
4

Recursos avançados

Neste capítulo são apresentados os tipos de dados matrizes, tabelas de símbolos (hash) e estruturas. É demonstrado o uso de ações de entrada de dados de forma interativa e de novas possibilidades de saída de dados. É indicado o uso de variáveis locais e a definição de macros. Destaque é dado ao uso de funcionalidades de tempo. É também apresentado o uso de funções anônimas.

4.1 TIPO DE DADO MATRIZ

Matrizes (**arrays**) são tipos de dados que podem ser definidos com uma ou múltiplas dimensões. Matriz é uma estrutura de dados contigua disposta na memória tendo seu endereço mais baixo como primeiro elemento e o endereço mais alto como último elemento, cada posição de armazenamento de certo elemento em uma matriz é chamado de **slot**. Os **slots** de uma matriz são gerenciados a partir da manipulação de índices formados por valores inteiros e positivos iniciados a partir de zero.

Uma matriz de uma dimensão é normalmente referenciada como vetor, com duas dimensões normalmente é referenciada como tabela. A partir de três dimensões usa-se como referência o termo matrizes multidimensionais.

A definição de matriz é realizada com o uso da função `make-array` a partir da sintaxe.

```
(make-array '<tamanho1> [<tamanho2> [<tamanhoN>]] [:chaves])
```

Onde, a indicação **tamanho1** refere-se a definição da quantidade de elementos como linhas para uma matriz de uma dimensão (vetor), **tamanho2** refere-se a definição da quantidade de elementos como colunas para uma matriz de duas

dimensões (tabela), **tamanhoN** refere-se a definição da quantidade de elementos como páginas para uma matriz de três dimensões (multidimensional) e assim por diante. O argumento opcional **chaves** refere-se a um conjunto de opções adicionais que podem ser estabelecidas para a definição de matrizes.

O conjunto de chaves que podem ser definidas na criação de matrizes são:

- **:element-type** - chave que especifica um tipo de dado para os elementos de uma matriz, o valor padrão é T especificando qualquer tipo de dado;
- **:initial-element** - chave que define valores iniciais para todos os elementos de uma matriz;
- **:initial-contents** – chave que define os valores iniciais de partes dos elementos que formam uma matriz;
- **:adjustable** - chave que auxilia a criação de matrizes redimensionáveis cuja memória subjacente pode ser automaticamente redimensionada, tendo como seu último elemento padrão o valor NIL. Esta chave é usada em conjunto com a chave **:fill-pointer**;
- **:fill-pointer** - chave que monitora o número de elementos armazenados em uma matriz redimensionável. Esta chave deve ser usada obrigatoriamente em conjunto com a chave **:adjustable**;
- **:displaced-to** - chave que auxilia a criação de matrizes com elementos compartilhados. Ambas as matrizes devem ter o mesmo tipo de elemento. Esta chave não é usada com as chaves **:initial-element** ou **:initial-contents**;
- **:displaced-index-offset** - chave que fornece o deslocamento do índice para uma matriz mapeada a partir de outra matriz a ela compartilhada. Para esta chave ser operada é preciso estar em uso a chave **:displaced-to**.

A função `make-array` para ser usada deve ser operada com as funções `setf` para criar a matriz na memória e `aref` para acessar os elementos da matriz, ambas já usadas.

Como exemplo, considere a definição de uma matriz de uma dimensão chamada ***MATRIZ-1D*** que armazene os quadrados dos valores inteiros de **10** a **20**. Atente para as etapas apresentadas.

Execute a instrução seguinte no programa CLISP.

```
[n]> (setf *MATRIZ-1D* (make-array '(10)))
#(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
```

Execute a instrução seguinte no programa SBCL.

```
* (defvar *MATRIZ-1D*)
*MATRIZ-1D*

* (setf *MATRIZ-1D* (make-array '(10)))
#(0 0 0 0 0 0 0 0 0 0)
```

A partir da definição da matriz ***MATRIZ-1D*** em memória é necessário definir as instruções que farão o cálculo dos quadrados de **10** a **20** dentro da matriz.

```
>>> (loop for I from 10 to 19
      do (setf (elt *MATRIZ-1D* (- I 10)) (expt I 2)))
NIL

>>> *MATRIZ-1D*
#(100 121 144 169 196 225 256 289 324 361)
```

No laço estabelecido a função `elt` (anteriormente usada para acessar determinado caracteres de uma cadeia) indica a matriz ***MATRIZ-1D*** a posição de acesso aos **slots** da matriz por meio da função `(- I 10)` para que a primeira contagem do laço em **10** seja subtraída e acesse o primeiro **slot** da matriz, ou seja, a posição zero. Isso é repetido até chegar ao valor **19** que dará acesso ao **slot 9**.

A princípio a estrutura de uma matriz e de uma lista diferenciam-se pelo uso do símbolo (`'`) aspas simples nas listas e (`#`) trilha nas matrizes, mas essa diferenciação é estética. Internamente a diferença entre listas e matrizes são mais acentuadas.

Uma lista é constituída de células `cons` e `nil`, sendo uma estrutura dinâmica de dados, uma vez que pode receber elementos sem se preocupar de antemão com seu tamanho. Para acessar, por exemplo, o último elemento é necessário percorrer sequencialmente a lista toda. Adicionar e remover certo elemento na frente da lista é uma tarefa barata, mas a adição ou remoção de elementos em outras posições

são mais caras e por conseguinte mais lentas. As listas podem sofrer sobrecarga de espaço na memória, pois cada elemento é armazenado em uma célula cons.

