
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 iniciado 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")  
"abcdefghijklm"
```

```
>>> (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-capitalize. Observe os seguintes exemplos.

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

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

```
>>> (string-capitalize "linguagem lisp")
"Language 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")
language lisp
NIL
```

```
>>> (format t "~:(~a~)" "linguagem lisp")
Language 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")
Language 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 megaugnîL"
```

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 posam 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)
"Language 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 desfeita. 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 BSCL.

```
>>> (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 operacionalizada 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;
- \geq função para: maior ou igual a;
- \leq 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-insensitive**).

```
>>> (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></code>	função para: maior que com case-sensitive;
<code>string-greaterp</code>	função para: maior que sem case-sensitive;
<code>string<</code>	função para: menor que com case-sensitive;
<code>string-lessp</code>	função para: menor que sem case-sensitive;
<code>string>=</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<=</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.

equal	igualdade de dados estruturalmente similares;
eq	igualdade de dados idênticos;
eql	igualdade de dados semelhantes em tipo e valor;
equap	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
```

```
>>> (eql 3 3)
```

```
T
```

```
>>> (eql 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 auxilio 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 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 JÁN 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 formam 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, trilha, 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***:

case	construtor que opera com decisão seletiva;
cond	construtor que verifica várias cláusulas de teste;
if	macro que opera decisão composta;
unless	macro que opera decisão simples com fluxo falso;
when	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 (recesso 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 ímpares 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 numéricos” se dados dois argumentos estes forem numéricos e caso um dos argumentos ou ambos não sejam numéricos apresentar a mensagem “Valores não numéricos”. 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 numéricos")
```

```
    (format t "~%Valores não numéricos"))
```

```
Valores numéricos
```

```
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 retornando 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 interativo ou iterativo. 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** decrescendo 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)>
  <blocos>)
```

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 partindo 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 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 retorna 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** **I** (* **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 factorial2 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 factorial1 no programa CLISP.

[n]> (**fatorial1 2486**)

Será apresentado o resultado da operação. Na sequência execute a instrução com a chamada da função factorial2 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 factorial1 no programa SBCL.

* (**fatorial1 64934**)

Será apresentado o resultado da operação. Na sequência execute a instrução com a chamada da função factorial2 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 factorial3 com uso do efeito de recursividade por cauda.

```
>>> (defun factorial3 (N &optional (BASE 1))
  (if (< N 0)
      (format t "Erro~%")
      (if (and (>= N 0) (< N 2))
          BASE
          (factorial3 (- N 1) (* BASE N)))))
```

A partir da definição da função factorial3 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.

```
>>> (factorial3 65000)
```

O resultado apresentado é gerado a partir da execução de uma função recursiva, forma mais elegante. Na função factorial3 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 factorial3 com o argumento obrigatório (- N 1) operando com o argumento opcional (* BASE N).

Após a execução da função factorial3 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 factorial3 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 factorial3.

```
[n]> (trace factorial3)
;; Tracing function FATORIAL3.
(FATORIAL3)
```

Observe a ativação do modo depuração no programa SBCL para a função recursiva factorial3.

```
[n]> (trace factorial3)
(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 factorial3 no programa CLISP.

```
[n]> (factorial3 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 factorial3 no programa SBCL.

```
* (factorial3 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.

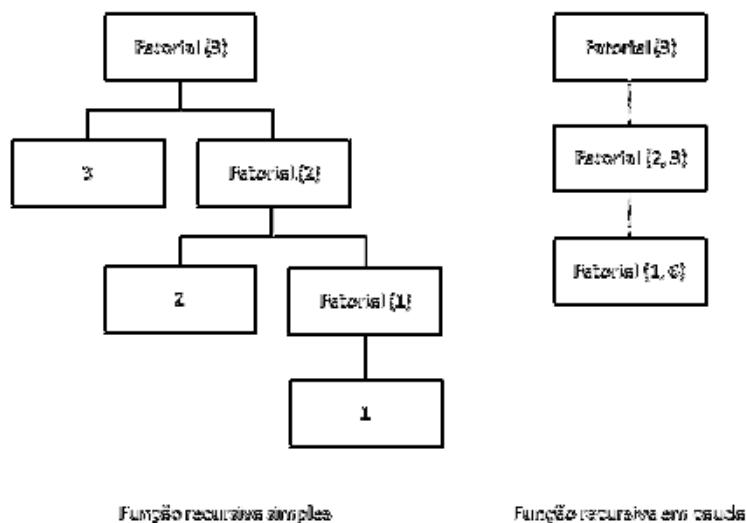


Figura 3.1 - Função simples e cauda: comportamento em memória.

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 `fibonacci2` 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 `fibonacci3` 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.

