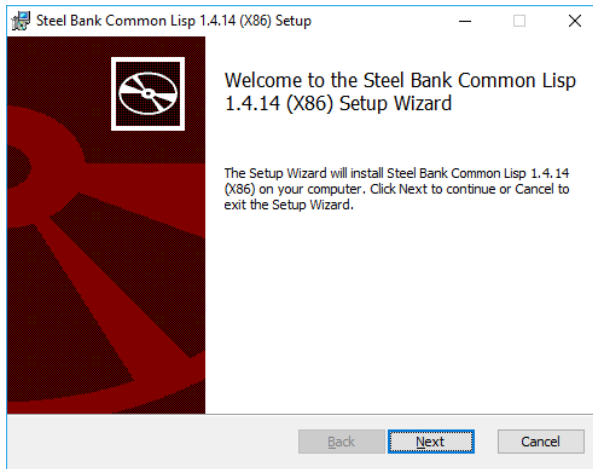


Figura 1.14 - Caixa de diálogo: Steel Bank Common Lisp 1.4.14 (X86) Setup.

Acione o botão Next e será apresentada a tela End-User License Agreement, como mostra a figura 1.15, após proceder a leitura e concordando com os termos acione a caixa de opção I accept the terms in the License Agreement e selecione na sequência o botão Next para que o processo de instalação seja continuado. Caso selecione o botão Cancel o processo de instalação é interrompido.

Após aceitar os termos do contrato é apresentada a tela Custom Setup como apresenta a figura 1.16. Mantenha as opções apresentadas e acione na sequência o botão Next para a próxima etapa.

Na tela Ready to install Steel Bank Common Lisp 1.4.14 (X86), como indica a figura 1.17, acione o botão Install.

Aguarde o término da instalação e ao final deste processo acione o botão Finish da tela Completed the Steel Bank Common Lisp 1.4.14 (X86) Setup Wizard como indica a figura 1.18.

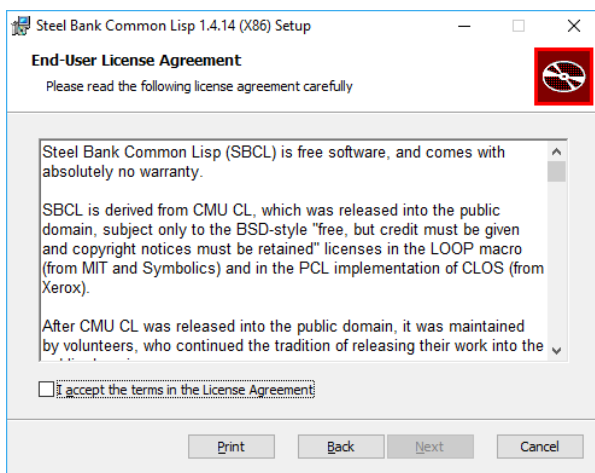


Figura 1.15 - Caixa de diálogo: End-User License Agreement.

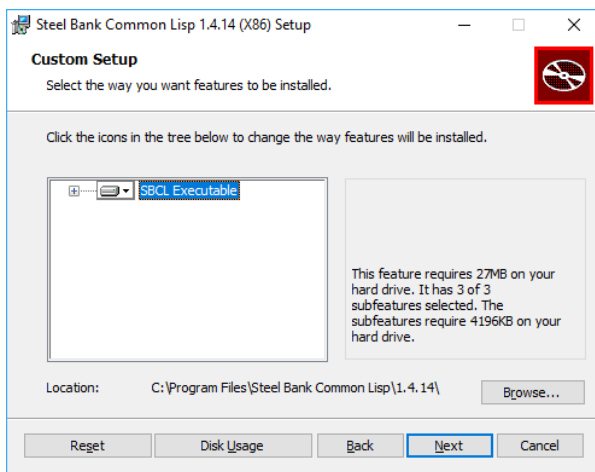


Figura 1.16 - Caixa de diálogo: Custom Setup.

Há duas outras formas de fazer uso do ambiente SBCL no Windows. A partir das teclas <Win> + <r>, que apresenta a figura 1.10, informe no campo Abrir da caixa de diálogo Executar a chamada de um dos programas de processamento de comandos CMD ou POWERSHELL e acione o botão OK.

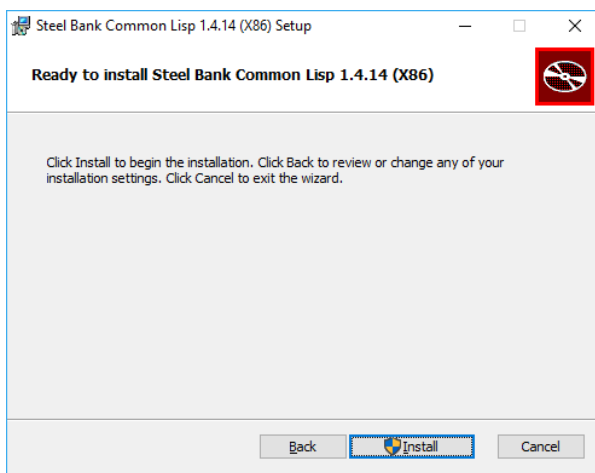


Figura 1.17 - Caixa de diálogo: Ready to install Steel Bank Common Lisp 1.4.14 (X86).

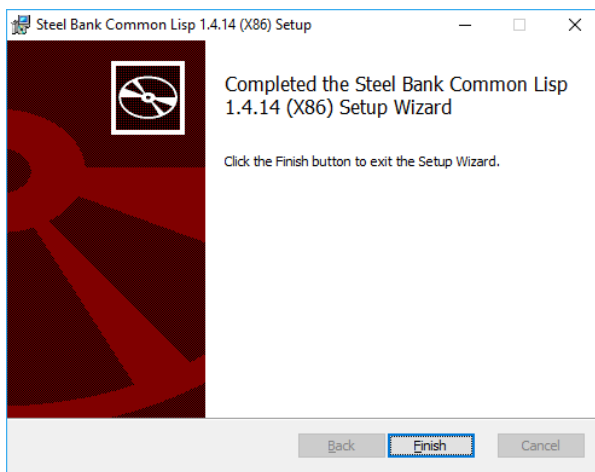
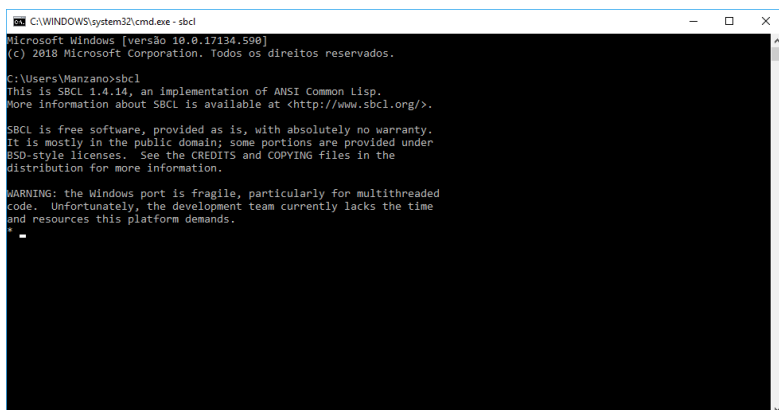


Figura 1.18 - Caixa de diálogo: tela Completed the Steel Bank Common Lisp 1.4.14 (X86) Setup Wizard.

Assim que o **prompt** do processador de comandos dos programas CMD/POWERSHELL for apresentado efetue a execução do comando SBCL e acione a tecla <Enter>.

Se efetuada a execução do ambiente CMD a apresentação da tela de operação do ambiente SBCL como mostra a figura 1.19.



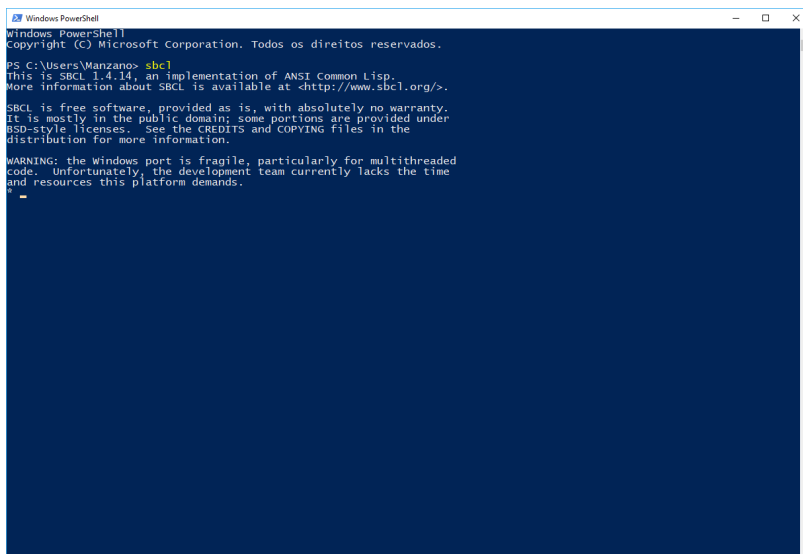
```
C:\WINDOWS\system32\cmd.exe - sbcl
Microsoft Windows [vers o 10.0.17134.590]
(c) 2018 Microsoft Corporation. Todos os direitos reservados.

C:\Users\Manzano>sbcl
This is SBCL 1.4.14, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses. See the CREDITS and COPYING files in the
distribution for more information.

WARNING: the Windows port is fragile, particularly for multithreaded
code. Unfortunately, the development team currently lacks the time
and resources this platform demands.
```

Figura 1.19 - Tela do ambiente SBCL a partir do programa CMD.



```
Windows PowerShell
Copyright (C) Microsoft Corporation. Todos os direitos reservados.

PS C:\Users\Manzano> sbcl
This is SBCL 1.4.14, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses. See the CREDITS and COPYING files in the
distribution for more information.

WARNING: the Windows port is fragile, particularly for multithreaded
code. Unfortunately, the development team currently lacks the time
and resources this platform demands.
```

Figura 1.20 - Tela do ambiente SBCL a partir do programa POWERSHELL.

Se efetuada a execu  o do ambiente POWERSHELL a apresenta  o da tela de opera  o do ambiente SBCL assemelha-se a figura 1.20.

Para encerrar a execu  o do ambiente SBCL execute o comando:

(QUIT) ou (quit)

Atente para o uso obrigat  rio dos parentes circundando o comando.

## 1.4 QUAL AMBIENTE USAR: CLISP OU SBCL?

---

Nos dois tópicos anteriores foram apresentadas duas opções de programas para execução e aprendizagem do código LISP indicado neste trabalho. Ambas as ferramentas são aqui descritas devido ao fato de serem ambientes populares.

O programa CLISP foi atualizado pela última vez (até a publicação deste trabalho) no ano de 2010 enquanto que o programa SBCL é atualizado com mais frequência. Segundo aponta o grupo **Common Lisp Brasil** (<https://lisp.com.br/>) a ferramenta CLISP é considerada um artefato histórico, ou seja, é uma ferramenta obsoleta e indica entre outras três opções o uso da ferramenta SBCL apontando esta como a mais recomendada.

A despeito das considerações apresentadas ambas as ferramentas são boas candidatas para o uso de iniciantes na linguagem e sua aprendizagem. O que deve ser considerado é que para aplicações robustas a ferramenta SBCL se apresenta mais adequada que a ferramenta CLISP. É importante salientar que CLISP em alguns momentos é mais permissivo que SBCL permitindo escrever operações que em SBCL geram certas mensagens de advertência, o que faz do SBCL uma ferramenta mais adequada, mas não desmerece o CLISP.

Nesta obra as duas ferramentas foram usadas e nelas foram testados cada um dos exemplos e exercícios desenvolvidos. Fica ao seu critério a opção em escolher a ferramenta que lhe seja mais agradável e adequada.

Na interface de comunicação dos ambientes a um **prompt** que indica a prontidão de uso. O símbolo de **prompt** é a forma com que a interface avisa que está pronta para receber determinado comando.

No programa CLISP o indicativo de prontidão é sinalizado com o símbolo “[n]>”, onde “n” indica um valor numérico sequencial informando as interações realizadas até então. Já no programa SBCL o indicativo de prontidão é sinalizado com o símbolo “\*” asterisco.

Devido à particularidade operacional das formas de indicação de **prompts** de cada ferramenta e para homogeneização das operações apresentadas nesta obra será usado como indicativo de **prompt** nos exemplos e exercícios de demonstração o símbolo “>>>” para as ações que se comportam igualmente em ambos os programas.

Quando uma ação interativa é executada em LISP a resposta poderá ocorrer em uma ou duas linhas imediatamente a indicação da ação. Em alguns casos a resposta apresentada poderá ocorrer diferentemente entre os dois programas, mesmo sendo a mesma a resposta. Quando esses efeitos ocorrerem serão indicados os ***prompts*** originais de cada uma das ferramentas.

## 1.5 CARACTERÍSTICAS BÁSICAS SOBRE LISP

---

A linguagem LISP é uma linguagem de programação dinâmica que permite a construção de programas na forma de funções simples (expressões simbólicas), onde, cada expressão mais genérica pode ser combinada para obter-se um programa com funcionalidade completa.

Para dar ideia sobre o potencial da linguagem LISP o programa AutoCAD da AutoDesk possui internamente como extensão de suas funcionalidades uma versão da linguagem LISP chamada AutoLISP, além do editor de textos Emacs e do programa de reserva de passagens aéreas Orbitz, entre outros escritos em LISP.

Pelo fato de ser LISP uma linguagem simples esta permite ao programador estender suas funcionalidades permitindo usar a linguagem como suporte a operação de outros paradigmas de programação. Originalmente LISP é uma linguagem pertencente ao paradigma declarativo funcional, mas devido a sua característica básica operacional permite que esta incorpore diversos outros paradigmas. Isso faz com que LISP consiga permanecer em uso atualmente, além de outras linguagens que surgiram e desapareceram.

LISP como uma das suas maiores características possui a capacidade de ser uma linguagem interativa, além da parte programada que possui. Cada função ou expressão criada para um programa pode ser testada, depurada e corrigida separadamente de forma rápida, sem que para isso seja necessário rever o programa todo ou mesmo compila-lo se for este o caso. LISP é uma linguagem incremental que facilita a construção de grandes programas com grande facilidade.

O menor programa LISP pode ser escrito sem nenhum conteúdo definido dentro de um par de parênteses na forma de lista que após execução retorna como resposta o resultado NIL indicando que nada foi realizado. NIL é um constante especial que representa uma resposta com valor falso após a execução de uma expressão, se uma ação retornar verdadeiro a constante especial usada é T. Tanto NIL como T podem ser expressos em caracteres minúsculos ou maiúsculos.



( )

NIL

O contexto definido dentro de parênteses é chamado em LISP de **expressão**, e por esta razão qualquer coisa escrita em LISP que esteja entre parênteses, até mesmo um programa inteiro é considerada uma expressão. Veja o grau de simplicidade que isso sugere. Todo comando LISP é por sua natureza uma função, ou seja, uma expressão indicada sempre dentro de parênteses que retorna um valor como resposta, mesmo quando o retorno é NIL ao indicar que nada foi realizado ou que o retorno é falso. É pertinente salientar que qualquer elemento informado dentro de parênteses ao interpretador LISP é chamado de **form**.

Um programa LISP mais sofisticado é baseado em uma estrutura de lista configurada a partir da sintaxe:

(ação [<argumento1> [<argumento2> [... <argumentoN>]]])

Onde o indicativo **ação** refere-se a definição de alguma operação a ser realizada pela linguagem sobre o(s) argumento(s) fornecido(s); **argumento1**, **argumento2** e **argumento** indicam a definição do que deverá ser processado na linguagem. A lista de argumentos fornecidos a uma ação é variável e depende da regra de uso da função aplicada.

**OBS:** *Os argumentos indicados nesta obra que se encontram entre os símbolos "<" e ">" caracterizam-se por serem obrigatórios e quando entre os símbolos "[" e "]" caracterizam-se por serem opcionais. De qualquer forma não é para proceder ao uso desses caracteres apenas dos conteúdos indicados por eles quando assim for necessário.*

As expressões LISP podem ser definidas de forma aninhada, uma dentro da outra, o que obriga a se ter alto grau de atenção para indicar o fechamento dos diversos parênteses utilizados em determinada expressão. Essa ocorrência é tão preocupante que ao longo dos anos alguns programadores LISP começaram a dizer que o significado da sigla LISP é **Lots of Irritant Stupid Parents**, ou seja, muitos parênteses estupidamente irritantes.

Uma linguagem de programação, seja ela qual for, opera a partir de dois elementos distintos: os dados e o código do programa. Os dados representam os valores a serem processados e o código representa o processamento dos dados em si. Em LISP o código do programa pode ser a chamada da **ação** e os dados podem ser a execução dos **argumentos**. No entanto, por vezes, o **argumento** pode ser definido como **ação** e a **ação** pode ser definida como **argumento**, ou seja, em LISP é pos-

sível implementar novos recursos a partir das estruturas de dados existentes ou definidas pelo programador, permitindo grande extensão de seus recursos a um nível alto de abstração. Quando uma linguagem de programação permite que uma ação possa ser interpretada como argumento e que um argumento possa ser interpretado como ação, a linguagem opera **homoiconicidade**, sendo LISP uma linguagem desta categoria.

Na linguagem LISP os dados a serem usados são considerados a partir de valores numéricos, variáveis e constantes. O código, quanto ao processamento, estabelece as regras de instruções que visam manipular os dados em memória (regras de negócio), sendo estes baseados em elementos primitivos da linguagem como a execução de operações matemáticas a partir de expressões aritméticas ou algébricas ou mesmo de expressões lógicas.

Como comentado a base da programação LISP é a definição de expressões simbólicas na forma de ações delimitadas entre parênteses que sempre retornam algum valor como resposta a ação. Uma expressão simbólica LISP é também referenciada como **sexp**.

A ação delimitada entre parênteses é composta de uma operação e seus argumentos representados pelos parâmetros fornecidos para a execução de certa ação.

A operação definida a uma expressão como ação é estabelecida a partir do uso de certo recurso, como: função (rotina de programa que retorna um valor como resposta a uma ação específica), macro (recurso que amplia certa função agregando mais funcionalidades a sua ação original), predicado (função que testa a validade de seus argumentos para a execução de condições específicas), construtor (função que cria dados compostos a partir de seus elementos mais simples), seletor (função que recebe um dado composto e devolve suas partes em separado), reconhecedor (função que identifica certo detalhe especial do tipo de dado definido) e teste (função de comparação de dados). Os recursos função, macro, predicado, construtor, seletor, reconhecedor e teste formam a base de definição para uma interface entre o uso e a implementação de certos dados, caracterizando-se a estrutura para definição de tipos abstratos de dados e informações em LISP.

Outra característica operacional da linguagem LISP é o fato de ser operada a partir do uso de notação pré-fixa (notação polonesa) que garante a definição de operações não ambíguas.

Se há o desejo de efetuar a expressão aritmética  $2 + 5$  para obter o resultado 7 deve-se escrevê-la na forma pré-fixada (notação polonesa) como:

(+ 2 5)

Dependendo de como os valores de certa expressão aritmética são dispostos e calculados  $2 + 3 * 5$  pode-se obter o resultado 17, se realizada  $2 + (3 * 5)$  ou pode-se obter o resultado 25, se for realizada  $(2 + 3) * 5$ . A expressão aritmética  $2 + 3 * 5$  caracteriza-se por ser uma expressão ambígua e uma forma de eliminar sua ambiguidade é expressá-la explicitamente da maneira que se deseja calcular como:

(+ 2 (\* 3 5))

(\* 5 (+ 2 3))

Observe nos exemplos indicados que em LISP cada expressão simbólica é representada por uma lista contendo como primeiro elementos um **operador** aplicado a uma lista de um ou mais **operandos**. Veja que em LISP os operandos são avaliados e computados antes de serem passados para o operador que executa a ação matemática desejada. O estilo (operador operandos) é conhecido na linguagem LISP como **avaliação estrita**.

Um recurso existente em LISP é sua capacidade de efetuar o tratamento de seu código como se esse fosse um dado e vice-versa como normalmente ocorre em linguagens funcionais, caracterizando-se desta forma operações com funções de primeira ordem.

Em especial o padrão COMMON LISP dá suporte ao paradigma da programação orientada a objetos com algumas peculiaridades como a definição de métodos não associados a certa classe dando suporte a classe em si, herança e o uso de sobrecarga. É pertinente salientar que a orientação a objetos em LISP se difere dos estilos usados em linguagens como C++, Java ou C# e não será tratado neste trabalho por fugir de seu escopo, merecendo uma obra particularizada para o tema.

Além dos detalhes apresentados LISP aceita o empacotamento (**packaging**) de programas em módulos, permite definição de macros, uso de funções genéricas, efetua o gerenciamento automático de memória com coleta de lixo.

O objetivo desta obra é apresentar de forma simples e introdutória a linguagem LISP para iniciantes na linguagem funcional de computadores. Desta forma, serão abordados durante as próximas páginas diversos exemplos de códigos que fornecerão ao final do estudo uma base para a compreensão mais acentuada de programas escritos em LISP.



```
This is SBCL 1.4.14, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses. See the CREDITS and COPYING files in the
distribution for more information.

WARNING: the Windows port is fragile, particularly for multithreaded
code. Unfortunately, the development team currently lacks the time
and resources this platform demands.
* (print "Alo, mundo!")

"Alo, mundo!"
"Alo, mundo!"
* _
```

Figura 1.22 - Tela com resultado da ação da função "print" no SBCL.

Similarmente a função `print` há a função `write` que efetua apresentação de conteúdo semelhante sem o salto antecipado de linha. Assim sendo execute a instrução:

```
(write "Alo, mundo!")
```

Observe a apresentação da mensagem "Alo, mundo!" sem o salto antecipado de linha passando o **prompt** a ser indicado como `[3]>`.

Além das funções `print` e `write` há uma terceira maneira de se realizar apresentações com um controle diferenciado por meio da função `format`. Assim sendo execute a instrução:

```
(format t "~%Alo, mundo!~%")
```

Neste caso ocorre com o uso do especificador `%` (porcentagem) um efeito de salto de linha antes da apresentação da mensagem e após a apresentação da mensagem. A mensagem é apresentada sem o uso dos símbolos de aspas.

Antes do especificador de salto de linha é usado a diretiva `~` (til) que indica o uso de certo efeito de apresentação junto a cadeia de caracteres delimita entre os símbolos de aspas inglesas.

Após a apresentação da mensagem ocorre o retorno do valor `NIL` indicando retorno de valor lógico como falso. A função `format` possui de forma simplificada a estrutura operacional (será a função documentada com mais detalhes no próximo capítulo).

```
(format <destino> <cadeia>)
```

O argumento **destino** indica o modo de apresentação do conteúdo no terminal de vídeo indicado em **cadeia**, podendo ser definido como nil (falso), t (verdadeiro) ou a indicação de uso de um arquivo. O argumento **cadeia** pode ser definido com o auxílio de códigos de formatação para estabelecer suas diretivas de ação.

O destino de saída t refere-se ao uso do fluxo de saída \*STANDARD-OUTPUT\* indicando que a saída deve ocorrer no terminal. O valor NIL indica que não deve ser apresentado absolutamente nada como resposta da função, mas indica que deve ocorrer a apresentação da cadeia de caracteres. A referência de uso de um arquivo indica que a saída deve ser realizada no arquivo e não no terminal.

A função format para a apresentação de dados fornece mais funcionalidades que as funções print e write a partir da aplicação de diversos especificadores, dos quais, alguns serão apresentados ao longo desta obra.

As figuras 1.23 e 1.24 apresentam trechos de tela com os resultados obtidos com o uso das funções print, write e format nos ambientes CLISP e SBCL.

```
[11]> (print "Alo, mundo!")
"Alo, mundo!"
"Alo, mundo!"
[21]> (write "Alo, mundo!")
"Alo, mundo!"
"Alo, mundo!"
[31]> (format t "~%Alo, mundo!~%")
Alo, mundo!
NIL
[41]> _
```

Figura 1.23 - Tela com resultado da ação das funções "print", "write" e "format" no CLISP.

```
* (print "Alo, mundo!")
"Alo, mundo!"
"Alo, mundo!"
* (write "Alo, mundo!")
"Alo, mundo!"
"Alo, mundo!"
* (format t "~%Alo, mundo!~%")
Alo, mundo!
NIL
*
```

Figura 1.24 - Tela com resultado da ação das funções "print", "write" e "format" no SBCL.

Observe na sequência a execução de algumas operações aritméticas simples de adição (+), subtração (-), multiplicação (\*) e divisão (/). Não entre no prompt o

símbolo ">>>" apenas informe a operação delimita entre parênteses incluindo-se os parênteses.

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

```
3
```

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

```
6
```

```
>>> (+ 1.5 2.3)
```

```
3.8
```

```
>>> (- 5 3)
```

```
2
```

```
>>> (- 5 3 1)
```

```
1
```

```
>>> (* 2 3)
```

```
6
```

```
>>> (* 1.5 3)
```

```
4.5
```

```
>>> (* 1.5 3 2)
```

```
9.0
```

```
>>> (/ 5.0 2)
```

```
2.5
```

```
>>> (/ 5.0 2 .5)
```

```
5.0
```

As operações com mais de dois argumentos são efetuadas sucessivamente não importando a função de operação utilizada (+, -, \* ou /). Atente para a indicação do **prompt** [n]>, onde “n” representa um valor numérico sequencial apresentado.

Além das funções (+, -, \* e /), tem-se um grande conjunto de funções primitivas (funções internas) para a realização de diversas operações. Veja alguns exemplos de funções para os cálculos de exponencial (exp), potência (expt), raiz quadrada (sqrt), maior valor (max), menor valor (min) e valor absoluto (abs).

```
>>> (exp 1)
```

```
2.7182817
```

```
>>> (exp 2.5)
```

```
12.182494
```

```
>>> (expt 2 3)
```

```
8
```

```
>>> (expt 2.5 3.8)
```

```
32.521606
```

```
>>> (sqrt 25)
```

```
5
```

```
>>> (max 1 2 3)
```

```
3
```

```
>>> (min 1 2 3)
```

```
1
```

```
>>> (abs -5)
```

```
5
```

A partir desta exposição você já possui o mínimo necessário para aprofundar seu conhecimento em LISP. Até o próximo capítulo.



## 2

## Recursos básicos

---

*Este capítulo apresenta alguns elementos básicos e introdutórios ao uso da linguagem CL como: átomos, listas e forms e alguns exemplos dessas definições. É dada ênfase aos tipos de dados numéricos suportados pela linguagem e como realizar a identificação dos tipos em uso. Outro ponto apresentado é a indicação de uso de algumas funções matemáticas existentes para o auxílio de diversas operações de cálculo. São orientados os procedimentos para a definição de variáveis e constantes. No sentido de melhorar a legibilidade da apresentação de valores numéricos o capítulo termina mostrando alguns recursos de formatação que podem ser utilizados com valores numéricos.*

### 2.1 AÇÕES INTERATIVAS

---

A linguagem CL é operada em um ambiente interativo. Neste sentido a linguagem funciona em um ciclo denominado **read-eval-print-loop** sendo seu ambiente interativo identificado pela sigla **REPL** que efetua a leitura, a avaliação da expressão informada e apresenta a escrita do resultado ocorrido.

LISP (de forma geral, indecentemente do sabor) é fundamentada sobre a estrutura de dois elementos básicos operacionais essenciais ao seu uso, chamados de **átomos** e **listas**. Tanto átomos como listas são elementos a serem avaliados pela linguagem. Qualquer elemento avaliado é considerado um **form** (formulário).

Uma lista é iniciada e finalizada com os caracteres parênteses, tendo dentro desses ao lado esquerdo a definição de um símbolo que representa a definição de uma função e ao lado direito de zero a mais expressões separadas por espaços em branco. Toda lista por sua natureza é um LISP uma expressão que retorna algum resultado a sua operação.

Um átomo é toda forma de expressão que não se configura como lista, destacando-se, por exemplo, o uso de símbolos indicados por caracteres, cadeias, valores lógicos ou valores numéricos. As expressões definidas dentro de uma lista são átomos e quando usadas separadamente retornam a si mesmas como resultado. Observe alguns exemplos de definição de átomos como números, caracteres, cadeias e símbolos.

```
>>> 26
```

```
26
```

```
>>> 3.14
```

```
3.14
```

```
>>> "A"
```

```
"A"
```

```
>>> "LISP"
```

```
"LISP"
```

```
>>> 'a
```

```
A
```

```
>>> 'Lisp
```

```
LISP
```

Os átomos caracterizam-se por serem elementos primitivos da linguagem, ou seja, são as entidades mais simples que a linguagem de programação consegue lidar, podendo-se verificar se certo elemento primitivo é ou não um átomo por meio da função `atom`. Veja exemplo que indica o resultado `T` quando o dado apontado é um átomo. Caso o dado indicado para a função `atom` não seja um átomo ocorrerá a apresentação de erro.

```
>>> (atom 26)
```

```
T
```

Veja uma ocorrência de erro no programa CLISP.

```
[n]> (atom X)
```

```
*** - SYSTEM::READ-EVAL-PRINT: 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
```

Veja uma ocorrência de erro no programa SBCL.

```
* (atom X)
```

```
debugger invoked on a UNBOUND-VARIABLE in thread
```

```
#<THREAD "main thread" RUNNING {10012E0613}>:
```

```
The variable X is unbound.
```

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

```
restarts (invokable by number or by possibly-abbreviated name):
```

```
0: [CONTINUE ] Retry using X.
1: [USE-VALUE ] Use specified value.
2: [STORE-VALUE] Set specified value and use it.
3: [ABORT ] Exit debugger, returning to top level.
```

```
(SB-INT:SIMPLE-EVAL-IN-LEXENV X #<NULL-LEXENV>)
```

```
0]
```

De acordo com avaliação da função `atom` sobre o dado `X` na memória do computador este não se trata de um dado válido, pois o mesmo não foi previamente definido.

O programa CLISP indica que o símbolo `X` não existe na memória e apresenta a mensagem de erro “variable X has no value”. Já o programa SBCL exibe a mensagem de erro “invoked on a UNBOUND-VARIABLE in thread”, informando que a variável `X` não está sendo invocada para uso na memória.

Quando uma ocorrência de erro é apresentada o avaliador de expressões do ambiente fica em ciclo de interação e necessita ser finalizado.

Para sair da ocorrência erro execute o comando `abort` e acione em seguida a tecla <Enter> para voltar ao *prompt*.

Listas são formas definidas por sequências indeterminadas de átomos ou por mesmo por outras listas delimitadas entre parênteses cujos seus elementos (as expressões) são separados por espaços em branco. O uso de listas se aplica a duas circunstâncias: armazenamento temporário de dados e para a chamada de funções internas da linguagem ou mesmo definidas pelo programador. No capítulo anterior os exemplos operacionais de cálculos matemáticos foram todos definidos dentro de listas. Lista não envolvida em operações de cálculo deve ser definida logo após o uso do caractere *aspas simples*. Observe alguns exemplos de definição de listas.

```
>>> (* 1 2 3 4 5)
120
```

```
>>> '(1 2 3 4 5)
(1 2 3 4 5)
```

```
>>> ()
NIL
```

```
>>> '()
NIL
```

```
>>> '(alo mundo)
(ALO MUNDO)
```

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

Uma lista vazia (sem elementos) é igual ao valor NIL. O valor NIL pode ser usado para indicar a existência de valor lógico falso ou indicar listas vazias.

As listas caracterizam-se por serem elementos complexos da linguagem, ou seja, entidades complexas formadas a partir da combinação de entidades simples.

Quando uma lista é avaliada em LISP é realizada primeiramente a identificação do primeiro símbolo após a abertura de parênteses caso não seja usado inicialmente o símbolo de aspas simples. Se o primeiro símbolo for uma função esta é procurada dentro do **kernel** da linguagem, existindo a função ocorre a avaliação de cada símbolo à direita da função uma única vez geralmente da esquerda para a direita passando esses parâmetros como argumentos da função indicada até o último símbolo definido antes do fechamento de parênteses. Ao ser a função processada ocorre o retorno do resultado de sua ação como resposta da lista indicada. Caso o primeiro símbolo for o nome de função não existente será retornado como resposta uma mensagem de erro.

Outro elemento da linguagem é a possibilidade de definição de comentários junto ao código CL para descrever algum comentário ilustrativo a partir do uso do símbolo (;) ponto-e-vírgula. Nesta etapa, você aprenderá a fazer uso deste recurso de maneira simplificada, ficando outros detalhes a serem apresentados no capítulo 5. Comentários não são processados pela linguagem, são usados para deixar o código de programa mais legível a programadores. Observe o uso de um comentário junto ao uso da função write-line seguinte.

```
>>> (write-line "Alo, mundo!") ; Mostra texto sem aspas  
Alo, mundo!  
"Alo, mundo!"
```

Note que ao ser processada a instrução apenas o conteúdo delimitado entre aspas é apresentado sem a indicação das aspas como ocorreu no capítulo anterior a partir do uso das funções print, write e format. Mas esteja atento ao fato de que a função write-line somente deve ser usada para a apresentação da cadeias de caracteres.

Veja que há em CL há algumas maneiras de efetuar a apresentação de dados no monitor de vídeo do computador em uso a partir das funções print, write, format e write-line.

## 2.2 TIPO DE DADO NUMÉRICO

Um dos principais elementos operacionais mais importantes em uma linguagem de programação são a manipulação e o processamento de dados, pois sem os dados o código de um programa não possui amplo sentido.

Há linguagens de programação que operam com o uso de tipos de dados estáticos onde é necessário estabelecer antes do uso a identificação do tipo de dado a ser manipulado. CL é uma linguagem que opera com tipos de dados dinâmicos. Isso significa que não é necessário conhecer de antemão o tipo de dado a ser usado, basta fazer uso do valor desejado respeitando-se a relação entre os valores, por exemplo, números podem ser operacionalizados matematicamente com outros números, mas não com cadeias.

Apesar da facilidade de CL identificar automaticamente o tipo de dado em uso é importante ao programador conhecer os tipos de dados que a linguagem consegue operar.

Um dos tipos de dados suportado por CL é o tipo **number**. O tipo **number** é na verdade uma categoria de tipo de dado que abrange um conjunto de quatro tipos de numéricos, os quais atendem a certas questões matemáticas, sendo, por conveniência, divididos em: **integer** (inteiro), **ratio** (racional), **floating-point** (real) e **complex** (complexo).

O tipo de dado **integer number** representa valores numéricos inteiros matemáticos positivos e/ou negativos. O padrão CL não impõe limite à magnitude numérica a partir do tamanho máximo endereçado pelo processador em bits; o armazenamento em memória ocorre de forma automática conforme necessidade em representar valores numéricos inteiros grandes (**bignum**).

```
>>> (expt 3 500) ; Estilo bignum para 3 ^ 500
363602917958699368423852670795433191180233850260016230403460358325806001
915838954841985082629793887833081797025344038557528559315170130661429924
309165620257800217712478476434501253428365658132099725903715901525787280
08385990139795377610001
```

A eficiência na representação numérica de valores inteiros dependerá da distribuição da linguagem em uso. No entanto, por questões de eficiência valores numéricos inteiros podem ser limitados a um número fixo de bits (**fixnum**) definindo-se certo intervalo com **most-positive-fixnum** (é o valor limite mais próximo ao infinito

positivo fornecido pela distribuição) ou **most-negative-fixnum** (é o valor limite mais próximo ao infinito negativo fornecido pela distribuição). Segundo o padrão ANSI INCITS 226-1994 a faixa fixa de valores opera entre  $-2^{15}$  até  $2^{15} - 1$ , ou seja, opera com tamanho mínimo de 16 bits.

Para o programa CLISP a constante com **most-positive-fixnum** opera com valor máximo em 16.777.215 e a constante **most-negative-fixnum** com valor mínimo em -16.777.216.

```
[n]> (write most-positive-fixnum) ; Estilo fixnum
16777215
16777215
```

```
[n]> (write most-negative-fixnum) ; Estilo fixnum
-16777216
-16777216
```

Para o programa SBCL a constante com **most-positive-fixnum** opera com valor máximo em 4.611.686.018.427.387.903 e a constante **most-negative-fixnum** com valor mínimo em -4.611.686.018.427.387.904.

```
* (write most-positive-fixnum) ; Estilo fixnum
4611686018427387903
4611686018427387903
```

```
* (write most-negative-fixnum) ; Estilo fixnum
-4611686018427387904
-4611686018427387904
```

O uso de memória entre os estilos **bignum** e **fixnum** ocorre automaticamente pela linguagem na medida em que certo valor numérico interior necessite ultrapassar o limite fixado em **fixnum**. Operações matemáticas realizadas com valores no limite **fixnum** são processadas de forma mais rápida.