Uma matriz é uma estrutura de dados estática formada por posições numeradas (**slots**) que apesar de permitir que seu tamanho seja ajustado durante o uso exige que os elementos existentes sejam recopiados para o novo tamanho, demandando mais tempo de processamento. Adicionar elementos é uma ação que se torna caro se a matriz para receber uma nova entrada necessita ter seu tamanho reajustado o que pode sobrecarregar o espaço de memória. Acessar elementos de uma matriz é muito barato uma vez que o acesso é definido a partir do índice de posição do **slot** desejado.

Considerando redefinir a matriz ***MATRIZ-1D*** de **10** para **15** elementos é necessário utilizar a função **adjust-array** que possui estrutura de uso idêntica a estrutura da função **make-array**, devendo esta função ser usada em conjunto com a função **setf**.

Execute a instrução seguinte no programa CLISP.

```
[n]> (setf *MATRIZ-1D* (adjust-array *MATRIZ-1D* '(15)))
#(100 121 144 169 196 225 256 289 324 361 NIL NIL NIL NIL NIL)
```

Execute a instrução seguinte no programa SBCL.

```
* (setf *MATRIZ-1D* (adjust-array *MATRIZ-1D* '(15)))
#(100 121 144 169 196 225 256 289 324 361 0 0 0 0 0)
```

A partir da definição do novo tamanho da matriz ***MATRIZ-1D*** é necessário proceder ao cálculo anteriores nas posições seguintes. Neste caso execute as instruções seguintes.

```
>>> (loop for I from 20 to 24
      do (setf (elt *MATRIZ-1D* (- I 10)) (expt I 2)))
NIL

>>> *MATRIZ-1D*
#(100 121 144 169 196 225 256 289 324 361 400 441 484 529 576)
```

A partir da estrutura ***MATRIZ-1D*** observe a instrução seguinte que apresenta os elementos da matriz e suas posições.

```
>>> (loop for I from 0 to 14
      do (format t "*MATRIZ-1D*[~2d] = ~a~%" I (elt *MATRIZ-1D* I)))
*MATRIZ-1D*[ 0] = 100
*MATRIZ-1D*[ 1] = 121
*MATRIZ-1D*[ 2] = 144
*MATRIZ-1D*[ 3] = 169
*MATRIZ-1D*[ 4] = 196
*MATRIZ-1D*[ 5] = 225
*MATRIZ-1D*[ 6] = 256
*MATRIZ-1D*[ 7] = 289
*MATRIZ-1D*[ 8] = 324
*MATRIZ-1D*[ 9] = 361
*MATRIZ-1D*[10] = 400
*MATRIZ-1D*[11] = 441
*MATRIZ-1D*[12] = 484
*MATRIZ-1D*[13] = 529
*MATRIZ-1D*[14] = 576
NIL
```

Como indicado existem algumas chaves que auxiliam a inicialização de matrizes, entre as chaves existentes a chave :initial-element em particular permite iniciar uma matriz com determinados valores. Atente para os exemplos seguintes que mostram como inicializar os elementos de uma matriz com a função make-array.

```
>>> (defvar *X1-1D*)
*X1-1D*

>>> (setf *X1-1D* (make-array '(5) :initial-element 'a))
#(A A A A A)

>>> (defvar *X2-1D*)
*X2-1D*
```

```

>>> (setf *X2-1D* (make-array '(5)
                                :element-type 'character :initial-element #\a))
"aaaaa"

>>> (defvar *X3-1D*)
*X3-1D*

>>> (setf *X3-1D* (make-array '(5)
                                :element-type 'bit :initial-element 1))
#*11111

```

A partir de uma visão básica sobre a definição de matrizes com uma dimensão veja um exemplo para a definição de uma matriz de duas dimensões. Assim sendo, considere definir uma matriz com três linhas e quatro colunas inicializada a partir de uma lista com todas as posições com valor 1.

```

>>> (defvar *MATRIZ-2D*)
*MATRIZ-2D*

>>> (setf *MATRIZ-2D* (make-array '(3 4) :initial-element '1))
#2A((1 1 1 1) (1 1 1 1) (1 1 1 1))

```

Veja que a ***MATRIZ-2D*** é formada por um conjunto de três listas (as linhas), onde cada lista é inicializada com quatro posições (colunas) indicadas com a definição do valor 1. Observe a apresentação dos elementos e as posições da matriz.

```

>>> (loop for I from 0 to 2 do
         (loop for J from 0 to 3
               do (format t
                           "*MATRIZ-2D*[~d,~d] = ~a~%" I J (aref *MATRIZ-2D* I J))))
*MATRIZ-2D*[0,0] = 1
*MATRIZ-2D*[0,1] = 1
*MATRIZ-2D*[0,2] = 1
*MATRIZ-2D*[0,3] = 1
*MATRIZ-2D*[1,0] = 1
*MATRIZ-2D*[1,1] = 1

```

```
*MATRIZ-2D*[1,2] = 1
*MATRIZ-2D*[1,3] = 1
*MATRIZ-2D*[2,0] = 1
*MATRIZ-2D*[2,1] = 1
*MATRIZ-2D*[2,2] = 1
*MATRIZ-2D*[2,3] = 1
NIL
```

Atente para o uso da função `aref` ao invés do uso da função `elt`. Esta mudança foi definida devido ao fato da função `elt` não permitir o uso de mais de dois argumentos em seu escopo de operação. Devido a esta questão talvez seja mais vantajoso usar a função `aref`.

A partir dos valores existentes em ***MATRIZ-2D*** definir para cada valor da matriz a soma deste com o valor **2** de modo que a matriz tenha em suas posições o valor **3**. Observe a instrução seguinte.

```
>>> (dotimes (I 3)
      (dotimes (J 4)
        (setf (aref *MATRIZ-2D* I J) (+ (aref *MATRIZ-2D* I J) 2))))
NIL
```

```
>>> (loop for I from 0 to 2
      do (loop for J from 0 to 3
              do (format t
                         "*MATRIZ-2D*[~d,~d] = ~a~%" I J (aref *MATRIZ-2D* I J)))
*MATRIZ-2D*[0,0] = 3
*MATRIZ-2D*[0,1] = 3
*MATRIZ-2D*[0,2] = 3
*MATRIZ-2D*[0,3] = 3
*MATRIZ-2D*[1,0] = 3
*MATRIZ-2D*[1,1] = 3
*MATRIZ-2D*[1,2] = 3
*MATRIZ-2D*[1,3] = 3
*MATRIZ-2D*[2,0] = 3
*MATRIZ-2D*[2,1] = 3
```

```
*MATRIZ-2D*[2,2] = 3  
*MATRIZ-2D*[2,3] = 3  
NIL
```

Quando se tem matrizes definidas em memória é possível verificar a quantidade de elementos que a matriz armazena a partir do uso da função `array-dimensions` que possui sintaxe semelhante a forma simplificada da função `make-array`. Observe em seguida as instruções que permitem apresentar os valores máximos existente nas dimensões das matrizes em uso.

```
>>> (array-dimensions *MATRIZ-1D*)  
(15)
```

```
>>> (array-dimensions *MATRIZ-2D*)  
(3 4)
```

Um recurso que pode ser aplicado é o uso da função `array-total-size` que apresenta o total de elementos de uma matriz independentemente da dimensão definida. A função `array-total-size` possui sintaxe semelhante a forma simplificada da função `make-array`. Observe em seguida as instruções que permitem apresentar a quantidade de elementos das matrizes em uso.

```
>>> (array-total-size *MATRIZ-1D*)  
15
```

```
>>> (array-total-size *MATRIZ-2D*)  
12
```

A quantidade de dimensões de uma matriz pode ser obtida por meio do uso da função `array-rank` que possui sintaxe semelhante a forma simplificada da função `make-array`. Observe em seguida as instruções que permitem apresentar a quantidade de dimensões das matrizes em uso.

```
>>> (array-rank *MATRIZ-1D*)  
1
```

```
>>> (array-rank *MATRIZ-2D*)  
2
```

Cabe apontar que a definição de matrizes é realizada como variáveis globais. Variáveis globais são criadas pelas funções `defvar` e `sef` (que particularmente mostra mensagem de advertência no programa SBCL). Mas, além dessas funções há outra forma de definir variáveis globais pela função `defparameter`.

As funções `defparameter` e `defvar` são semelhantes e realizam as mesmas operações, tendo como diferença o fato de que a função `defparameter` exige a inicialização de um valor associado a variável global. Seu uso não indica nenhuma mensagem de advertência em nenhum dos interpretadores CL.

Observe em seguida as instruções para definição de variável global como matriz a partir do uso da função `defparameter`.

```
>>> (defparameter *MATRIZ-X1* #(1 2 3 4 5 6))
>>> *MATRIZ-X1*
#(1 2 3 4 5 6)

>>> *MATRIZ-X1*
#(1 2 3 4 5 6)

>>> (defparameter *MATRIZ-X2* #2A((1 2 3) (4 5 6)))
*MATRIZ-X2*
#2A((1 2 3) (4 5 6))
```

A partir de uma visão básica da definição e da especificação de algumas operações em matrizes é pertinente apresentar a funcionalidade das demais chaves de definição.

Considere definir uma matriz que seja inicializada com dados de um tipo específico. Veja a seguir os exemplos para criação de uma matriz de bits inicializada com valor 1 e uma matriz de caracteres inicializada com a letra “a”.

```
>>> (defvar *MAT-BIT*)
*MAT-BIT*
```

```
>>> (setf *MAT_BIT* (make-array '(5)
  :element-type 'bit
  :initial-element 1))
#*11111

>>> (defvar *MAT-CHAR*)
*MAT-CHAR*

>>> (setf *MAT_CHAR* (make-array '(5)
  :element-type 'character
  :initial-element #\a))
"aaaaa"
```

A chave :initial-element é usada para inicializar elementos de uma matriz de qualquer dimensão, desde que todos os elementos tenham o mesmo valor. Para matrizes multidimensionais essa chave pode não atender.

Imagine a criação de uma matriz com duas linhas e três colunas, contendo os valores da primeira linha como **1** e os elementos da segunda linha como **2**. Observe as instruções a seguir.

```
>>> (defvar *2X3*)
*2X3*

>>> (setf *2X3* (make-array '(2 3)
  :initial-contents '((1 1 1) (2 2 2))))
#2A((1 1 1) (2 2 2))
```

O ajuste do tamanho de uma matriz após sua definição sem a necessidade de usar a função adjust-array pode ser definido previamente com a chave :adjustable que para surtir o efeito necessita ser operacionalizada em conjunto com a chave :fill-pointer que monitora o número de elementos armazenados em uma matriz redimensionável

Considerando a definição de uma matriz chamada ***MAT-ADJ*** com a capacidade de armazenar inicialmente três elementos e que possa essa ser ajustada para mais elementos de forma automática sem o uso da função adjust-array deve ser definida a partir da instrução.

```
>>> (defvar *MAT-ADJ*)  
*MAT-ADJ*  
  
>>> (setf *MAT-ADJ* (make-array '(3)  
:adjustable t  
:fill-pointer 0  
:initial-element 0))  
#(0 0 0)
```

A partir da criação da matriz com a ativação da chave :adjustable com valor T de verdadeiro é possível, quando necessário, executar ações que ampliem o número de posições da matriz.