Para operações com valores numéricos inteiros existem algumas funções matemáticas exclusivas como: `isqrt`, `gcd`, `lcm`, entre outras. Observe alguns exemplos.

```
>>> (isqrt 25) ; Raiz quadrada de valor inteiro
5
```

```
>>> (gcd 60 42) ; Maximo divisor comum  
6
```

```
>>> (lcm 25 30) ; Minimo multiplo comum  
150
```

O tipo de dado **ratio number** representa a razão matemática existente entre dois valores numéricos inteiros. Uma razão é representada em CL com a função (/) divisão. A função divisão apresenta valores na forma fracionária. Observe alguns exemplos.

```
>>> (/ 20 50) ; razao de 20/50 = 2/5, ou seja, 0.4  
2/5
```

```
>>> (/ 30 50) ; razao de 30/50 = 3/5, ou seja, 0.6  
3/5
```

```
>>> (/ 40 20) ; razao de 20/20 = 2, ou seja, 2/1  
2
```

Observe que as operações de divisão indicadas não retornam os resultados de seus quocientes, mas sim a razão existente entre os valores operacionalizados. Caso deseje realizar uma operação de divisão que apresente como resultado o quociente de dois valores e não sua razão use a função float antes da operação de divisão.

```
>>> (float (/ 20 50))  
0.4
```

O tipo de dado **floating-point number** representa os valores numéricos reais positivos e negativos de ponto flutuante. Um valor numérico real é um valor numérico racional formado por uma porção inteira (expoente) com valores decimais após um ponto (mantissa). O padrão CL prevê para os dados de tipo **floating-point** sub-classificações definidas como: **short-float**, **single-float**, **double-float** e **long-float**. O intervalo de operação numérica flutuante depende da distribuição em uso e podem ser verificadas a partir do uso das constantes **short-float-epsilon** (precisão mínima de 13 bits com expoente mínimo de 5 bits), **single-float-epsilon** (precisão mínima de 24 bits com expoente mínimo de 8 bits), **double-float-epsilon** (precisão



mínima de 50 bits com expoente mínimo de 8 bits) e ***long-float-epsilon*** (precisão mínima de 50 bits com expoente mínimo de 8 bits).

Para o programa CLISP as constantes ***short-float-epsilon*** e ***long-float-epsilon*** operam com os valores.

```
[n]> (write short-float-epsilon)
```

```
7.6295s-6
```

```
7.6295s-6
```

```
[n]> (write long-float-epsilon)
```

```
5.4210108624275221706L-20
```

```
5.4210108624275221706L-20
```

Para o programa SBCL as constantes ***short-float-epsilon*** e ***long-float-epsilon*** operam com os valores.

```
* (write short-float-epsilon)
```

```
5.960465e-8
```

```
5.960465e-8
```

```
* (write long-float-epsilon)
```

```
1.1102230246251568d-16
```

```
1.1102230246251568d-16
```

Os programas CLISP e SBCL retornam para as constantes ***single-float-epsilon*** e ***double-float-epsilon*** os mesmos resultados:

```
>>> (write single-float-epsilon)
```

```
5.960465E-8
```

```
5.960465E-8
```

```
>>> (write double-float-epsilon)
```

```
1.1102230246251568d-16
```

```
1.1102230246251568d-16
```

As letras *s*, *e*, *d* e *l* apresentadas junto aos valores numéricos (ou suas equivalentes em formato maiúsculo) especificam respectivamente o uso dos formatos de tipo **short** (curto), **single** (simples), **double** (duplo) e (**long**) longo.

O tipo de dado **complex** é a definição de um valor representado em forma cartesiana formado por uma parte real e outra imaginária, cada qual sendo um valor numérico não complexo (real ou racional), desde que ambos os valores sejam simultaneamente do mesmo tipo.

Valores complexos podem se escritos em CL com a macro `#c` ou com a função `complex` seguidos de uma lista de par: real e imaginário. A macro `#c` não permite o uso de expressões como partes reais e imaginárias, isso somente é realizado com o uso da função `complex` que apresenta sua saída levemente diferente entre os programas CLISP e SBCL. Observe a definição de alguns valores complexos.

```
>>> #c(8 -3)
```

```
#C(8 -3)
```

```
>>> #c(1 1)
```

```
#C(1 1)
```

```
>>> #c(3 0)
```

```
3
```

```
>>> #c(4.0 0.0)
```

```
#C(4.0 0.0)
```

Observe o uso da função `complex` no programa CLISP.

```
[n]> (complex(+ 2.5 1.5) 7)
```

```
#C(4.0 7)
```

Observe o uso da função `complex` no programa SBCL.

```
* (complex(+ 2.5 1.5) 7)
```

```
#C(4.0 7.0)
```

Se fosse realizada a operação por meio de macro: `(#c(+ 2.5 1.5) 7)` ao invés da operação por meio de expressão (função): `(complex(+ 2.5 1.5) 7)` ocorreria um erro de sintaxe.

Use a função `complex` apenas se houver a necessidade de estabelecer operações com expressões em um de seus argumentos. Caso contrário, por questões de praticidade, use a macro `#c`.

A partir da definição de valores complexos é possível fazer a extração apenas da parte real ou da parte imaginária do valor. Para tanto, use a função `realpart` para extrair o primeiro valor do par e a função `imagpart` para extrair o segundo valor do par. Observe o uso das funções `realpart` e `imagpart`.

```
>>> (realpart #C(1 2))
```

```
1
```

```
>>> (imagpart #C(1 2))
```

```
2
```

O conjunto de funções matemáticas da linguagem CL normalmente opera com argumentos complexos e com retorno de resultados complexos. Observe alguns exemplos.

```
>>> (exp #C(0.0 0.5))
```

```
#C(0.87758255 0.47942555)
```

```
>>> (tan #C(1.0 1.0))
```

```
#C(0.2717526 1.0839233)
```

```
[n]> (sin #C(1.0 1.0))
```

```
#C(1.2984576 0.6349639)
```

```
* (sin #C(1.0 1.0))
```

```
#C(1.2984576 0.63496387)
```

```
>>> (cos #C(1.0 1.0))
```

```
#C(0.83373 -0.9888977)
```

Observe o uso da função matemática `sqrt` no programa CLISP.

```
[n]> (sqrt -4.0)
#C(0 2.0)
```

Observe o uso da função matemática `sqrt` no programa SBCL.

```
* (sqrt -4.0)
#C(0.0 2.0)
```

Foram neste tópico apresentados apenas os tipos de dados numéricos. Outros tipos de dados da linguagem serão apresentados na medida em que se fizerem necessários,

## 2.3 IDENTIFICAÇÃO DE TIPOS DE DADOS

---

Para realizar a identificação de um tipo de dado de certo valor operacionalizado pode-se utilizar a função `type-of` que tem por finalidade a capacidade de retornar um especificador de tipo de dado para um elemento em uso. Observe os exemplos de uso.

```
>>> (type-of 1)
BIT
```

```
>>> (type-of 1.5)
SINGLE-FLOAT
```

```
>>> (type-of 1.5d0)
DOUBLE-FLOAT
```

```
>>> (type-of 2/3)
RATIO
```

Observe o uso da função matemática `type-of` no programa CLISP.

```
[n]> (type-of 10)
(INTEGER 0 16777215)
```

```
[n]> (type-of -3)
(INTEGER -16777216 (0))
```

```
[n]> (type-of 1.5l0)
LONG-FLOAT
```

```
[n]> (type-of (expt 3 500))
(INTEGER (16777215))
```

```
[n]> (type-of #C(1.0 1.0))
COMPLEX
```

Observe o uso da função matemática `type-of` no programa SBCL.

```
* (type-of 10)
(INTEGER 0 4611686018427387903)
* (type-of -3)
FIXNUM
```

```
* (type-of 1.5l0)
DOUBLE-FLOAT
```

```
* (type-of (expt 3 500))
(INTEGER 4611686018427387904)
```

```
* (type-of #C(1.0 1.0))
(COMPLEX (SINGLE-FLOAT 1.0 1.0))
```

## 2.4 FUNCIONALIDADES MATEMÁTICAS

---

Como já apresentado, em alguns exemplos anteriores, as ações matemáticas em CL são realizadas a partir do uso de funções destinadas a esta finalidade. Veja a relação de funções utilizadas até o presente momento.

+	função para adição;
-	função para subtração;
/	função para divisão (flutuante) ou obtenção de razão (inteiro);
*	função para multiplicação;
abs	função para apresentar o valor positivo;
complex	função para obter valor complexo;
cos	função para obter o cosseno;
exp	função para obter a potência de $e^n$ ;
expt	função para potência de base elevado ao expoente de $b^e$ ;
gcd	função para obter o máximo divisor comum;
imagpart	função para obter a parte imaginária de valor complexo;
isqrt	função para obter raiz quadrada de número inteiro;
lcm	função para obter o mínimo múltiplo comum;
max	função para obter o maior valor de uma lista de valores;
min	função para obter o menor valor de uma lista de valores;
realpart	função para obter a parte real de valor complexo;
sin	função para obter o seno;
sqrt	função para obter raiz quadrada de qualquer número;
tan	função para obter a tangente.

Na sequência observe um conjunto de funções matemáticas importantes e não apresentada até então.

1-	função para decremento de 1;
1+	função para incremento de 1;

<code>acos</code>	função para obter o arco cosseno;
<code>asin</code>	função para obter o arco seno;
<code>atan</code>	função para obter o arco tangente;
<code>ceiling</code>	função para obter o teto do valor numérico;
<code>floor</code>	função para obter o piso do valor numérico;
<code>log</code>	função para obter o logaritmo de $\log_n m$ ;
<code>mod / rem</code>	função para obter o resto de uma divisão de inteiros;
<code>random</code>	função para obter valor randômico entre 1 e n-1;
<code>round</code>	função para obter arredondamento de valores;
<code>signum</code>	função para verificar se valor é positivo, negativo ou zero;
<code>truncate</code>	função para obter o expoente de valor flutuante.

Observe os exemplos de uso das funcionalidades matemáticas anteriores ainda não demonstradas.

```
>>> (1+ 5)
```

```
6
```

```
>>> (1- 5)
```

```
4
```

```
>>> (acos -1)
```

```
3.1415927
```

```
>>> (asin -1)
```

```
-1.5707964
```

```
>>> (log 8.0 2) ; logaritmo na base 2
```

```
3.0
```

```
>>> (log 100.0 10) ; logaritmo na base 10  
2.0
```

```
>>> (log 2.0) ; logaritmo neperiano  
0.6931472
```

```
>>> (log -1.0) ; logaritmo neperiano  
#C(0.0 3.1415927)
```

```
>>> (mod 8 3)  
2
```

```
>>> (rem 8 3)  
2
```

```
>>> (random 3) ; retorna 0, 1 ou 2  
0
```

```
>>> (+ (random 3) 1) ; retorna 1, 2 ou 3  
3
```

```
>>> (signum 9) ; retorna 1 se valor maior que zero  
1
```

```
>>> (signum -9) ; retorna -1 se valor menor que zero  
-1
```

```
>>> (signum 9) ; retorna 0 se valor igual a zero  
0
```

Observe o uso das funcionalidades matemáticas não utilizadas até o presente momento no programa CLISP.

```
[n]> (atan -1)  
-0.7853981
```



```
[n]> (round 2.5)
```

```
2 ;
```

```
0.5
```

```
[n]> (round 2.6)
```

```
3 ;
```

```
-0.4000001
```

```
[n]> (floor 1.1)
```

```
1 ;
```

```
0.100000024
```

```
[n]> (floor 1.9)
```

```
1 ;
```

```
0.9
```

```
[n]> (ceiling 1.1)
```

```
2 ;
```

```
-0.9
```

```
[n]> (ceiling 1.9)
```

```
2 ;
```

```
-0.100000024
```

```
[n]> (truncate 0.3)
```

```
0 ;
```

```
0.3
```

```
[n]> (truncate -0.3)
```

```
0 ;
```

```
-0.3
```

```
>>> (expt 2 (float (/ 6 3))) ; Raiz cubica de 2 ^ 6
```

```
4.0
```

Observe o uso das funcionalidades matemáticas não utilizadas até o presente momento no programa SBCP.

**\* (atan -1)**

-0.7853982

**\* (round 2.5)**

2

0.5

**\* (round 2.6)**

3

-0.4000001

**\* (floor 1.1)**

1

0.100000024

**\* (floor 1.9)**

1

0.9

**\* (ceiling 1.1)**

2

-0.9

**\* (ceiling 1.9)**

2

-0.100000024

**\* (truncate 0.3)**

0

0.3

```
* (truncate -0.3)
0
-0.3
```

Das funções apresentadas é importante tomar o cuidado de não confundir as ações das funções ceiling, floor, round e truncate. Assim sendo, atente para os detalhes do trecho seguinte e observe cada informação retornada pela ação de cada uma das funções.

Argumento	floor	ceiling	truncate	round
2.9	2	3	2	3
2.5	2	3	2	2
2.1	2	3	2	2
0.9	0	1	0	1
0.1	0	1	0	0
-0.1	-1	0	0	0
-0.9	-1	0	0	-1
-2.1	-3	-2	-2	-2
-2.5	-3	-2	-2	-2
-2.9	-3	-2	-2	-3

Nesta etapa do estudo estão sendo apresentadas algumas das funções matemáticas existentes na linguagem CL. Nos próximos capítulos serão apresentadas outras funções não só matemáticas, além de serem desenvolvidas funções particulares.

## 2.5 VARIÁVEIS E CONSTANTES

Variável é um dos elementos de programação mais importantes no desenvolvimento de alguma tarefa, pois são essas ferramentas que possibilitam efetuar o armazenamento de certo valor em memória para processá-los posteriormente.

Uma variável é por assim dizer uma região de memória (memória principal) usada por um programa para armazenar determinado valor por certo espaço de tempo. O valor armazenado pode ser usado para, basicamente, dois tipos de processamen-

to: matemático e lógico. O processamento matemático caracteriza-se pelo uso de variáveis de ação e o processamento lógico caracteriza-se pelo uso de variáveis de controle.

Toda variável é em CL um símbolo (átomo) que para ser definida necessita possuir um nome de identificação formado por caracteres alfabéticos, numéricos e de pontuação.

O nome de identificação de uma variável não poderá ser formado apenas por números ou por símbolos, mas poderão possuir na sua composição apenas símbolos e números.

Nomes formados somente por caracteres alfabéticos são permitidos, mas não são permitidos em nomes de variáveis são o espaço em branco e os símbolos de parênteses.

Variáveis em CL podem ser declaradas com as funções `defvar` (para definição de variáveis globais) e `let` (para definição de variáveis locais). Variáveis globais quando definidas ficam ativas durante todo o tempo de uso do ambiente da linguagem e locais ficam ativas apenas dentro do escopo ao qual foi definida.

A definição de variáveis globais em CL pode ser realizada com a função `defvar` a partir da estrutura de sintaxe:

```
(defvar *<variável>* [<valor>])
```

Onde, a indicação **variável** refere-se ao nome de identificação a ser atribuído a certa variável e o indicativo **valor** refere-se a um conteúdo, que poderá ser declarado de forma opcional a fim de ser associada a variável definida.

As variáveis globais definidas em memória podem ser removidas quando necessário. Para esta ação use a função `makunbound` que possui a sintaxe.

```
(makunbound '<variável>)
```

Onde a indicação **variável** é a indicação do nome da variável a ser removida da memória.

No caso de se definir uma variável sem valor de inicialização será necessário realizar a atribuição de certo valor para que a variável possa ser utilizada posteriormente, sob o risco de ocorrência de erro ao acesso de seu conteúdo. Dito isso, é mais

conveniente sempre criar variáveis com a definição de certo valor. Caso não se saiba de antemão o valor a ser definido a uma variável use NIL em sua definição.

A função `defvar` tem por finalidade permitir a criação de variáveis do tipo global. Uma variável global é aquela que fica ativa na memória durante toda a execução do ambiente CL. No entanto, existe a possibilidade de criar variáveis que sejam locais (tema que será tratado no tópico sobre funções definidas) e para diferenciar tais variáveis costuma-se definir as variáveis globais entre os símbolos de asterisco.

Quando uma variável global é definida pela função `defvar` com a inicialização de um valor este fica vinculado a variável criada. Se for executada nova definição da mesma variável com a função `defvar` com outro valor fica mantido o primeiro valor. Observe esta ocorrência.

```
>>> (defvar *x* 9)
```

```
*x*
```

```
>>> *x*
```

```
9
```

```
>>> (defvar *x* 1)
```

```
*x*
```

```
>>> *x*
```

```
9
```

Variáveis globais são consideradas especiais e devem, por esta razão, ter sua identificação diferenciada das chamadas variáveis léxicas, ou seja, das variáveis locais. Isso é feito com o uso dos símbolos de asterisco entre o nome da variável.

A remoção da variável `*X*` da memória é realizada pela instrução.

```
>>> (makunbound '*x*')
```

```
*x*
```

Observe os exemplos de definição de variáveis (globais) e a indicação dos valores atribuídos.

```
>>> (defvar *a* 1)
```

```
*A*
```

```
>>> *a*
```

```
1
```

```
>>> (defvar *b* nil)
```

```
*B*
```

```
>>> *b*
```

```
NIL
```

No caso do programa CLISP a definição de uma variável global sem a atribuição de um valor inicial apesar de aceita gera erro quando é solicitada a apresentação do conteúdo da variável. Observe as instruções seguintes.

```
[n] (defvar *c*)
```

```
*C*
```

```
[n]> *c*
```

```
*** - SYSTEM::READ-EVAL-PRINT: variable *C* has no value
```

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

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

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

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

Note que a definição da variável **\*C\*** realizada no programa CLISP sem a indicação de um valor de inicialização gera a apresentação de mensagem de erro apesar da variável **\*C\*** existir na memória. Para encerrar a ocorrência de erro execute o código “:R3” e acione a tecla <Enter>.

No caso do programa SBCL a definição de uma variável global sem a atribuição de um valor inicial apesar de ser aceita gera erro quando é solicitada a apresentação do conteúdo da variável. Observe as instruções seguintes.

```
* (defvar *C*)
```

```
*C*
```

```
* *C*
```

```
debugger invoked on a UNBOUND-VARIABLE in thread
```

```
#<THREAD "main thread" RUNNING {10012E0613}>:
```

```
  The variable *C* is unbound.
```

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

```
restarts (invokable by number or by possibly-abbreviated name):
```

```
  0: [CONTINUE   ] Retry using *C*.
```

```
  1: [USE-VALUE  ] Use specified value.
```

```
  2: [STORE-VALUE] Set specified value and use it.
```

```
  3: [ABORT      ] Exit debugger, returning to top level.
```

```
(SB-INT:SIMPLE-EVAL-IN-LEXENV *C* #<NULL-LEXENV>)
```

```
0]
```

Observe que a definição da variável **\*C\*** realizada no programa SBCL sem a indicação de um valor de inicialização gera a apresentação de mensagem de erro apesar da variável **\*C\*** existir na memória. Para sair da ocorrência de erro execute o código “3” e acione a tecla <Enter>.

A definição de um valor a uma variável global criada sem a atribuição de um valor inicial pode ser realizada na linguagem LISP por meio da função set que opera a partir da sintaxe:

```
(set '*<variável>* <valor>)
```

Onde, a indicação **variável** refere-se ao nome da variável definida em memória e o indicativo **valor** refere-se valor que será atribuído a variável.

Observe o uso da função set para definir um valor a variável **\*C\*** e em seguida a apresentação de seu conteúdo.

```
>>> (set 'c* 3)
```

```
3
```

```
>>> *c*
```

```
3
```

Note que a função `set` usa antes do nome da variável a indicação de um símbolo (`'`) aspas simples que pode ser substituído pelo **form** especial `quote` que opera a partir da sintaxe:

```
(quote *<variável>*)
```

O formulário `quote` ou o símbolo aspas simples (`'`) tem por finalidade ignorar as regras de avaliação padrão da linguagem passando para a função que a utiliza exatamente o conteúdo que se encontra escrito na forma em que foi escrito sem nenhuma avaliação. Desta forma, a definição de um valor a uma variável criada sem valor pode ser feito com a instrução.

```
>>> (set (quote *c*) 3)
```

```
3
```

As formas usadas anteriormente para a definição de valores a uma variável criada são os estilos que a linguagem CL possuía antes da existência de variáveis léxicas. Com a evolução da linguagem foi acrescida função `setq` (**set quote**), que opera a partir da sintaxe.

```
(setq *<variável>* <valor>)
```

Desta forma, a definição de um valor a uma variável criada com a função `defvar` sem valor pode ser feito com a instrução.

```
>>> (defvar *xx*)
```

```
*XX*
```

```
>>> (setq *xx* 1)
```

```
1
```

Além da função `setq` é possível usar para a mesma tarefa a macro (o tema macro será tratado mais adiante) `setf` que realiza a mesma operação de `setq` por ser baseada na função `setq` com maior flexibilidade, como segue.



```
>>> (defvar *yy*)
*YY*
```

```
>>> (setf *yy* 1)
1
```

Segundo o que é exposto na obra "*Practical Common Lisp*" de Peter Norvig a macro `setf` é o principal recurso de atribuição de valores a variáveis para o padrão CL. Por esta razão, a função `setf` é bastante usada em relação ao uso das funções `setq` e `set`.

A função `setq` é um recurso considerado obsoleto advindo de versões antigas da linguagem LISP anteriores ao padrão CL e mantida por questão de compatibilidade, sendo sua ação produzida em baixo nível enquanto `setf` opera em alto nível.

Observe a tentativa de criação de uma variável com a função `setq` no programa CLISP.

```
[n]> (setq *d* 1)
```

```
*** - EVAL: undefined function SETD
```

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

USE-VALUE	:R1	Input a value to be used instead of (FDEFINITION 'SETD).
RETRY	:R2	Retry
STORE-VALUE	:R3	Input a new value for (FDEFINITION 'SETD).
ABORT	:R4	Abort main loop

Observe a tentativa de criação de uma variável com a função `setq` no programa SBCL.

```
* (setd *d* 1)
```

```
debugger invoked on a UNBOUND-VARIABLE in thread
```

```
#<THREAD "main thread" RUNNING {10012E0613}>:
```

```
The variable *D* is unbound.
```

```
Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.
```

restarts (invokable by number or by possibly-abbreviated name):

- 0: [CONTINUE ] Retry using \*D\*.
- 1: [USE-VALUE ] Use specified value.
- 2: [STORE-VALUE] Set specified value and use it.
- 3: [ABORT ] Exit debugger, returning to top level.

(SB-INT:SIMPLE-EVAL-IN-LEXENV \*D\* #<NULL-LEXENV>)

0]

Uma maneira de se verificar que a macro `setf` usa para sua ação a função `setq` é fazer uso da função `macroexpand` que retornará o valor verdadeiro (T) se `setf` usa `setq`, além de indicar a função usada “(SETQ \*D\* 4)”.

Observe o uso da função `macroexpand` no programa CLISP.

```
>>> (macroexpand '(setf *d* 4))
(SETQ *D* 4) ;
T
```

Observe o uso da função `macroexpand` no programa SBCL.

```
>>> (macroexpand '(setf *d* 4))
(SETQ *D* 4)
T
```

Observe que o retorno da ação da função `macroexpand` mostra que a macro `setf` faz uso da função `setf`. A variável **D** não foi criada na memória. O que ocorreu foi apenas a apresentação da informação de que `setf` usa para sua ação a função `setq`, comprovando o que já foi mencionado.

A definição de variáveis locais em CL pode ser realizada com a função `let` a partir da estrutura de sintaxe:

```
(let <variável> <valor>)
```

Onde, a indicação **variável** refere-se ao nome de identificação a ser atribuído a certa variável e o indicativo **valor** refere-se a um conteúdo, que poderá ser declarado de forma opcional a fim de ser associada a variável definida. A criação efetiva de variáveis locais para ser operada necessita estar dentro de um contexto opera-

cional, tema que será visto no estudo de definições de funções pelo próprio programador mais adiante nesta obra.

Observe o uso da função `let` para definir valor a variável **E** do tipo local fora de um contexto favorável ao seu uso com valor **3** no programa CLISP e o pedido de sua apresentação que gera um erro indicando que a variável **E** não possui valor definido.

```
[n]> (let e 3)
```

```
3
```

```
[n]> e
```

```
*** - SYSTEM::READ-EVAL-PRINT: variable E has no value
```

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

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

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

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

A tentativa de criar uma variável local no programa SBCL como feito no programa CLISP gera um erro logo na definição da variável.

```
* (let e 3)
```

```
; in: LET E
```

```
; (LET E
```

```
; 3)
```

```
;
```

```
; caught ERROR:
```

```
; Malformed LET bindings: E.
```

```
;
```

```
; compilation unit finished
```

```
; caught 1 ERROR condition
```

```
debugger invoked on a SB-INT:COMPILED-PROGRAM-ERROR in thread
```

```
#<THREAD "main thread" RUNNING {10012E0613}>:
```

```
Execution of a form compiled with errors.
```

Form:

```
(LET E
  3)
```

Compile-time error:

Malformed LET bindings: E.

Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.

restarts (invokable by number or by possibly-abbreviated name):

0: [ABORT] Exit debugger, returning to top level.

```
((LAMBDA ()))
```

```
  source: (LET E
```

```
    3)
```

```
0]
```

A definição de variáveis locais será aprofundada em momento mais adiante nesta obra. Assim, foque apenas neste momento a definição de variáveis globais.

Além da definição de variáveis há a possibilidade de se fazer a definição de constantes na linguagem CL. Assim como as variáveis as constantes são elementos essenciais a diversas ações operacionais para a linguagem. Uma constante é uma entidade que possui um valor definido o qual não sofre nenhuma espécie de alteração. A definição de constantes é realizada com a macro `defconstant` a partir da sintaxe:

```
(defconstant <constante> <valor>)
```

Onde, a indicação **constante** refere-se ao nome da constante definida em memória e o indicativo **valor** refere-se valor definido a constante.

Constantes possuem uma característica, pois são de fato variáveis que não alteram seus valores durante a execução de um programa. Desta forma, constantes podem ser consideradas variáveis imutáveis.

A CL possui internamente a definição da função `pi` que retorna a razão entre o perímetro da circunferência de um círculo e o seu diâmetro, como indicado.

Observe a seguir a apresentação do valor da constante `pi` no programa CLISP.

```
[n]> pi  
3.1415926535897932385L0
```

Observe a seguir a apresentação do valor da constante `pi` no programa SBCL.

```
* pi  
3.141592653589793d0
```

Note que a apresentação do valor da constante `pi` nos programas CLISP e SBCL ocorre com sua precisão expressa de forma diferente. A maneira como a precisão numérica é indicada depende de como a implementação da linguagem CL é realizada pelo programa em uso. Por exemplo, se a implementação CL trata o tipo numérico *floating-point* como *long float* ou *double float* ocorre a apresentação do marcador de expoente “d” como em SBCL ou “L” como em CLISP.

Uma forma de ajustar a aproximação de valores de ponto flutuante, como é o caso da constante `pi`, pode ser feita com o uso da função `coerce` que efetua a coerção de valores numéricos de um tipo de dado numérico para outro tipo de dado numérico. A função `coerce` possui a sintaxe.

```
(coerce <valor> '<tipo>)
```

Onde *valor* é a definição de uma variável ou constante e *tipo* a indicação de um tipo de dado a ser convertido.

Observe os exemplos de coerção de tipos de dados no programa CLISP.

```
[n]> (coerce pi 'short-float)  
3.1416s0
```

```
[n]> (coerce pi 'long-float)  
3.1415926535897932385L0
```

```
[n]> (coerce pi 'single-float)  
3.1415927
```

```
[n]> (coerce pi 'double-float)
3.141592653589793d0
```

Observe os exemplos de coerção de tipos de dados no programa SBCL.

```
* (coerce pi 'short-float)
3.1415927
```

```
* (coerce pi 'long-float)
3.141592653589793d0
```

```
* (coerce pi 'single-float)
3.1415927
```

```
* (coerce pi 'double-float)
3.141592653589793d0
```

Quando se define nomes para constantes (e mesmo variáveis) é importante tomar o cuidado de não fazer uso de um nome que seja de certo recurso da linguagem. Neste sentido, uma técnica que pode ser usada é possuir uma forma de identificar o nome da certa constante a partir de certo identificador. Por exemplo, para a definição de constantes matemáticas usar um indicativo como “ $\pi$ ”. Assim sendo, observe exemplo de definição da constante de Euler.

```
>>> (defconstant  $\pi_e$  2.718281828459045235360287471352662497757)
 $\pi_e$ 
```

```
>>>  $\pi_e$ 
2.7182817
```

Observe os exemplos de coerção a partir da definição da constante  $\pi_e$  no programa CLISP.

```
[n]> (coerce  $\pi_e$  'short-float)
2.7183s0
```

```
[n]> (coerce m_e 'long-float)
```

```
2.7182817459106445313L0
```

```
[n]> (coerce m_e 'single-float)
```

```
2.7182817
```

```
[n]> (coerce m_e 'double-float)
```

```
2.7182817459106445d0
```

Observe os exemplos de coerção a partir da definição da constante `m_e` no programa SBCL.

```
* (coerce m_e 'short-float)
```

```
2.7182817
```

```
* (coerce m_e 'long-float)
```

```
2.7182817459106445d0
```

```
* (coerce m_e 'single-float)
```

```
2.7182817
```

```
* (coerce m_e 'double-float)
```

```
2.7182817459106445d0
```

É pertinente salientar que a função `coerce` pode ser usada para atuar na conversão de tipos numéricos em outros tipos numéricos suportados na linguagem CL. Na medida em que os tipos de dados forem sendo apresentados outros exemplos da função `coerce` serão apresentados.

## 2.6 FORMATAÇÃO NUMÉRICA

---

A apresentação de dados em CL pode ser determinada por ações de formatação a partir do uso da função `format` com o objetivo de deixar o visual dos dados apresentados mais elegantes e convenientes.

A função `format` usada de forma simplificada no capítulo anterior usou o código de formatação `%` (porcentagem) para gerar um salto de linha no ponto de indicação da cadeia. Neste sentido, observe a seguir a “nova” estrutura sintática da função:

**(format <destino> <cadeia> <dado>)**

Onde, o argumento **destino** pode ser definido a partir dos valores: `T` para indicar a saída formatada no terminal de vídeo (fluxo de saída `*STANDARD-OUTPUT*`), `NIL` para devolver a cadeia formatada como uma cadeia delimitada entre aspas ou indicar o destino como um arquivo (**stream**) ou cadeia (**string**) para adicionar a saída ao final da cadeia, exceto se destino for `NIL` com retorno **string** que retornará `NIL`. O argumento **cadeia** pode ser formado por uma sequência de caracteres a ser apresentada com ou sem a definição de diretivas de formação, seus códigos. Este argumento pode ser definido de forma simples como ocorre com o uso do código de salto de linha `%` para gerar um salto de linha ou de forma mais complexa formado por um conjunto de códigos. O argumento **dado** refere-se ao conteúdo a ser apresentado como complemento do argumento **cadeia**.

As diretivas de código de formatação são iniciadas com o símbolo (`~`) til e seguidas de um caractere escrito em letra maiúsculo ou minúsculo que identifica o código a ser definido sobre os dados indicados entre aspas inglesas. Entre o símbolo til e o código de formatação podem ser usados símbolos de formatação auxiliares como vírgula, arroba, dois pontos, entre outros. Pode ser usado dentro das aspas inglesas mais de um código de formatação.

Observe os exemplos a seguir a partir do uso dos códigos (`$`) **monetary** para a apresentação de valores financeiros, (`f`) **floating point** para a apresentação de valores com ponto flutuante mais elaborado, (`d`) **decimal integer** para valores inteiros decimais, (`b`) **binary integer** para valores inteiros binários, (`o`) **octal integer** para valores inteiros octais, (`x`) **hexadecimal integer** para valores inteiros hexadecimais, (`r`) **spell an integer** para valores inteiros em forma cardinal, ordinal, romano e romano antigo, (`e`) código **scientific notation** para valores numéricos no estilo exponencial e (`a`) **aesthetic** para mostrar valores numéricos simples como texto.

```
>>> (format t "~$" 100) ; "$" valor com duas casas decimais
100.00
NIL
```



```
>>> (format t "~4$" 100) ; "4$" valor com quatro casas decimais
100.0000
NIL
```

```
>>> (format t "~f" 2) ; "f" valor como flutuante
2.0
NIL
```

```
>>> (format t "~,3f" pi) ; ",3f" valor com três casas decimais
3.142
NIL
```

```
>>> (format nil "~7,2f" pi) ; "7,2f" usa mascara 9999.99
"  3.14"
```

```
>>> (format t "~d" 123456789) ; "d" para valor inteiro
123456789
NIL
```

```
>>> (format t "~:d" 123456789) ; ":d" valor 123,456,789
123,456,789
NIL
```

```
>>> (format t "~@d" 123456789) ; "@d" valor +123456789
+123456789
NIL
```

```
>>> (format t "~:@d" 123456789) ; ":@d" valor +123,456,789
+123,456,789
NIL
```

```
>>> (format t "~,'.:d" 123456789) ; ",',.:d" troca "," por "."
123.456.789
NIL
```

```
>>> (format t "~,,',.4:d" 123456789) ; ",,,',.4:d" troca "," por "."  
1.2345.6789  
NIL
```

```
>>> (format t "~8,'0d" 123) ; ":@d" valor 00000123  
00000123  
NIL
```

```
>>> (format t "~2,'0d/~2,'0d/~4,'0d" 26 4 1965) ; data 26/04/1965  
26/04/1965  
NIL
```

```
>>> (format t "~b" 10) ; "b" valor binario  
1010  
NIL
```

```
>>> (format t "~o" 10) ; "o" valor octal  
12  
NIL
```

```
>>> (format t "~x" 10) ; "x" valor hexadecimal  
A  
NIL
```

```
>>> (format t "Valor negativo: ~:d" -123) ; mostra mensagem e valor  
Valor negativo: -123  
NIL
```

```
>>> (format t "Valor: ~a" 123) ; mostra mensagem com valor anexo  
Valor: 123  
NIL
```

```
>>> (format t "Valor: ~a" "zero") ; mostra mensagem concatenada
Valor: zero
NIL
```

```
>>> (format t "Valores ~a e ~a." 1 2) ; mostra valores e mensagem
Valores 1 e 2.
NIL
```

```
>>> (format t "~r" 10) ; mostra cardinal do valor
ten
NIL
```

```
>>> (format t "~:r" 10) ; mostra ordinal do valor
tenth
NIL
```

```
>>> (format t "~@r" 4) ; mostra romano do valor
IV
NIL
```

```
>>> (format t "~:@r" 4) ; mostra romano do valor
IIII
NIL
```

```
[n]> (format t "~@e" 4) ; mostra valor no formato exponencial sinalizado
+4.0E+0
NIL
```

```
[n]> (format t "~@e" 4) ; mostra valor no formato exponencial sinalizado
+4.0e+0
NIL
```

Observe os exemplos de formatação numérica no programa CLISP.

```
[n]> (format t "~f" pi) ; "f" valor como flutuante  
3.1415926535897932385  
NIL
```

```
[n]> (format t "~e" 4) ; mostra valor no formato exponencial simples  
4.0E+0  
NIL
```

Observe os exemplos de formatação numérica no programa SBCL.

```
* (format t "~f" pi) ; "f" valor como flutuante  
3.141592653589793  
NIL
```

```
* (format t "~e" 4) ; mostra valor no formato exponencial simples  
4.0e+0  
NIL
```

As diretivas de códigos de formatação indicadas são uma parte de um conjunto maior que será apresentado mais adiante com outros detalhes. Neste tópico está sendo concentrado apenas os códigos de formatação relacionados a formatação de valores numéricos.

## 3

## Recursos intermediários

---

*Este capítulo amplia detalhes da linguagem apresentando informações sobre tipos de dados caracteres, lista, símbolos e funções; conversão entre dados numéricos e textos e vice versa. São indicados o uso de operações de decisão e laços, além do uso de funcionalidades relacionais e lógicas. Na parte de apresentação de funções é indicado o uso de recursividade simples e de cauda.*

### 3.1 TIPO DE DADO CARACTERE E CADEIA

---

Além dos dados numéricos apresentados, CL opera outros tipos de dados, entre eles destaca-se o tipo de dado vinculado a manipulação de caracteres chamada **character**. O tipo **character** é usado, entre outras coisas, para a definição de **strings** (cadeias) que são em essência um conjunto de caracteres, ou seja, um vetor de caracteres. Por vezes os tipos **character** e **string** se confundem. Os caracteres manipulados isoladamente ou na forma de cadeias estão de acordo com a estrutura da tabela ASCII (**American Standard Code for Information Interchange**) e da tabela UNICODE.

De um ponto de vista operacional um caractere em CL é um conteúdo único inicializado pelo símbolo “#\" e uma cadeia é uma sequência de caracteres delimitados entre aspas inglesas.

Observe alguns exemplos de definição e apresentação de caracteres simples.

```
>>> #\a  
#\a
```

```
>>> #\A
#\A
```

Observe alguns exemplos de definição e apresentação de cadeias simples.

```
>>> "lisp"
"lisp"
```

```
>>> (write "A")
"A"
"A"
```

```
>>> (write "LISP")
"LISP"
"LISP"
```

É possível extrair de uma cadeia um caractere a partir do uso da função `char` junto a determinado conteúdo delimitado entre aspas. Observe os exemplos seguintes.

```
>>> (char "A" 0)
#\A
```

```
>>> (char "LISP" 1)
#\I
```

A função `char` apresenta o caractere de uma cadeia a partir da posição sequencial (do índice) informada.