Assim, sendo para efetuar a entrada de dados na matriz ***MAT-ADJ*** execute as instruções seguintes a partir do uso da função vector-push que após inserir o conteúdo dentro da matriz retorna o valor da posição do **slot** usado para o armazenamento.

```
>>> (vector-push '1 *MAT-ADJ*)  
0  
  
>>> (vector-push '2 *MAT-ADJ*)  
1  
  
>>> (vector-push '3 *MAT-ADJ*)  
2  
  
>>> *MAT-ADJ*  
#(1 2 3)
```

Observe que foram inseridos na matriz três valores e se realizada uma nova tentativa de entrada de um quarto valor com a função vector-push ocorrerá um erro na entrada indicando o valor NIL, ou seja, a entrada não foi efetivada. Veja esta ocorrência.

```
>>> (vector-push '4 *MAT-ADJ*)  
NIL
```

Para poder realizar a entrada de mais um elemento em uma matriz redimensionável de forma automática use a função `vector-push-extend` ao invés da função `vector-push`. Veja a próxima instrução.

```
>>> (vector-push-extend '4 *MAT-ADJ*)  
3
```

```
>>> *MAT-ADJ*  
#(1 2 3 4)
```

Observe que as funções `vector-push-extend` e `vector-push` realizam a entrada de valores no final da matriz.

Para retirar elementos do final de uma matriz use a função `vector-pop`. Observe as instruções seguintes.

```
>>> (vector-pop *MAT-ADJ*)  
4
```

```
>>> (vector-pop *MAT-ADJ*)  
3
```

```
>>> *MAT-ADJ*  
#(1 2)
```

Outra ação possível de ser realizada é a de definir matrizes compartilhadas a partir do uso da chave `:displaced-to`.

Considere a definição de duas matrizes compartilhadas, sendo a ***MAT-DIS-1*** e a matriz ***MAT-DIS-2*** que quando os dados da matriz ***MAT-DIS-1*** forem alterados serão refletidos na matriz ***MAT-DIS-2***.

```
>>> (defvar *MAT-DIS-1*)  
*MAT-DIS-1*
```

```
>>> (defvar *MAT-DIS-2*)  
*MAT-DIS-2*
```

```
>>> (setf *MAT-DIS-1* (make-array '(3) :initial-element 0))
#(0 0 0)

>>> (setf *MAT-DIS-2* (make-array 3 :displaced-to *MAT-DIS-1*))
#(0 0 0)

>>> *MAT-DIS-1*
#(0 0 0)

>>> *MAT-DIS-2*
#(0 0 0)

>>> (setf (aref *MAT-DIS-1* 0) 4)
4

>>> *MAT-DIS-1*
#(4 0 0)

>>> *MAT-DIS-2*
#(4 0 0)
```

Além de vincular matrizes a função :displaced-to permite alterar a estrutura de dimensão de uma matriz. Assim sendo, considere a definição de uma matriz de uma dimensão com quatro elementos e sua redefinição como sendo uma matriz de duas dimensões com duas linhas e duas colunas.

```
>>> (defvar *MAT-TRANS-1*)
*MAT-TRANS-1*

>>> (defvar *MAT-TRANS-2*)
*MAT-TRANS-1*

>>> (setf *MAT-TRANS-1* (make-array '(4)
                                         :initial-contents '(1 2 3 4)))
#(1 2 3 4)
```

```
>>> (setf *MAT-TRANS-2* (make-array '(2 2) :displaced-to *MAT-TRANS-1*))  
#2A((1 2) (3 4))  
  
>>> *MAT-TRANS-2*  
#2A((1 2) (3 4))
```

Para transformar uma matriz de duas dimensões em uma dimensão basta fazer o uso de ação inversa. Teste a instrução.

```
>>> (defvar *MAT-TRANS-3*)  
*MAT-TRANS-3*  
  
>>> (setf *MAT-TRANS-3* (make-array '(4) :displaced-to *MAT-TRANS-2*))  
#(1 2 3 4)  
  
>>> *MAT-TRANS-3*  
 #(1 2 3 4)
```

A transformação de matrizes de uma dimensão em duas dimensões ou vice versa é possível uma vez que matrizes unidimensionais são armazenadas na memória como sendo uma matriz de uma dimensão na ordem das linhas existentes.

A definição de matrizes pode ser realizada com a chave :displaced-index-offset que para ser usada necessita do uso prévio da chave :displaced-to. A chave :displaced-index-offset usa um valor inteiro positivo que determina o valor de deslocamento do índice para acesso a matriz copiada a partir do deslocamento matriz principal.

Observe a seguir as instruções para criação da matriz principal ***MAT-IND-1*** com os valores de 1 até 5 e da matriz ***MAT-IND-2*** que conterá os três últimos valores da matriz ***MAT-IND-1***.

```
>>> (defvar *MAT-IND-1*)  
*MAT-IND-1*  
  
>>> (setf *MAT-IND-1* (make-array '(5)  
:initial-contents '(1 2 3 4 5)))  
#(1 2 3 4 5)
```

```
>>> (defvar *MAT-IND-2*)
*MAT-IND-2*

>>> (setf *MAT-IND-2* (make-array '(3
:displaced-to *MAT-IND-1*
:displaced-index-offset 2))
#(3 4 5)
```

Note que o trecho :displaced-index-offset 2 cria a matriz ***MAT-IND-2*** a partir da matriz ***MAT-IND-1*** considerando todos os elementos da matriz ***MAT-IND-1*** a partir do índice 2.