No primeiro exemplo a função `char` usa o valor (0) zero para indicar a apresentação do caractere da posição atual da cadeia, ou seja, o primeiro caractere da cadeia (e neste caso o único) é o de posição zero.

No segundo exemplo a função `char` usa o valor (1) um para indicar a apresentação do segundo caractere da cadeia indicada, ou seja, efetivamente o segundo caractere da cadeia.

Outra maneira de obter caracteres de uma cadeia pode ser com o uso das funções `aref` e `elt`. Note os seguintes exemplos.

```
>>> (aref "LIST" 1)
#\I
```

```
>>> (elt "LISP" 1)
#\I
```

A função `aref` retorna o elemento da posição da cadeia informada. A diferença entre `aref` e `char` é que `aref` pode ser usada com dados numéricos. Outra função que pode ser usada com outro tipo de dado é a função `elt`.

As funções `char`, `aref` e `elt` operam apenas com os caracteres de uma cadeia, em contrapartida a função `string` opera cadeias como um todo. Observe os exemplos seguintes.

```
>>> (string "LISP")
"LISP"
```

```
>>> (string 'LISP)
"LISP"
```

```
>>> (string #\A)
"A"
```

A função `string` quando usada juntamente da indicação de um caractere o converte em cadeia como ocorreu com o uso da instrução `(string #\A)`.

A partir da visão básica em saber identificar a diferença entre caracteres e cadeias é possível realizar operações de junção de cadeias ou de caracteres por meio da função `concatenate`. Observe os seguintes exemplos.

```
>>> (concatenate 'string "abc" "def")
"abcdef"
```

```
>>> (concatenate 'string "abc" "def" "ghi")
"abcdefghi"
```

```
>>> (concatenate 'string '(\a #\b #\c #\e #\f))
"abcdef"
```

A função `concatenate` para operar a junção de cadeias se utiliza como principal argumento a indicação do tipo de junção a ser executada, neste caso a definição de uso da função `string`, a partir do segundo argumento faz-se a indicação dos conteúdos que serão concatenados, não existindo um limite para a junção desses argumentos.

Além do apresentado é possível converter caracteres em forma de cadeias e vice versa a partir da função `coerce`. Veja os exemplos seguintes.

```
>>> (coerce "a" 'character)
#\a

>>> (coerce "LISP" 'list)
(#\L #\I #\S #\P)

>>> (coerce '(\L #\I #\S #\P) 'string)
"LISP"

>>> (concatenate 'string '(\a #\b #\c #\e #\f))
"abcdef"
```

A função `coerce` pode ser substituída pela função `character` para converter uma cadeia de caractere em caractere. Observe o exemplo a seguir.

```
>>> (character "a")
#\a
```

A apresentação de cadeias pode ocorrer a partir dos formatos maiúsculos, minúsculos e capitalizados com as funções `string-upcase`, `string-downcase` e `string-capitalise`. Observe os seguintes exemplos.

```
>>> (string-upcase "linguagem lisp")
"LINGUAGEM LISP"

>>> (string-downcase "LINGUAGEM LISP")
"linguagem lisp"
```



```
>>> (string-capitalize "linguagem lisp")  
"Linguagem Lisp"
```

Caracteres de uma cadeia podem ser grafados em maiúsculos, minúsculos e capitalizados a partir do uso da função `format` com o código de formatação “(~a~)”  
Observe os exemplos a seguir.

```
>>> (format t "~:@(~a~)" "linguagem lisp")  
LINGUAGEM LISP  
NIL
```

```
>>> (format t "~(~a~)" "LINGUAGEM LISP")  
linguagem lisp  
NIL
```

```
>>> (format t "~:(~a~)" "linguagem lisp")  
Linguagem Lisp  
NIL
```

Além dos efeitos apresentados há ainda com a função `format` a possibilidade de capitalizar apenas a primeira letra da primeira palavra de uma cadeia que seja formada por um conjunto de palavras. Observe o seguinte.

```
>>> (format t "~@(~a~)" "linguagem lisp")  
Linguagem lisp  
NIL
```

Além da possibilidade em tratar cadeias com letras capitalizadas, letras maiúsculas e letras minúsculas, é possível aplicar os recursos de efeito maiúsculos e minúsculos de forma isolada sobre caracteres a partir das funções `char-upcase` e `char-downcase`. Veja os exemplos seguintes.

```
>>> (char-upcase #\a)  
#\A
```

```
>>> (char-upcase #\A)  
#\A
```

```
>>> (char-upcase #\@)
```

```
#\@
```

```
>>> (char-downcase #\A)
```

```
#\a
```

```
>>> (char-downcase #\a)
```

```
#\a
```

Outro efeito no uso de cadeias é realizar operações de extração de sub cadeias a partir de uma cadeia principal existente. Há três maneiras de se fazer a extração de parte de uma cadeia a partir de uma cadeia principal com as funções: `string-trim` (retorna uma sub cadeia de uma cadeia indicada, com todos os caracteres removidos do início e do fim da cadeia principal), `string-left-trim` (retira caracteres do começo de uma cadeia) e `string-right-trim` (retira caracteres do final de uma cadeia). Veja os exemplos seguintes.

```
>>> (string-trim "xyz" "xyzabacatexyz")
```

```
"abacate"
```

```
>>> (string-left-trim "xyz" "xyzabacatexyz")
```

```
"abacatexyz"
```

```
>>> (string-right-trim "xyz" "xyzabacatexyz")
```

```
"xyzabacate"
```

Um efeito que pode ser operacionalizado com cadeias de caracteres é capturar o tamanho em caracteres de uma cadeia por meio da função `length`. Observe os exemplos a seguir.

```
>>> (length "Linguagem LISP")
```

```
14
```

```
>>> (length "LISP")
```

```
4
```

A definição de uma cadeia pode ser apresentada de forma invertida a partir do uso da função `reverse`. Veja em seguida exemplo desta aplicação.

```
>>> (reverse "Linguagem LISP")  
"PSIL megaugniL"
```

A função `reverse` pode ser usada para a inversão de dados pertencentes a listas e vetores, tema a ser abordado mais adiante.

A manipulação de caracteres e cadeias pode fazer uso de diversas outras possibilidades de operação. Uma delas é a extração de parte de uma cadeia escolhendo-se as posições de apresentação com a função `subseq`. Observe os exemplos seguintes.

```
>>> (subseq "computador" 7)  
"dor"
```

```
>>> (subseq "computador" 0 3)  
"com"
```

```
>>> (subseq "computador" 3 7)  
"puta"
```

A função `subseq` usa no mínimo dois argumentos e no máximo três argumentos. Quando em uso dois argumentos é possível apresentar o conteúdo do primeiro argumento a partir da posição indicada até o fim da cadeia. Quando em uso três argumentos, o segundo argumento indica a partir de que posição será realizada a extração, o terceiro argumento indica a quantidade de caracteres que será extraída do primeiro argumento.

O uso, particularmente, de cadeias permite que certas edições sobre determinada cadeia como remoção, substituição e alteração respectivamente possam ser definidas a partir das funções `remove`, `substitute` e `replace`.

A função `remove` retira de certa cadeia determinado caractere. Veja o exemplo seguinte.

```
>>> (remove #\a "abacate")  
"bcte"
```

A função `remove` efetua a retirada do caractere indicado de toda a cadeia definida. Esta função faz uso de dois argumentos, sendo o primeiro o caractere e o segundo a cadeia a ser alterada.

A função `substitute` efetua a substituição em certa cadeia de um caractere indicado por outro caractere informado. Veja o exemplo seguinte.

```
>>> (substitute #\o #\a "abacate")
"obocote"
```

A função `substitute` faz uso de três argumentos, onde o primeiro argumento refere-se ao caractere que será substituído pelo caractere informado no segundo argumento sobre a cadeia definida no terceiro argumento.

A função `replace` substitui parte de uma cadeia por trecho de cadeia informada. Veja os exemplos seguintes.

```
>>> (replace "Linguagem Java" "LISP" :start1 10)
"Linguagem LISP"
```

```
>>> (replace "Joao Alves" "Juca" :end1 5)
"Juca Alves"
```

```
>>> (replace "galinha" "katucha" :start1 2 :end1 6 :start2 2 :end2 6)
"gatucha"
```

```
>>> (replace "galinha" "katucha" :start1 2 :end1 6 :start2 2)
"gatucha"
```

```
>>> (replace "galinha" "ti" :start1 2 :end1 4)
"gatinha"
```

A função `replace` substitui os caracteres do primeiro argumento pelos caracteres do segundo argumento retornando o conteúdo modificado junto ao primeiro argumento. A modificação do primeiro argumento após efetivação da troca não poderá ser desfeito. O primeiro e o segundo argumentos possuem como complemento os argumentos `start1` e `end1` para indicar o local de alteração que a ação de manipulação sobre o primeiro argumento será realizada, os argumentos `start2` e `end2` indicam o trecho de caracteres que será usado para alterar o primeiro argumento.

As operações com cadeias podem fazer uso das funções `search` e `mismatch` para indicar respectivamente uma subsequência de caracteres em uma cadeia e para indicar quando duas sequências se divergem. Observe as instruções seguintes.

```
>>> (search "LISP" "Estudo de LISP")  
10
```

```
>>> (mismatch "Estudo de LISP" "Estu")  
4
```

A função `search` retorna a posição a partir da qual o primeiro argumento existe na indicação do segundo argumento. Caso o primeiro argumento não exista no segundo argumento a função retorna `NIL`.

A função `mismatch` retorna a posição do primeiro par de elementos incompatíveis a partir da qual o segundo argumento diverge do primeiro. Caso os argumentos sejam correspondentes a função retorna `NIL`.

Cada caractere definido para uma cadeia possui como sua representação computacional um valor inteiro referente a tabela ASCII ou a tabela UNICODE dependendo apenas da distribuição da linguagem em uso. Assim sendo para mostrar o código numérico padronizado de certo caractere usa-se a função `code-char`. Observe os seguintes exemplos.

```
>>> (code-char 65)  
#\A
```

```
>>> (code-char 97)  
#\a
```

```
>>> (code-char 32)  
#\Space
```

```
>>> (code-char 200)  
#\LATIN_CAPITAL_LETTER_E_WITH_GRAVE
```

```
>>> (code-char 195)  
#\LATIN_CAPITAL_LETTER_A_WITH_TILDE
```

```
>>> (code-char 1488)
```

```
#\HEBREW_LETTER_ALEF
```

```
>>> (code-char 1050)
```

```
#\CYRILLIC_CAPITAL_LETTER_KA
```

A operação inversa é produzida com a função `char-code`. Observe os seguintes exemplos.

```
>>> (char-code #\A)
```

```
65
```

```
>>> (char-code #\a)
```

```
97
```

```
>>> (char-code #\Space)
```

```
32
```

```
>>> (char-code #\LATIN_CAPITAL_LETTER_E_WITH_GRAVE)
```

```
200
```

```
>>> (char-code #\LATIN_CAPITAL_LETTER_A_WITH_TILDE)
```

```
195
```

```
>>> (char-code #\HEBREW_LETTER_ALEF)
```

```
1488
```

```
>>> (char-code #\CYRILLIC_CAPITAL_LETTER_KA)
```

```
1050
```

Apesar de ser o conjunto de funções apresentadas um pouco extenso, este conjunto é apenas uma parte do conjunto de funções disponíveis para manipulação de caracteres e cadeias.

## 3.2 CONVERSÃO NÚMERO TEXTO E VICE VERSA

---

A linguagem de programação LISP possui funções para o tratamento de conversão de número em texto e de texto em número. Para converter um valor numérico em texto usa-se a função `read-from-string` e para converter texto em número usa-se a função `write-to-string`.

Observe em seguida os exemplos de conversão de valores numéricos na forma de cadeias em valores numéricos puros com a função `read-from-string`.

Veja os exemplos comuns aos programas CLISP e BSL.

```
>>> (type-of (read-from-string "1.5"))  
SINGLE-FLOAT
```

```
>>> (type-of (read-from-string "1.23e5"))  
SINGLE-FLOAT
```

Veja os exemplos no programa CLISP.

```
[n]> (read-from-string "1.5")  
1.5 ;  
3
```

```
[n]> (read-from-string "99")  
99 ;  
2
```

```
[n]> (type-of (read-from-string "99"))  
(INTEGER 0 16777215)
```

```
[n]> (read-from-string "#xA") ; A hexadecimal = 10 decimal  
10 ;  
3
```

```
[n]> (type-of (read-from-string "#xA"))  
(INTEGER 0 16777215)
```

```
[n]> (read-from-string "1.23e5")  
123000.0 ;  
6
```

Veja os exemplos no programa CLISP.

```
* (read-from-string "1.5")  
1.5  
3
```

```
* (read-from-string "99")  
99  
2
```

```
* (type-of (read-from-string "99"))  
(INTEGER 0 4611686018427387903)
```

```
* (read-from-string "#xA") ; A hexadecimal = 10 decimal  
10  
3
```

```
* (type-of (read-from-string "#xA"))  
(INTEGER 0 4611686018427387903)
```

```
* (read-from-string "1.23e5")  
123000.0  
6
```

A função `read-from-string` ao ser operacionaliza retorna dois dados, sendo o primeiro dado o valor convertido em número e o segundo dado a quantidade de caracteres que forma o conteúdo convertido.



Observe em seguida os exemplos de conversão de números em cadeias com a função `write-to-string`.

```
>>> (write-to-string 100)
"100"
```

```
>>> (type-of (write-to-string 100))
(SIMPLE-BASE-STRING 3)
```

```
>>> (write-to-string 9.99)
"9.99"
```

```
>>> (type-of (write-to-string 9.99))
(SIMPLE-BASE-STRING 4)
```

```
>>> (write-to-string #xA) ; A hexadecimal = 10 decimal
"10"
```

```
>>> (type-of (write-to-string #xA))
(SIMPLE-BASE-STRING 2)
```

```
>>> (write-to-string 1.23e5)
"123000.0"
```

```
>>> (type-of (write-to-string 1.23e5))
(SIMPLE-BASE-STRING 8)
```

A função `write-to-string` ao ser operacionalizada retorna o dado numérico na forma de cadeia ao apresentá-lo delimitado entre aspas inglesas.

Com os exemplos de uso das funções `read-from-string` e `write-to-string` está sendo usada a função `type-of` (já conhecida). Note que a função `type-of` tem por finalidade indicar o tipo de dado que certo valor pertence, mas ao usá-la com dados do tipo cadeia seu retorno indica a cadeia como sendo `SIMPLE-BASE-STRING` e mostra um valor numérico que indica o tamanho da cadeia, tamanho esse que pode ser obtido com o uso da função `length`.

### 3.3 FUNCIONALIDADES RELACIONAIS

---

Assim como as funcionalidades focadas a operações matemáticas, há um conjunto de funções relacionais pra validação lógica que auxiliam operações de comparação (funções relacionais) e tomada de decisão a partir da definição de condições baseada em predicados. LISP fornece um conjunto específico de predicados para o tratamento de dados numéricos, caracteres e cadeias. Além de um conjunto com funcionalidades genéricas que podem ser aplicadas a qualquer tipo de dado.

Na sequência observe um conjunto de funções relacionais para a definição de operações relacionais importantes para a validação de dados numéricos.

- = função para: igual a;
- > função para: maior que;
- < função para: menor que;
- >= função para: maior ou igual a;
- <= função para: menor ou igual a;
- /= função para: diferente de.

Observe os exemplos de uso das funcionalidades relacionais para verificação de resultados verdadeiros (T) e falsos (NIL) de algumas relações lógicas.

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

```
T
```

```
>>> (= 1 2)
```

```
NIL
```

```
>>> (> 1 2)
```

```
NIL
```

```
>>> (< 1 2)
```

```
T
```

```
>>> (>= 1 2)
```

```
NIL
```

```
>>> (<= 1 2)
```

```
T
```

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

```
T
```

Na sequência observe um conjunto de funções relacionais para a definição de operações relacionais importantes para a validação de dados caracteres.

char=	função para: igual a com case-sensitive;
char-equal	função para: igual a sem case-sensitive;
char>	função para: maior que com case-sensitive;
char-greaterp	função para: maior que sem case-sensitive;
char<	função para: menor que com case-sensitive;
char-lessp	função para: menor que sem case-sensitive;
char>=	função para: maior ou igual a com case-sensitive;
char-not-lessp	função para: maior ou igual a sem case-sensitive;
char<=	função para: menor ou igual a com case-sensitive;
char-not-greaterp	função para: menor ou igual a sem case-sensitive;
char/=	função para: diferente de com case-sensitive;
char-not-equal	função para: diferente de sem case-sensitive.

Observe alguns exemplos de uso de algumas funcionalidades relacionais para verificação de resultados verdadeiros (T) e falsos (NIL) de algumas relações lógicas com caracteres. O estilo com ou sem **case-sensitive** refere a diferenciação de caracteres no formato maiúsculo de minúsculo (com **case-sensitive**) ou de não diferenciar caracteres maiúsculo de minúsculo (com **case-sensitive**).

```
>>> (char= #\a #\a)
```

```
T
```

```
>>> (char= #\A #\a)
```

```
NIL
```

```
>>> (char-equal #\a #\a)
```

```
T
```

```
>>> (char-equal #\A #\a)
```

```
T
```

```
>>> (char> #\a #\b)
```

```
NIL
```

```
>>> (char> #\b #\a)
```

```
T
```

```
>>> (char< #\a #\b)
```

```
T
```

```
>>> (char< #\b #\a)
```

```
NIL
```

```
>>> (char/= #\a #\a)
```

```
NIL
```

```
>>> (char/= #\a #\b)
```

```
T
```

```
>>> (char-not-equal #\a #\a)
```

```
NIL
```

Na sequência observe um conjunto de funções relacionais para a definição de operações relacionais importantes para a validação de dados cadeias, ou seja, de dados delimitados entre aspas inglesas.

<code>string=</code>	função para: igual a com case-sensitive;
<code>string-equal</code>	função para: igual a sem case-sensitive;
<code>string&gt;</code>	função para: maior que com case-sensitive;
<code>string-greaterp</code>	função para: maior que sem case-sensitive;
<code>string&lt;</code>	função para: menor que com case-sensitive;
<code>string-lessp</code>	função para: menor que sem case-sensitive;
<code>string&gt;=</code>	função para: maior ou igual a com case-sensitive;
<code>string-not-lessp</code>	função para: maior ou igual a sem case-sensitive;
<code>string&lt;=</code>	função para: menor ou igual a com case-sensitive;
<code>string-not-greaterp</code>	função para: menor ou igual a sem case-sensitive;
<code>string/=</code>	função para: diferente de com case-sensitive;
<code>string-not-equal</code>	função para: diferente de sem case-sensitive.

Observe alguns exemplos de uso de algumas funcionalidades relacionais para verificação de resultados verdadeiros (T) e falsos (NIL) de algumas relações lógicas com cadeias.

```
>>> (string= "alo" "alo")
```

```
T
```

```
>>> (string= "AL0" "alo")
```

```
NIL
```

```
>>> (string-equal "AL0" "alo")
```

```
T
```

Na sequência observe um conjunto de funções relacionais para a definição de operações genéricas de dados.

<code>equal</code>	igualdade de dados estruturalmente similares;
<code>eq</code>	igualdade de dados idênticos;
<code>eq1</code>	igualdade de dados semelhantes em tipo e valor;
<code>equalp</code>	igualdade de dados similares de mesmo tipo ou iguais;

Observe alguns exemplos de uso de algumas funcionalidades relacionais para verificação de resultados verdadeiros (T) e falsos (NIL) de algumas relações lógicas genéricas.

```
>>> (equal 3 3)
```

```
T
```

```
>>> (equal 3 3.0)
```

```
NIL
```

```
>>> (eq 3 3)
```

```
T
```

```
>>> (eq 3 3.0)
```

```
NIL
```

```
>>> (eq1 3 3)
```

```
T
```

```
>>> (eq1 3 3.0)
```

```
NIL
```

```
>>> (equalp 3 3)
```

```
T
```

```
>>> (equalp 3 3.0)
```

```
T
```

Aparentemente as funções `equal`, `eq` e `eql` parecem ser iguais, mas possuem diferenças significativas.

A função `equal` retorna o valor verdadeiro (T) se os dados informados como argumentos forem estruturalmente similares, ou seja, isomórficos e possuírem valores de mesma configuração. Caso contrário, a função retorna o valor falso (NIL).

A função `eq` retorna o valor verdadeiro (T) se os dados informados como argumentos forem idênticos em tipo e valor. Caso contrário, a função retorna o valor falso (NIL).

A função `eql` retorna o valor verdadeiro (T) se os dados informados como argumentos forem idênticos como em `eq`, se forem números de mesmo tipo e valor ou se forem caracteres que representem o mesmo caractere. Caso contrário, a função retorna o valor falso (NIL).

Daqui em diante poderá ocorrer naturalmente em outros exemplos de recursos da linguagem LISP o uso de funções lógicas para o estabelecimento de relações condicionais na composição de ações condicionais. No entanto, a lista anterior é apenas uma parte das funções existente e caso seja usada uma função lógica não retratada neste tópico a mesma será explicada e exemplificada.

## 3.4 TIPO DE DADO LISTA

---

Além dos dados numéricos e caracteres já apresentados há um tipo de dado muito utilizado chamado *list* (lista). Lista é uma estrutura de dados flexível e simples que pode armazenar elementos (átomos) de tipos numéricos ou caracteres delimitando-os dentro de parênteses separados por espaços em branco, tendo sempre como último elemento a existência do valor NIL.

Listas podem ser definidas de três formas diferentes: com o uso do símbolo (') aspas simples, pela função `quote` ou propriamente por meio da função `list`. Veja alguns exemplos de listas.

```
>>> '(1 2 3 4 5)
(1 2 3 4 5)
```

```
>>> '("A" "B" "C" "D" "E")
("A" "B" "C" "D" "E")
```

```
>>> (quote ("a" "b" "c" "d" "e"))  
("a" "b" "c" "d" "e")
```

```
>>> '("A" "B" "C" 1 2 3)  
("A" "B" "C" 1 2 3)
```

```
>>> (list "A" "B" "C" "D" "E")  
("A" "B" "C" "D" "E")
```

```
>>> (list "A" "B" "C" 1 2 3)  
("A" "B" "C" 1 2 3)
```

A definição de uma lista oferece para uso uma série de funcionalidades que são apresentadas neste capítulo. Listas são formadas por elementos encadeados, onde cada elemento "conhece" seu valor e "sabe" qual é o próximo elemento da lista.

Observe o exemplo seguinte que define uma lista **A** como variável global contendo os cinco primeiros números inteiros de **1** a **5**.

```
>>> (defvar *A* (list 1 2 3 4 5))  
*A*
```

```
>>> *A*  
(1 2 3 4 5)
```

As listas possuem duas partes: cabeça (**head**) acessada pela função `car` e cauda (**tail**) acessada pela função `cdr`. A cabeça é identificada pelo primeiro elemento da lista apontada e a cauda é composta por todos os demais elementos de uma lista excetuando-se o primeiro elemento. Observe os detalhes sobre obtenção da cabeça e cauda de uma lista.

```
>>> (car *A*)  
1
```

```
>>> (cdr *A*)  
(2 3 4 5)
```



Note que a função `car` indica o primeiro elementos da lista como um valor numérico fora da lista. Já a função `cdr` apresenta todos os elementos da lista excetuando-se o primeiro na forma de lista. Caso deseja apresentar o primeiro elemento da lista na forma de lista execute a instrução.

```
>>> (list (car *A*))  
(1)
```

Alternativamente ao invés de usar as funções `car` e `cdr` pode-se usar as funções `first` para obter a cabeça e `rest` para obter a cauda de uma lista. Observe os exemplos seguintes.

```
>>> (list (first *A*))  
(1)
```

```
>>> (rest *A*)  
(2 3 4 5)
```

Assim como é possível obter o primeiro elementos de uma lista com as funções `car` ou `first` é possível obter o último elemento de uma lista com a função `last`. Observe o exemplo seguinte.

```
>>> (last *A*)  
(5)
```

Diferentemente das funções `car` e `first` a função `last` apresenta o último valor da lista delimitado entre parêntese.

Além da função `last` há a função `butlast` que remove da apresentação o último elemento apresentado os demais.

```
>>> (butlast *A*)  
(1 2 3 4)
```

A partir de uma visão básica inicial execute as instruções a seguir para a definição de uma lista vazia e sua apresentação a partir da definição de uma variável global vazia.

```
>>> (defvar *LISTA* nil)  
*LISTA*
```

```
>>> *LISTA*
```

```
NIL
```

Antes de operacionalizar ações sobre a lista é necessário inicializa-la com algum valor. Assim sendo execute a instrução para acrescenta o conteúdo “A”.

```
>>> (setf *LISTA* '("A"))
```

```
("A")
```

A partir da definição de uma lista com conteúdo inicial, esta pode ser usada para diversas operações de gerenciamento sobre seus elementos. Assim sendo, observe as etapas a seguir que efetuam as ações de entrada na cauda de uma lista existente com auxílio da função `nconc`.

```
>>> (nconc *LISTA* '("B"))
```

```
("A" "B")
```

```
>>> (nconc *LISTA* '("C"))
```

```
("A" "B" "C")
```

```
>>> (nconc *LISTA* '("D"))
```

```
("A" "B" "C" "D")
```

```
>>> (nconc *LISTA* '("E" "F"))
```

```
("A" "B" "C" "D" "E" "F")
```

```
>>> *LISTA*
```

```
("A" "B" "C" "D" "E" "F")
```

```
>>> (nconc *A* '(6 7))
```

```
(1 2 3 4 5 6 7)
```

```
>>> *A*
```

```
(1 2 3 4 5 6 7)
```

A função `nconc` utiliza dois argumentos, sendo o primeiro argumento a indicação da lista a ser alterada e o segundo argumento caracteriza-se por ser o elemento a ser inserido fisicamente ao final da lista.

Caso queira acrescentar elementos no início da lista, ou seja, em sua cabeça utilize a função `push`, a qual faz uso de dois argumentos, sendo o primeiro argumento a indicação do conteúdo a ser inserido e o segundo argumento a indicação da lista a ser operacionalizada. Observe a seguir algumas inserções no início da lista.

```
>>> (push 0 *A*)  
(0 1 2 3 4 5 6 7)
```

```
>>> *A*  
(0 1 2 3 4 5 6 7)
```

Para realizar a remoção física de um elemento da cabeça da lista use a função `pop` indicando o nome da tabela. Observe a remoção do elemento zero da lista em uso.

```
>>> (pop *A*)  
0
```

```
>>> *A*  
(1 2 3 4 5 6 7)
```

Para realizar a remoção física de um elemento do final da cauda da lista use a função `nbutlast` indicando o nome da tabela em seu primeiro argumento e no segundo argumento a quantidade de elementos que serão retirados da cauda. Observe a remoção do elemento zero da lista em uso.

```
>>> (nbutlast *a* 1)  
(1 2 3 4 5 6)
```

```
>>> *A*  
(1 2 3 4 5 6)
```

O conteúdo de uma lista pode ser apresentado de forma invertida com a função `reverse`. Veja os exemplos seguintes.

```
>>> (reverse *LISTA*)  
("F" "E" "D" "C" "B" "A")
```

```
>>> (reverse *A*)  
(6 5 4 3 2 1)
```

Há uma alternativa que efetua a inserção de elementos no início da lista de forma "virtual", ou seja, a inserção não é registrada fisicamente. Para tanto, use a função `append`. Observe os exemplos seguintes.

```
>>> (append *LISTA* '("G"))  
("A" "B" "C" "D" "E" "F" "G")
```

```
>>> (append *LISTA* '("H" "I"))  
("A" "B" "C" "D" "E" "F" "H" "I")
```

```
>>> *LISTA*  
("A" "B" "C" "D" "E" "F")
```

```
>>> (append *A* '(8))  
(1 2 3 4 5 6 7 8)
```

```
>>> *A*  
(1 2 3 4 5 6 7)
```

Observe que a função `append` diferentemente da função `nconc` efetua a ação de inserção de elementos apenas no momento de sua ação, ou seja, de forma virtual não mantendo a inserção fisicamente na lista resultante como ocorre com a função `nconc`.

A função `append` opera com estrutura de ação semelhante a estrutura da função `nconc`: primeiro argumento lista e segundo argumento conteúdo inserido. Veja que a função `append` acrescenta virtualmente um elemento ao final da lista.

Assim como é possível realizar inserção virtual de elementos em uma lista é possível fazer a remoção virtual de elementos de uma lista. Para tanto, utilize a instrução seguinte.

```
>>> (reverse (cdr (reverse *a*)))  
(1 2 3 4 5)
```

```
>>> *A*  
(1 2 3 4 5 6)
```

Para saber a quantidade de elementos de uma lista usa-se a função `length` já usada anteriormente. Observe os seguintes exemplos.

```
>>> (length *LISTA*)  
6
```

```
>>> (length *A*)  
7
```

Para a apresentação de um elemento de uma posição específica de uma lista usa-se a função `nth`. Veja os exemplos seguintes.

```
>>> (nth 0 *LISTA*)  
"A"
```

```
>>> (nth 5 *LISTA*)  
"F"
```

```
>>> (nth 7 *A*)  
NIL
```

```
>>> (nth 5 *A*)  
6
```

Outra ação que pode ser executada é verificar se certo elemento pertence a uma lista em operação. Esta ação pode ser realizada com a função `find`. Para tanto, execute as instruções.

```
>>> (find "A" *LISTA* :test #'equal)  
"A"
```

```
>>> (find "X" *LISTA* :test #'equal)
NIL

>>> (find 3 '(1 2 3 4 5))
3

>>> (find 3 '(1 2 3 4 5) :test #'equal)
3

>>> (find 3 *A*)
3

>>> (find 9 '(1 2 3 4 5))
NIL
```

A função `find` verifica se certo elemento pertence ou não a lista indicada. Quando o elemento existe na lista a função retorna o próprio valor do elemento. Se não existir o elemento na lista o retorno da função `find` será o valor `NIL`. Esta função opera basicamente a partir de dois argumentos, sendo o primeiro argumento a indicação do elemento a ser verificado e o segundo elemento a lista a ser pesquisada, aceitando ainda opcionalmente outros dois argumentos, podendo ser `test` para definir uma função de comparação e `key` para definir uma função para extrair um valor antes da execução do teste de comparação. No exemplo anterior está sendo usado no argumento `test` a indicação de comparação com a função `equal`, permitindo verificar se há na lista apontada no segundo argumento algum elemento igual ao indicado no primeiro argumento. A indicação de comparação com a opção  `#'equal` é formada de duas partes, sendo a indicação de ação  `#'` e a indicação da função lógica `equal`. O uso de `equal` é necessário com dados do tipo cadeia ou caractere. Para dados de tipo número a função é opcional.

Uma lista pode ser concatenada com outra lista adicionando seus elementos em uma só lista a partir da função `concatenate` já apresentada. Veja as instruções a seguir para definição de duas novas listas.

```
>>> (defvar *LISTA_LET* (list "A" "B" "C"))
*LISTA_LET*
```

```
>>> (defvar *LISTA_NUM* (list 1 2 3))
*LISTA_NUM*
```

Há duas maneiras de se concatenar listas, uma virtualmente e outra fisicamente. Observe os exemplos seguintes.

```
>>> (concatenate 'list *LISTA_LET* *LISTA_NUM*)
("A" "B" "C" 1 2 3)
```

```
>>> (defvar *LISTA_TOT* (concatenate 'list *LISTA_LET* *LISTA_NUM*))
*LISTA_TOT*
```

```
>>> *LISTA_TOT*
("A" "B" "C" 1 2 3)
```

Observe que na variável **\*LISTA\_TOT\*** fica realizada a concatenação de forma que seu registro fique “fisicamente” armazenado na memória junto a variável global **\*LIST\_TOT\***, pois a ação executada pela função `concatenate` não mantém o efeito de concatenação permanentemente memorizado.

Assim como a inserção de elementos pode ser feito fisicamente (`nconc`) ou virtualmente (`append`) o mesmo pode ser realizado com a remoção de elementos a partir das funções `remove` (ação virtual) e `delete` (ação física).

Observe os exemplos seguintes para ações de remoção virtual, lembrado que uma ação virtual pode ser armazenada junto a definição de uma variável global.

```
>>> (remove 2 *A*)
(1 3 4 5 6 7)
```

```
>>> *A*
(1 2 3 4 5 6 7)
```

```
>>> (remove "A" *LISTA* :test #'equal)
("B" "C" "D" "E" "F")
```

```
>>> *LISTA*
("A" "B" "C" "D" "E" "F")
```

Observe os exemplos seguintes para ações de remoção física com alteração dos dados existentes na lista operacionalizada.

```
>>> (delete 2 *A*)  
(1 3 4 5 6 7)
```

```
>>> *A*  
(1 3 4 5 6 7)
```

```
>>> (delete "C" *LISTA* :test #'equal)  
("A" "B" "D" "E" "F")
```

```
>>> *LISTA*  
("A" "B" "D" "E" "F")
```

A função `delete` usa o argumento opcional `test` segundo as mesmas regras apresentadas para a função `find`.

Se uma lista possui elementos numéricos, esta pode processar algumas operações matemáticas. Por exemplo, indicar o somatório dos elementos de uma lista com a função `apply`. Observe as instruções a seguir.

```
>>> (apply '+ '(1 2 3 4 5))  
15
```

```
>>> (apply '+ *A*) ; elementos (1 3 4 5 6 7)  
26
```

Observe no próximo exemplo a apresentação do valor do somatório dos valores de **1 a 5** definidos em uma lista.

```
>>> (apply '* '(1 2 3 4 5))  
120
```

A função `apply` aplica a partir de seu primeiro argumento um designador de função que é iterado sobre o segundo argumento caso este seja uma lista. Se for um valor numérico simples `apply` aplica sua ação diretamente sobre o valor. Note alguns exemplos de uso da função `apply`.



```
>>> (apply #' + 1 2 3 '(4 5 6)) ; soma valores com lista
21
```

```
>>> (apply #' cos '(1.0))
0.5403023
```

Anteriormente foram apresentadas as funções `nconc` e `append` para realizar respectivamente a inserção de dados em listas de forma física e virtual. Chegou a hora de conhecer a função `cons` que pode ser usada para definir ações de concatenação em listas e definir células de valores (pares de valores).

Diferentemente da função `append` que armazena virtualmente um elemento ao final da lista a função `cons` adiciona virtualmente um elemento no início da lista. Observe os exemplos a seguir.

```
>>> *A*
(1 3 4 5 6 7)

>>> (cons 9 *A*)
(9 1 3 4 5 6 7)

>>> (cons 1 '(2 3 4))
(1 2 3 4)

>>> *A*
(1 3 4 5 6 7)
```

Veja que a função `cons` insere o elemento indicado no primeiro argumento como componente da lista informada no segundo argumento. No entanto, o segundo argumento da função `cons` pode ser definido por outra função `cons`, desde que o último `cons` indique além do último elemento o valor `NIL` para informar o final da lista. Veja o exemplo seguinte.

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

Uma estrutura `cons` gerenciada pela função `cons` pode ser representada de duas maneiras diferentes. Uma na forma de lista como apresentado anteriormente, outra

na forma de pares-com-ponto (células de valores), onde o primeiro elemento antes do ponto é um `car` e o segundo elemento depois do ponto é um `cdr`, formada pelos pares (`car . cdr`). Por exemplo, (`JAN . 1`) é um `cons` cujo `car` é o símbolo `JAN` e o `cdr` é o valor numérico 1. Observe a seguir os exemplo de uso de pares separados por pontos.

```
>>> (cons 1 2)
```

```
(1 . 2)
```

```
>>> (car (cons 1 2))
```

```
1
```

```
>>> (cdr (cons 1 2))
```

```
2
```

```
>>> (cons 'Jan 1)
```

```
(JAN . 1)
```

```
>>> (cons '(A) '(B C D))
```

```
((A) B C D)
```

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

```
((1 . 2) 3 . 4)
```

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

```
(1 . 2)
```

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

```
1
```

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

```
(3 . 4)
```

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

```
4
```

As funções `car` e `cdr` possuem variantes de funções para o tratamento de níveis subsequentes de pares, como: `(car (car x))` e `(cdr (cdr x))`. Neste caso, considere respectivamente as funções `caar` e `cddr`. Veja os exemplos a seguir.

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

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

Para o tratamento de níveis subsequentes de pares há um conjunto maior de funções. É importante que você efetue consulta das documentações da linguagem. Um bom lugar para começar é o sítio <http://clhs.lisp.se/Front/index.htm>.

O tipo de dado *list* é formado tendo seu segundo elemento como uma nova célula `cons` sem o uso de ponto, tendo como último elemento o valor `NIL`. Observe a seguir as definições de duas listas uma definida a partir da função `list` e outra definida a partir da função `cons`.

```
>>> (cons '1 (cons '2 '()))  
(1 2)
```

```
>>> (list '1 '2)  
(1 2)
```

A criação de lista com as funções `list` e `cons` possui uma pequena diferença operacional. A lista anterior criada por `cons` omitiu no segundo argumento indicado por `(cons '2 '())` o valor `NIL` ao fazer uso do indicativo `'()` gerando uma lista `(1 2)` ao invés da célula `(1 . 2)`. O símbolo `'()` substitui a definição explícita de `NIL`. Note a seguir alguns exemplos variados de definição de listas.

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

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

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

```
>>> '(a b c)  
(A B C)
```

```
>>> (list 'a 'b 'c)  
(A B C)
```

É óbvio, por questões de praticidade, que fazer uso da função `list` ou do símbolo `(')` aspas simples para definir listas é mais vantajosa que usar a função `cons`, mas a função `cons` deve sempre ser considerada para a criação de células de pares de valores.

Para a localização e apresentação de elementos de uma lista já foram apresentadas as funções `car`, `cdr`, `first`, `rest`, `last` e `find`. Dentro deste escopo mais algumas funções que podem ser usadas para a localização de elementos, destacando-se `first`, `second`, `third`, `fourth`, `fifth`, `sixth`, `seventh`, `eighth`, `ninth`, `tenth` e `subseq`.

As funções `first`, `second`, `third`, `fourth`, `fifth`, `sixth`, `seventh`, `eighth`, `ninth` e `tenth` permitem obter a apresentação do primeiro até o décimo elemento se estes existirem na lista indicada. Caso seja pedido um valor de uma posição inexistente será retornado como resposta o valor `NIL`. Veja alguns exemplos de uso dessas funções.

```
>>> (first (list 'a 'b 'c))  
A
```

```
>>> (sixth (list 'a 'b 'c))  
NIL
```

Uma função de apresentação de elementos interessante é a `subseq` que apresenta valores subsequentes existentes em certa lista. Observe os exemplos seguintes.

```
>>> (subseq (list 'a 'b 'c) 0 1)  
(A)
```

```
>>> (subseq *LISTA* 0 1)
("A")
```

```
>>> (subseq (list 'a 'b 'c) 1 2)
(B)
```

```
>>> (subseq (list 'a 'b 'c) 0 2)
(A B)
```

```
>>> (subseq (list 'a 'b 'c) 0 3)
(A B C)
```

```
>>> (subseq (list 'a 'b 'c) 1 3)
(B C)
```

A função `subseq` faz uso de três argumentos, sendo o primeiro argumento a lista a ser manipulado, o segundo argumento determina a posição que se deseja de certo elemento e o terceiro argumento a quantidade de elementos a ser apresentada a partir da posição indicada no segundo argumento.

Uma lista pode ainda ser formada por elementos, seus átomos, na forma de letras, números, caracteres, cadeias e pela junção desses elementos. Observe a seguir uma lista formada com um operador de adição e três elementos numéricos.

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

Observe que a função `cons` permitiu concatenar o operador de adição (+) aos elementos (1 2 3) formando a lista (+ 1 2 3).

A partir da lista (+ 1 2 3) é possível efetuar a adição dos valores com auxílio da função `eval`. Observe a instrução seguinte.

```
>>> (eval (cons '+ '(1 2 3)))
6
```

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

A função `eval` efetua o processamento de elementos de uma lista a partir da indicação de um argumento na forma de uma expressão válida, ou seja, força a avaliação da expressão indicada.

Uma lista pode ter seus elementos ordenados a partir de certa ordem. A definição de uma lista por ser formada com elementos a partir de qualquer ordem de disposição. A ordenação de listas pode ser realizada com as funções `sort` e `stable-sort`. Veja as instruções seguintes para ordenação de lista numérica.

```
>>> (sort '(2 3 1 7 5 8 4 9 6 0) #'<)
(0 1 2 3 4 5 6 7 8 9)
```

```
>>> (sort '(2 3 1 7 5 8 4 9 6 0) #'>)
(9 8 7 6 5 4 3 2 1 0)
```

A função `sort` opera sua ação a partir de dois argumentos, sendo o primeiro argumento a indicação de uma lista não ordenada, o segundo argumento a definição de uma função de comparação que pode ser crescente (`'>`) ou decrescente (`'<`).

A função `sort` efetua sua ação fisicamente na lista, ou seja, ela ordena os valores na lista apontada e não permite o retorno a disposição original da lista. Caso necessite manter a lista original aconselha-se fazer uso da função `copy-list`. Observe o exemplo seguinte.

```
>>> (defvar *LISTA_ORIG* '(1 3 5 2 4))
*LISTA_ORIG*
```

```
>>> *LISTA_ORIG*
(1 3 5 2 4)
```

```
>>> (defvar *LISTA_COPY* (copy-list *LISTA_ORIG*))
*LISTA_COPY*
```

```
>>> *LISTA_COPY*
(1 3 5 2 4)
```

```
>>> (sort *LISTA_COPY* #'>)
(5 4 3 2 1)
```

```
>>> *LISTA_ORIG*
```

```
(1 3 5 2 4)
```

```
>>> *LISTA_COPY*
```

```
(5 4 3 2 1)
```

A função `copy-list` faz a cópia da lista indicada como seu argumento e para seu efetivo uso deve ser associada a uma ação de atribuição como demonstrado com o uso da função `defvar`.

Além da função `sort` há a função `stable-sort`. A diferença entre essas funções é que a função `stable-sort` garante que dois elementos idênticos serão apresentados na mesma ordem em que foram definidos junto à lista original.

Para realizar a ordenação de caracteres e cadeias é necessário fazer uso de funções de comparação pertinentes aos dados **char** e **string**. Veja em seguida alguns exemplos de ordenação de caracteres e cadeias.

```
>>> (sort "afcbed" #'char-lessp)
```

```
"abcdef"
```

```
>>> (sort "afcbed" #'char-greaterp)
```

```
"fedcba"
```

```
>>> (sort '(#\L #\I #\S #\P) #'char-greaterp)
```

```
(#\S #\P #\L #\I)
```

```
>>> (sort '(#\L #\I #\S #\P) #'char-lessp)
```

```
(#\I #\L #\P #\S)
```

```
>>> (sort '("silvio" "silvia" "carlos") #'string-lessp)
```

```
("carlos" "silvia" "silvio")
```

```
>>> (sort '("silvio" "silvia" "carlos") #'string-greaterp)
```

```
("silvio" "silvia" "carlos")
```

Observe que para a ordenação de elementos textuais foram usadas as funções de comparação `char-greaterp`, `char-lessp`, `string-lessp` e `string-greaterp`.

Além da classificação de elementos de listas é possível realizar a junção de duas ou listas por meio da função `merge` que possui a sintaxe básica.

**`merge (<tipo> <sequência1> <sequência2> <#'função>)`**

Onde ***tipo*** representa o tipo de retorno da ação efetuada pela função; ***sequência1*** e ***sequência2*** são as definições da coleção dos elementos a serem mesclados; ***função*** estabelece uma ação a ser aplicada sobre a lista gerada.

Assim sendo, observe as instruções seguintes.

```
>>> (merge 'list '(1 3 5) '(2 4 6) #'<)
(1 2 3 4 5 6)
```

```
>>> (merge 'list '(1 3 5) '(2 4 6) #'=)
(1 3 5 2 4 6)
```

```
>>> (merge 'list '(1 3 5) '(2 4 6) #'>)
(2 4 6 1 3 5)
```

Na função `merge` quando se usa a função “<” obtém-se a junção de duas listas em ordem crescente. Se usada a função “=” será apresentada a junção com a mesma ordem de definição e se usada a função “>” será apresentada a junção com a segunda lista a frente da primeira lista.

## 3.5 TIPO DE DADO SÍMBOLO

---

O tipo de dado ***symbol*** (símbolo) serve para uso em vários propósitos, caracterizados por sequências de caracteres sem o uso de caracteres que representem espaços em branco, parênteses, tralha, aspa simples, aspa inglesa ou ponto. É muito comum na representação de símbolos fazer uso dos caracteres hífen e asterisco.

A manipulação da lista de propriedades de um símbolo é realizada a partir funções fornecidas, dando a possibilidade de tratar o símbolo como se fosse uma estrutura



de dado vinculada a registros. Um registro é composto por um conjunto de campos com a finalidade de representar um conjunto de dados na forma de informação.

Os símbolos podem ser usados para representar certos tipos de variáveis. Neste caso, existirá um conjunto de funções para o tratamento deste tipo de operação. Símbolos como variáveis são constituídos com o uso das funções `setq` e `setf` já apresentadas quando da definição de variáveis, mas sem o uso prévio da função `defvar`.

A definição do nome de um símbolo aceita o uso de caracteres alfabéticos maiúsculos, numéricos ou caracteres de acentuação excetuando-se os caracteres parênteses e espaço em branco.

É pertinente salientar que os nomes de símbolos podem ser grafados com caracteres maiúsculos, minúsculas ou ambas as formas. No entanto, o ambiente CLISP converte caracteres minúsculos em maiúsculos.

Observe o exemplo seguinte de definição do símbolo **\*VLR\*** com valor **3** como se variável global.

```
[n]> (setq *vlr* 3)
3
```

No caso do programa SBCL o uso das funções `setq` e `setf` apesar de surtirem o efeito desejado da definição do símbolo de representação das variáveis apresentam mensagem de advertência informando que a variável indicada não está previamente definida. Para evitar esta ocorrência deve-se usar antecipadamente a função `defvar`. Observe a ocorrência.

```
* (setq *vlr* 3)
; in: SETQ *VLR*
; (SETQ *VLR* 3)
;
; caught WARNING:
; undefined variable: *VLR*
;
; compilation unit finished
; Undefined variable:
; *VLR*
```

```
; caught 1 WARNING condition
3
```

Uma vez definido um símbolo é possível obter o valor armazenado no componente da lista de propriedades com a função `symbol-value`. Observe os exemplos seguintes.

```
>>> (symbol-value '*vlr*')
3
```

Para definir símbolos como registros usa-se a função `get` (integrada a função `setf`) que possui como estrutura sintática a forma.

```
(get <símbolo> <campo> <conteúdo>)
```

Onde, a indicação **símbolo** refere-se ao nome de identificação do símbolo em uso, **campo** refere-se a definição do campo de dados associado ao símbolo e o indicativo **conteúdo** refere-se ao conteúdo associado ao campo.

Observe o exemplo seguinte de definição do símbolo **CARRO** com a definição dos campos: **FABRICANTE**, **MODELO**, **VALOR** formando a estrutura de um registro de dados.

```
>>> (setf (get 'carro 'fabricante) 'Chevrolet)
CHEVROLET
```

```
>>> (setf (get 'carro 'modelo) 'Camaro)
CAMARO
```

```
>>> (setf (get 'carro 'valor) '"R$ 304.645,00")
("R$ 304.645,00")
```

Para visualizar a lista de propriedades definidas para o símbolo **CARRO** usa-se a função `symbol-plist`. Observe o exemplo seguinte.

```
>>> (symbol-plist 'carro)
(VALOR "R$ 304.645,00" MODELO CAMARO FABRICANTE CHEVROLET)
```

Caso deseje remover algum item (propriedade) do registro de dados basta usar a função `remprop` e indicar para ela no primeiro argumento o nome do símbolo e no segundo argumento o nome do campo a ser removido.

Note instrução de remoção de elemento do registro no programa CLISP.

```
[n]> (remprop 'carro 'valor)  
T
```

Note instrução de remoção de elemento do registro no programa SBCL.

```
* (remprop 'carro 'valor)  
(VALOR "R$ 304.645,00" MODELO CAMARO FABRICANTE CHEVROLET)
```

Para verificar a remoção do elemento do registro execute a seguinte instrução.

```
>>> (symbol-plist 'carro)  
(MODELO CAMARO FABRICANTE CHEVROLET)
```

Em CL toda forma de expressão escrita, seja no uso de uma função interna aplicada ou um argumento definido é considerado um dado do tipo símbolo. Tudo o que foi apresentado até o presente momento e o que for apresentado logo após este tópico é em LISP um símbolo.

## 3.6 USO DE DECISÕES

---

Operações de tomada de decisão em CL são obtidas a partir de formulários especiais que usam uma ou mais condições a serem avaliadas para que possa executar certa operação se a condição for verdadeira ou outras operações se a condição for falsa. As ações de tomada de decisão são executadas com os **forms**:

<code>case</code>	construtor que opera com decisão seletiva;
<code>cond</code>	construtor que verifica várias cláusulas de teste;
<code>if</code>	macro que opera decisão composta;
<code>unless</code>	macro que opera decisão simples com fluxo falso;
<code>when</code>	macro que opera decisão simples com fluxo verdadeiro.

O uso de funcionalidades relacionadas a tomada de decisões a partir das macros indicadas mostram-se com código mais elaborado em relação aos exemplos indicados até o momento. Percebe-se então o uso de quantidade maior de parênteses. Essa ocorrência pode dificultar a visualização da indicação de certa ação operacional. Neste sentido, usa-se uma regra de indentação (recoo de código) para cada uma das macros a fim de acomodar melhor a visualização da instrução.

Observe como exemplo a proposta de efetivação da operação aritmética  $(2 + 3) * (8 - 2) / (3 * 5)$  que pode ficar confusa sem o uso de indentação.

```
>>> (float (/ (* (+ 2 3) (- 8 2)) (* 3 5)))
2.0
```

Note o mesmo exemplo com indentação que permite melhor visualização da operação aritmética  $(2 + 3) * (8 - 2) / (3 * 5)$ .

```
>>> (float (/
      (*
        (+ 2 3)
        (- 8 2))
      (* 3 5)))
2.0
```

O uso de indentação refere-se à quantidade de espaços em branco que devem ser definidos no início das linhas de código e como essas linhas se alinham indicando continuidade ou não da instrução em uso. O estilo de indentação CL atualmente adotado evoluiu ao longo de vários anos. A regra de indentação, no geral, é definida, no mínimo a cada duas posições. No entanto, outros espaçamentos podem ser aplicados dependendo do recurso como será em seguida abordado.

O objetivo da indentação é tornar a leitura de uma instrução o mais clara possível. Assim observe atentamente a apresentação das próximas instruções e do estilo de indentação adotado.

A macro `when` opera ação de decisão simples de fluxo verdadeiro a partir da estrutura sintática.

```
(when <(condição)>
      <(ação verdadeira)>)
```

Onde **condição** é o uso de uma função condicional que pode retornar um valor verdadeiro (T) ou falso (NIL) e **ação verdadeira** é a definição do que será executado pela macro `when` se a condição for verdadeira. Observe os exemplos seguintes.

```
>>> (defvar *IDADE*)
*IDADE1*

>>> (setf *IDADE* 19)
19

>>> (when (>= *IDADE* 18)
      (format t "Maior de idade"))
Maior de idade
NIL

>>> (setf *IDADE* 17)
17

>>> (when (>= *IDADE* 18)
      (format t "Maior de idade"))
NIL
```

Perceba que ao se definir a variável **\*IDADE\*** com valor **19** a execução da macro `when` apresenta a mensagem “Maior de idade” se a idade for maior ou igual a **18**. Ao se definir a variável **\*IDADE\*** com valor **17** não ocorre a apresentação de nenhuma mensagem.

A macro `unless` opera ação de decisão simples de fluxo falso a partir da estrutura sintática.

```
(unless <(condição)>
      <(ação falsa)>)
```

Onde **condição** é o uso de uma função condicional que pode retornar um valor verdadeiro (T) ou falso (NIL) e **ação falsa** é a definição do que será executado pela macro `unless` se a condição for falsa. Observe os exemplos seguintes.

```
>>> (defvar *SAUDE*)
*SAUDE*

>>> (setf *SAUDE* "Esta doente")
"Esta doente"

>>> (unless (string= *SAUDE* "Saudavel")
      (format t "Va ao medico"))
Va ao medico
NIL
```

Perceba que ao se definir a variável **\*SAUDE\*** com valor **"Esta doente"** a execução da macro `unless` apresenta a mensagem "Va ao medico" a menos que esteja saudável.

A macro `if` opera ação de decisão composta a partir da estrutura sintática.

```
(if <(condição)>
    <(ação verdadeira)>
    <(ação falsa)>)
```

Onde **condição** é o uso de uma função condicional que pode retornar um valor verdadeiro (T) ou falso (NIL), **ação verdadeira** é a definição do que será executado pela macro `if` se a condição for verdadeira e **ação falsa** é a definição do que será executado pela macro `if` se a condição for falsa. Observe os exemplos seguintes.

```
>>> (defvar *VALOR*)
*VALOR*

>>> (setf *VALOR* 1)
1

>>> (if (evenp *VALOR*)
      (format t "Par")
      (format t "Impar"))
Impar
NIL
```

```
>>> (setf *VALOR* 2)
2

>>> (if (evenp *VALOR*)
        (format t "Par")
        (format t "Impar"))

Par
NIL
```

Veja que ao se definir a variável **\*VALOR\*** com valor **1** a execução da macro **if** apresenta a mensagem “Impar”. Ao se definir a variável **Valor** com valor **2** a execução da macro **if** apresenta a mensagem “Par”.

Para validar se o valor da variável **\*VALOR\*** é par é usada a função **evenp**. Caso deseje validar valores impares pode-se usar a função **oddp**. As funções **evenp** e **oddp** operam com argumentos de valores numéricos inteiros retornando um valor booleano como resposta, sendo T para verdadeiro e NIL para falso a partir da estrutura sintática.

```
(evenp <valor numérico inteiro>)
(oddp <valor numérico inteiro>)
```

O construtor **case** permite definir várias cláusulas de decisão a serem tomadas, a partir da avaliação de certa condição. Observe a estrutura sintática.

```
(case <variável>
  (<valor1> (ação1 ação2 ... açãoN))
  (<valor2> (ação1 ação2 ... açãoN))
  ...
  (<valorN> (ação1 ação2 ... açãoN)))
```

Onde **variável** é o uso de uma variável com a definição de um valor que será avaliado, **valor1**, **valor2** e **valorN** é a definição do valor avaliado a partir do conteúdo definido na **variável**; **ação1**, **ação2** e **açãoN** são as definições das ações a serem realizada para cada um dos valores especificados. Observe os exemplos seguintes.

```
>>> (defvar *DIA-SEMANA*)
*DIA_SEMANA*
```

```
>>> (setf *DIA-SEMANA* 7)
7
>>> (case *DIA-SEMANA*
      (1 (format t "~%domingo~%"))
      (2 (format t "~%segunda-feira~%"))
      (3 (format t "~%terca-feira~%"))
      (4 (format t "~%quarta-feira~%"))
      (5 (format t "~%quinta-feira~%"))
      (6 (format t "~%sexta-feira~%"))
      (7 (format t "~%sabado~%")))
```

sabado

NIL

Para o construtor `case` se for fornecido um valor para a variável `*DIA_SEMANA*` que não esteja especificado na lista de valores ocorrerá apenas a apresentação do valor NIL.

O construtor `cond` permite definir várias cláusulas de decisão a serem tomadas, a partir da avaliação de certa condição. Observe a estrutura sintática.

```
(cond
  ((<condição1>) (<ação1>))
  ((<condição2>) (<ação2>))
  (...)
  ((<condiçãoN> <açãoN>))
  [(<t> (<açãoextra>))])
```

Onde **condição1**, **condição2** e **condiçãoN** é o estabelecimento de uma função condicional que retorne T (verdadeiro) ou NIL (falso) e **ação1**, **ação2** e **açãoN** são as especificações de ações executadas se as condições forem verdadeiras. Se nenhuma condição for satisfeita poderá ocorrer a execução opcional da **açãoextra** definida a partir de uma condição genérica como verdadeira se nenhuma ação condicional for anteriormente executada. Observe os exemplos seguintes.

```
>>> (defvar *TRIMESTRE*)
*TRIMESTRE*
```



```
>>> (setf *TRIMESTRE* 2)
2
>>> (cond
      ((= *TRIMESTRE* 1) (format t "1o. trimestre~%"))
      ((= *TRIMESTRE* 2) (format t "2o. trimestre~%"))
      ((= *TRIMESTRE* 3) (format t "3o. trimestre~%"))
      ((= *TRIMESTRE* 4) (format t "4o. trimestre~%"))
      (t (format t "Trimestre incorreto~%")))

2o. trimestre
NIL
```

Se definido para a variável **\*TRIMESTRE\*** um valor diferente de **1**, **2**, **3** ou **4** ocorrerá a apresentação da mensagem "Trimestre incorreto".

## 3.7 FUNCIONALIDADES LÓGICAS

---

Além das funcionalidades matemáticas para a realização de cálculos e das funcionalidades relacionais destinadas a ações condicionais como predicados, há a existência de um conjunto de funcionalidades lógicas, destinadas a ações de processamento condicional formada por uma função e duas macros.

Na sequência observe algumas funcionalidades do conjunto de recursos operacionais lógicos da linguagem LISP para a realização de operações lógicas.

**and**                      macro para conjunção condicional de valores booleanos;

**not**                      função para inversão condicional;

**or**                        macro para disjunção condicional de valores booleanos.

A função **not** retorna **T** (verdadeiro) se seu argumento é **NIL** (falso), mas se for **NIL** retorna **T**. Observe a estrutura sintática.

```
(not <condição>)
```

Esta função inverte o resultado lógico do argumento por ela apontado, sendo que o argumento em uso pode ser representado por dados de qualquer. Veja alguns exemplos de uso da função `not`.

```
>>> (not NIL)
```

```
T
```

```
>>> (not T)
```

```
NIL
```

```
>>> (not (not T))
```

```
T
```

```
>>> (not 4.5)
```

```
NIL
```

```
>>> (not 'LISP)
```

```
NIL
```

A macro `and` usa no mínimo dois argumentos booleanos como **forms** e retorna um resultado booleano de sua avaliação. Cada **form** usado é avaliado da esquerda para a direita. Se algum dos argumentos avaliados for `NIL` o resultado `NIL` é automaticamente retornado sem avaliar os demais argumentos, qualquer outra situação é considerada não `NIL` (verdadeiro). Essa macro pode ser usada para operações lógicas onde `NIL` significa resultado falso e não `NIL` significa resultado verdadeiro como expressão condicional. Se nenhum argumento for fornecido a macro `and` retorna como resultado o valor `T`. Observe a estrutura sintática.

```
(and <condição1> <condição2> ... <condiçãoN>)
```

Veja alguns exemplos de uso da macro `and` considerando argumentos que resultam em valores lógicos de forma simplificada.

```
>>> (and T T)
```

```
T
```

```
>>> (and T NIL)
NIL
```

```
>>> (and NIL T)
NIL
```

```
>>> (and NIL NIL)
NIL
```

Quando se utiliza argumentos booleanos para a macro `and` obtém-se como resultado o valor lógico não `NIL` (indicado como `T`), ou seja, verdadeiro se os argumentos booleanos da macro `and` forem verdadeiros. Considere como exemplo a apresentação da mensagem “Valores numericos” se dados dois argumentos estes forem numéricos e caso um dos argumentos ou ambos não sejam numéricos apresentar a mensagem “Valores nao numericos”. Observe as instruções seguintes.

```
>>> (defvar *X1*)
*X1*
```

```
>>> (setf *X1* 2.7)
2.7
```

```
>>> (defvar *X2*)
*X2*
```

```
>>> (setf *X2* 4.3)
4.3
```

```
>>> (if (and (numberp *X1*) (numberp *X2*))
      (format t "~%Valores numericos")
      (format t "~%Valores nao numericos"))
```

```
Valores numericos
NIL
```

```
>>> (setf *X2* "A")
"A"

>>> (if (and (numberp *X1*) (numberp *X2*))
        (format t "~%Valores numericos")
        (format t "~%Valores nao numericos"))
```

```
Valores nao numericos
NIL
```

Para o teste apresentado foi usada a função `numberp` que tem por finalidade verificar de forma lógica se certo valor indicado é ou não numérico. Se o valor passado no argumento é numérico a função retorna T (verdadeiro), caso contrário é retornado NIL (falso).

Além da função `numberp` há outras para verificar se um argumento em uso pertence ou não a certo tipo de dado, destacando-se: `listp` (verifica se o argumento passado é uma lista), `symbolp` (verifica se o argumento passado é um símbolo), `integerp` (verifica se o argumento passado é um valor inteiro), `floatp` (verifica se o argumento passado é um valor de ponto flutuante), `complexp` (verifica se o argumento passado é um valor complexo), `stringp` (verifica se o argumento passado é um dado do tipo cadeia), entre outros existentes na linguagem que são baseadas sob a estrutura sintática.

### (funçãop <conteúdo>)

Onde **funçãop** é a identificação de uma função de verificação de dados como `numberp`, `listp`, `symbolp`, `integerp`, `floatp`, `complexp` e `stringp`; **conteúdo** refere-se a indicação do dado a ser verificado.

A macro `or` usa no mínimo dois argumentos booleanos como **forms** e retorna um resultado booleano de sua avaliação. Cada **form** usado é avaliado da esquerda para a direita. Se pelo menos um dos argumentos avaliados for T o resultado T é automaticamente retornado sem avaliar os demais argumentos. Se nenhum argumento for fornecido a macro `or` retorna como resultado o valor NIL. Observe a estrutura sintática.

```
(or <condição1> <condição2> ... <condiçãoN>)
```

Observe alguns exemplos de uso da macro `or` considerando argumentos que resultam em valores lógicos de forma simplificada.

```
>>> (or T T)
T
```

```
>>> (or T NIL)
T
```

```
>>> (or NIL T)
T
```

```
>>> (or NIL NIL)
NIL
```

Quando se utiliza argumentos booleanos para a macro `or` obtém-se como resultado o valor lógico não `NIL` (indicado como `T`), ou seja, verdadeiro se pelo menos um dos argumentos booleanos da macro `or` for verdadeiro. Considere como exemplo a apresentação da mensagem “Valores validos” se dados dois argumentos pelos menos um seja numérico de ponto flutuante ou o outro seja uma cadeia de caracteres. Caso ambos os valores não sejam válidos apresentar a mensagem “Definicao invalida”. Observe as instruções seguintes.

```
>>> (defvar *Y*)
*Y*
```

```
>>> (setf *Y* 2.7)
2.7
```

```
>>> (defvar *S*)
*S*
```

```
>>> (setf *S* "LISP")
"LISP"
```

```
>>> (if (or (floatp *Y*) (stringp *S*))
      (format t "~%Valores validos")
      (format t "~%Definicao invalida"))
```

Valores validos

NIL

```
>>> (setf *Y* "LIVRO")
```

100

```
>>> (setf *S* 100)
```

100

```
>>> (if (or (floatp *Y*) (stringp *S*))
      (format t "~%Valores validos")
      (format t "~%Definicao invalida"))
```

Definicao invalida

NIL

Para o teste apresentado foram usadas as funções `floatp` e `stringp` que tem por finalidade verificar de forma lógica se os valores indicados são respectivamente dos tipos de dados numérico (com ponto flutuante) e cadeia. Se um dos valores passados em um dos argumentos for válido ocorrerá a apresentação do resultado T (verdadeiro), caso contrário é retornado NIL (falso).

---

## 3.8 Uso DE LAÇOS

---

Uma ação recorrente na atividade de programação é a necessidade de se repetir determinado trecho de código algum número de vezes.

Laços podem, dependendo da macro em uso, serem executados para ações interativas e iterativas. Neste tópico são apresentados apenas laços para ações iterativas, pois para tratar ações interativas será necessário a execução de entrada de dados, tema a ser tratado mais adiante nesta obra.

Para atender a este requisito CL oferece funcionalidades específicas para a ação de laços, destacando-se as macros:

do	laço iterativo estruturada;
dolist	laço iterativo sobre cada elemento de uma lista;
dotimes	laço com número fixo de iterações;
loop for	laço para iteração;
loop repeat	laço iterativo de repetição;
loop while	laço condicional para interação ou iteração;
loop	laço condicional executado até encontrar declaração de retorno.

A macro `loop` pode estabelecer laços condicionais ao estilo pré-teste ou pós-teste iterativo ou interativo. Esta macro usa três argumentos estabelecidos a partir das formas sintáticas.

```
(loop <ação> <incremento> <condição>)  
(loop <condição> <ação> <incremento>)
```

Onde, **ação** caracteriza-se por ser a operação executada pelo laço; **incremento** a definição da operação de continuidade do laço até que ser condição seja satisfeita e **condição** a determinação da referência de validação da execução do laço com o uso de uma macro condicional, sendo que a condição pode ser posicionada dentro da macro `loop` em qualquer posição de argumento caracterizando um laço do tipo seletivo.

Veja alguns exemplos de uso de laço `loop` para apresentar os valores da contagem de 1 a 5 de 1 em 1 de forma iterativa.

```
>>> (defvar *I*)  
*I*  
  
>>> (setf *I* 1)  
1
```

```
>>> (loop
      (unless (<= *I* 5)
        (return))
      (format t "~a " *I*)
      (setf *I* (+ *I* 1)))
```

```
1 2 3 4 5
```

```
NIL
```

```
>>> (setf *I* 1)
```

```
1
```

```
>>> (loop
      (when (> *I* 5)
        (return))
      (format t "~a " *I*)
      (setf *I* (+ *I* 1)))
```

```
1 2 3 4 5
```

```
NIL
```

```
>>> (setf *I* 1)
```

```
1
```

```
>>> (loop
      (format t "~a " *I*)
      (setf *I* (+ *I* 1))
      (unless (<= *I* 5)
        (return)))
```

```
1 2 3 4 5
```

```
NIL
```

```
>>> (setf *I* 1)
```

```
1
```



```
>>> (loop
      (format t "~a " *I*)
      (setf *I* (+ *I* 1))
      (when (> *I* 5)
        (return)))
1 2 3 4 5
NIL
```

Na definição de um laço iterativo com a macro `loop` três elementos operacionais são necessários, sendo: a inicialização da contagem do laço realizada pela instrução `(setf *I* 1)`; a verificação do final de contagem, neste caso, com a instrução `(when (> *I* 4) (return))` ou `(unless (<= *I* 4) (return))` e o incremento da contagem com a instrução `(setf *I* (+ *I* 1))`.

A instrução `(setf *I* (+ *I* 1))` faz com que a variável `*I*` seja definida com seu valor atual mais uma unidade a partir da ação `(+ *I* 1)` definida para a função `setf`.

A finalização do laço ao estilo `loop` ocorre com o uso da macro `when` ou da macro `unless` que quando verdadeiras farão a saída do escopo da macro `loop` efetuando o retorno do último valor da variável avaliada no laço por meio da macro `return`. Um laço do tipo `loop` é encerrado quando é encontrada a macro `return`, sendo `return` obrigatório.

O laço controlado pela macro `do` é usado para realizar laços iterativos a partir de uma estrutura de iteração com definição local de variáveis que permite estabelecer um conjunto de variáveis e valores que ao serem verificados condicionalmente determinam se o laço continua ou seja encerrado. Este laço executa um bloco de instruções enquanto uma condição é verdadeira. A macro `do` usa a seguinte forma sintática.

```
(do
  ((<variável1> <valor1> (<incremento1>))
   (<variável2> <valor2> (<incremento2>))
   ...)
  (<variávelN> <valorN> (<incrementoN>)))
(<condição>))
(<ação>))
```

Onde, **variável1**, **variável2** e **variávelN** caracterizam-se por serem as definições de variáveis locais a serem usadas no escopo da macro **do**; os argumentos **valor1**, **valor2** e **valorN** determinam o valor inicial de cada variável; os argumentos **incremento1**, **incremento2** e **incrementoN** determina o incremento a ser definido para cada variável definida. O argumento **condição** permite estabelecer a condição de saída do laço e o argumento **ação** especifica o que deve ser realizado pelo laço. Após cada iteração a condição é avaliada e sendo diferente de zero, ou seja, verdadeira o valor de retorno avaliado é retornado. O argumento **ação** é opcional. Se presente será executado após cada iteração até que o valor lógico da condição seja verdadeiro.

Observe as instruções seguintes para a execução de um laço **do** que apresenta os valores das coordenadas **X** iniciando em **1** com incremento de **1** em **1** e **Y** iniciado em **20** com decremento de **1** em **1** até que o valor da coordena **X** seja maior que **10**.

```
>>> (do
      ((X 1 (+ X 1))
       (Y 20 (- Y 1)))
      (> X 10))
      (format t "coordenada x = ~2,' d | y = ~2,' d~%" X Y))
coordenada x = 1 | y = 20
coordenada x = 2 | y = 19
coordenada x = 3 | y = 18
coordenada x = 4 | y = 17
coordenada x = 5 | y = 16
coordenada x = 6 | y = 15
coordenada x = 7 | y = 14
coordenada x = 8 | y = 13
coordenada x = 9 | y = 12
coordenada x = 10 | y = 11
NIL
```

Os valores iniciais estabelecidos para as variáveis **X** e **Y** são avaliados e vinculados à própria variável respectivamente com os valores **1** e **20**. O valor atualizado no argumento **incremento** especifica como os valores das variáveis são atualizados em cada iteração, sendo **X** acrescido de **1** em **1** (**X 1 (+ X 1)**) e **Y** decrescido de **1** em **1** (**Y 20 (- Y 1)**). A cada iteração o teste condicional (**> X 10**) é

avaliado, sendo verdadeiro ocorre o encerramento do laço, caso contrário o laço é processado mais uma vez e a apresentação dos valores das variáveis **X** e **Y** são exibidos.

O laço `loop for` é a maneira mais simples e sofisticada de realizar diversas ações iterativas. Esta forma de laço permite realizar:

- configurar variáveis para iteração;
- estabelecer condição que permite encerrar condicionalmente a iteração;
- estabelecer condição que permite executar alguma ação a cada iteração;
- estabelecer condição que permite executar alguma ação antes de sair do loop

O laço `loop for` pode ser definido para realizar a contagem de um valor inicial até um valor final para uma variável com incremento de uma unidade, a partir da sintaxe.

```
(loop for <var> from <vlr inicial> to/downto <vlr final> [by <inc>]  
do (<ação>))
```

Onde, **ação** é a definição do que será efetivamente repetido na execução do laço; **var** é a definição de variável local que será usada para controlar o número de vezes que a iteração será processada que pode ser a partir da definição de um **vlr inicial** e **vlr final** com a definição opcional **inc** que determina o valor do incremento da contagem.

A cláusula `from` permite estabelecer o início da contagem que poderá ser crescente quando usada a cláusula `to` ou decrescente quando usada a cláusula `downto`.

Observe as instruções seguintes para a execução de um laço `loop for` que apresenta os valores de **1** a **5** com incremento de uma unidade (sem a necessidade da cláusula `by`) obtidos a partir de uma iteração.

```
>>> (loop for I from 1 to 5
      do (format t "~d~%" I))
1
2
3
4
5
NIL
```

Note que a apresentação dos valores foi feito no sentido vertical, diferentemente dos exemplos anteriores que ocorreram no sentido horizontal. A mudança se faz necessária, pois se mantido a ação (`format t "~a " *I*`) os valores apresentados na linha seriam todos iguais, sendo necessário ter um salto de linha para que a ação do incremento ocorra.

O laço `loop for` com os complementos `from` e `to` estabelece a faixa de abrangência entre os valores definidos para o início e final da contagem, incluindo-se os valores definidos.

Observe as instruções seguintes para a execução de um laço `loop for` que apresenta os valores de **1** a **10** de **2** em **2** obtidos a partir de uma iteração.

```
>>> (loop for I from 1 to 10 by 2
      do (format t "~d~%" I))
1
3
5
7
9
NIL
```

Observe as instruções seguintes para a execução de um laço `loop for` que apresenta os valores de **5** a **1** com incremento de uma unidade.

```
>>> (loop for I from 5 downto 1
      do (format t "~d~%" I))
5
4
3
2
1
NIL
```

O laço `loop for` pode ser definido para realizar a iteração de elementos sobre uma lista definida de valores com auxílio da cláusula `in`, a partir da sintaxe.

```
(loop for <variável> in <lista> [by #'<função>]
      do (<ação>))
```

Onde, **ação** é a definição do que será efetivamente repetido na execução do laço; **variável** é a definição de variável local que será usada para controlar o número de vezes que a iteração será processada; **lista** a indicação da coleção de valores que será iterada e cláusula opcional `by` determina a aplicação de uma função sobre os valores da lista indicada.

Observe as instruções seguintes para a execução de um laço `loop for` que apresenta os valores de **1 a 5** definidos dentro de uma lista com iteração por unidade.

```
>>> (loop for I in '(1 2 3 4 5)
      do (format t "~d~%" I))
1
2
3
4
5
NIL
```

O laço `loop for` com o complemento `in` define um iterador que percorre todos os elementos da lista indicada transferindo cada valor para a variável em uso.