Quando se utiliza a chave :displaced-index-offset as matrizes não são comparadas como ocorre com o uso isolado da chave :displaced-to.

A partir de uma visão básica sobre a definição e uso essencial de matrizes é importante conhecer algumas funcionalidades operacionais para seu gerenciamento em memória como contar elementos, localizar elementos, remover elementos, substituir elementos, entre outras operações.

Observe as instruções seguintes.

```
>>> (length (vector 1 2 3 4 5))
5

>>> (length #(1 2 3 4 5))
5

>>> (count 3 #(1 2 3 4 3 2 1 0 3 6 1 3))
4

>>> (remove 0 #(0 1 0 2 0 3 0 4 0 5))
#(1 2 3 4 5)

>>> (substitute 99 3 #(1 2 3 4 3 2 1))
#(1 2 99 4 99 2 1)
```

```
>>> (find 3 #(1 2 3 4 3 2 1))  
3  
  
>>> (position 4 #(1 2 3 4 5 4 3 2 1))  
3  
  
>>> (remove-duplicates #(1 2 3 4 3 5 3 3 6))  
#(1 2 4 5 3 6)
```

As funções `length`, `remove`, `substitute` e `find` são conhecidas e operam da forma já orientada. A função `vector` define uma matriz de uma dimensão (vetor) cujo tamanho corresponde ao número de elementos estabelecidos. Já a função `count` efetua a contagem do elemento indicado dentro de uma matriz retornando a quantidade de vezes que o elemento se encontra. Caso o elemento não seja encontrado o valor 0 é retornado. A função `position` retorna o valor cardinal do índice em que se encontra o primeiro elemento conforme o elemento indicado para pesquisa. Se nada for localizado é retornado o valor NIL e a função `remove-duplicates` retira de uma matriz (ou lista) todos os elementos que se repetem na estrutura.

As funções `count`, `find`, `remove` e `substitute` possuem variações condicionais que podem auxiliar algumas operações de uso, como `count-if`, `find-if`, `remove-if` e `substitute-if`. Assim sendo, observe os exemplos seguintes.

```
>>> (count-if #'evenp #(1 2 3 4 5)) ; quantidade de pares  
2  
  
>>> (count-if #'oddp #(1 2 3 4 5)) ; quantidade de ímpares  
3  
  
>>> (remove-if #'oddp #(1 2 3 4 5)) ; remove ímpares  
#(2 4)  
  
>>> (remove-if #'evenp #(1 2 3 4 5)) ; remove pares  
#(1 3 5)  
  
>>> (find-if #'evenp '(1 4 3 2 5)) ; localiza primeiro par  
4
```

```
>>> (find-if #'oddp '(2 4 3 1 5)) ; localiza primeiro ímpar  
3  
  
>>> (substitute-if 99 #'oddp '(1 2 3 4 5)) ; substitui ímpares por 99  
(99 2 99 4 99)
```

As funções `count-if`, `find-if`, `remove-if` e `substitute-if` possuem como variações opostas as funções `count-if-not`, `find-if-not`, `remove-if-not` e `substitute-if-not`.

Uma ação que também pode ser aplicada sobre matrizes é o efeito de junção com a função `merge`. Assim sendo, observe as instruções seguintes.

```
>>> (merge 'vector '(1 3 5) '(2 4 6) #'<)  
#(1 2 3 4 5 6)  
  
>>> (merge 'vector #(1 3 5) '(2 4 6) #'=)  
#(1 3 5 2 4 6)  
  
>>> (merge 'vector #(1 3 5) #(2 4 6) #'>)  
#(2 4 6 1 3 5)
```

A ação de junção com a função `merge` aceita que as coleções de elementos indicadas para uso sejam tanto matrizes como listas. Anteriormente para mesclar coleções de listas em lista usou-se o tipo `list`. Para mesclar coleções de listas e/ou matrizes em matrizes (**arrays**) usa-se o tipo `vector`.

Há casos em que o número de dimensões de uma matriz não é importante e neste sentido podem ser usadas as funções `row-major-aref` que acessa os elementos de uma matriz multidimensional usando um único índice e `array-total-size` para detectar o tamanho da matriz.

Veja as instruções seguintes.

```
>>> (defparameter *MAT2D* #2A((1 2 3) (4 5 6) (7 8 9)))  
*MAT2D*
```

```
>>> (loop for I from 0 below (array-total-size *MAT2D* )
      do (format t "~a " (row-major-aref *MAT2D* I)))
1 2 3 4 5 6 7 8 9
NIL
```

Veja que a função `array-total-size` é usada para delimitar a quantidade de iterações que é realizada sobre todos os elementos da matriz ***MAT2D*** acessados pela função `row-major-aref`.

Os dois tipos de dados mais populares são as listas e as matrizes. Essas formas de representação podem ser intercambiadas a partir de ações de coerção (**casting**) transformando listas em matrizes e vice versa a partir da função `coerce`.

Observe os exemplos seguintes mostram a conversão de uma lista em matriz e de uma matriz em lista.

```
>>> (coerce '(A B C D E) 'vector)
#(A B C D E)

>>> (coerce #(A B C D E) 'list)
(A B C D E)
```

No conjunto de funções para a manipulação de listas e de matrizes havendo a eventualidade de uma função não poder ser usada com uma lista ou uma matriz a conversão passa a ser uma ação desejada. Isso faz com que os recursos possam ser amplamente utilizados.