Observe as instruções seguintes para a execução de um laço `loop for` que apresenta na faixa de valores de **1 a 5** com incremento **2**.

```
>>> (loop for I in '(1 2 3 4 5) by #'cddr
      do (format t "~d~%" I))
1
3
5
NIL
```

A função `cddr` é um recurso assessor que possibilita acesso aos elementos de uma lista, sendo está uma forma reduzida de executar ação similar com `(cdr (cdr X))`, onde `X` é o restante da lista, lembrando que a função `cdr` apresenta todos os elementos da lista excetuando-se o primeiro na forma de lista.

O laço `loop for` além do complemento `in` pode ser operado com as cláusulas `upto` para contagem positiva de zero até o limite estabelecido e `below` para contagem positiva do valor zero até o anterior do limite estabelecido.

Estrutura de contagem positiva.

```
(loop for <variável> upto <limite> [by <incremento>]
      do (<ação>))
```

Estrutura de contagem negativa.

```
(loop for <variável> below <limite> [by <incremento>]
      do (<ação>))
```

Onde, **ação** é a definição do que será efetivamente repetido na execução do laço; **variável** é a definição de variável local que será usada para controlar o número de vezes que a iteração será processada; **limite** é a do valor máximo da contagem e a cláusula opcional `by` determina o estabelecimento do passo de contagem.

Observe as instruções seguintes para a execução de um laço `loop for` com a cláusula `upto` de zero a 5.

```
>>> (loop for I upto 5
      do (format t "~d~%" I))
0
1
2
3
4
5
NIL
```

A ação anterior apresenta a contagem do valor zero e segue até o valor do limite **5** estabelecido após upto.

Observe as instruções seguintes para a execução de um laço loop for com a cláusula upto de zero a **5** de **2** em **2**.

```
>>> (loop for I upto 5 by 2
      do (format t "~d~%" I))
0
2
4
NIL
```

Observe as instruções seguintes para a execução de um laço loop for com a cláusula below de zero até **4** a partir da definição do limite **5**.

```
>>> (loop for I below 5
      do (format t "~d~%" I))
0
1
2
3
4
NIL
```

A ação anterior apresenta a contagem do valor zero e segue até o valor anterior ao limite **5** estabelecido após below.

Além das cláusulas `in`, `from`, `downto`, `to`, `upto`, `below` e `by` usadas com `loop for` é possível fazer uso de outras cláusulas, destacando-se nesta obra algumas mais importantes: `while` (não confundir com `loop while`), `until`, `when`, `if`, `above` (contrário de `below` já apresentado), `on`, `downfrom`, `upfrom`, `downto` (contrário de `upto` já apresentado) `across` e `collect`.

A cláusula `while` tem por finalidade iterar uma lista enquanto certa condição não é satisfeita. Veja a seguir a apresentação dos valores pares enquanto um valor ímpar não for encontrado. Encontrado um valor ímpar a lista é encerrada mesmo que haja valores pares após o valor ímpar.

```
>>> (loop for I in '(2 4 6 7 8) while (evenp I) do (format t "~a " I))
2 4 6
NIL
```

A cláusula `until` tem por finalidade iterar uma lista até que certa condição seja satisfeita. Veja a seguir a apresentação dos valores até que seja encontrado o primeiro valor par.

```
>>> (loop for I in '(1 3 6 7 8) until (evenp I) do (format t "~a " I))
1 3
NIL
```

A cláusula `when` tem por finalidade iterar uma lista quando nela é encontrada a certa condição. Veja a seguir a apresentação dos valores maiores que quatro e menores que três.

```
>>> (loop for I in '(1 2 3 4 5 6)
      when (> I 4)
      do (format t "Maior: ~a " I)
      when (< I 3)
      do (format t "Menor: ~a " I))
Menor: 1 Menor: 2 Maior: 5 Maior: 6
NIL
```

A cláusula `if` tem por finalidade iterar uma lista quando certa condição é verdadeira. Veja a seguir a apresentação dos valores maiores que quatro.



```
>>> (loop for I in '(1 2 3 4 5 6) if (> I 4)
      do (format t "~a " I))
5 6
NIL
```

A cláusula `above` tem por finalidade iterar uma lista com valores acima do valor apontado. Veja a seguir a apresentação dos valores até cinco acima de três.

```
>>> (loop for I from 5 above 3
      do (format t "~a " I))
5 4
NIL
```

A cláusula `on` tem por finalidade iterar uma lista gerando a partir dela sub listas menores a até não existir nenhum elemento. Veja a seguir a apresentação dos valores de uma lista e suas sub listas.

```
>>> (loop for I on '(1 2 3 4 5) do (format t "~a~%" I))
(1 2 3 4 5)
(2 3 4 5)
(3 4 5)
(4 5)
(5)
NIL
```

A cláusula `upfrom` tem por finalidade iterar uma lista de forma crescente de um valor inicial até um valor final. Veja a seguir a apresentação dos valores de uma lista com os valores de **1** a **5**.

```
>>> (loop for I upfrom 1 to 5 do (format t "~a " I))
1 2 3 4 5
NIL
```

A cláusula `downfrom` tem por finalidade iterar uma lista de forma decrescente de um valor inicial até um valor final. Veja a seguir a apresentação dos valores de uma lista com os valores de **5** a **1**.

```
>>> (loop for I downfrom 5 to 1 do (format t "~a " I))
5 4 3 2 1
NIL
```

A cláusula `downto` tem por finalidade iterar uma lista de forma decrescente de um valor inicial até um valor final. Veja a seguir a apresentação dos valores de uma lista com os valores de **5** a **1**.

```
>>> (loop for I from 5 downto 1 do (format t "~a " I))
5 4 3 2 1
NIL
```

A cláusula `across` tem por finalidade iterar uma matriz. Veja a seguir a apresentação a obtenção do somatório dos elementos de uma matriz.

```
>>> (loop for I across #(1 2 3 4 5) sum I)
15
NIL
```

A cláusula `collect` tem por finalidade coletar cada valor iterado e armazená-lo em uma lista. Esta cláusula não funciona dentro de formulários ou laços aninhados. Note a seguir os exemplos de uso da cláusula `collect` para gerar listas de valores.

```
>>> (loop for I to 5 collect I)
(0 1 2 3 4 5)
```

```
>>> (loop for I from 1 to 5 collect I)
(1 2 3 4 5)
```

```
>>> (loop for I from 1 to 5 if (oddp I) collect I)
(1 3 5)
```

```
>>> (reverse (loop for I from 1 to 5 if (oddp I) collect I))
(5 3 1)
```

O laço `loop while` é um recurso que pode ser usado para a execução condicional de laços iterativos ou interativos. Observe a sequência a seguir para a apresentação iterativa de valores de **1** a **5** com `loop while`.

```
>>> (setf *I* 1)
1

>>> (loop while (<= *I* 5)
      do (format t "~d~%" *I*)
      (setf *I* (+ *I* 1)))
1
2
3
4
5
NIL
```

Observe a sequência a seguir para a apresentação iterativa de valores de **1** a **5** de **2** em **2** com loop while.

```
>>> (setf *I* 1)
1

>>> (loop while (<= *I* 5)
      do (format t "~d~%" *I*)
      (setf *I* (+ *I* 2)))
1
3
5
NIL
```

Observe a sequência a seguir para a apresentação iterativa de valores de **5** a **1** com loop while.

```
>>> (setf *I* 5)
5
```

```
>>> (loop while (>= *I* 1)
      do (format t "~d~%" *I*)
      (setf *I* (- *I* 1)))
5
4
3
2
1
NIL
```

O laço controlado pela macro `dotimes` permite definir um valor fixo que será usado na ação de contagem da variável indicada para uso. Esta macro usa três argumentos estabelecidos a partir das formas sintáticas.

**(dotimes (<variável> <limite>) (ação>))**

Onde, **variável** é a definição da variável que será usada como contador; **limite** é a definição do valor máximo de contagem e **ação** é a definição da operação que será repetida.

Observe as instruções seguintes para a execução de um laço `dotimes` que apresenta os valores de **0** a **4** obtidos a partir de uma iteração.

```
>>> (dotimes (I 5)
      (format t "~d~%" I))
0
1
2
3
4
NIL
```

O valor definido para a variável de controle de um laço `dotimes` opera a partir de uma contagem cardinal. Desta forma ao se estabelecer o valor limite **5** obtém-se os valores de **0** até **4**.

O laço controlado pela macro `dolist` assemelha-se a ação do laço `loop for` com complemento `in`. Esta macro usa três argumentos estabelecidos a partir das formas sintáticas.

```
(dolist (<variável> <('lista)>))
```

Onde, **variável** é a definição da variável que será usada para iterar os elementos de uma lista e **'lista** é a definição do dado a ser iterado.

Observe as instruções seguintes para a execução de um laço `dolist` que apresenta os valores de **1** a **5** definidos dentro de uma lista.

```
>>> (dolist (I '(1 2 3 4 5))
      (format t "~d~%" I))
1
2
3
4
5
NIL
```

O laço `dolist` estabelece um iterador que percorre todos os elementos da lista indicada transferindo cada valor para a variável em uso.

É importante salientar que as macros para as ações dos laços `loop`, `loop while` e `do` podem ser usados para a definição de laços iterativos com contador crescente e decrescente. Os laços `loop for`, `dotimes` e `dolist` só podem ser usados para criar laços eminentemente iterativos de contagem crescente.

## 3.9 TIPO DE DADO FUNÇÃO

---

A linguagem CL opera todas as suas operações baseadas em funções. Algumas funções usadas na linguagem são primitivas, ou seja, são funções internas, existentes no **kernel** da linguagem como diversos recursos já apresentados. No entanto, é possível definir funções externas que sejam criadas por quem esteja programando o ambiente.

Uma função, do ponto de vista computacional, caracteriza-se por ser uma rotina ou módulo de sub-rotina de um programa maior que tem por finalidade retornar certo resultado como resposta a sua ação. Uma sub-rotina que não retorne valor, mesmo em linguagens de programação que operem com rotinas de funções, são referenciadas como procedimento. O uso de funções permite resolver grandes problemas em partes menores, levando-se em consideração que uma função deve ser definida da maneira mais simples possível.

Em LISP as funções são consideradas tipos de dados e são estabelecidas com o uso da função `defun` que possui como sintaxe a estrutura.

```
(defun <nome> <(argumento1 argumento2 ... argumentoN)>
  <bloco>)
```

Onde, **nome** estabelece o rótulo de identificação da função no ambiente global da linguagem, **argumento1**, **argumento2** e **argumentoN** estabelecem a relação de parâmetros que serão passados a função e o **bloco** estabelece as instruções realizadas pela função.

A definição de uma função poderá ser feita em uma linha de código se a função for pequena, mas sendo grande aconselha-se trabalhar com a definição da estrutura de indentação.

Como exemplo de definição de função considere a criação de sub-rotina que apresente o resultado do quociente inteiro da divisão de dois valores numéricos (dividendo e divisor) indicados, tanto como valores numéricos inteiros, bem como valores numéricos de ponto flutuante. Neste caso, é conveniente usar duas outras operações, sendo a função `floor` e a macro `multiple-value-bind`.

A função `floor`, já apresentada, retorna o valor do quociente inteiro tendendo este ao infinito negativo e o valor do resto da divisão a partir do fornecimento de dois argumentos numéricos, sendo o primeiro o valor do dividendo e o segundo o valor do divisor. Se usado apenas um argumento a função `floor` retorna a parte inteira, o expoente, do valor numérico de ponto flutuante como quociente e devolve o resto do valor expresso como a mantissa. Quando usado dois argumentos a função efetua a divisão do primeiro valor pelo segundo e retorna o quociente inteiro e o resto em ponto flutuante dos valores indicados. Se executada a operação (`floor 1.1`) ocorre a apresentação do valor de expoente 1 e do valor da mantissa 0.100000024, mas se executada a operação (`floor 7 2`) ocorre a apresentação do valor de expoente 3 e do valor de resto da divisão 1.

A macro `multiple-value-bind` tem por objetivo criar ligações entre argumentos e ações representadas por **forms** associados aos argumentos fornecidos permitindo a criação de resultados especiais a partir de alguma ação efetuada.

**(multiple-value-bind (<variáveis>) <operação> <resultado>)**

Onde, **variáveis** é a definição de uma lista de argumentos associadas a certa ação a partir de variáveis locais, **operação** é o uso de algum recurso da linguagem como **form** associado as variáveis definidas para uso e **resultado** é o que se deseja efetivamente obter como outro **form**.

Veja exemplo para obter uma lista com os valores truncados do expoente e mantissa a partir de um valor de ponto flutuante indicado com a função `truncate`.

```
>>> (multiple-value-bind (V1 V2) (truncate 0.2) (list V1 V2))  
(0 0.2)
```

Observe que são definidos para a macro `multiple-value-bind` como argumentos as variáveis **V1** e **V2** que são associadas ao resultado `(list V1 V2)`, onde a variável **V1** recebe o primeiro valor e a variável **V2** recebe o segundo valor da operação realizada pela função `truncate` e são assim apresentadas com o uma lista formada pelo expoente e a mantissa.

A partir das funcionalidades `floor` e `multiple-value-bind` observe a definição da função externa `div` que apresenta o resultado do quociente inteiro da divisão de um dividendo por um divisor.

```
>>> (defun div (A N)  
      (multiple-value-bind (Q) (floor A N) Q))
```

Apesar de poder ser definida a função `div` em uma só linha o uso do efeito de indentação permite melhor visualização e entendimento da ação da função definida.

Veja que na definição da função externa `div` pela função `defun` há a recepção dos argumentos representados pelas variáveis locais **A** (para o dividendo) e **N** (para o divisor) que são usadas no cálculo da função `floor` que ao ser processada retornará dois valores um sendo o quociente e outro sendo o resto da divisão. No entanto, apenas o valor do quociente está sendo recuperado junto a variável **Q** (quociente) associada a operação da função `floor` pela macro `multiple-value-bind`.

A partir da função `div` observe a seguir a instrução de uso da função com a sintaxe.

```
>>> (div 7 2)
3
```

Note que é apresentado apenas o valor do resultado do quociente obtido a partir dos dois valores definidos para a função `div`.

Considerando o desenvolvimento de funções para o cálculo da fatorial de um valor inteiro qualquer são propostas algumas formas diferentes. O cálculo de uma fatorial ocorre com a obtenção do produto dos números inteiros consecutivos de **1** até o valor definido como limite da fatorial. Veja a seguir três exemplos indicados para calcular o resultado da fatorial de um valor qualquer. Nestes exemplos são indicados em conjunto o uso de funções de validação como `cond`, `if` e/ou `and` para auxiliar o cálculo de cada função apresentada.

A função `fatorial1` a seguir efetua a ação de cálculo a partir do uso de um laço do aninhado a função de decisão `cond` para a realização da operação ao estilo iterativo.

```
>>> (defun fatorial1 (N)
      (cond
        ((< N 0) (format t "Erro~%"))
        ((>= N 0) (do (
                        (I 1 (+ I 1))
                        (FAT 1 (* FAT I)))
                      ((> I N) FAT)))))
```

Inicialmente a função `fatorial1` por meio da função `cond` verifica se o valor fornecido a variável local **N** é menor que zero com a condição (`< N 0`), sendo o valor menor que **0** (zero) ocorre a apresentação da mensagem **Erro**. No entanto, se o valor de **N** for maior ou igual a **0** (zero) com a condição (`>= N 0`) ocorre a efetivação do cálculo da fatorial, pois **N** determina a quantidade de vezes que o laço do será processado até que o valor da variável **I** seja maior que o valor da variável **N** definido em (`> I N`). Quando a variável **I** tiver valor maior que a variável **N** ocorrerá o encerramento do laço e o retorno do valor da variável **FAT** definida em (`(> I N) FAT`). O laço, a partir da inicialização da variável **I** em (`I 1 (+ I 1)`), faz seu incremento de **1** em **1** e dá continuidade ao cálculo da fatorial junto a variável **FAT**



iniciada em **1** e multiplicada sucessivamente com os valores de **FAT** e **I** como indicado em `(FAT 1 (* FAT I))`).

Para verificar a funcionalidade da função `fatorial1` execute a instrução.

```
>>> (fatorial1 5)
120
```

Funções que utilizam laços iterativos para realizar operações de cálculo podem ser codificadas com o uso da ação de recursividade, ou seja, podem realizar chamadas a si mesmas, desde que controladas por certa condição.

O uso de recursividade proporciona a escrita de um código elegante com alto grau de abstração. Mas para que uma função recursiva seja bem definida, é importante levar em consideração duas propriedades (LIPSCHUTZ & LIPSON, 2013, p. 51):

- a função recursiva deve possuir argumentos chamados de *valores bases*, para os quais a função em hipótese alguma faça referência a si mesma;
- cada vez que a função se retira a si mesma, seu resultado deve ficar mais próximo à resposta esperada a partir do parâmetro base utilizado.

Se essas propriedades não forem consideradas, haverá o risco de se implementar funções circulares, as quais produzem chamadas infinitas a si mesmas, consumindo recursos de memória e não produzindo nenhuma resposta.

Uma função recursiva não pode chamar a si mesma indiscriminadamente, pois, se assim fizer, entrara em um processo infinito de ação. Portanto, é necessário que a função recursiva tenha a definição de uma condição de encerramento, que é a solução mais simples e menor para o problema avaliado.

Veja a seguir a função `fatorial2` com uso do efeito de recursividade utilizando-se um estilo de indentação mais tradicional que as formas anteriormente indicadas neste trabalho.

```
>>> (defun fatorial2 (N)
      (if (< N 0)
          (format t "Erro~%")
          (if (= N 0)
              1
              (* N (fatorial2 (- N 1)))))))
```

A função `fatorial2` é iniciada com a função `if` verificando inicialmente se a variável local **N** é menor que **0** (zero) com a condição (`< N 0`), sendo é apresentada a mensagem indicado **Erro** na operação. Caso contrário, a segunda função `if` verifica se **N** é igual a **0** (zero) (`<= N 0`), sendo esta condição verdadeira ocorre o retorno do valor **1** e a execução da função é encerrada, caso contrário a função chama a si mesma atribuindo um valor menor em **1** a cada execução de (`- N 1`) até chegar a **0** (zero). A cada retorno sucessivo efetuado o valor obtido anteriormente é multiplicado até que o último retorno ocorra com o valor desejado.

A recursividade indicada caracteriza-se por ser processada a partir de ação conhecida como **recursividade direta**. Apesar de funcionar esta forma faz com que haja grande consumo de memória limitando a ação da função impedindo que se obtenha o cálculo desejado.

Para realizar um teste execute a instrução com a chamada da função `fatorial1` no programa CLISP.

```
[n]> (fatorial1 2486)
```

Será apresentado o resultado da operação. Na sequência execute a instrução com a chamada da função `fatorial2` no programa CLISP e observe a resposta apresentada indicando que a memória ficou esgotada.

```
[n]> (fatorial2 2486)
```

```
*** - Program stack overflow. RESET
```

Para realizar um teste execute a instrução com a chamada da função `fatorial1` no programa SBCL.

```
* (fatorial1 64934)
```

Será apresentado o resultado da operação. Na sequência execute a instrução com a chamada da função `fatorial2` no programa SBCL e observe que ocorrerá um erro na execução do programa e este será encerrado sumariamente.

```
* (fatorial2 64934)
```

O uso de recursividade é muitas vezes mais desejável que o uso de laços e para neutralizar o efeito de sobrecarga de memória que pode ocasionar a apresentação de uma mensagem de erro ou mesmo a interrupção de funcionamento do programa usa-se o estilo chamado **recursividade indireta**.

A recursividade indireta ocorre quando uma função chama outra função e a função chamada retorna seu resultado a função chamadora criando um ciclo de chamadas encadeadas atuando sobre a recursividade. Esta forma de operação é conhecida como **recursividade por cauda**.

A recursividade em cauda garante menor uso de memória, por não fazer uso da pilha de memória. O resultado final da chamada recursiva em cauda é o resultado da própria função.

O uso de função recursiva em cauda deve ser feito com o uso da definição de argumento opcional sinalizado pelo símbolo `&optional` que permite indicar que o argumento a sua frente é opcional e poderá ser omitido na chamada da função.

Veja a seguir a função `fatorial3` com uso do efeito de recursividade por cauda.

```
>>> (defun fatorial3 (N &optional (BASE 1))
      (if (< N 0)
          (format t "Erro~%")
          (if (and (>= N 0) (< N 2))
              BASE
              (fatorial3 (- N 1) (* BASE N)))))
```

A partir da definição da função `fatorial3` é possível fazer uso de valores acima de **2485** e **64933** respectivamente com os programas CLISP e SBCL sem ter a apresentação de alguma mensagem de erro ou mesmo a interrupção do funcionamento do programa.

Assim sendo, execute a instrução seguinte e observe o resultado apresentado em ambos os programas interpretadores CL.

```
>>> (fatorial3 65000)
```

O resultado apresentado é gerado a partir da execução de uma função recursiva, forma mais elegante. Na função `fatorial3` se o valor da variável local **N** for menor que **0** (zero) ocorrerá a apresentação da mensagem **Erro**. Se o valor de **N** for maior ou igual a **0** e menor que **2** será retornado o valor estabelecido para o argumento opcional **BASE** definido inicialmente com valor **1** a partir da definição de argumento `&optional (BASE 1)`. O efeito recursivo ocorre com a execução da chamada da função `fatorial3` com o argumento obrigatório `(- N 1)` operando com o argumento opcional `(* BASE N)`.

Após a execução da função `fatorial3` com o valor **5** para a variável **N** e nenhum valor para o argumento opcional da variável **BASE** (primeira ação) é realizada uma chamada recursiva, segunda ação, definindo para a variável **N** o valor **4** devido a ação `(- N 1)` e como segundo argumento o valor **1** da **BASE** multiplicado pelo valor anterior **5** de **N** devido a ação `(* BASE N)` gerando o valor **5** para **BASE**. Assim sendo, ocorre nesta etapa a chamada da terceira ação recursiva da função `fatorial3` com os valores **4** e **5**.

A terceira chamada recursiva da função é realizada com o valor **3** para **N** e **BASE** com o valor **5** multiplicado pelo valor anterior de **N** com **4** gerando o valor **20**. Como o valor de **N** não é menor que **2** a operação continua.

A quarta chamada recursiva da função é realizada com o valor **2** para **N** e **BASE** com o valor **20** multiplicado pelo valor anterior de **N** com **3** gerando o valor **60**. Como o valor de **N** não é menor que **2** a operação continua.

A quinta chamada recursiva da função é realizada com o valor **1** para **N** e da **BASE** com o valor **60** multiplicado pelo valor anterior de **N** com **2** gerando o valor **120**. Como o valor de **N** agora é menor que **2** não ocorre nova recursão a operação é encerrada retornando o último valor de **BASE** que neste momento é **120**. A partir deste instante toda a sequência recursiva é em sentido oposto encerrada retornando o último valor de **BASE**.

O acompanhamento da execução de uma função pode ser rastreado com o uso da função `trace` que opera a partir da sintaxe.

**(trace <função>)**

Onde o argumento ***função*** é o nome da função que será rastreada quando esta for colocada em uso.

Observe a ativação do modo depuração no programa CLISP para a função recursiva `fatorial3`.

```
[n]> (trace fatorial3)
;; Tracing function FATORIAL3.
(FATORIAL3)
```

Observe a ativação do modo depuração no programa SBCL para a função recursiva `fatorial3`.

```
[n]> (trace fatorial3)
(FATORIAL3)
```

A partir da ativação do modo depuração e a indicação da função a ser rastreada, basta fazer uso da função de forma normal e ver o resultado apresentado.

Veja o resultado apresentado da função `fatorial3` no programa `CLISP`.

```
[n]> (fatorial3 3)
1. Trace: (FATORIAL2 '3)
2. Trace: (FATORIAL2 '2)
3. Trace: (FATORIAL2 '1)
4. Trace: (FATORIAL2 '0)
4. Trace: FATORIAL2 ==> 1
3. Trace: FATORIAL2 ==> 1
2. Trace: FATORIAL2 ==> 2
1. Trace: FATORIAL2 ==> 6
6
```

Veja o resultado apresentado da função `fatorial3` no programa `SBCL`.

```
* (fatorial3 3)
0: (FATORIAL3 3)
  1: (FATORIAL3 2 3)
    2: (FATORIAL3 1 6)
      2: FATORIAL3 returned 6
    1: FATORIAL3 returned 6
  0: FATORIAL3 returned 6
6
```

O uso do modo de depuração `trace` é um recurso que auxilia o acompanhamento e entendimento da execução de funções, principalmente com o uso de funções recursivas.

**OBS:** *Veja que a apresentação do modo de depuração no programa `SBCL` possui visual mais fácil de ser observado e analisado em relação a apresentação realizada pelo programa `CLISP`.*

A figura 3.1 apresenta um diagrama esquemático e comparativo de como funções de recursividade direta (simples) e indireta (cauda) são executadas na memória.

Note que pela imagem indicada na figura 3.1 observa-se que a recursividade simples ocupa mais memória que a recursividade por cauda.

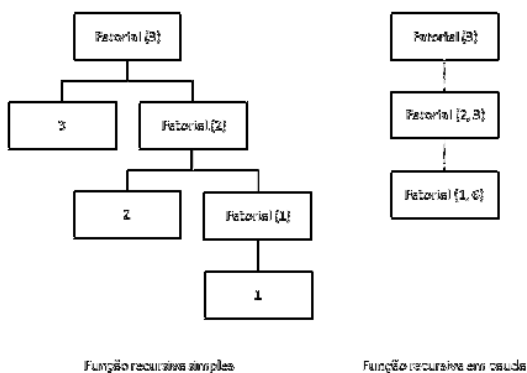


Figura 3.1 - Função simples e cauda: comportamento em memória.

A partir da figura 3.1, é possível ter ideia de como se comportam os tipos de recursividade simples e em cauda. O tempo de execução de ambas as funções é idêntico, uma vez que a função recursiva chama a si mesma apenas uma vez por instância.

Para desativar o modo de depuração basta fazer uso da função `untrace` que utiliza a mesma sintaxe que a função `trace`.

O efeito de economia de memória com recursividade por cauda é perceptível em funções que efetuam mais de uma chamada a si mesmas no mesmo escopo de operação como é o caso do cálculo do termo da série de Fibonacci.

Assim sendo, considere três versões de uma função para cálculo de Fibonacci: forma iterativa, recursiva simples e recursiva por cauda. A sequência de Fibonacci é formada por uma ordem numérica infinita onde **1** é o valor de seu primeiro e segundo termos, e os demais termos são formados pela soma de seus dois termos antecessores: **1, 1, 2, 3, 5, 8, 13** e assim por diante.

A função `fibonacci1` a seguir efetua o cálculo do termo da sequência indicado em **N** utilizando-se para o cálculo ação com laço iterativo.

```
>>> (defun fibonacci1 (N)
      (cond
        ((<= N 0) (format t "Erro~%"))
        ((>= N 1) (do (
                        (I N (- I 1))
                        (ANTERIOR 0 ATUAL)
                        (ATUAL 1 (+ ANTERIOR ATUAL))))
                    ((= I 0) ANTERIOR)))))
```

A função `fibonacci2` a seguir efetua o cálculo do termo da sequência indicado em `N` utilizando-se para o cálculo recursividade simples.

```
>>> (defun fibonacci2 (N)
      (cond
        ((<= N 0) (format t "Erro~%"))
        ((or (= N 1) (= N 2)) 1)
        ((+ (fibonacci2 (- N 1)) (fibonacci2 (- N 2))))))
```

A função `fibonacci3` a seguir efetua o cálculo do termo da sequência indicado em `N` utilizando-se para o cálculo recursividade de cauda.

```
>>> (defun fibonacci3 (N &optional (ANTERIOR 0) (ATUAL 1))
      (cond
        ((<= N 0) (format t "Erro~%"))
        ((or (= N 1) (= N 2)) (+ ATUAL ANTERIOR))
        ((fibonacci3 (- N 1) ATUAL (+ ANTERIOR ATUAL)))))
```

Uma maneira de checar a eficiência de execução de funções é fazer uso da macro `time` que apresenta informações sobre o tempo de execução de um **form**. A precisão da informação apresentada depende de diversos fatores operacionalizados pelo sistema operacional e pela implementação da linguagem em uso.

Tenha cuidado com o valor definido para o termo do cálculo que poderá ser excessivamente demorado para a macro `fibonacci2`, principalmente no programa CLISP. Execute as instruções a seguir e veja os resultados apresentados que poderão ser um pouco diferentes em seu sistema, dos aqui indicados.

Veja os resultados apresentados a partir da execução da macro `time` em conjunto com as funções `fibonacci2` e `fibonacci3` no programa CLISP.

```
[n]> (time (fibonacci2 34))
```

```
Real time: 13.824161 sec.
```

```
Run time: 13.8125 sec.
```

```
Space: 0 Bytes
```

```
5702887
```

```
[n]> (time (fibonacci3 34))
```

```
Real time: 0.0 sec.
```

```
Run time: 0.0 sec.
```

```
Space: 0 Bytes
```

```
5702887
```

Veja os resultados apresentados a partir da execução da macro `time` em conjunto com as funções `fibonacci2` e `fibonacci3` no programa `SBCL`.

```
* (time (fibonacci2 34))
```

```
Evaluation took:
```

```
0.514 seconds of real time
```

```
0.515625 seconds of total run time (0.515625 user, 0.000000 system)
```

```
100.39% CPU
```

```
1,228,877,671 processor cycles
```

```
0 bytes consed
```

```
5702887
```

```
* (time (fibonacci3 34))
```

```
Evaluation took:
```

```
0.000 seconds of real time
```

```
0.000000 seconds of total run time (0.000000 user, 0.000000 system)
```

```
100.00% CPU
```

```
12,520 processor cycles
```

```
0 bytes consed
```

```
5702887
```



Note que o tempo de execução da função `fibonnaci2` com a indicação do termo **34** em CLISP dura a média de **13,8** segundos e no programa SBCL o mesmo valor dura a média de **0,5** segundo. Já a função `fibonnaci3` com o termo **34** dura em ambos os programas o tempo de **0,0** segundo.

Operações baseadas em recursão do tipo simples podem chegar a horas de processamento antes de fornecerem uma resposta aceitável. Por esta razão optar por recursão de cauda é mais vantajoso e adequado.



## 4

## Recursos avançados

---

*Neste capítulo são apresentados os tipos de dados matrizes, tabelas de símbolos (hash) e estruturas. É demonstrado o uso de ações de entrada de dados de forma interativa e de novas possibilidades de saída de dados. É indicado o uso de variáveis locais e a definição de macros. Destaque é dado ao uso de funcionalidades de tempo. É também apresentado o uso de funções anônimas.*

### 4.1 TIPO DE DADO MATRIZ

---

Matrizes (**arrays**) são tipos de dados que podem ser definidos com uma ou múltiplas dimensões. Matriz é uma estrutura de dados contigua disposta na memória tendo seu endereço mais baixo como primeiro elemento e o endereço mais alto como último elemento, cada posição de armazenamento de certo elemento em uma matriz é chamado de **slot**. Os **slots** de uma matriz são gerenciados a partir da manipulação de índices formados por valores inteiros e positivos iniciados a partir de zero.

Uma matriz de uma dimensão é normalmente referenciada como vetor, com duas dimensões normalmente é referenciada como tabela. A partir de três dimensões usa-se como referência o termo matrizes multidimensionais.

A definição de matriz é realizada com o uso da função `make-array` a partir da sintaxe.

```
(make-array '(<tamanho1> [<tamanho2> [<tamanhoN>]] [<:chaves>]))
```

Onde, a indicação **tamanho1** refere-se a definição da quantidade de elementos como linhas para uma matriz de uma dimensão (vetor), **tamanho2** refere-se a definição da quantidade de elementos como colunas para uma matriz de duas

dimensões (tabela), **tamanhoN** refere-se a definição da quantidade de elementos como páginas para uma matriz de três dimensões (multidimensional) e assim por diante. O argumento opcional **chaves** refere-se a um conjunto de opções adicionais que podem ser estabelecidas para a definição de matrizes.

O conjunto de chaves que podem ser definidas na criação de matrizes são:

- `:element-type` - chave que especifica um tipo de dado para os elementos de uma matriz, o valor padrão é T especificando qualquer tipo de dado;
- `:initial-element` - chave que define valores iniciais para todos os elementos de uma matriz;
- `:initial-contents` - chave que define os valores iniciais de partes dos elementos que formam uma matriz;
- `:adjustable` - chave que auxilia a criação de matrizes redimensionáveis cuja memória subjacente pode ser automaticamente redimensionada, tendo como seu último elemento padrão o valor NIL. Esta chave é usada em conjunto com a chave `:fill-pointer`;
- `:fill-pointer` - chave que monitora o número de elementos armazenados em uma matriz redimensionável. Esta chave deve ser usada obrigatoriamente em conjunto com a chave `:adjustable`;
- `:displaced-to` - chave que auxilia a criação de matrizes com elementos compartilhados. Ambas as matrizes devem ter o mesmo tipo de elemento. Esta chave não é usada com as chaves `:initial-element` ou `:initial-contents`;
- `:displaced-index-offset` - chave que fornece o deslocamento do índice para uma matriz mapeada a partir de outra matriz a ela compartilhada. Para esta chave ser operada é preciso estar em uso a chave `:displaced-to`.

A função `make-array` para ser usada deve ser operada com as funções `setf` para criar a matriz na memória e `aref` para acessar os elementos da matriz, ambas já usadas.

Como exemplo, considere a definição de uma matriz de uma dimensão chamada **\*MATRIZ-1D\*** que armazene os quadrados dos valores inteiros de **10** a **20**. Atente para as etapas apresentadas.

Execute a instrução seguinte no programa CLISP.

```
[n]> (setf *MATRIZ-1D* (make-array '(10)))
#(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
```

Execute a instrução seguinte no programa SBCL.

```
* (defvar *MATRIZ-1D*)
*MATRIZ-1D*

* (setf *MATRIZ-1D* (make-array '(10)))
#(0 0 0 0 0 0 0 0 0 0)
```

A partir da definição da matriz **\*MATRIZ-1D\*** em memória é necessário definir as instruções que farão o cálculo dos quadrados de **10** a **20** dentro da matriz.

```
>>> (loop for I from 10 to 19
      do (setf (elt *MATRIZ-1D* (- I 10)) (expt I 2)))
NIL

>>> *MATRIZ-1D*
#(100 121 144 169 196 225 256 289 324 361)
```

No laço estabelecido a função `elt` (anteriormente usada para acessar determinado caracteres de uma cadeia) indica a matriz **\*MATRIZ-1D\*** a posição de acesso aos **slots** da matriz por meio da função `(- I 10)` para que a primeira contagem do laço em **10** seja subtraída e acesse o primeiro **slot** da matriz, ou seja, a posição zero. Isso é repetido até chegar ao valor **19** que dará acesso ao **slot 9**.

A princípio a estrutura de uma matriz e de uma lista diferenciam-se pelo uso do símbolo `'` aspas simples nas listas e `#` tralha nas matrizes, mas essa diferenciação é estética. Internamente a diferença entre listas e matrizes são mais acentuadas.

Uma lista é constituída de células `cons` e `nil`, sendo uma estrutura dinâmica de dados, uma vez que pode receber elementos sem se preocupar de antemão com seu tamanho. Para acessar, por exemplo, o último elemento é necessário percorrer sequencialmente a lista toda. Adicionar e remover certo elemento na frente da lista é uma tarefa barata, mas a adição ou remoção de elementos em outras posições

são mais caras e por conseguinte mais lentas. As listas podem sofrer sobrecarga de espaço na memória, pois cada elemento é armazenado em uma célula cons.

Uma matriz é uma estrutura de dados estática formada por posições numeradas (**slots**) que apesar de permitir que seu tamanho seja ajustado durante o uso exige que os elementos existentes sejam recopiados para o novo tamanho, demandando mais tempo de processamento. Adicionar elementos é uma ação que se torna caro se a matriz para receber uma nova entrada necessita ter seu tamanho reajustado o que pode sobrecarregar o espaço de memória. Acessar elementos de uma matriz é muito barato uma vez que o acesso é definido a partir do índice de posição do **slot** desejado.

Considerando redefinir a matriz **\*MATRIZ-1D\*** de **10** para **15** elementos é necessário utilizar a função `adjust-array` que possui estrutura de uso idêntica a estrutura da função `make-array`, devendo esta função ser usada em conjunto com a função `setf`.

Execute a instrução seguinte no programa CLISP.

```
[n]> (setf *MATRIZ-1D* (adjust-array *MATRIZ-1D* '(15)))
#(100 121 144 169 196 225 256 289 324 361 NIL NIL NIL NIL NIL)
```

Execute a instrução seguinte no programa SBCL.

```
* (setf *MATRIZ-1D* (adjust-array *MATRIZ-1D* '(15)))
#(100 121 144 169 196 225 256 289 324 361 0 0 0 0 0)
```

A partir da definição do novo tamanho da matriz **\*MATRIZ-1D\*** é necessário proceder ao cálculo anteriores nas posições seguintes. Neste caso execute as instruções seguintes.

```
>>> (loop for I from 20 to 24
      do (setf (elt *MATRIZ-1D* (- I 10)) (expt I 2)))
NIL
```

```
>>> *MATRIZ-1D*
#(100 121 144 169 196 225 256 289 324 361 400 441 484 529 576)
```

A partir da estrutura **\*MATRIZ-1D\*** observe a instrução seguinte que apresenta os elementos da matriz e suas posições.

```
>>> (loop for I from 0 to 14
      do (format t "~a~%" I (elt *MATRIZ-1D* I)))
*MATRIZ-1D*[ 0] = 100
*MATRIZ-1D*[ 1] = 121
*MATRIZ-1D*[ 2] = 144
*MATRIZ-1D*[ 3] = 169
*MATRIZ-1D*[ 4] = 196
*MATRIZ-1D*[ 5] = 225
*MATRIZ-1D*[ 6] = 256
*MATRIZ-1D*[ 7] = 289
*MATRIZ-1D*[ 8] = 324
*MATRIZ-1D*[ 9] = 361
*MATRIZ-1D*[10] = 400
*MATRIZ-1D*[11] = 441
*MATRIZ-1D*[12] = 484
*MATRIZ-1D*[13] = 529
*MATRIZ-1D*[14] = 576
NIL
```

Como indicado existem algumas chaves que auxiliam a inicialização de matrizes, entre as chaves existentes a chave `:initial-element` em particular permite iniciar uma matriz com determinados valores. Atente para os exemplos seguintes que mostram como inicializar os elementos de uma matriz com a função `make-array`.

```
>>> (defvar *X1-1D*)
*X1-1D*

>>> (setf *X1-1D* (make-array '(5) :initial-element 'a))
#(A A A A A)

>>> (defvar *X2-1D*)
*X2-1D*
```

```
>>> (setf *X2-1D* (make-array '(5)
      :element-type 'character :initial-element #\a))
"aaaaa"
```

```
>>> (defvar *X3-1D*)
*X3-1D*
```

```
>>> (setf *X3-1D* (make-array '(5)
      :element-type 'bit :initial-element 1))
#*11111
```

A partir de uma visão básica sobre a definição de matrizes com uma dimensão veja um exemplo para a definição de uma matriz de duas dimensões. Assim sendo, considere definir uma matriz com três linhas e quatro colunas inicializada a partir de uma lista com todas as posições com valor 1.

```
>>> (defvar *MATRIZ-2D*)
*MATRIZ-2D*

>>> (setf *MATRIZ-2D* (make-array '(3 4) :initial-element '1))
#2A((1 1 1 1) (1 1 1 1) (1 1 1 1))
```

Veja que a **\*MATRIZ-2D\*** é formada por um conjunto de três listas (as linhas), onde cada lista é inicializada com quatro posições (colunas) indicadas com a definição do valor 1. Observe a apresentação dos elementos e as posições da matriz.

```
>>> (loop for I from 0 to 2 do
      (loop for J from 0 to 3
        do (format t
          "~*MATRIZ-2D*[~d,~d] = ~a~%" I J (aref *MATRIZ-2D* I J))))
*MATRIZ-2D*[0,0] = 1
*MATRIZ-2D*[0,1] = 1
*MATRIZ-2D*[0,2] = 1
*MATRIZ-2D*[0,3] = 1
*MATRIZ-2D*[1,0] = 1
*MATRIZ-2D*[1,1] = 1
```



```

*MATRIZ-2D*[1,2] = 1
*MATRIZ-2D*[1,3] = 1
*MATRIZ-2D*[2,0] = 1
*MATRIZ-2D*[2,1] = 1
*MATRIZ-2D*[2,2] = 1
*MATRIZ-2D*[2,3] = 1
NIL

```

Atente para o uso da função `aref` ao invés do uso da função `elt`. Esta mudança foi definida devido ao fato da função `elt` não permitir o uso de mais de dois argumentos em seu escopo de operação. Devido a esta questão talvez seja mais vantajoso usar a função `aref`.

A partir dos valores existentes em **\*MATRIZ-2D\*** definir para cada valor da matriz a soma deste com o valor **2** de modo que a matriz tenha em suas posições o valor **3**. Observe a instrução seguinte.

```

>>> (dotimes (I 3)
      (dotimes (J 4)
        (setf (aref *MATRIZ-2D* I J) (+ (aref *MATRIZ-2D* I J) 2))))
NIL

```

```

>>> (loop for I from 0 to 2
      do (loop for J from 0 to 3
        do (format t
          "~*MATRIZ-2D*[,~d] = ~a~%" I J (aref *MATRIZ-2D* I J))))
*MATRIZ-2D*[0,0] = 3
*MATRIZ-2D*[0,1] = 3
*MATRIZ-2D*[0,2] = 3
*MATRIZ-2D*[0,3] = 3
*MATRIZ-2D*[1,0] = 3
*MATRIZ-2D*[1,1] = 3
*MATRIZ-2D*[1,2] = 3
*MATRIZ-2D*[1,3] = 3
*MATRIZ-2D*[2,0] = 3
*MATRIZ-2D*[2,1] = 3

```

```
*MATRIZ-2D*[2,2] = 3
```

```
*MATRIZ-2D*[2,3] = 3
```

```
NIL
```

Quando se tem matrizes definidas em memória é possível verificar a quantidade de elementos que a matriz armazena a partir do uso da função `array-dimensions` que possui sintaxe semelhante a forma simplificada da função `make-array`. Observe em seguida as instruções que permitem apresentar os valores máximos existente nas dimensões das matrizes em uso.

```
>>> (array-dimensions *MATRIZ-1D*)
```

```
(15)
```

```
>>> (array-dimensions *MATRIZ-2D*)
```

```
(3 4)
```

Um recurso que pode ser aplicado é o uso da função `array-total-size` que apresenta o total de elementos de uma matriz independentemente da dimensão definida. A função `array-total-size` possui sintaxe semelhante a forma simplificada da função `make-array`. Observe em seguida as instruções que permitem apresentar a quantidade de elementos das matrizes em uso.

```
>>> (array-total-size *MATRIZ-1D*)
```

```
15
```

```
>>> (array-total-size *MATRIZ-2D*)
```

```
12
```

A quantidade de dimensões de uma matriz pode ser obtida por meio do uso da função `array-rank` que possui sintaxe semelhante a forma simplificada da função `make-array`. Observe em seguida as instruções que permitem apresentar a quantidade de dimensões das matrizes em uso.

```
>>> (array-rank *MATRIZ-1D*)
```

```
1
```

```
>>> (array-rank *MATRIZ-2D*)
```

```
2
```

Cabe apontar que a definição de matrizes é realizada como variáveis globais. Variáveis globais são criadas pelas funções `defvar` e `sef` (que particularmente mostra mensagem de advertência no programa SBCL). Mas, além dessas funções há outra forma de definir variáveis globais pela função `defparameter`.

As funções `defparameter` e `defvar` são semelhantes e realizam as mesmas operações, tendo como diferença o fato de que a função `defparameter` exige a inicialização de um valor associado a variável global. Seu uso não indica nenhuma mensagem de advertência em nenhum dos interpretadores CL.

Observe em seguida as instruções para definição de variável global como matriz a partir do uso da função `defparameter`.

```
>>> (defparameter *MATRIZ-X1* #(1 2 3 4 5 6))
>>> *MATRIZ-X1*

>>> *MATRIZ-X1*
#(1 2 3 4 5 6)

>>> (defparameter *MATRIZ-X2* #2A((1 2 3) (4 5 6)))
*MATRIZ-X2*

>>> *MATRIZ-X2*
#2A((1 2 3) (4 5 6))
```

A partir de uma visão básica da definição e da especificação de algumas operações em matrizes é pertinente apresentar a funcionalidade das demais chaves de definição.

Considere definir uma matriz que seja inicializada com dados de um tipo específico. Veja a seguir os exemplos para criação de uma matriz de bits inicializada com valor 1 e uma matriz de caracteres inicializada com a letra “a”.

```
>>> (defvar *MAT-BIT*)
*MAT-BIT*
```

```
>>> (setf *MAT_BIT* (make-array '(5)
      :element-type 'bit
      :initial-element 1)))
```

```
#*11111
```

```
>>> (defvar *MAT-CHAR*)
```

```
*MAT-CHAR*
```

```
>>> (setf *MAT_CHAR* (make-array '(5)
      :element-type 'character
      :initial-element #\a)))
```

```
"aaaaa"
```

A chave `:initial-element` é usada para inicializar elementos de uma matriz de qualquer dimensão, desde que todos os elementos tenham o mesmo valor. Para matrizes multidimensionais essa chave pode não atender.

Imagine a criação de uma matriz com duas linhas e três colunas, contendo os valores da primeira linha como **1** e os elementos da segunda linha como **2**. Observe as instruções a seguir.

```
>>> (defvar *2X3*)
```

```
*2X3*
```

```
>>> (setf *2X3* (make-array '(2 3)
      :initial-contents '((1 1 1) (2 2 2)))))
```

```
#2A((1 1 1) (2 2 2))
```

O ajuste do tamanho de uma matriz após sua definição sem a necessidade de usar a função `adjust-array` pode ser definido previamente com a chave `:adjustable` que para surtir o efeito necessita ser operacionalizada em conjunto com a chave `:fill-pointer` que monitora o número de elementos armazenados em uma matriz redimensionável.

Considerando a definição de uma matriz chamada **\*MAT-ADJ\*** com a capacidade de armazenar inicialmente três elementos e que possa essa ser ajustada para mais elementos de forma automática sem o uso da função `adjust-array` deve ser definida a partir da instrução.

```
>>> (defvar *MAT-ADJ*)
*MAT-ADJ*

>>> (setf *MAT-ADJ* (make-array '(3)
                                :adjustable t
                                :fill-pointer 0
                                :initial-element 0))
#(0 0 0)
```

A partir da criação da matriz com a ativação da chave `:adjustable` com valor `T` de verdadeiro é possível, quando necessário, executar ações que ampliem o número de posições da matriz.

Assim, sendo para efetuar a entrada de dados na matriz **\*MAT-ADJ\*** execute as instruções seguintes a partir do uso da função `vector-push` que após inserir o conteúdo dentro da matriz retorna o valor da posição do *slot* usado para o armazenamento.

```
>>> (vector-push '1 *MAT-ADJ*)
0

>>> (vector-push '2 *MAT-ADJ*)
1

>>> (vector-push '3 *MAT-ADJ*)
2

>>> *MAT-ADJ*
#(1 2 3)
```

Observe que foram inseridos na matriz três valores e se realizada uma nova tentativa de entrada de um quarto valor com a função `vector-push` ocorrerá um erro na entrada indicando o valor `NIL`, ou seja, a entrada não foi efetivada. Veja esta ocorrência.

```
>>> (vector-push '4 *MAT-ADJ*)
NIL
```

Para poder realizar a entrada de mais um elemento em uma matriz redimensionável de forma automática use a função `vector-push-extend` ao invés da função `vector-push`. Veja a próxima instrução.

```
>>> (vector-push-extend '4 *MAT-ADJ*)  
3
```

```
>>> *MAT-ADJ*  
#(1 2 3 4)
```

Observe que as funções `vector-push-extend` e `vector-push` realizam a entrada de valores no final da matriz.

Para retirar elementos do final de uma matriz use a função `vector-pop`. Observe as instruções seguintes.

```
>>> (vector-pop *MAT-ADJ*)  
4
```

```
>>> (vector-pop *MAT-ADJ*)  
3
```

```
>>> *MAT-ADJ*  
#(1 2)
```

Outra ação possível de ser realizada é a de definir matrizes compartilhadas a partir do uso da chave `:displaced-to`.

Considere a definição de duas matrizes compartilhadas, sendo a **\*MAT-DIS-1\*** e a matriz **\*MAT-DIS-2\*** que quando os dados da matriz **\*MAT-DIS-1\*** forem alterados serão refletidos na matriz **\*MAT-DIS-2\***.

```
>>> (defvar *MAT-DIS-1*)  
*MAT-DIS-1*
```

```
>>> (defvar *MAT-DIS-2*)  
*MAT-DIS-2*
```

```
>>> (setf *MAT-DIS-1* (make-array '(3) :initial-element 0))
#(0 0 0)

>>> (setf *MAT-DIS-2* (make-array 3 :displaced-to *MAT-DIS-1*))
#(0 0 0)

>>> *MAT-DIS-1*
#(0 0 0)

>>> *MAT-DIS-2*
#(0 0 0)

>>> (setf (aref *MAT-DIS-1* 0) 4)
4

>>> *MAT-DIS-1*
#(4 0 0)

>>> *MAT-DIS-2*
#(4 0 0)
```

Além de vincular matrizes a função `:displaced-to` permite alterar a estrutura de dimensão de uma matriz. Assim sendo, considere a definição de uma matriz de uma dimensão com quatro elementos e sua redefinição como sendo uma matriz de duas dimensões com duas linhas e duas colunas.

```
>>> (defvar *MAT-TRANS-1*)
*MAT-TRANS-1*

>>> (defvar *MAT-TRANS-2*)
*MAT-TRANS-1*

>>> (setf *MAT-TRANS-1* (make-array '(4)
                                   :initial-contents '(1 2 3 4)))
#(1 2 3 4)
```

```
>>> (setf *MAT-TRANS-2* (make-array '(2 2) :displaced-to *MAT-TRANS-1*))  
#2A((1 2) (3 4))
```

```
>>> *MAT-TRANS-2*  
#2A((1 2) (3 4))
```

Para transformar uma matriz de duas dimensões em uma dimensão basta fazer o uso de ação inversa. Teste a instrução.

```
>>> (defvar *MAT-TRANS-3*)  
*MAT-TRANS-3*
```

```
>>> (setf *MAT-TRANS-3* (make-array '(4) :displaced-to *MAT-TRANS-2*))  
#(1 2 3 4)
```

```
>>> *MAT-TRANS-3*  
#(1 2 3 4)
```

A transformação de matrizes de uma dimensão em duas dimensões ou vice versa é possível uma vez que matrizes unidimensionais são armazenadas na memória como sendo uma matriz de uma dimensão na ordem das linhas existentes.

A definição de matrizes pode ser realizada com a chave `:displaced-index-offset` que para ser usada necessita do uso prévio da chave `:displaced-to`. A chave `:displaced-index-offset` usa um valor inteiro positivo que determina o valor de deslocamento do índice para acesso a matriz copiada a partir do deslocamento matriz principal.

Observe a seguir as instruções para criação da matriz principal **\*MAT-IND-1\*** com os valores de **1** até **5** e da matriz **\*MAT-IND-2\*** que conterá os três últimos valores da matriz **\*MAT-IND-1\***.

```
>>> (defvar *MAT-IND-1*)  
*MAT-IND-1*
```

```
>>> (setf *MAT-IND-1* (make-array '(5)  
    :initial-contents '(1 2 3 4 5)))  
#(1 2 3 4 5)
```



```
>>> (defvar *MAT-IND-2*)  
*MAT-IND-2*  
  
>>> (setf *MAT-IND-2* (make-array '(3)  
:displaced-to *MAT-IND-1*  
:displaced-index-offset 2))  
#(3 4 5)
```

Note que o trecho `:displaced-index-offset 2` cria a matriz **\*MAT-IND-2\*** a partir da matriz **\*MAT-IND-1\*** considerando todos os elementos da matriz **\*MAT-IND-1\*** a partir do índice **2**.

Quando se utiliza a chave `:displaced-index-offset` as matrizes não são compartilhadas como ocorre com o uso isolado da chave `:displaced-to`.

A partir de uma visão básica sobre a definição e uso essencial de matrizes é importante conhecer algumas funcionalidades operacionais para seu gerenciamento em memória como contar elementos, localizar elementos, remover elementos, substituir elementos, entre outras operações.

Observe as instruções seguintes.

```
>>> (length (vector 1 2 3 4 5))  
5  
  
>>> (length #(1 2 3 4 5))  
5  
  
>>> (count 3 #(1 2 3 4 3 2 1 0 3 6 1 3))  
4  
  
>>> (remove 0 #(0 1 0 2 0 3 0 4 0 5))  
#(1 2 3 4 5)  
  
>>> (substitute 99 3 #(1 2 3 4 3 2 1))  
#(1 2 99 4 99 2 1)
```

```
>>> (find 3 #(1 2 3 4 3 2 1))
```

```
3
```

```
>>> (position 4 #(1 2 3 4 5 4 3 2 1))
```

```
3
```

```
>>> (remove-duplicates #(1 2 3 4 3 5 3 3 6))
```

```
 #(1 2 4 5 3 6)
```

As funções `length`, `remove`, `substitute` e `find` são conhecidas e operam da forma já orientada. A função `vector` define uma matriz de uma dimensão (vetor) cujo tamanho corresponde ao número de elementos estabelecidos. Já a função `count` efetua a contagem do elemento indicado dentro de uma matriz retornando a quantidade de vezes que o elemento se encontra. Caso o elemento não seja encontrado o valor 0 é retornado. A função `position` retorna o valor cardinal do índice em que se encontra o primeiro elemento conforme o elemento indicado para pesquisa. Se nada for localizado é retornado o valor `NIL` e a função `remove-duplicates` retira de uma matriz (ou lista) todos os elementos que se repetem na estrutura.

As funções `count`, `find`, `remove` e `substitute` possuem variações condicionais que podem auxiliar algumas operações de uso, como `count-if`, `find-if`, `remove-if` e `substitute-if`. Assim sendo, observe os exemplos seguintes.

```
>>> (count-if #'evenp #(1 2 3 4 5)) ; quantidade de pares
```

```
2
```

```
>>> (count-if #'oddp #(1 2 3 4 5)) ; quantidade de ímpares
```

```
3
```

```
>>> (remove-if #'oddp #(1 2 3 4 5)) ; remove ímpares
```

```
 #(2 4)
```

```
>>> (remove-if #'evenp #(1 2 3 4 5)) ; remove pares
```

```
 #(1 3 5)
```

```
>>> (find-if #'evenp '(1 4 3 2 5)) ; localiza primeiro par
```

```
4
```

```
>>> (find-if #'oddp '(2 4 3 1 5)) ; localiza primeiro impar
3
```

```
>>> (substitute-if 99 #'oddp '(1 2 3 4 5)) ; substitui impares por 99
(99 2 99 4 99)
```

As funções `count-if`, `find-if`, `remove-if` e `substitute-if` possuem como variações opostas as funções `count-if-not`, `find-if-not`, `remove-if-not` e `substitute-if-not`.

Uma ação que também pode ser aplicada sobre matrizes é o efeito de junção com a função `merge`. Assim sendo, observe as instruções seguintes.

```
>>> (merge 'vector '(1 3 5) '(2 4 6) #'<)
#(1 2 3 4 5 6)
```

```
>>> (merge 'vector #(1 3 5) '(2 4 6) #'=)
#(1 3 5 2 4 6)
```

```
>>> (merge 'vector #(1 3 5) #(2 4 6) #'>)
#(2 4 6 1 3 5)
```

A ação de junção com a função `merge` aceita que as coleções de elementos indicadas para uso sejam tanto matrizes como listas. Anteriormente para mesclar coleções de listas em lista usou-se o tipo `list`. Para mesclar coleções de listas e/ou matrizes em matrizes (**arrays**) usa-se o tipo `vector`.

Há casos em que o número de dimensões de uma matriz não é importante e neste sentido podem ser usadas as funções `row-major-aref` que acessa os elementos de uma matriz multidimensional usando um único índice e `array-total-size` para detectar o tamanho da matriz.

Veja as instruções seguintes.

```
>>> (defparameter *MAT2D* #2A((1 2 3) (4 5 6) (7 8 9)))
*MAT2D*
```

```
>>> (loop for I from 0 below (array-total-size *MAT2D* )
      do (format t "~a " (row-major-aref *MAT2D* I)))
1 2 3 4 5 6 7 8 9
NIL
```

Veja que a função `array-total-size` é usada para delimitar a quantidade de iterações que é realizada sobre todos os elementos da matriz **\*MAT2D\*** acessados pela função `row-major-aref`.

Os dois tipos de dados mais populares são as listas e as matrizes. Essas formas de representação podem ser intercambiadas a partir de ações de coerção (**casting**) transformando listas em matrizes e vice versa a partir da função `coerce`.

Observe os exemplos seguintes mostram a conversão de uma lista em matriz e de uma matriz em lista.

```
>>> (coerce '(A B C D E) 'vector)
#(A B C D E)

>>> (coerce #(A B C D E) 'list)
(A B C D E)
```

No conjunto de funções para a manipulação de listas e de matrizes havendo a eventualidade de uma função não poder ser usada com uma lista ou uma matriz a conversão passa a ser uma ação desejada. Isso faz com que os recursos possam ser amplamente utilizados.

## 4.2 ENTRADA E SAÍDA DE DADOS

---

As operações de saída (exibição) e entrada (leitura) de dados são ações de conversão de dados manipulados pela linguagem na forma textual e vice-versa.

Algumas operações de saída foram apresentadas anteriormente a partir do uso das funções `write` que envia a saída sem pular linha antes da impressão mostrando o retorno do valor em uso, se numérico mostra o valor numérico e se cadeia ou caractere mostra conteúdo entre aspas; `print` que envia a saída pulando uma linha antes da impressão mostrando o retorno do valor em uso, se numérico mostra o valor numérico e se cadeia ou caractere mostra conteúdo entre aspas e `format` que

produz como saída uma estrutura de texto formatada. No entanto, há outras funções de saída que podem ser usadas, destacando-se as principais:

<code>pprint</code>	saída sem indicar retorno e pula linha antes;
<code>prin1</code>	saída com salto de linha ao final;
<code>princ</code>	saída sem salto de linha ao final;
<code>terpri</code>	saída com um linha em branco.

Observe alguns exemplos de uso das principais funções de saída.

```
>>> (prin1 "ABC")
```

```
"ABC"
```

```
"ABC"
```

```
>>> (pprint "ABC")
```

```
<pula linha em branco>
```

```
"ABC"
```

```
>>> (princ "ABC")
```

```
ABC
```

```
"ABC"
```

```
>>> (terpri)
```

```
<pula linha em branco>
```

```
NIL
```

Além das funcionalidades que operam ações de saída há a função `read` que permite realizar a entrada de dados de forma interativa. Observe o exemplo seguinte, no qual dever ser fornecido um valor após a execução da ação. Assim que o valor é fornecido o mesmo é imediatamente apresentado.

```
>>> (write (read))
```

```
abc
```

```
ABC
```

```
ABC
```

A partir das ações de entrada e saída é possível criar funções interativas que proporcionam melhor operação com um usuário. Considere uma função que apresente o resultado da área de uma circunferência.

```
>>> (defun area-circ()
      (princ "Entre o valor do raio: ")
      (finish-output)
      (defparameter raio (read))
      (princ "Area: ")
      (write (* (expt raio 2) pi)))
```

```
>>> (area-circ)
Entre o valor do raio: 3
Area: 28.274333882308139147L0
28.274333882308139147L0
```

Note que na definição da função `area-circ` está sendo usada a função `princ`, que é a função de saída que melhor se ajusta a apresentação da mensagem e a recepção do valor informado para o cálculo.

O uso da função `finish-output` tem por finalidade realizar a finalização da ação de saída junto ao buffer. Esta função em alguns interpretadores LISP pode ser desnecessária. No entanto, deve sempre ser considerado seu uso após o uso de uma função de saída para evitar efeitos colaterais onde a mensagem de saída venha a ser apresentada após a entrada de dados quando deveria, de fato, for indicada antes da operação de entrada.

## 4.3 VARIÁVEIS LOCAIS

---

Até este ponto foram usados nos exemplos apresentados variáveis globais, exceto algumas operações com ações de laço de alto nível que obrigatoriamente utilizam variáveis locais. Apesar de não ter sido utilizado exemplos com variáveis locais esta forma de variáveis é em CL bastante popular. A definição de variáveis locais é realizada, como comentado, com a função `let` a partir da sintaxe.

```
(let ((<var1 [v1r1]>) [<var2 [v1r2]>]) [...] [<varN [v1rN]>]) <ação>)
```

Onde, as indicações **var1**, **var2** e **varN** referem-se a definição das variáveis locais; **v1r1**, **v1r2** e **v1rN** referem-se a definição opcional de valores iniciais para as variáveis locais e **ação** é a indicação das operações realizadas com as variáveis locais.

Observe a instrução seguinte que define a criação de duas variáveis **A** e **B** locais inicializadas, respectivamente, com os valores **1** e **2** e apresenta a soma dos valores definidos.

```
>>> (let ((A 1) (B 2))
      (+ A B))
3
```

A diferença entre variáveis globais e locais é que as variáveis globais são “visíveis” a qualquer momento durante o uso do ambiente de programação. Já as variáveis locais são “visíveis” dentro de certo escopo, ou seja, dentro do escopo operacionalizado dentro do contexto da função `let`.

Considere como demonstração a definição de uma função que solicita a entrada de um valor par e o apresenta. Caso seja fornecido um valor ímpar o pedido de entrada deve ser repetido. Observe as instruções.

```
>>> (defun entra-numero ())
      (format t "Informe um valor par: ")
      (finish-output)
      (let ((valor (read)))
        (if (evenp valor)
            (format t "Valor informado: ~a" valor)
            (entra-numero))))
```

ENTRA-NUMERO

```
>>> (entra-numero)
Informe um valor par: 3
Informe um valor par: 5
Informe um valor par: 2
Valor informado: 2
NIL
```

Ao ser executada a função `entra-numero` e um valor par não foi informado ocorrerá a execução recursiva da função até um valor par ser fornecido.

A função `let` efetua o efeito de amarração ou atamento (***binding***). É a ação de se atribuir valores iniciais a variáveis que possuem validade apenas dentro do escopo do corpo de uma função em uso, fora da função essas variáveis não existem.

Caso venha a existir a definição de uma variável local com o mesmo nome de uma variável global o acesso ocorrerá no contexto de amarração apenas na variável local. Para verificar este efeito considere a definição de variável global e local com o mesmo nome de identificação.

```
>>> (defparameter TESTE 1)
TESTE
```

```
>>> (let ((TESTE 9)) TESTE)
9
```

```
>>> TESTE
1
```

Devido a dificuldade em diferenciar variáveis globais e locais é que se tem o costume de usar a definição de variáveis globais entre os símbolos de asterisco. Uma variável global de nome ***\*X\**** é referenciada com a identificação ***estrela-xis-estrela***, se fosse local seria referenciada apenas por ***X***.

É possível fazer uso das funções `setq` ou `setf` dentro do escopo de definição de certa variável local. Observe como exemplo o armazenamento temporário do resultado da soma a partir de dois valores definidos localmente junto a variável local ***R***.

```
>>> (let ((A 1) (B 2) (R 0))
      (setf R (+ A B)) R)
3
```

No entanto, com a função `let` não é possível junto a definição de variáveis locais fazer com que o valor de uma variável seja usado para inicializar outra variável local. O trecho seguinte acarreta na apresentação de uma mensagem de erro.



```
>>> (let ((A 1) (B (+ A 1)) (R 0))  
      (setf R (+ A B)) R)
```

Não é possível definir para a variável **B** a inicialização do valor **2** a partir da soma do valor da variável **A** com mais **1**. A mensagem de erro estará informando que a variável **A** não se encontra definida. No entanto, se ai invés de usar a função `let` for usada a função `let*` o resultado será diferente.

```
>>> (let* ((A 5) (B (+ A 1)) (R 0))  
        (setf R (+ A B)) R)  
11
```

Com o uso da função `let*` ocorreu a apresentação do resultado **11** sendo a soma do valor **5** da variável **A** com o valor **6** da variável **B** advindo da soma do valor da variável **A** com o valor **1**.

A função `let` não permite criar vínculo de um **form** como valor de inicialização de uma variável, apenas aceita a definição de valores isolados, daí o motivo pelo qual a tentativa da ação `(B (+ A 1))` não gerar o efeito esperado, o que é possível de ser realizado pela `let*`.

## 4.4 MACROS

---

O recurso denominado macro caracteriza-se por ser uma forma de expansão de recursos em CL que permitem estender as funcionalidades existentes da linguagem. É um mecanismo que permite otimizar e customizar operações da linguagem que venham a atender necessidades mais específicas.

Alguns dos recursos usados em CL são definições de macros que se não forem assim indicadas, passam naturalmente como sendo funções, destacando-se, por exemplo, as macros `if`, `when` e `unless`, entre outras tantas.

A definição simplificada de macros é realizada com a macro de definição `defmacro` a partir da estrutura sintática.

```
(defmacro <nome> <ação>)
```

Onde, a indicação **nome** refere-se ao nome da macro que pode ou não ter uma lista de argumentos e **ação** o que a macro deve realizar,

Para um exemplo simples e inicial considere uma macro que efetue uma operação de soma utilizando notação polonesa inversa como ocorre em algumas calculadoras financeiras. Observe as instruções a seguir.

```
>>> (defmacro npr (ARGUMENTOS)
      (reverse ARGUMENTOS))
```

NPR

```
>>> (npr (1 2 +))
```

3

Veja que a macro definida `npr` (Notação Polonesa Reversa) inverte a forma habitual de cálculo permitindo uma nova forma de acesso.

Os dois próximos exemplos caracterizam-se pela definição de duas macros que mostram respectivamente a cabeça e a cauda de uma lista.

```
>>> (defmacro cabeca (LISTA)
      (list 'car LISTA))
```

CABECA

```
>>> (defmacro cauda (LISTA)
      (list 'cdr LISTA))
```

CAUDA

```
>>> (cabeca '(1 2 3 4 5))
```

1

```
>>> (cauda '(1 2 3 4 5))
```

(2 3 4 5)

Os três próximos exemplos demonstram como as macros `se` (`if`), `quando` (`when`) e `a-nao-ser-que` (`unless`) podem estar definidas na linguagem CL.

Observe as instruções a seguir que definem a macro `se`.

```
>>> (defmacro se (CONDICAO ENTAO SENA0)
      `(cond (,CONDICAO ,ENTAO)
              (t ,SENA0)))
SE

>>> (se (evenp 2) (format t "Par") (format t "Impar"))
Par
NIL

>>> (se (evenp 3) (format t "Par") (format t "Impar"))
Impar
NIL
```

Na definição da macro `se` a lista de argumentos está sendo definida a partir de três componentes, sendo: `CONDICAO`, `ENTAO` e `SENA0`. Note que antes de usar cada um dos componentes no corpo da macro define-se o uso de uma vírgula imediatamente antes de cada componente (não pode haver espaço em branco entre a vírgula e o componente), indicando em se tratar de argumentos que possuem valores recebidos externamente. Outro detalhe a ser considerado é o uso do símbolo grave indicado antes da definição da ação que permite tratar o que vem a frente do símbolo como dado e não código.

Observe as instruções a seguir que definem a macro `quando`.

```
>>> (defmacro quando (CONDICAO &rest CORPO)
      `(if ,CONDICAO (progn ,@CORPO)))
QUANDO

>>> (quando (>= 20 18) (format t "Maior de idade"))
Maior de idade
NIL

>>> (quando (>= 2 18) (format t "Maior de idade"))
NIL
```

A macro quando usa junto a definição do argumento **CONDICAO** o marcador &rest que tem por finalidade aceitar qualquer quantidade de argumentos que são atribuídos junto ao argumento **CORPO**.

O uso do operador especial progn junto a função if tem for finalidade avaliar um conjunto de **forms** na ordem em que são definidos retornando prioritariamente o valor do último formulário. Nesta aplicação este recurso define um bloco de operações que serão executadas quando a condição da função if for verdadeira, assemelhando-se a definição de blocos em outras linguagens de programação como feito em C com o uso de chaves ou Pascal com o uso de begin e end, entre outras. Alternativamente a função progn há a função prog1 que retorna de um conjunto o primeiro valor.

Observe os exemplos de uso das funções progn e prog1 respectivamente a seguir.

```
>>> (if (> 4 2)
      (progn
        (print 'Soma)
        (+ 4 2))
      (progn
        (print 'Subtracao)
        (- 4 2)))
```

SOMA

6

```
>>> (if (> 4 2)
      (prog1
        (+ 4 2)
        (print 'Soma))
      (prog1
        (- 4 2)
        (print 'Subtracao)))
```

SOMA

6

Veja que o resultado apresentado para os dois exemplos é o mesmo, mas atente para a ordem de definição da operação considerada no uso das funções `progn` e `prog1`.

Observe as instruções a seguir que definem a macro `a-nao-ser-que`.

```
>>> (defmacro a-nao-ser-que (CONDICAO &rest CORPO)
      `(if (not ,CONDICAO) (progn ,@CORPO)))
A-NAO-SER-QUE

>>> (a-nao-ser-que (= 2 3) (format t "Execute a acao"))
Execute a acao
NIL

>>> (a-nao-ser-que (= 2 3) (format t "Execute a acao"))
Execute a acao
NIL
```

É importante esclarecer que quando certa operação não é precedida dos símbolos `apóstrofo` ou `grave` isto indica que a operação será tratada apenas como código. Se precedida de um símbolo `apóstrofo` isto indica que a operação será tratada apenas como dado e se precedida de um símbolo `grave` contendo seus argumentos precedidos de `vírgula` isto indica que a operação será tratada como dado, permitindo que os argumentos indicados sejam inseridos dentro do código da macro. Observe os detalhes indicados.

```
>>> (defparameter *MAC-V1* 1)
*MAC-V1*

>>> (defparameter *MAC-V2* 4)
*MAC-V2*

>>> (+ *MAC-V1* *MAC-V2*) ; código
5

>>> '(+ *MAC-V1* *MAC-V2*) ; dado
(+ *MAC-V1* *MAC-V2*)
```

```
>>> `(+ ,*MAC-V1* ,*MAC-V2*) ; dado com argumento externo
(+ 1 4)
```

```
>>> (eval `(+ ,*MAC-V1* ,*MAC-V2*))
5
```

A partir do conjunto de detalhes apresentados observe em seguida a definição da macro enquanto que opera um laço do tipo pré-teste com fluxo de ação para condição verdadeira. Observe o código seguinte.

```
>>> (defmacro enquanto (CONDICAO &rest CORPO)
      `(tagbody INICIO
        (if ,CONDICAO (progn ,@CORPO
          (go INICIO)))))
ENQUANTO
```

```
>>> (defparameter *I* 1)
*I*
```

```
>>> (enquanto (<= *I* 5)
      (princ *I*)
      (terpri)
      (setf *I* (+ *I* 1)))
1
2
3
4
5
NIL
```

Para a definição da macro enquanto é realizado uso do operador especial tagbody que indica um ponto de retorno pela definição de um rótulo de identificação, neste caso o rótulo INICIO. O retorno é produzido pelo operador especial go que transfere o controle para o operador tagbody criando um laço de repetição contínuo.

## 4.5 FUNÇÕES DE TEMPO

---

As funções de tempo são recursos usados para a manipulação de dados relacionados as informações de data e hora, podendo ser representado de três formas diferentes: tempo decodificado, tempo universal e tempo interno.

As representações de tempo decodificado e universal ocorrem sobre o tempo baseado no calendário do computador tendo sua precisão definida em segundos. O tempo interno é voltado a medição de ciclos computacionais com precisão determinada a partir de certa grandeza de fração em segundos, determinada pelo ambiente CL em uso.

O padrão para hora decodificada indica o tempo de forma absoluta e os padrões de tempo universal e interno indicam respectivamente o tempo absoluto e relativo.

O tempo decodificado indica o tempo de calendário a partir de um número serial formado pelos componentes: segundo (0-59), minuto (0-59), hora (0-23), dia (1-31), mês (1-12), ano (na forma Anno Domini), dia da semana (0-6: 0 = segunda-feira até 6 = domingo), horário de verão (T horário de verão em uso, se NIL horário de verão não em uso) e fuso horário (número inteiro de horas a oeste do meridiano de Greenwich).

O tempo universal indica o tempo como um valor inteiro positivo na forma serial que se visto do ponto de vista relativo é um número indicado em segundos; se visto do ponto de vista absoluto é um número de segundos contados a partir da meia-noite de 1 de janeiro de 1900 GMT. Assim, o valor de tempo 3764413260 (segundos) corresponde ao tempo 11:21:00 de 16/04/2019 GMT.

O tempo interno também indica o tempo como um valor serial inteiro. O tempo relativo é medido como um valor da unidade de tempo usada pelo ambiente CL. O tempo absoluto é baseado a um valor de tempo arbitrário que normalmente usa a hora em que o ambiente CL começou a ser executado.

Na sequência veja o uso das funções de manipulação de data e hora e o significado do retorno apresentado de cada função usada.