4.2 ENTRADA E SAÍDA DE DADOS

As operações de saída (exibição) e entrada (leitura) de dados são ações de conversão de dados manipulados pela linguagem na forma textual e vice-versa.

Algumas operações de saída foram apresentadas anteriormente a partir do uso das funções `write` que envia a saída sem pular linha antes da impressão mostrando o retorno do valor em uso, se numérico mostra o valor numérico e se cadeia ou caractere mostra conteúdo entre aspas; `print` que envia a saída pulando uma linha antes da impressão mostrando o retorno do valor em uso, se numérico mostra o valor numérico e se cadeia ou caractere mostra conteúdo entre aspas e `format` que

produz como saída uma estrutura de texto formatada. No entanto, há outras funções de saída que podem ser usadas, destacando-se as principais:

pprint	saída sem indicar retorno e pula linha antes;
prin1	saída com salto de linha ao final;
princ	saída sem salto de linha ao final;
terpri	saída com um linha em branco.

Observe alguns exemplos de uso das principais funções de saída.

```
>>> (prin1 "ABC")
"ABC"
"ABC"

>>> (pprint "ABC")
<pula linha em branco>
"ABC"

>>> (princ "ABC")
ABC
"ABC"

>>> (terpri)
<pula linha em branco>
NIL
```

Além das funcionalidades que operam ações de saída há a função `read` que permite realizar a entrada de dados de forma interativa. Observe o exemplo seguinte, no qual dever ser fornecido um valor após a execução da ação. Assim que o valor é fornecido o mesmo é imediatamente apresentado.

```
>>> (write (read))
abc
ABC
ABC
```

A partir das ações de entrada e saída é possível criar funções interativas que proporcionam melhor operação com um usuário. Considere uma função que apresente o resultado da área de uma circunferência.

```
>>> (defun area-circ()
  (princ "Entre o valor do raio: ")
  (finish-output)
  (defparameter raio (read))
  (princ "Área: ")
  (write (* (expt raio 2) pi)))  
  
>>> (area-circ)  
Entre o valor do raio: 3  
Área: 28.27433882308139147L0  
28.27433882308139147L0
```

Note que na definição da função `area-circ` está sendo usada a função `princ`, que é a função de saída que melhor se ajusta a apresentação da mensagem e a recepção do valor informado para o cálculo.

O uso da função `finish-output` tem por finalidade realizar a finalização da ação de saída junto ao buffer. Esta função em alguns interpretadores LISP pode ser desnecessária. No entanto, deve sempre ser considerado seu uso após o uso de uma função de saída para evitar efeitos colaterais onde a mensagem de saída venha a ser apresentada após a entrada de dados quando deveria, de fato, ter sido indicada antes da operação de entrada.

4.3 VARIÁVEIS LOCAIS

Até este ponto foram usados nos exemplos apresentados variáveis globais, exceto algumas operações com ações de laço de alto nível que obrigatoriamente utilizam variáveis locais. Apesar de não ter sido utilizado exemplos com variáveis locais esta forma de variáveis é em CL bastante popular. A definição de variáveis locais é realizada, como comentado, com a função `let` a partir da sintaxe.

```
(let ((<var1 [vlr1]>) [<var2 [vlr2]>] [...] [<varN [vlrN]>]) <ação>)
```

Onde, as indicações **var1**, **var2** e **varN** referem-se a definição das variáveis locais; **vlr1**, **vlr2** e **vlrN** referem-se a definição opcional de valores iniciais para as variáveis locais e **ação** é a indicação das operações realizadas com as variáveis locais.

Observe a instrução seguinte que define a criação de duas variáveis **A** e **B** locais inicializadas, respectivamente, com os valores **1** e **2** e apresenta a soma dos valores definidos.

```
>>> (let ((A 1) (B 2))
      (+ A B))
3
```

A diferença entre variáveis globais e locais é que as variáveis globais são “visíveis” a qualquer momento durante o uso do ambiente de programação. Já as variáveis locais são “visíveis” dentro de certo escopo, ou seja, dentro do escopo operacionalizado dentro do contexto da função let.

Considere como demonstração a definição de uma função que solicita a entrada de um valor par e o apresenta. Caso seja fornecido um valor ímpar o pedido de entrada deve ser repetido. Observe as instruções.

```
>>> (defun entra-numero ()
      (format t "Informe um valor par: ")
      (finish-output)
      (let ((valor (read)))
          (if (evenp valor)
              (format t "Valor informado: ~a" valor)
              (entra-numero))))
```

ENTRA-NUMERO

```
>>> (entra-numero)
Informe um valor par: 3
Informe um valor par: 5
Informe um valor par: 2
Valor informado: 2
NIL
```

Ao ser executada a função `entra-numero` e um valor par não foi informado ocorrerá a execução recursiva da função até um valor par ser fornecido.

A função `let` efetua o efeito de amarração ou atamento (*binding*). É a ação de se atribuir valores iniciais a variáveis que possuem validade apenas dentro do escopo do corpo de uma função em uso, fora da função essas variáveis não existem.

Caso venha a existir a definição de uma variável local com o mesmo nome de uma variável global o acesso ocorrerá no contexto de amarração apenas na variável local. Para verificar este efeito considere a definição de variável global e local com o mesmo nome de identificação.

```
>>> (defparameter TESTE 1)
```

```
TESTE
```

```
>>> (let ((TESTE 9)) TESTE)
```

```
9
```

```
>>> TESTE
```

```
1
```

Devido a dificuldade em diferenciar variáveis globais e locais é que se tem o costume de usar a definição de variáveis globais entre os símbolos de asterisco. Uma variável global de nome ***X*** é referenciada com a identificação **estrela-xis-estrela**, se fosse local seria referenciada apenas por **X**.