A função `get-decoded-time` retorna uma lista de valores de tempo decodificado no formato linear com a indicação dos dados: segundo, minuto, hora, dia, mês, ano, dia da semana, horário de verão e fuso horário.

```
>>> (get-decoded-time)
```

```
0
```

```
21
```

```
11
```

```
16
```

```
4
```

```
2019
```

```
1
```

```
NIL
```

```
3
```

A função `get-universal-time` retorna um valor serial único que representa internamente o conjunto de dados: segundo, minuto, hora, dia, mês, ano, dia da semana, horário de verão e fuso horário.

```
>>> (get-universal-time)
```

```
3764413260
```

A função `decode-universal-time` decodifica o valor de tempo universal indicado mostrando separadamente o conjunto de dados: segundo, minuto, hora, dia, mês, ano, dia da semana, horário de verão e fuso horário. Esta função possui de forma opcional um segundo argumento o qual determina a informação de fuso horário como um número de horas compensadas a partir do meridiano de Greenwich.

```
>>> (decode-universal-time 3764413260)
```

```
0
```

```
21
```

```
11
```

```
16
```

```
4
```

```
2019
```

```
1
```

```
NIL
```

```
3
```



```
>>> (decode-universal-time 3764413260 5)
0
21
9
16
4
2019
1
NIL
5
```

O argumento de fuso horário quando em uso caracteriza-se por ser um valor múltiplo racional de  $1/3600$  entre -24 e 24 que representa um número de horas compensadas pelo tempo médio de Greenwich. No exemplo anterior o valor **5** estabelece

A função `encode-universal-time` codifica o valor de tempo universal indicado em: segundo, minuto, hora, dia, mês, ano e o uso opcional do fuso horário na forma de tempo decodificado. Não se usa nesta função o dado que representa o horário de verão por ser um dado de tipo lógico e nem a indicação do dia da semana.

```
>>> (encode-universal-time 0 21 11 16 4 2019 3)
3764413260
```

A função `get-internal-run-time` retorna um valor absoluto único serial baseado no formato de hora interna do sistema. O valor de tempo apresentado depende da implementação CL em uso.

```
>>> (get-internal-run-time)
234
```

A função `get-internal-real-time` retorna um valor relativo único serial baseado no formato de hora interna do sistema. O valor de tempo apresentado depende da implementação CL em uso.

```
>>> (get-internal-real-time)
15073340
```

A função `sleep` faz com que a execução de certa tarefa fique inativa um determinado tempo real em segundos, após o que a execução é retornada indicando o valor `NIL`. Esta função opera com valores positivos inteiros ou de ponto flutuante.

```
>>> (sleep 3)
NIL
```

Além das funções indicadas o recurso de tempo de um ambiente CL possui a constante `internal-time-units-per-second` que indica a medida de unidade de tempo interna por segundo usada. O valor de tempo apresentado depende da implementação CL em uso.

```
>>> internal-time-units-per-second
1000
```

Como exemplo de uso de recursos de tempo considere uma função que fornecida uma data indique o dia da semana correspondente. Para tanto, siga as instruções.

```
>>> (defun dia-semana (DIA MES ANO)
      (case (nth-value 6
                    (decode-universal-time
                     (encode-universal-time 0 0 0 DIA MES ANO 0)
                     0))
          (0 (format t "segunda-feira~%"))
          (1 (format t "terça-feira~%"))
          (2 (format t "quarta-feira~%"))
          (3 (format t "quinta-feira~%"))
          (4 (format t "sexta-feira~%"))
          (5 (format t "sabado~%"))
          (6 (format t "domingo~%")))))
```

```
DIA-SEMANA
```

```
>>> (dia-semana 26 04 1965)
segunda-feira
NIL
```

Ao ser executada a função `dia-semana` com a definição de uma data no formato dia, mês e ano ocorre a apresentação do dia da semana da data informada. Veja

que a parte (encode-universal-time 0 0 0 DIA MES ANO 0) efetua a codificação da data informada em um valor universal serial que é então decodificado pela função decode-universal-time com argumento 0 de fuso horário. A partir dessas duas ações ocorre a obtenção da série de valores segundo, minuto, hora, dia, mês, ano, dia da semana, horário de verão e fuso horário.

Note que a informação de dia da semana é a sétima informação existente, ou seja, é a informação de número 6, contando a partir de zero. Por esta razão usa-se o valor 6 na indicação de uso da macro nth-value que é operada com a sintaxe.

**(nth-value <índice> <formulário>)**

Onde, a indicação **índice** refere-se a indicação do índice de acesso posicional do conteúdo existente no argumentos e **formulário** que indica uma coleção de valores da qual um item é usado.

Observe alguns exemplos de uso simplificado da macro nth-value.

```
>>> (nth-value 1 (values 'a 'b 'c 'd 'e))
```

```
B
```

```
>>> (nth-value 4 (values 1 2 3 4 5))
```

```
5
```

```
>>> (nth-value 6 (decode-universal-time 3764413260)) ; dia da semana
```

```
1
```

```
>>> (nth-value 5 (decode-universal-time 3764413260)) ; ano
```

```
2019
```

Após capturar o valor do sétimo item da coleção de valores gerados a partir da data fornecida a função case verifica qual dos valores da data referente ao dia da semana foi retornado e apresenta o extenso do dia correspondente.

## 4.6 RANDOMIZAÇÃO

---

Randomização é a ação de realizar algo de forma aleatório. A linguagem CL oferece para a geração de números pseudoaleatórios a função `random` que é usada a partir da seguinte sintaxe.

```
(random <limite> [<estado>])
```

Onde, a indicação **limite** refere-se a definição do valor numérico positivo inteiro ou de ponto flutuante limite da geração de valores aleatórios de 0 (inclusivo) até `limite` (exclusivo) e **estado** refere-se a definição opcional do estado de aleatoriedade `make-random-state` com valor `T` a ser empregado, sendo o padrão por omissão definido junto a variável global interna **\*RANDOM-STATE\*** que atua como se fosse a definição de uma semente implícita de geração de números aleatórios, pontuando que CL não permite que seja definido de forma explícita um valor de semente para a geração de números aleatórios.

O estado para geração de números aleatórios muda a cada chamada efetuada a função `random` na mesma instância em que o ambiente CL está sendo executado. Se o ambiente for encerrado e inicializado novamente a geração de números aleatórios anteriores se repetirá, comprovando ser uma geração de números pseudoaleatórios. O efeito de repetição de números aleatórios a cada execução do ambiente pode ser neutralizado com o uso do estado (`make-random-state t`).

Observe a seguir a geração de valores aleatórios (em seu sistema os valores poderão ser diferentes) sem a definição do estado de geração de aleatoriedade com valores inteiros entre 0 e 9.

```
>>> (random 10)
```

```
7
```

```
>>> (random 10)
```

```
2
```

```
>>> (random 10)
```

```
6
```

```
>>> (random 10)
8
```

Saia do ambiente e retorne a ele novamente e repita as quatro etapas anteriores e observe a apresentação dos mesmos valores anteriormente sorteados.

Saia e retorne ao ambiente e veja a geração de valores aleatórios com a definição do estado de geração de aleatoriedade com valores inteiros entre **0** e **9**.

```
>>> (random 10 (make-random-state t))
3
```

```
>>> (random 10 (make-random-state t))
4
```

```
>>> (random 10 (make-random-state t))
4
```

```
>>> (random 10 (make-random-state t))
2
```

Saia do ambiente e retorne a ele novamente e repita as quatro etapas anteriores e observe a apresentação dos valores anteriormente sorteados dispostos de forma diferente.

Usar a geração de números aleatórios com a definição de estado de aleatoriedade passa a ser mais interessante na ação de simulação de geração de valores aleatórios.

Como comentado a geração de números aleatórios é realizada a partir de zero até o limite exclusivo definido. Caso deseje, por exemplo, gerar números aleatórios entre **1** e **10** use a instrução.

```
>>> (+ 1 (random 10 (make-random-state t)))
2
```

Em sentido mais amplo pode-se definir uma função que efetue a geração de números aleatórios entre a definição de um valor inicial e final. Assim, observe o código seguinte.

```
>>> (defun sorteio (INICIO FIM)
      (+ INICIO
         (random (1+ (- FIM INICIO))
                  (make-random-state t))))
```

```
10
```

A função `sorteio` caracteriza-se por ser a definição de uma forma que estabelece uma maneira mais cômoda em realizar a geração de números aleatórios. Nesta função está sendo usada a função de incremento “+” anexa a um valor de incremento como em “1+” para (1+ (- FIM INICIO)), gerando o mesmo efeito no uso de (+ 1 (- FIM INICIO)).

## 4.7 FUNÇÃO ANÔNIMA (LAMBDA)

Função anônima ou lambda caracteriza-se por ser uma sub-rotina trivial sem nome de identificação com o objetivo de retornar uma resposta rápida a execução de certa ação operacional de cunho genérico.

A estrutura de definição de uma função lambda (anônima) segue o modelo matemático  $\lambda a.(ab)$ , onde a função anônima  $[\lambda]$  de  $[a]$  a qual  $[.]$  efetua a ação prevista na operação  $[(ab)]$  tem desta ação seu resultado.

Uma função anônima é definida em CL com a macro `lambda`, ficando a disposição para uso uma única vez na memória a partir ou em conjunto com a execução da função `funcall`. Após a função anônima ser usada esta é removida automaticamente da memória.

A definição de uma função anônima é realizada pela instrução.

```
(lambda (<argumentos>) <corpo>)
```

Onde, a indicação **argumentos** refere-se a definição da lista de argumentos a ser usada pela função e **corpo** refere-se a estrutura operacional a ser executada a partir dos parâmetros fornecidos.

A chamada e execução de uma função anônima é realizada pela instrução.

```
(funcall <função> <argumentos>)
```

Onde, a indicação **função** refere-se a definição da função anônima a ser usada e **argumentos** o conteúdo passado para a execução da função indicada.

Considerando a função  $\lambda x.(x + 1)$  que apresenta o sucessor do argumento fornecido, onde o argumento **X** seja **9** tem-se como forma matemática escrita a operação  $\lambda x.(x + 1).9$ , que efetuado o cálculo para a operação  $\lambda x.(9 + 1)$ , terá como resultado o valor **10** para  $\lambda = 10$  e poderá ser declarada em CL a partir da função `lambda`.

Para tanto, execute as instruções a seguir no ambiente CLISP.

```
[n]> (lambda (X) (+ X 1))
#<FUNCTION :LAMBDA (X) (+ X 1)>
```

Para tanto, execute as instruções a seguir no ambiente SBCL.

```
* (lambda (X) (+ X 1))
#<FUNCTION (LAMBDA (X)) {10025D002B}>
```

Após a definição da função anônima esta pode ser executada a partir da instrução.

```
>>> (funcall * 9)
10
```

Na sequência se for solicitada nova execução da instrução `(funcall * 9)` ocorrerá a apresentação de mensagem de erro, pois a função anônima definida não está mais presente na memória. Isto ocorre, pois o argumento definido para uma função `lambda` permite referência de uso apenas no âmbito do corpo da função, mas nunca fora da função.

A execução de uma função anônima pode ser produzida de uma única vez com o uso conjunto da função `funcall` com a macro `lambda`. Observe o exemplo seguinte que mostra o quadrado do valor **4** definido na execução da chamada da função.

```
>>> (funcall (lambda (X) (expt X 2)) 4)
16
```

Neste caso o uso da função anônima ficou um pouco mais restritivo por exigir que o valor a ser usado no cálculo seja fornecido em conjunto com a função anônima definida.

A definição de função anônima pode ser realizada e utilizada em conjunto de outras funções, como por exemplo, apresentar o resultado de certa operação. Veja a seguir a definição de função anônima que apresenta o resultado da soma de dois valores.

```
>>> (write ((lambda (X Y) (+ X Y)) 3 7))  
10  
10
```

Ao ser executado o código anterior ocorre a apresentação do resultado calculado pela função anônima.

O uso de funções anônimas, nesta obra, será visto na parte que se refere ao uso da linguagem CL na forma funcional.

## 4.8 TIPO DE DADO TABELA DE SÍMBOLO (HASH)

---

As tabelas hash (**hashables**) ou tabelas de símbolos são estruturas usadas para o gerenciamento de coleção de dados esparsos (dados que não seguem uma ordem predefinida) de propósito geral definidos com pares de elementos compostos de chaves e valores organizados com base na definição da chave em si.

No uso de tabelas hash é possível criar e excluir entradas, além de efetuar a localização de conteúdo associado a certa chave. A localização de elementos é realizada de forma mais rápida do que as ações efetuadas com listas ou matrizes, pois se utiliza da chave para acessar os elementos na coleção.

Tabelas hash associam um valor a uma chave, de modo que, se houver a tentativa de adicionar novo valor a uma chave existente ocorrerá a substituição natural do valor existente pelo novo valor.

Tabelas hash são criadas a partir do uso da função `make-hash-table` que opera a partir da sintaxe.

```
(make-hash-table [<chave> <teste> <tamanho> <aumentar> <limite>])
```

Onde, a indicação **chave** refere-se a definição de uma chave para a tabela, **teste** define a maneira pela qual as chaves são comparadas, **tamanho** define o tamanho inicial da tabela a partir da indicação de um valor numérico inteiro maior que zero, **aumentar** define o valor do quanto aumentar o tamanho da tabela quando ficar



cheia e **limite** que determina o tamanho que a tabela deve possuir antes de ficar cheia. É oportuno pontuar que a função `make-hash-table` pode ser executada sem a indicação de argumentos, os quais são aqui omitidos por estarem fora do escopo deste trabalho.

Observe em seguida a instrução para a definição de uma tabela de símbolo (hash) chamada **\*ALUNO\***, definida como sendo uma variável global.

```
>>> (defvar *ALUNO* (make-hash-table))  
*ALUNO*
```

Observe como a criação da tabela hash é identificada no programa CLISP.

```
[n]> *ALUNO*  
#S(HASH-TABLE :TEST FASTHASH-EQL)
```

Observe como a criação da tabela hash é identificada no programa SBCL.

```
* *ALUNO*  
#<HASH-TABLE :TEST EQL :COUNT 0 {1002576DF3}>
```

A partir da definição da tabela hash **\*ALUNO\*** é possível adicionar elementos e recuperá-los para apresentação, por exemplo, por meio de uso da função `gethash` que opera a partir da sintaxe.

```
(gethash <chave> <tabela> [<padrão>])
```

Onde, a indicação **chave** refere-se a definição de uma chave existente na tabela, **tabela** refere-se a definição da tabela a ser pesquisada e padrão a definição de um valor opcional a ser retornado.

Para fazer uso da função `gethash` na ação de adição de elementos em uma tabela usa-se em conjunto a função `setf`. Assim, observe as instruções a seguir que adicionam os nomes dos alunos a tabela **\*ALUNO\***.

```
>>> (setf (gethash '001 *ALUNO*) '(Augusto))  
(AUGUSTO)
```

```
>>> (setf (gethash '002 *ALUNO*) '(Sandra))  
(SANDRA)
```

Observe os detalhes indicados para a tabela **\*ALUNO\*** no programa CLISP.

```
[n]> *ALUNO*
#S(HASH-TABLE :TEST FASTHASH-EQL (2 . (SANDRA)) (1 . (AUGUSTO)))
```

Observe os detalhes indicados para a tabela **\*ALUNO\*** no programa SBCL.

```
* *ALUNO*
#<HASH-TABLE :TEST EQL :COUNT 2 {1002576DF3}>
```

Para apresentar especificamente um dos elementos cadastrados na tabela hash pode-se fazer uso de uma das funções de saída conhecidas, como indicado a seguir.

```
>>> (princ (gethash '002 *ALUNO*))
(SANDRA)
(SANDRA)
```

Como experimento de nova entrada de dado na tabela e apresentação do conteúdo existente execute em seguida as instruções.

```
>>> (princ (gethash '001 *ALUNO*))
(AUGUSTO)
(AUGUSTO)
```

```
>>> (setf (gethash '001 *ALUNO*) '(Audrey))
(AUDREY)
```

```
>>> (princ (gethash '001 *ALUNO*))
(AUDREY)
(AUDREY)
```

```
>>> (princ (gethash '002 *ALUNO*))
(SANDRA)
(SANDRA)
```

Observe que na sequência anterior de instruções ao se definir com a chave **001** o valor **AUDREY** este foi sobreposto ao valor **AUGUSTO** anterior. É importante tomar cuidado com este tipo de operação.

Para realizar a apresentação dos elementos de uma tabela de hash pode-se usar a função `maphash` que possui a sintaxe.

```
(maphash <função> <tabela>)
```

Onde, a indicação **função** refere-se a uma função aplicada sobre os elementos da tabela em uso e **tabela** a indicação da tabela hash a ser usada.

Observe a instrução a seguir para apresentação do conteúdo da tabela **\*ALUNO\*** no programa CLISP.

```
[n]> (maphash (lambda (CHAVE VALOR)
                (format t "~a: ~a~%" CHAVE VALOR)) *ALUNO*)
2: (SANDRA)
1: (AUDREY)
NIL
```

Observe a instrução a seguir para apresentação do conteúdo da tabela **\*ALUNO\*** no programa SBCL.

```
* (maphash (lambda (CHAVE VALOR)
              (format t "~a: ~a~%" CHAVE VALOR)) *ALUNO*)
1: (AUDREY)
2: (SANDRA)
NIL
```

O uso da função `maphash` anterior tem como primeiro argumento a definição da função `lambda (CHAVE VALOR)` que é aplicada sobre a função `format` que apresenta o conteúdo indicado por `CHAVE` e `VALOR`. A função `maphash` efetua uma ação de iteração sobre todos os elementos da tabela em uso.

Outra maneira de obter o conteúdo de uma tabela hash é fazendo uso da macro `loop for ...` do a partir da instrução.

```
>>> (loop for C being the hash-keys in *ALUNO* using (hash-value V)
      do (format t "~a: ~a~%" C V))
```

Ao ser executada a instrução anterior o resultado apresentado será o mesmo gerado pela função `maphash`. Observe que a definição da variável local **C** se refere ao uso da chave e a definição da variável local **V** se refere ao uso do valor existente para cada elemento da tabela **\*ALUNO\***.

Para “pegar” a chave com o laço `loop` e associá-la a variável **C** está sendo usada a ação “`C being the hash-keys in`” que atribui o valor da chave (`hash-keys`) a variável **C** da tabela indicada e em seguida por meio da ação “`using (hash-value V)`” pega o valor correspondente (`hash-value`) para a variável **V** associando **C** (chave) e **V** (valor) e definindo assim o elemento a ser utilizado. Desta forma `loop` percorre toda a tabela e permite apresentar os elementos existentes em **\*VALORES\*** por meio da função `format`.

A expressão de iteração definida com `loop for ...` do pode ser usada a partir de quatro variações.

1. CHAVE being each hash-key of <tabela>
2. CHAVE being the hash-keys in <tabela> using (hash-value VALOR)
3. VALOR being each hash-value of <tabela> using (hash-key CHAVE)
4. VALOR being the hash-values in <tabela>

Os termos `hash-key/hash-keys` e `hash-value/hash-values` (atente para a forma no plural e no singular pois atuam de forma diferente na ação) são usados respectivamente para identificar em uma tabela hash a chave e o valor de um elemento. O termo `each ... of` refere-se a ação de “pegar” cada chave do elemento se usado `hash-key` ou os valores dos elementos se usado `hash-value` e realizar alguma ação com o conteúdo existente na tabela. O termo `the ... in` refere-se a ação de “pegar” as chaves dos elementos se usado `hash-keys` ou o valor do elemento se usado `hash-values` e realizar alguma ação sobre o conteúdo da tabela. O termo `using` coloca em uso para a operação a chave (`hash-key`) ou o valor (`hash-value`) quando respectivamente está em uso `being each hash-values` e `being the hash-keys in`.

A título de ilustração observe a seguir exemplos de uso das formas de iteração **1**, **3** e **4**, sendo que a forma **2** já fora apresentada.

```
>>> (loop for C being each hash-key of *ALUNO*
      do (format t "~a~%" C))
```

O trecho anterior faz a apresentação apenas das chaves dos elementos existentes na tabela **\*ALUNO\***.

```
>>> (loop for V being the hash-values in *ALUNO*
      do (format t "~a~%" V))
```

O trecho anterior faz a apresentação apenas dos valores dos elementos existentes na tabela **\*ALUNO\***.

```
>>> (loop for V being each hash-value of *ALUNO* using (hash-key C) do
      (format t "~a: ~a~%" C V))
```

O trecho anterior faz a apresentação das chaves e valores dos elementos existentes na tabela **\*ALUNO\***.

**OBS:** Usar os termos *hash-key*, *hash-keys*, *hash-value* ou *hash-values* no singular ou plural pode não interferir a resposta da ação desejada, mas dependendo do interpretador de CL em uso será apresentada mensagem de advertência informado para usar o termo adequado.

A quantidade de elementos armazenados em uma tabela hash pode ser obtida com o uso da função `hash-table-count` a partir da sintaxe.

```
(hash-table-count <tabela>)
```

Onde, a indicação **tabela** refere-se a definição da tabela hash a ser usada para ter a quantidade de elementos existentes.

Veja em seguida a instrução para saber a quantidade de elementos armazenados na tabela **\*ALUNO\***.

```
>>> (hash-table-count *ALUNO*)
2
```

Os elementos definidos em uma tabela hash podem ser removidos com o uso da função `remhash` que possui a sintaxe.

```
(remhash <chave> <tabela>)
```

Onde, a indicação **chave** refere-se a definição de uma chave existente na tabela e **tabela** refere-se a definição da tabela a ser pesquisada.

Observe a seguir a instrução para remover o elemento de chave **001** da tabela **\*ALUNO\*** com a função `remhash`.

```
>>> (remhash 001 *ALUNO*)
```

```
T
```

```
>>> (maphash (lambda (CHAVE VALOR)
```

```
  (format t "~a: ~a~%" CHAVE VALOR)) *ALUNO*)
```

```
2: (SANDRA)
```

```
NIL
```

Diferentemente das outras funções de gerenciamento de tabela hash a função `remhash` não usa o sinal de apóstrofo antes do valor da chave. Note ainda que após a remoção da chave **001** ficou definido na tabela apenas o elemento correspondente a chave **002**.

Para a remoção de elementos de uma matriz tem-se também a função `clrhash` que efetua a remoção de todos os elementos definidos para uma tabela operada a partir da sintaxe.

```
(clrhash <tabela>)
```

Onde, a indicação **tabela** refere-se a definição da tabela hash a ter todos os elementos removidos, deixando a tabela vazia.

Veja o resultado da execução da função `clrhash` executado no programa CLISP.

```
[n]> (clrhash *ALUNO*)
```

```
#S(HASH-TABLE :TEST FASTHASH-EQL)
```

Veja o resultado da execução da função `clrhash` executado no programa SBCL.

```
* (clrhash *ALUNO*)
```

```
#<HASH-TABLE :TEST EQL :COUNT 0 {1002576DF3}>
```

Na sequência se solicitada a apresentação dos elementos existentes na tabela ter-se-á a resposta NIL.

```
>>> (maphash (lambda (CHAVE VALOR)
              (format t "~a: ~a~%" CHAVE VALOR)) *ALUNO*)

NIL
```

Como demonstração mais acentuada do gerenciamento de elementos em tabelas hash considere a criação de uma tabela chamada **\*VALORES\*** formada por um conjunto de elementos configurados com chaves alfabéticas e valores numéricos inteiros. Observe as instruções seguintes.

```
>>> (defvar *VALORES* (make-hash-table :size 6))
>>> (setf (gethash 'A *VALORES*) 1)
>>> (setf (gethash 'B *VALORES*) 2)
>>> (setf (gethash 'C *VALORES*) 3)
>>> (setf (gethash 'D *VALORES*) 4)
>>> (setf (gethash 'E *VALORES*) 5)
>>> (setf (gethash 'F *VALORES*) 6)
```

A partir dos valores definidos considere realizar uma operação de atualização nos valores dos elementos existentes. A instrução que fará esta operação deve considerar que sendo um valor ímpar este deverá ser retirado da tabela e caso contrário deve o valor ser elevado ao cubo.

```
>>> (maphash (lambda (CHAVE VALOR)
              (if (oddp VALOR)
                  (remhash CHAVE *VALORES*)
                  (setf (gethash CHAVE *VALORES*) (expt VALOR 3))))
      *VALORES*)

NIL
```

Em seguida a fim de visualizar os valores que ficaram na tabela elevados ao cubo execute a instrução.

```
>>> (maphash (lambda (CHAVE VALOR) (format t "~a: ~a~%" CHAVE VALOR))
      *VALORES*)
```

Serão então apresentados os elementos **B** com valor **8**, **D** com valor **64** e **F** com valor **216**.

Para verificar se dada variável definida em memória é de fato uma tabela de hash pode-se fazer uso da função `hash-table-p` que retorna T se a verificação for verdadeira ou NIL se a verificação for falsa.

A função `hash-table-p` é usada a partir da sintaxe.

```
(hash-table-p <variável>)
```

Onde, a indicação ***variável*** refere-se a definição de uma variável a ser verificada.

Observe a instrução a seguir.

```
>>> (hash-table-p *VALORES*)
T
```

Uma ação que pode ser interessante ou até mesmo necessária é criar um mecanismo que venha a popular uma tabela hash de forma iterativa. Assim sendo, observe as instruções seguintes que criam uma tabela chamada ***\*TAB-X1\**** e efetua a definição de quatro chaves e seus respectivos valores.

```
>>> (defvar *TAB-X1* (make-hash-table))
*TAB-X1*

>>> (loop for (CHAVE VALOR) on
      '(um "um" dois "dois" tres "tres" quatro "quatro") by #'cddr
      do (setf (gethash CHAVE *TAB-X1*) VALOR))
NIL

>>> (loop for V being each hash-value of *TAB-X1* using (hash-key C)
      do (format t "~a: ~a~%" C V))
```

Veja que na apresentação do conteúdo as chaves são expressas em formato maiúsculo e os valores que foram definidos entre aspas inglesas ficam mantidos em caracteres minúsculos.



## 4.9 TIPO DE DADO ESTRUTURA

O tipo de dado estrutura permite que sejam criados conjuntos de dados formados a partir de dados de tipos diferentes para representarem registros definidos a partir da macro `defstruct` que é operada a partir da sintaxe simplista e resumida.

```
(defstruct <nome> (<campo1 [<opcao1>]) [(<campoN [<opcaoN>])])>
```

Onde, a indicação **nome** refere-se ao nome de identificação da estrutura; as indicações **campo1** e **campoN** referem-se ao conjunto de elementos informativos que formam a base do registro a partir do uso opcional da definição de opções **opcao1** e **opcaoN** de configuração que podem conter a definição de valores iniciais de cada campo, do tipo de dado a ser aceito por cada campo, além de outras possibilidades.

Considere a definição de uma estrutura que modele um conjunto de dados para o gerenciamento de alunos. Para tanto, é necessário indicar um campo para o nome do aluno, um campo para registro das notas e outro campo para o armazenamento da média. Para tanto, siga as instruções.

```
>>> (defstruct CADALUNO
      (nome "" :type string)
      (notas (make-array 4 :element-type 'single-float))
      (media 0.0 :type single-float))
```

Veja que a definição da estrutura **CADALUNO** é configurada com o campo `nome` sem a indicação de nenhum conteúdo inicial ("") do tipo cadeia a partir da indicação `:type string`. O campo `notas` é configurado como uma matriz de uma dimensão para quatro notas bimestrais `make-array 4` com elementos do tipo ponto flutuante simples a partir da indicação `:element-type 'single-float`. Por último o campo `media` é também inicializado com valor zero definido para uso de valor de ponto flutuante simples.

Com a definição da estrutura **CADALUNO** são implicitamente definidas de forma automática **funções de acesso** aos campos existentes, sendo representados por: `cadaluno-nome`; `cadaluno-notas` e `cadaluno-media`, os quais receberão um argumento como componente da representação de seu dado informativo.

Assim que um tipo estrutura está definido na memória é possível criar variáveis que farão uso dessa estrutura. Para tanto, execute o uso da função `defvar` para definir

a variável e use a função `setf` para inserir dados na variável criada. Assim sendo, siga as instruções seguintes.

```
>>> (defvar *ALUNO*)
*ALUNO*

>>> (setf *ALUNO* (make-cadaluno
      :nome "Jose"
      :notas #(2.5 7.5 8.5 4.5)))
#S(CADALUNO :NOME "Jose" :NOTAS #(2.5 7.5 8.5 4.5) :MEDIA 0.0)
```

Note o uso da função construtora implícita `make+<nome da estrutura>`, neste caso, `make-cadaluno` que tem por finalidade modelar a variável global **\*ALUNO\*** com os campos estabelecidos para a estrutura **CADALUNO**. Observe que a função `make-cadaluno` quando invocada, cria na variável em uso a estrutura de dados adequada para uso com as funções de acesso relacionadas aos campos estabelecidos, ou seja, cria uma instância com o construtor definido. Uma estrutura definida é identificada com o prefixo “#S” podendo ser usada para operações de leitura e escrita como será demonstrada.

Observe que ao se definir os dados para a variável estruturada está sendo omitido o valor da média, pois este será internamente atribuído a zero automaticamente pela simples omissão de seu dado.

Para visualizar o conteúdo total da variável execute a instrução.

```
>>> *ALUNO*
#S(CADALUNO :NOME "Jose" :NOTAS #(2.5 7.5 8.5 4.5) :MEDIA 0.0)
```

Veja que foram indicados o nome do aluno e informada as quatro notas bimestrais e deixado com valor zero o campo média, pois esta ação será realizada a partir do cálculo da média sobre as notas existentes. Para tanto, execute a instrução.

```
>>> (setf (cadaluno-media *ALUNO*)
      (/ (apply '+ (coerce (cadaluno-notas *ALUNO*) 'list)) 4))
5.75
```

Para o cálculo a função `setf` atribui no campo `media` da estrutura `cadaluno-media` o valor da média calculada. Para calcular a média é feita primeiramente a conver-

são da matriz em lista (`coerce (cadaluno-notas *aluno*) 'list`)). Assim que os dados da matriz são convertidos como lista a função `apply` efetua a soma dos elementos da lista e em seguida é dividido por quatro.

A partir da definição dos dados de uma estrutura é possível apresentá-los separadamente ou juntos a partir das instruções.

```
>>> (cadaluno-nome *ALUNO*)
```

```
Jose
```

```
>>> (cadaluno-notas *ALUNO*)
```

```
 #(2.5 7.5 8.5 4.5)
```

```
>>> (cadaluno-media *ALUNO*)
```

```
(cadaluno-media *aluno*)
```

```
>>> *ALUNO*
```

```
#S(CADALUNO :NOME "Jose" :NOTAS #(2.5 7.5 8.5 4.5) :MEDIA 5.75)
```

Para definição de mais alunos basta definir novas variáveis globais estruturadas e replicar as ações comentadas anteriormente sobre a “nova” variável.

Desejando maior dinamismo na operação você poderá definir funções que automatizem os passos mostrados anteriormente.



## 5

## Programação CL

---

*Neste capítulo são apresentados alguns exemplos complementares de programação em CL, tais como: introdução a programação estruturada e funcional; desenvolvimento de pacotes e o estabelecimento das regras para definição de comentários de código de programas.*

### 5.1 PROGRAMAÇÃO ESTRUTURADA

---

A partir da exposição das diversas funcionalidades para a linguagem CL passa-se a ter em mãos as ferramentas necessárias para escrever programas mais complexos na forma de **scripts**.

Para demonstrar o uso da técnica de programação estruturada em CL considere um programa que a partir de um menu indique a possibilidade de escolha de uma entre quatro operações aritméticas que seja operada a partir do fornecimento de dois valores numéricos. O programa deve possuir a indicação de cinco opções de operação, sendo quatro para as operações aritméticas de soma, subtração, divisão e multiplicação e uma para a saída do programa. Se fornecido opção de menu diferente das opções esperadas o programa deve apresentar mensagem de erro. Caso seja indicado divisor zero para a operação de divisão a resposta deverá ser a mensagem “Erro”.

Programas escritos em estilo estruturado seguem em CL o padrão **bottom-up** de codificação, onde primeiro codifica-se as sub-rotinas de nível mais baixo do programa e em seguida se codifica as sub-rotinas de nível mais alto. Note que primeiramente é codificada as rotinas das operações aritméticas e no final é codificado o trecho principal do programa.

O código seguinte deve ser escrito em um editor de texto simples e gravado com o nome **calc1.lisp** ou **calc1.lisp**, onde **LSP** ou **LISP** (*Lisp Source Program*) indicam a identificação da extensão de arquivos de programas codificados em CL. A extensão **LSP** é uma forma mais antiga de representação, atualmente utiliza-se mais a forma **LISP**.

Note que todos os detalhes usados neste programa foram apresentados ao longo das páginas anteriores. Atente para um fato novo em que a definição de uma função pode conter internamente um conjunto de funções. Grave o programa proposto em uma pasta chamada "**FonteLISP**" a partir da unidade **C:**.

```
(defun calculadora ()

  (defun ADICAO ()
    (let ((A 0) (B 0) (R 0))
      (format t "~%Rotina de Adicao~%")
      (format t "-----~%~%")
      (princ "Entre valor <A> .: ") (finish-output)
      (setf A (read))
      (princ "Entre valor <B> .: ") (finish-output)
      (setf B (read))
      (setf R (+ A B))
      (format t "Resultado .....: ~,2f~%" R)))

  (defun SUBTRACAO ()
    (let ((A 0) (B 0) (R 0))
      (format t "~%Rotina de Subtracao~%")
      (format t "-----~%~%")
      (princ "Entre valor <A> .: ") (finish-output)
      (setf A (read))
      (princ "Entre valor <B> .: ") (finish-output)
      (setf B (read))
      (setf R (- A B))
      (format t "Resultado .....: ~,2f~%" R)))
```

```

(defun MULTIPLICACAO ()
  (let ((A 0) (B 0) (R 0))
    (format t "~%Rotina de Multiplicacao~%")
    (format t "-----~%~%")
    (princ "Entre valor <A> .: ") (finish-output)
    (setf A (read))
    (princ "Entre valor <B> .: ") (finish-output)
    (setf B (read))
    (setf R (* A B))
    (format t "Resultado .....: ~,2f~%" R)))

(defun DIVISAO ()
  (let ((A 0) (B 0) (R 0))
    (format t "~%Rotina de Divisao~%")
    (format t "-----~%~%")
    (princ "Entre valor <A> .: ") (finish-output)
    (setf A (read))
    (princ "Entre valor <B> .: ") (finish-output)
    (setf B (read))
    (if (= B 0)
        (format t "Resultado .....: Erro~%~%")
        (progn
         (setf R (/ A B))
         (format t "Resultado .....: ~,2f~%" R)))))

(loop
  (let ((OPCAO 0))
    (format t "~%Programa Calculadora~%")
    (format t "-----~%~%")
    (format t "[1] - Adicao~%")
    (format t "[2] - Subtracao~%")
    (format t "[3] - Multiplicacao~%")
    (format t "[4] - Divisao~%")
    (format t "[5] - Fim de programa~%~%")

```

```
(format t "Entre uma opcao: ") (finish-output)
(setf OPCA0 (read))
(when (/= OPCA0 5)
  (progn
    (cond
      ((= OPCA0 1) (ADICAO))
      ((= OPCA0 2) (SUBTRACAO))
      ((= OPCA0 3) (MULTIPLICACAO))
      ((= OPCA0 4) (DIVISAO))
      (t (format t "Opcao invalida - tente novamente~%")))))
(when (= OPCA0 5)
  (return))))
```

A partir do **script** de programa gravado é possível fazer seu uso de duas maneiras diferentes: uma executando o programa de dentro do ambiente e outra executando o programa no sistema operacional a partir do ambiente e mantendo-se no sistema.

Para executar o programa `calc1.lisp` no **prompt** do sistema operacional e manter-se ao final da execução no próprio sistema sem que ocorra na memória o carregamento do ambiente CL sigas as instruções seguintes.

Com o uso do programa CLISP execute a instrução.

```
C:\> clisp calc1.lisp
```

Com o uso do programa SBCL execute a instrução.

```
C:\> sbcl --script calc1.lisp
```

Para executar o programa **calc1.lisp** de dentro dos ambientes CLISP e SBCL siga as seguintes instruções.

Com o programa CLISP carregado execute a instrução seguinte que efetua o carregamento do programa **calc1.lisp** na memória.

```
[n]> (load "C:\\FonteLISP\\calc1.lisp")
;; Loading file C:\\FonteLISP\\calc1.lisp ...
;; Loaded file C:\\FonteLISP\\calc1.lisp
T
```



Com o programa SBCL carregado execute a instrução seguinte que efetua o carregamento do programa **calc1.lisp** na memória.

```
* (load "C:\\FonteLISP\\calc1.lisp")
T
```

Após o carregamento na memória do programa **calc1.lisp** para usá-lo basta apenas executar a instrução.

```
>>> (calculadora)
```

A partir deste momento escolha a opção desejada no menu e informe os valores solicitados. Veja cada uma das opções. Teste a entrada de valor zero para o divisor. Tente efetuar a entrada de opções diferentes das opções indicadas no menu. Veja todas as possibilidades do programa.

## 5.2 DEFINIÇÃO DE COMENTÁRIOS

Além das definições de instruções, funções e programas escritos em CL é adequado conhecer as regras de definição das linhas de comentários dos códigos de um programa, que podem ser estabelecidas a partir de três símbolos diferentes com: comentários alados (#| |#), ponto e vírgula (;) e aspas inglesas ("").