É possível fazer uso das funções `setq` ou `setf` dentro do escopo de definição de certa variável local. Observe como exemplo o armazenamento temporário do resultado da soma a partir de dois valores definidos localmente junto a variável local **R**.

```
>>> (let ((A 1) (B 2) (R 0))
```

```
  (setf R (+ A B)) R)
```

```
3
```

No entanto, com a função `let` não é possível junto a definição de variáveis locais fazer com que o valor de uma variável seja usado para inicializar outra variável local. O trecho seguinte acarreta na apresentação de uma mensagem de erro.

```
>>> (let ((A 1) (B (+ A 1)) (R 0))
      (setf R (+ A B)) R)
```

Não é possível definir para a variável **B** a inicialização do valor **2** a partir da soma do valor da variável **A** com mais **1**. A mensagem de erro estará informando que a variável **A** não se encontra definida. No entanto, se ai invés de usar a função `let` for usada a função `let*` o resultado será diferente.

```
>>> (let* ((A 5) (B (+ A 1)) (R 0))
      (setf R (+ A B)) R)
```

11

Com o uso da função `let*` ocorreu a apresentação do resultado **11** sendo a soma do valor **5** da variável **A** com o valor **6** da variável **B** advindo da soma do valor da variável **A** com o valor **1**.

A função `let` não permite criar vinculo de um **form** como valor de inicialização de uma variável, apenas aceita a definição de valores isolados, daí o motivo pelo qual a tentativa da ação (`(B (+ A 1))`) não gerar o efeito esperado, o que é possível de ser realizado pela `let*`.

4.4 MACROS

O recurso denominado macro caracteriza-se por ser uma forma de expansão de recursos em CL que permitem estender as funcionalidades existentes da linguagem. É um mecanismo que permite otimizar e customizar operações da linguagem que venham a atender necessidades mais específicas.

Alguns dos recursos usados em CL são definições de macros que se não forem assim indicadas, passam naturalmente como sendo funções, destacando-se, por exemplo, as macros `if`, `when` e `unless`, entre outras tantas.

A definição simplificada de macros é realizada com a macro de definição `defmacro` a partir da estrutura sintática.

```
(defmacro <nome> <ação>)
```

Onde, a indicação **nome** refere-se ao nome da macro que pode ou não ter uma lista de argumentos e **ação** o que a macro deve realizar,

Para um exemplo simples e inicial considere uma macro que efetue uma operação de soma utilizando notação polonesa inversa como ocorre em algumas calculadoras financeiras. Observe as instruções a seguir.

```
>>> (defmacro npr (ARGUMENTOS)
      (reverse ARGUMENTOS))
NPR
```

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

Veja que a macro definida `npr` (Notação Polonesa Reversa) inverte a forma habitual de cálculo permitindo uma nova forma de acesso.

Os dois próximos exemplos caracterizam-se pela definição de duas macros que mostram respectivamente a cabeça e a cauda de uma lista.

```
>>> (defmacro cabeca (LISTA)
      (list 'car LISTA))
CABECA
```

```
>>> (defmacro cauda (LISTA)
      (list 'cdr LISTA))
CAUDA
```

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

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

Os três próximos exemplos demonstram como as macros `se (if)`, `quando (when)` e `a-nao-ser-que (unless)` podem estar definidas na linguagem CL.

Observe as instruções a seguir que definem a macro se.

```
>>> (defmacro se (CONDICAO ENTAO SENA0)
      `(cond (,CONDICAO ,ENTAO)
             (t ,SENAO)))
SE

>>> (se (evenp 2) (format t "Par") (format t "Impar"))
Par
NIL

>>> (se (evenp 3) (format t "Par") (format t "Impar"))
Impar
NIL
```

Na definição da macro se a lista de argumentos está sendo definida a partir de três componentes, sendo: CONDICAO; ENTAO e SENA0. Note que antes de usar cada um dos componentes no corpo da macro define-se o uso de uma vírgula imediatamente antes de cada componente (não pode haver espaço em branco entre a vírgula e o componente), indicando em se tratar de argumentos que possuem valores recebidos externamente. Outro detalhe a ser considerado é o uso do símbolo grave indicado antes da definição da ação que permite tratar o que vem a frente do símbolo como dado e não código.

Observe as instruções a seguir que definem a macro quando.

```
>>> (defmacro quando (CONDICAO &rest CORPO)
      `'(if ,CONDICAO (progn ,@CORPO)))
QUANDO

>>> (quando (>= 20 18) (format t "Maior de idade"))
Maior de idade
NIL

>>> (quando (>= 2 18) (format t "Maior de idade"))
NIL
```

A macro quando usa junto a definição do argumento **CONDICAO** o marcador &rest que tem por finalidade aceitar qualquer quantidade de argumentos que são atribuídos junto ao argumento **CORPO**.

O uso do operador especial progn junto a função if tem por finalidade avaliar um conjunto de **forms** na ordem em que são definidos retornando prioritariamente o valor do último formulário. Nesta aplicação este recurso define um bloco de operações que serão executadas quando a condição da função if for verdadeira, assemelhando-se a definição de blocos em outras linguagens de programação como feito em C com o uso de chaves ou Pascal com o uso de begin e end, entre outras. Alternativamente a função progn há a função prog1 que retorna de um conjunto o primeiro valor.