Os comentários alados são iniciados com (#|) e finalizados com (|#) servindo para a definição de blocos de comentários que ocupem mais de uma linha aceitando-se o uso de blocos aninhados simulando sub níveis. Por vezes essa forma de comentário pode ser usada para indicar a eliminação temporária de trechos de códigos na fase de testes de um programa por permitir abranger grupos de linhas.

**#| Sub-rotina destinada a geração de valores aleatórios definidos em uma faixa numérica delimitada entre um valor inicial representado pelo argumento INICIO e um valor final representado pelo argumento FIM.**

```
#|
```

Os valores dos argumentos INICIO e FIM poderão ser definidos a partir do uso números inteiros e/ou reais.

```
|#
```

```
|#
```

```
(defun sorteio (INICIO FIM)
  (+ INICIO
    (random (1+ (- FIM INICIO)))
    (make-random-state t))))
```

Quanto ao uso do símbolo ponto e vírgula (;) este pode ser definido a partir de quatro variações descritas a seguir.

A indicação de comentários com o uso de um ponto e vírgula (;) são usados para descrever as ações de certa linha de código. Quando usados vários comentários em esses devem ser alinhados a partir da mesma posição usada para o comentário da coluna anterior.

```
(defun sorteio (INICIO FIM)      ; Função "sorteio" com INICIO e FIM
  (+ INICIO                      ; soma o valor do INICIO ao sorteio
    (random (1+ (- FIM INICIO))) ; do valor entre 1 e FIM - INICIO a
    (make-random-state t))))    ; partir do modo randômico variável
```

A indicação de comentários com dois pontos e vírgulas (;;) são usados para descrever o propósito das linhas de código ou o estado operacional de certo programa naquele ponto, são usados na definição de comentários comuns. Este tipo de comentário fica alinhado com o mesmo nível de recuo do código quando assim for usado.

```
;; Função "sorteio" que efetua a apresentação de valor randômico entre
;; um valor inicial (INICIO) e valor final (FIM) informados.
(defun sorteio (INICIO FIM)
  ;; Função sorteio com simulação de geração automática de semente
  (+ INICIO
    (random (1+ (- FIM INICIO)))
    (make-random-state t))))
```

A indicação de comentários com três pontos e vírgulas (;;;) são usados para descrever uma seção de programa. Esses comentários devem ser iniciados na margem esquerda do código.

**;;; Função para geração de valores aleatórios**

```
(defun sorteio (INICIO FIM)
  (+ INICIO
     (random (1+ (- FIM INICIO))
              (make-random-state t))))
```

A indicação de comentários com quatro pontos e vírgulas (;;;) são usadas para descrever observações em nível de arquivo dando normalmente explicações gerais sobre o conteúdo do arquivo contendo um conjunto de funcionalidades. Esses comentários devem ser iniciados na margem esquerda do código.

**;;; Instruções para teste de execução recursiva de cálculo de  
;;; fatorial que indicam os resultados de 0 a 15.**

```
;;; Função recursiva para cálculo de fatorial
(defun fat (N)
  ;; Cálculo de fatorial ocorrerá apenas com valores numéricos
  ;; inteiros.
  (if (<= N 0) ; Se N menor ou igual a zero
      1 ; retorna 1.
      (* N (fat (- N 1)))))

;;; Parte principal que apresenta as fatoriais de 0 a 15.
(loop for I from 0 to 15
  do (format t "~2d! = ~13d~%" I (fat I)))
```

A indicação de comentários entre aspas inglesas (") são usadas para descrever uma forma de documentação incorporada ao código de uma função que pode ser visualizado com a função `documentation`. Isto é válido desde que esse comentário seja a primeira expressão na forma de cadeia associada a partir da definição de uma função podendo ser uma descrição com qualquer quantidade de linhas. A tabulação no uso desse tipo de comentário ocorre no mesmo nível de definição da primeira instrução em relação a definição da função.

```
(defun fat (N)
  "Efetua o cálculo de N fatorial"
  (if (<= N 0) ; Se N menor ou igual a zero
      1          ; retorna 1.
      (* N (fat (- N 1)))))
```

A partir da definição de comentários para documentação é possível obtê-los a partir do uso da instrução.

```
>>> (documentation 'fat 'function)
"Efetua cálculo de fatorial"
```

A função `documentation` pode ser usada para obter a descrição de certo recurso da linguagem CL como, por exemplo, ver a descrição de alguma função interna. No entanto, se a descrição não existir será retornado o valor `NIL`.

Os detalhes aqui expostos seguem orientações gerais existentes e recomendadas na documentação disponibilizada para a linguagem CL. Existem variações praticadas que não estão sendo aqui consideradas. Manteve-se o senso comum usado por grande parte dos programadores CL.

## 5.3 PACOTES (PACKAGE)

---

O sistema da linguagem CL mantém internamente em seu ambiente operacional uma tabela contendo os símbolos disponíveis no sistema e também aqueles definidos na etapa da programação, como: variáveis especiais (globais), funções, macros, entre outros. Quando um novo símbolo é definido este é automaticamente adicionado a esta tabela (conhecida como pacote do sistema).

Os pacotes são um recurso existente na linguagem CL que permite dividir o estado de memória em áreas de trabalho denominadas nomes de espaços (***namespace***). Cada pacote permite definir conjuntos de símbolos formados por variáveis globais, funções, macros ou outro elemento que ficam separados do restante do sistema.

O motivo para se usar pacotes (há quem os chame de módulos) é permitir a definição controlada de uma área de trabalho isolada onde os programadores podem criar seus símbolos sem se preocupar se determinado símbolo criado por ele entra em conflito com outro símbolo usado no sistema ou mesmo escrito por outro programador. Os pacotes reduzem eventuais conflitos entre códigos produzidos inde-

pendentemente por programadores, podendo também ser úteis na definição de arquivos de programas como bibliotecas.

O ambiente CL possui um conjunto de pacotes (que podem ser diferentes entre as diversas distribuições CL existentes), dos quais se destacam três pacotes padrão mais comuns e existentes em qualquer distribuição, sendo:

- **common-lisp** - pacote padrão que possui os símbolos relacionados a todas as funções e variáveis globais definidas pelo padrão da linguagem e disponibilizados para o uso geral da linguagem;
- **keyword** - pacote usado para a identificação de todos os nomes internos da linguagem iniciados por dois pontos;
- **common-lisp-user** – pacote, também referenciado como **cl-user** que utiliza além do pacote padrão *common-lisp*, todos os demais pacotes relacionados as operações de edição e depuração do ambiente de trabalho.

Para visualizar o pacote ativo em certo momento basta solicitar que seja apresentado o conteúdo definido na variável especial **\*PACKAGE\*** a partir da instrução.

```
>>> *package*
```

No programa CLISP é apresentada a resposta.

```
#<PACKAGE COMMON-LISP-USER>
```

No programa SBCL é apresentada a resposta.

```
#<PACKAGE "COMMON-LISP-USER">
```

O programa SBCL mostra o nome do pacote COMMON-LISP-USER entre aspas inglesas diferentemente do programa CLISP que o indica sem aspas. Além do uso da apresentação do conteúdo da variável **\*PACKAGE\*** é possível obter essa informação por meio das instruções.

```
>>> common-lisp:*package*
```

```
>>> cl:*package*
```

Para ver todos os pacotes existentes no ambiente execute a instrução.

```
>>> (list-all-packages)
```

A lista de pacotes apresentada é diferente entre os ambientes de programação CL existentes, pois isso depende da distribuição do ambiente CL. Considerando as

distribuições CLISP e SBCL tem-se como padrão os três pacotes antes indicados, como: `common-lisp`, `keyword` e `common-lisp-user`.

A criação de pacotes é realizada com o uso da função `defpackage` a partir da sintaxe.

```
(defpackage <:nome>
  (:use <:pacote1> [<:pacote2> ... [<:pacoteN>]])
  (:export <:simbolo1> [<:simbolo2> ... [<:simboloN>]]))
```

Onde **:nome** representa a definição do nome do pacote que deve ser definido sem o uso de espaços em branco, **:use** se refere ao vínculo e uso de pacotes necessários ao desenvolvimento do pacote criado e **:export** se refere ao uso opcional de símbolos que são definidos externamente para o pacote criado.

Execute as instruções seguintes para criar três pacotes no programa CLISP.

```
[n]> (defpackage :pacote1 (:use :common-lisp))
#<PACKAGE PACOTE1>
```

```
[n]> (defpackage :pacote2 (:use :common-lisp))
#<PACKAGE PACOTE2>
```

```
[n]> (defpackage :pacote3 (:use :common-lisp))
#<PACKAGE PACOTE3>
```

Execute as instruções seguintes para criar três pacotes no programa SBCL.

```
* (defpackage :PACOTE1 (:use :common-lisp))
#<PACKAGE "PACOTE1">
```

```
* (defpackage :PACOTE2 (:use :common-lisp))
#<PACKAGE "PACOTE2">
```

```
* (defpackage :PACOTE3 (:use :common-lisp))
#<PACKAGE "PACOTE3">
```

Na sequência se executada a função `list-all-packages` será apresentado na lista de pacotes do ambiente a existência dos três pacotes criados.

Quando um pacote é criado este pode ter seu nome modificado a qualquer momento com o uso da função `rename-package` operada a partir da sintaxe.

```
(rename-package '<nome_antigo>' '<nome_novo>')
```

Onde ***nome\_antigo*** representa o nome do pacote existente em memória e a indicação ***nome\_novo*** se refere ao novo nome do pacote existente.

Observe a seguir a instrução de mudança do nome do pacote PACOTE1 para o nome PACOTE4 no programa CLISP.

```
[n]> (rename-package 'PACOTE1 'PACOTE4)  
#<PACKAGE PACOTE4>
```

Observe a seguir a instrução de mudança do nome do pacote PACOTE1 para o nome PACOTE4 no programa SBCL.

```
* (rename-package 'PACOTE1 'PACOTE4)  
#<PACKAGE "PACOTE4">
```

Um pacote criado pode ser removido com o uso da função `delete-package`, a partir da sintaxe.

```
(delete-package '<nome>')
```

Onde ***nome*** representa o nome do pacote a ser removido.

Observe a instrução para remoção dos pacotes anteriormente definidos.

```
>>> (delete-package 'PACOTE2)  
T
```

```
>>> (delete-package 'PACOTE3)  
T
```

```
>>> (delete-package 'PACOTE4)  
T
```

A partir da apresentação de uma visão básica sobre o gerenciamento de pacotes será definido um novo pacote chamado TESTE a partir da instrução.

```
>>> (defpackage :TESTE (:use :common-lisp))
```

Para se fazer uso efetivo de um pacote é necessário usar a função `in-package` que possui a sintaxe.

```
(in-package <nome>)
```

Onde **nome** representa o nome do pacote a ser colocado em uso na memória.

No programa CLISP execute a instrução a seguir e veja a mudança do **prompt**.

```
[n]> (in-package TESTE)
#<PACKAGE TESTE>
TESTE[n]>
```

No programa SBCL execute a instrução seguinte.

```
* (in-package TESTE)
#<PACKAGE "TESTE">
*
```

No programa CLISP o pacote em uso é indicado ao lado esquerdo do **prompt**, mas no programa SBCL o **prompt** padrão é mantido. Uma forma segura de verificar em qual pacote o programa SBCL está a fazer uso é utilizar uma das instruções.

```
* *package*
* common-lisp:*package*
* cl:*package*
```

A partir da definição e ativação do nome de espaço TESTE será neste local definida a função mensagem que indicará a apresentação do texto "Acao executada na area TESTE" quando executada no nome de espaço TESTE. Assim sendo, crie a seguinte função.

```
>>> (defun mensagem () "Mensagem em TESTE")
MENSAJEM
```

Na sequência execute a chamada da função mensagem e observe o resultado apresentado.

```
>>> (mensagem)
"Mensagem em TESTE"
```



Para sair do nome de espaço TESTE e retornar ao nome de espaço padrão execute a instrução.

```
>>> (in-package COMMON-LISP-USER)
```

Neste momento se executada a função mensagem ocorrerá um erro por esta função não estar ativa na área de memória COMMON-LISP-USER. Desta forma, para executar a função mensagem da área TESTE execute a instrução.

```
>>> (teste::mensagem)
```

```
"Mensagem em TESTE"
```

Veja que para acionar a ação de uma função de um nome de espaço não ativo é necessário indicar o nome do pacote e o nome da função separada por um operador de escopo indicado pelos símbolos duplos de dois pontos (::).

A partir do que foi exposto é possível criar de forma simples um pacote matemático que contenha algumas constantes não existentes no ambiente CL e algumas funções auxiliares. Para tanto, escreva em um editor de texto simples o código seguinte gravando-o na pasta **FonteLISP** com o nome **math.lisp**.

```
;;; Pacote .....: MATH
;;; Autor .....: Augusto Manzano
;;; Finalidade ...: Apoio a operacoes matematicas

;;; Definicao do pacote
(defpackage :MATH
  (:use :common-lisp))

;;; Pacote colocado em uso
(in-package MATH)

;;; Definicao de constantes de apoio
(defconstant M_E 2.718281828459045235360287471352662497757)
(defconstant M_LOG2E 1.4426950408889634073599)
(defconstant M_LOG10E 0.434294481903251827651128)
(defconstant M_LN2 0.693147180559945309417232121458176568075)
(defconstant M_LN10 2.302585092994045684017991)
```

```

(defconstant M_PI2 1.570796326794896619231321691639751442098)
(defconstant M_PI4 0.785398163397448309615660845819875721049)
(defconstant M_SQRT2 1.41421356237309504880168872420969807)
(defconstant M_SQRT1_2 0.707106781186547524400844)

;;; Função para obtencao de quociente inteiro
(defun div (A N)
  (multiple-value-bind (Q) (floor A N) Q))

;;; Função recursiva para cálculo de fatorial
(defun fact (N &optional (BASE 1))
  (if (< N 0)
      (format t "Error~%")
      (if (and (>= N 0) (< N 2))
          BASE
          (fact (- N 1) (* BASE N))))))

;;; Função ambigüidade
;;; Dada uma lista de valores a funcao escolhera apenas um dos
;;; valores, sem que se saiba de antemao qual valor sera escolhido
(defun amb (&rest VALORES)
  ;; VALORES representa uma lista de itens separados por espaco
  ;; em branco
  (nth (random (length VALORES)) VALORES))

;;; Função para geracao de faixa de valores
(defun range (INICIO FIM)
  (loop for VALOR from INICIO to FIM
        collect VALOR))

(in-package COMMON-LISP-USER)

```

Assim que o arquivo **math.lisp** estiver escrito e gravado saia do ambiente CL. Faça novo acesso e execute a instrução.

```
>>> (load "math.lisp")
```

E em seguida, para testar o pacote carregado, execute as instruções.

```
>>> math::m_pi2
```

```
1.5707964
```

```
>>> (math::fact 5)
```

```
120
```

```
>>> (math::range 1 5)
```

```
(1 2 3 4 5)
```

```
>>> (math::amb 5 3 1)
```

```
5
```

A definição da função `amb` (ambiguidade) do pacote `math` tem por finalidade devolver aleatoriamente um dos elementos indicados na lista de valores definida. Devido ao uso de `&rest` é possível especificar uma quantidade indefinida de elementos para a lista de valores da função.

Um uso prático para a função `amb` pode ser aplicado a necessidade de saber quais valores em certa equação gera determinada resposta. Por exemplo, quando que os valores das variáveis **A** e **B** geram o resultado **5** a partir de  $5 = A + B$ . Note a instrução a seguir.

```
>>> (let ((a (math::amb 1 2 3 4))
          (b (math::amb 1 2 3 4)))
      (if (= 5 (+ a b)) (list a b)))
```

Veja que serão apresentados os valores para as variáveis **A** e **B** quando a soma desses valores for igual a **5**. Caso não seja o resultado **5** o retorno será **NIL**. Para testar a instrução anterior execute-a algumas vezes em seu ambiente até que os valores que geram o resultado **5** seja apresentado.

A função definida como `range` gera uma lista de valores numéricos a partir da indicação de um valor menor e outro maior informado como seus argumentos. Para criar uma lista de valores usa-se a cláusula de construção `collect` que na ação de iteração controlada pelo laço e ao receber certo valor o coloca sequencialmente em

uma lista. Note que quando se usa `collect` não se faz uso da cláusula `do` para a macro `loop`.

A necessidade de indicar o nome do pacote e a funcionalidade a ser utilizada é que garante a possibilidade de se fazer uso de nomes de recursos repetidos em pacotes diferentes.

## 5.4 PROGRAMAÇÃO FUNCIONAL

---

O paradigma de programação imperativa teve seu início marcado com o lançamento da linguagem FORTRAN em 1954, dando origem inicialmente ao paradigma de programação semiestruturada, tornando-se mais tarde paradigma de programação estruturada com o lançamento da linguagem ALGOL de 1958, a qual influenciou posteriormente outras linguagens, destacando-se em 1965 a linguagem Simula, primeira linguagem a utilizar o paradigma de programação orientada a objetos.

De certa forma, em contradição ao estilo proposto pelo paradigma imperativo (semiestruturado, estruturado e orientado a objetos) surgiu em 1958 o paradigma declarativo funcional, tendo como seu primeiro exemplar a linguagem LISP como seu representante.

A programação semiestruturada, estruturada e orientada a objetos, por serem um ramo da programação imperativa descreve as operações computacionais como ações executadas por meio de instruções que mudam o estado dos dados armazenados em memória por intermédio de variáveis, ou seja, enfatiza ações com mudanças no estado do programa. Já a programação funcional por ser um paradigma declarativo trata suas operações a partir da avaliação de funções matemáticas, evitando o uso de estados mutáveis comum no paradigma imperativo.

Na programação funcional, de um ponto de vista geral, um dos tipos de dados mais usados e populares é a **lista**, que permite simular computacionalmente a aplicação de conjuntos do ponto de vista estudado pela **teoria de conjuntos**.

As operações relacionadas a manipulação de conjuntos em CL são principalmente realizadas com o uso dos tipos de dados `list`, `array` e `function`. Assim sendo, são indicados neste tópico apenas os detalhes relacionados a aplicação matemática sobre conjuntos e suas relações a partir do uso de funções.

A programação funcional tem como princípio operacional básico o uso de funções de ordem superior, também chamadas de funções de alta ordem que possuem

como característica operacional a capacidade de receber como argumento uma função ou retornar como resultado uma função.

Considere como exemplo o uso da função simples *soma-raizes* que tem por finalidade somar as raízes quadradas definidas entre dois valores inteiros delimitados pelos argumentos I (início) e F (fim) e obtidos a partir do uso da função *sqrt*.

```
>>> (defun soma-raizes (I F)
      (if (> I F)
          0
          (+ (sqrt I) (soma-raizes (1+ I) F))))
```

Veja que a função *soma-raizes* faz uso do recurso *(1+ I)* para incrementar o valor **1** junto a variável *I* a cada vez que a ação recursiva é processada na soma do resultado de uma raiz com o próximo valor.

Para verificar a ação da operação da função *soma-raizes* execute a instrução seguinte e observe a apresentação do resultado da soma das raízes de **1** a **5**.

```
>>> (soma-raizes 1 5)
8.382332
```

As funções *soma-raizes* e *sqrt* não estão sendo usadas como funções de primeira ordem, pois são funções simples, sendo necessário extrair a raiz quadrada de cada valor para em seguida produzir o somatório de cada um dos valores calculados. E se quiser fazer o somatório dos valores pares entre dois limites? Seria necessário criar outra função de somatório para atender a esta necessidade. Agora considere usar apenas uma função para calcular o somatório, por exemplo, das raízes quadradas, dos quadrados ou qualquer outra ação entre os limites estabelecidos.

Funções de ordem superior são maneiras de se definir abstrações para a realização de certas operações sem se preocupar como essas ações são efetivamente feitas, uma vez que já estão definidas e podem ser usadas a qualquer momento.

Como exemplo, considere uma função, com toque genérico, chamada *somatorio* que fará uso de alguma outra função passada a ela como argumento. Desta forma, observe o código da função a seguir.

```
>>> (defun somatorio (I F FUNCAO)
      (if (> I F)
          0
          (+ (funcall FUNCAO I) (somatorio (+ I 1) F FUNCAO))))
```

A função `somatorio` recebe para sua operação três argumentos em sua chamada, representados pelas variáveis `I` (início), `F` (fim) e `FUNCAO` (uma função de ordem superior a ser fornecida). Enquanto o valor da variável `I` não for maior que o valor da variável `F` ocorrerá o processamento da soma por recursão dos resultados aplicados pela função passada como argumento em `(funcall FUNCAO I)`, onde `FUNCAO` representa a função passada no argumento de `somatorio` e `funcall` estabelece a execução da função passada como argumento.

Para verificar a ação da função `somatorio` execute a instrução seguinte.

```
>>> (somatorio 1 5 (function sqrt))
8.382332
```

Observe que para aplicar uma função de ordem superior como argumento de outra função é necessário defini-la com o uso do comando `function` entre parênteses, como feito na função `somatorio` a partir de `(function sqrt)`.

A função `somatorio` poderá ser usada para diversas ações sobre certa sequência de valores. Por exemplo, para apresentar o somatório dos quadrados dos valores de **1** a **5** será definida uma função simples que será usada como função de ordem superior chamada `quadrado`.

```
>>> (defun quadrado (NUM)
      (expt NUM 2))
```

Agora execute a instrução.

```
>>> (somatorio 1 5 (function quadrado))
55
```

Note que a função `somatorio` pode ser usada para a obtenção da soma de diversas operações baseadas em funções de ordem superior que utilizem um argumento.

Como comentado o paradigma de programação funcional considera sua computação a partir do uso de estruturas de conjuntos, onde conjunto é em essência uma coleção de elementos que possuem características comuns (contexto matemático).

Do ponto de vista matemático um conjunto é delimitado entre chaves, tendo seus elementos separados por vírgulas e associados a uma letra maiúscula. Em CL o conjunto será representado por uma coleção de valores delimitados entre parênteses, separados por espaço em branco na forma de listas que poderão ou não estar associados a uma variável.

As operações previstas pela teoria dos conjuntos são: pertinência, inclusão, união, intersecção, diferença e igualdade.

A pertinência indica se determinado elemento pertence ou não a certo conjunto. Essa ação pode ser realizada com o uso da função `find`, já conhecida, que retorna a indicação do valor se este existir na lista ou retorna o valor `NIL` quando o elemento não está contido.

Veja os exemplos.

```
>>> (find 3 '(1 2 3 4 5)) ; o elemento 3 pertence ao conjunto
3
```

```
>>> (find 6 '(1 2 3 4 5)) ; o elemento 6 nao pertence ao conjunto
NIL
```

A inclusão ocorre quando certo conjunto está contido em outro conjunto (ou seja, quando um conjunto com outro conjunto) e se certo conjunto não está contido em outro conjunto. Essa ação é realizada com o uso da função `subsetp` que verifica se o primeiro conjunto encontra-se contido no segundo conjunto retornando `T` se a operação for verdadeira ou `NIL` se a operação for falsa.

Veja os exemplos.

```
>>> (subsetp '(1 2) '(1 2 3)) ; conjunto 1 contido no conjunto 2
T
```

```
>>> (subsetp '(1 2 3) '(1 2)) ; conjunto 1 nao contido no conjunto 2
NIL
```

O resultado da união corresponde a junção dos elementos de dois conjuntos. Essa ação é realizada com o uso da função `union`.

Veja os exemplos no programa CLISP.

```
[n]> (union '(1 2 3) '(4 5 6))  
(1 2 3 4 5 6)
```

```
[n]> (union '(1 2 2 3 4) '(4 5 5 6))  
(1 2 2 3 4 5 5 6)
```

Veja os exemplos no programa SBCL.

```
* (union '(1 2 3) '(4 5 6))  
(3 2 1 4 5 6)
```

```
* (union '(1 2 2 3 4) '(4 5 5 6))  
(3 2 2 1 4 5 5 6)
```

O resultado da intersecção entre dois conjuntos corresponde aos elementos comuns aos conjuntos. Essa ação é realizada com o uso da função `intersection`.

Veja os exemplos no programa CLISP.

```
[n]> (intersection '(1 2 3 4 5) '(3 4 5 6))  
(3 4 5)
```

Veja os exemplos no programa SBCL.

```
* (intersection '(1 2 3 4 5) '(3 4 5 6))  
(5 4 3)
```

O resultado da diferença entre conjuntos é formado pelos elementos do primeiro conjunto que não estão presentes no segundo conjunto. Essa ação é realizada com o uso da função `set-difference`.

Veja os exemplos no programa CLISP.

```
[n]> (set-difference '(1 2 3 4) '(1 2))  
(3 4)
```



Veja os exemplos no programa SBCL.

```
* (set-difference '(1 2 3 4) '(1 2))  
(4 3)
```

A igualdade entre conjuntos ocorre quando os conjuntos possuem independentemente da ordem de disposição os mesmos elementos.

A linguagem CL não possui uma função específica para esta ação. Esta operação pode ser realizada a partir do uso adicional de outras funções de apoio, como o uso das funções `null` e `set-exclusive-or`.

A função `null` retorna `T` se a lista estiver vazia, caso contrário o retorno será `NIL`, sua ação assemelha-se a função `not` e a função `set-exclusive-or` possui funcionamento operacional contrário ao funcionamento da função `intersection`. Desta forma, esta função retorna como resposta uma lista dos elementos que não se repetem nas duas listas indicadas. Se as listas possuírem os mesmos elementos independentemente da ordem que estejam definidos retornará o valor `NIL`.

Veja os exemplos.

```
>>> (null (set-exclusive-or '(1 2 3) '(1 2 3)))  
T
```

```
>>> (null (set-exclusive-or '(3 2 1) '(1 3 2)))  
T
```

```
>>> (null (set-exclusive-or '(1 2 3) '(1 2 9)))  
NIL
```

A partir da visão sobre manipulação de listas e sua relação com a teoria de conjuntos incluindo-se as operações de pertinência, inclusão, união, interseção, diferença e igualdade é possível expandir essas operações com ações de filtragem, redução, mapeamento e transposição aplicados sobre conjuntos a partir do uso de funções, de um ponto de vista matemático.

Computacionalmente uma função, pode ser entendida como sendo uma rotina de programa ou a definição de uma expressão aritmética ou lógica que a partir de um argumento de entrada fornece sempre um valor de saída como resposta de sua ação.

Matematicamente uma função é uma maneira de expressar dois valores existentes em diferentes conjuntos. Desta forma, todos os elementos do primeiro conjunto, chamado domínio, devem ser relacionados aos valores do segundo conjunto, contradomínio. No entanto, pode acontecer de existir no segundo conjunto valores que não estejam relacionados com os elementos do primeiro conjunto.

Algebricamente uma função pode ser definida como  $f: A \rightarrow B$ , onde "efe" representa a relação existente entre (seta para à direita) os elementos do primeiro conjunto "A" com os elementos do segundo conjunto "B". A partir dessa relação tem-se a definição da operação representada por  $y = f(x)$  que será computacionalmente indicada como  $f(x) = y$ .

A partir da definição  $f: A \rightarrow B$  tem-se como domínio o conjunto "A" e contradomínio o conjunto "B". O conjunto domínio refere-se aos valores independentes que determinam a partir da regra algébrica  $y = f(x)$  os valores dependentes do conjunto contradomínio.

Poderá existir no conjunto contradomínio algum valor que não tenha relação com um valor do conjunto domínio, ou seja, não pertencente ao conjunto imagem definido a partir da função existente entre os conjuntos. Assim sendo, o conjunto imagem é por sua natureza um subconjunto do conjunto contradomínio possuindo os valores que correspondem diretamente aos valores do conjunto domínio.

A ação conhecida por **mapeamento** visa aplicar uma função a todos os elementos existentes em um conjunto domínio gerando a partir dessa ação um conjunto contradomínio com o resultado da operação definida como função. Essa ação é realizada com o uso da função `mapcar`.

Considerando um conjunto domínio com os elementos **1, 2, 3, 4 e 5** será definido o conjunto contradomínio com o triplo de cada elemento do domínio a partir da função  $f(x) = x \times 3$ . Veja a seguir a instrução que permite obter o conjunto imagem **3, 6, 9, 12 e 15**.

```
>>> (mapcar #'(lambda (x) (* x 3)) '(1 2 3 4 5))
(3 6 9 12 15)
```

A função `mapcar` é usada basicamente a partir de dois argumentos, sendo o primeiro a definição de uma função (geralmente do tipo `lambda`) e o segundo argumento a definição de uma lista de valores. Observe mais alguns exemplos.

```
>>> (mapcar #'(lambda (x) (expt x 2)) '(1 2 3 4 5))
(1 4 9 16 25)
```

```
>>> (mapcar #'(lambda (x) (* x -1)) '(1 2 3 4 5))
(-1 -2 -3 -4 -5)
```

Dada a função  $f: \{-2, 0, 2, 4, 6\} \rightarrow \{8, 6, 4, 2, 0, -2, -4, -6, -8\}$  apresentar seu conjunto imagem.

Observe a instrução executada em CLISP.

```
[n]> (intersection '(8 6 4 2 0 -2 -4 -6 -8)
  (mapcar #'(lambda (x) (- x (* 3 x))) '(-2 0 2 4 6)))
(4 0 -4 -8)
```

Observe a instrução executada em SBCL.

```
* (intersection '(8 6 4 2 0 -2 -4 -6 -8)
  (mapcar #'(lambda (x) (- x (* 3 x))) '(-2 0 2 4 6)))
(-8 -4 0 4)
```

A ação de **filtragem** visa aplicar uma função a todos os elementos existentes em um conjunto domínio gerando a partir dessa ação um conjunto contradomínio com o resultado da operação definida como função apenas para os resultados retornados verdadeiros. A linguagem CL não possui função específica, devendo então ser codificada uma função para esta operação. Assim sendo, considere o seguinte código.

```
>>> (defun filter (FUNCAO LISTA)
  (cond
    ((null LISTA) '())
    ((funcall FUNCAO (first LISTA))
     (cons (first LISTA) (filter FUNCAO (rest LISTA)))))
  (t (filter FUNCAO (rest LISTA)))))
```

A função `filter` recebe para operação os argumentos `FUNCAO` que será a indicação da função a ser aplicado sobre os elementos de uma lista representada pelo argumento `LISTA`. O trecho `((null LISTA) '())` em `cond` verifica se a lista é vazia (`null LISTA`) e se a condição for verdadeira retorna `NIL` a partir da indicação `'()`.

O trecho `((funcall FUNCAO (first LISTA))` verifica se o primeiro elemento existente na lista satisfaz como condição a função indicada em `FUNCAO`, sendo a condição verdadeira será criada uma lista com o elemento validado a partir do trecho

(cons (first LISTA) (filter FUNCAO (rest LISTA))), caso contrário será executado o trecho (t (filter FUNCAO (rest LISTA))) que recursivamente pega o restante da lista e o reaplica para novo teste.

Considerando um conjunto domínio com os elementos **1, 2, 3 e 4** será definido o conjunto contradomínio formado apenas pelos valores ímpares do domínio a partir da função  $f(x) = x - 2 \lfloor x / 2 \rfloor \mid x \sim 0$ . Veja a seguir a instrução que permite obter o conjunto imagem **1 e 3**.

```
>>> (filter #'oddp '(1 2 3 4))
(1 3)
```

Tomando por base o conjunto domínio  $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 0\}$  para obter o conjunto imagem a partir da filtragem formada pelos valores do domínio que sejam pares. Observe as seguintes instruções.

```
>>> (filter #'evenp '(1 2 3 4 5 6 7 8 9 0))
(2 4 6 8 0)
```

A ação conhecida por **redução** visa aplicar uma função a todos os elementos existentes em um conjunto domínio gerando a partir dessa ação um único valor como resposta. Essa ação é realizada com o uso da função reduce.

Considerando um conjunto domínio com os elementos **1, 2, 3 e 4** será definido o conjunto contradomínio de um elemento formado pela soma dos valores do domínio a partir da função  $f(x) = \sum x$ . Veja a seguir a instrução que permite obter o conjunto imagem **10**.

```
>>> (reduce #' + '(1 2 3 4))
10
```

A função reduce aceita o uso de qualquer função que opere com dois argumentos. Veja os próximos exemplos.

```
>>> (reduce #' * '(1 2 3 4))
24
```

```
>>> (reduce #' - '(1 2 3 4))
-8
```

```
>>> (float (reduce #'/ '(1 2 3 4)))  
0.041666668
```

A ação conhecida por **transposição** visa criar um conjunto formado com os elementos intercalados de outros conjuntos. A transposição ocorrerá com a junção dos primeiros elementos dos conjuntos informados e assim por diante até que acabe os elementos dos conjuntos. Se os conjuntos a serem transpostos forem de tamanho diferentes a transposição usará sempre como base o conjunto com menor número de elementos. Essa ação é feita com o uso das funções `mapcar` e `list`.

Considerando a definição de dois conjuntos, um contendo os elementos **1, 2 e 3** e outro contendo os elementos **4, 5 e 8** será definido o conjunto resultando formado pelos valores **(1 4) (2 5) e (3 6)**. Veja a seguir a instrução que permite obter a transposição dos conjuntos mencionados.

```
>>> (mapcar #'list '(1 2 3) '(4 5 6))  
((1 4) (2 5) (3 6))
```

Note que a operação de transposição faz a distribuição intercalada dos elementos um a um combinando-os em um conjunto resultante.

Veja alguns outros exemplos de transposição.

```
>>> (mapcar #'list '(1 2 3) '(4 6))  
((1 4) (2 6))
```

```
>>> (mapcar #'list '(1 2 3) '(4 6 5 7))  
((1 4) (2 6) (3 5))
```

Os exemplos anteriores mostram os resultados que são obtidos quando se opera com conjuntos de tamanho diferentes.





---

## Referências bibliográficas

---

AVELINO, T. **História do Lisp: abra os olhos para programação funcional.**

iMasters.com, 2018. Disponível em: <<https://imasters.com.br/desenvolvimento/historia-lisp-abra-os-olhos-para-programacao-funcional>>. Acesso em: 17 mar. 2019.

BARSKI, C. **Land of LISP: Learn to program in Lisp, one game at a time.** San Francisco: No Starch Press, 2010.

BERKELEY, E. C. & BOBROW, D. G. **The programming language LISP: Its operation and applications.** 3th. Massachusetts: MIT Press, 1974.

GRAHAM, P. **ANSI Common Lisp.** New Jersey: Prentice-Hall, 1995.

\_\_\_\_\_. **OnLisp: Advanced techniques for COMMON LISP.** New Jersey: Prentice-Hall, 1993.

KEENE, S. E. **Object-oriented programming in Common LISP: A programmer's guide to CLOS.** New York: Addison-Wesley, 1989.

LÉVÉNEZ, É. **Computer Languages History.** Paris: VIERLING, 2018. Disponível em: <<https://www.levenez.com/lang/>>. Acesso em: 17 mar. 2019.

LIPSCHUTZ, S.; LIPSON, M. **Matemática discreta.** 3. ed. Porto Alegre: Bookman, 2013.

McCARTHY, John. **LISP 1.5 Programmer's Manual.** Massachusetts: MIT Press, 1962.

McJONES, P. **The LISP 2 Project.** Washington: IEEE Computer Society, 2017.

MIRANKER, D. P. **TREAT: A new and efficient match algorithm for AI production systems.** California: Morgan Kaufmann Publishers, Inc. 1990.

MOON, D. A. **MacLISP reference manual.** Massachusetts: MIT Press, 1974.

NORVIG, P. **Paradigms of artificial intelligence programming: Case studies in COMMON LISP**. California: Morgan Kaufmann Publishers, Inc. 1992.

QUARESMA, P. **ANSI and GNU Common Lisp Document**. Coimbra: DEMUC - Departamento Matemática Universidade de Coimbra, 2018. Disponível em: <[http://www.mat.uc.pt/~pedro/cientificos/funcional/lisp/gcl\\_toc.html](http://www.mat.uc.pt/~pedro/cientificos/funcional/lisp/gcl_toc.html)>. Acesso em: 17 mar. 2019.

SEIBEL, P. **Practical Common LISP**. New York: Apress, 2005.

SHAPIRO, S. C. **Common LISP: An Interactive Approach (Principles of Computer Science Series)**. New York: W H Freeman & Co, 1991.

STEELE Jr. G. L. **Common LISP: The language**. Massachusetts: Digital Press, 1990.

SYMBOLICS. **Symbolics Common Lisp: Language Concepts**. Massachusetts: Symbolics, Inc., 1986.

TEITELMAN, W. **InterLISP reference manual**. California: XEROX, 1974.

TIERNEY, L. **LISP-STAT: An object-oriented environment for statistical computing and dynamic graphics**. New York: John Wiley & Sons, 1990.

TOURETZKY, D. S. **Common LISP: A gentle introduction to symbolic computation**. California: The Benjamin/Cummings, 2013.

WEITZ, E. **Common LISP Recipes: A problem-solution approach**. New York: Apress, 2016.

YUASA, T. & HAGIYA, M. **Introduction to Common LISP**. California: Morgan Kaufmann. 1987.