Observe os exemplos de uso das funções progn e prog1 respectivamente a seguir.

```
>>> (if (> 4 2)
      (progn
        (print 'Soma)
        (+ 4 2))
      (progn
        (print 'Subtracao)
        (- 4 2)))
```

SOMA

6

```
>>> (if (> 4 2)
      (prog1
        (+ 4 2)
        (print 'Soma))
      (prog1
        (- 4 2)
        (print 'Subtracao)))
```

SOMA

6

Veja que o resultado apresentado para os dois exemplos é o mesmo, mas atente para a ordem de definição da operação considerada no uso das funções `progn` e `prog1`.

Observe as instruções a seguir que definem a macro `a-nao-ser-que`.

```
>>> (defmacro a-nao-ser-que (CONDICAO &rest CORPO)
      `(if (not ,CONDICAO) (progn ,@CORPO)))
```

A-NAO-SER-QUE

```
>>> (a-nao-ser-que (= 2 3) (format t "Execute a acao"))
```

Execute a acao

NIL

```
>>> (a-nao-ser-que (= 2 3) (format t "Execute a acao"))
```

Execute a acao

NIL

É importante esclarecer que quando certa operação não é precedida dos símbolos apóstrofo ou grave isto indica que a operação será tratada apenas como código. Se precedida de um símbolo apóstrofo isto indica que a operação será tratada apenas como dado e se precedida de um símbolo grave contendo seus argumentos precedidos de vírgula isto indica que a operação será tratada como dado, permitindo que os argumentos indicados sejam inseridos dentro do código da macro. Observe os detalhes indicados.

```
>>> (defparameter *MAC-V1* 1)
```

MAC-V1

```
>>> (defparameter *MAC-V2* 4)
```

MAC-V2

```
>>> (+ *MAC-V1* *MAC-V2*) ; codigo
```

5

```
>>> '(+ *MAC-V1* *MAC-V2*) ; dado
```

(+ *MAC-V1* *MAC-V2*)

```
>>> `(+ ,*MAC-V1* ,*MAC-V2*) ; dado com argumento externo  
(+ 1 4)  
  
>>> (eval `(+ ,*MAC-V1* ,*MAC-V2*))  
5
```

A partir do conjunto de detalhes apresentados observe em seguida a definição da macro enquanto que opera um laço do tipo pré-teste com fluxo de ação para condição verdadeira. Observe o código seguinte.

```
>>> (defmacro enquanto (CONDICAO &rest CORPO)  
      `(tagbody INICIO  
        (if ,CONDICAO (progn ,@CORPO  
                      (go INICIO))))  
ENQUANTO  
  
>>> (defparameter *I* 1)  
*I*  
  
>>> (enquanto (<= *I* 5)  
              (princ *I*)  
              (terpri)  
              (setf *I* (+ *I* 1)))  
1  
2  
3  
4  
5  
NIL
```

Para a definição da macro enquanto é realizado uso do operador especial tagbody que indica um ponto de retorno pela definição de um rótulo de identificação, neste caso o rótulo INICIO. O retorno é produzido pelo operador especial go que transfere o controle para o operador tagbody criando um laço de repetição contínuo.

4.5 FUNÇÕES DE TEMPO

As funções de tempo são recursos usados para a manipulação de dados relacionados as informações de data e hora, podendo ser representado de três formas diferentes: tempo decodificado, tempo universal e tempo interno.

As representações de tempo decodificado e universal ocorrem sobre o tempo baseado no calendário do computador tendo sua precisão definida em segundos. O tempo interno é voltado a medição de ciclos computacionais com precisão determinada a partir de certa grandeza de fração em segundos, determinada pelo ambiente CL em uso.

O padrão para hora decodificada indica o tempo de forma absoluta e os padrões de tempo universal e interno indicam respectivamente o tempo absoluto e relativo.

O tempo decodificado indica o tempo de calendário a partir de um número serial formado pelos componentes: segundo (0-59), minuto (0-59), hora (0-23), dia (1-31), mês (1-12), ano (na forma Anno Domini), dia da semana (0-6: 0 = segunda-feira até 6 = domingo), horário de verão (T horário de verão em uso, se NIL horário de verão não em uso) e fuso horário (número inteiro de horas a oeste do meridiano de Greenwich).

O tempo universal indica o tempo como um valor inteiro positivo na forma serial que se visto do ponto de vista relativo é um número indicado em segundos; se visto do ponto de vista absoluto é um número de segundos contados a partir da meia-noite de 1 de janeiro de 1900 GMT. Assim, o valor de tempo 3764413260 (segundos) corresponde ao tempo 11:21:00 de 16/04/2019 GMT.

O tempo interno também indica o tempo como um valor serial inteiro. O tempo relativo é medido como um valor da unidade de tempo usada pelo ambiente CL. O tempo absoluto é baseado a um valor de tempo arbitrário que normalmente usa a hora em que o ambiente CL começou a ser executado.

Na sequência veja o uso das funções de manipulação de data e hora e o significado do retorno apresentado de cada função usada.

A função `get-decoded-time` retorna uma lista de valores de tempo decodificado no formato linear com a indicação dos dados: segundo, minuto, hora, dia, mês, ano, dia da semana, horário de verão e fuso horário.

```
>>> (get-decoded-time)
0
21
11
16
4
2019
1
NIL
3
```

A função `get-universal-time` retorna um valor serial único que representa internamente o conjunto de dados: segundo, minuto, hora, dia, mês, ano, dia da semana, horário de verão e fuso horário.

```
>>> (get-universal-time)
3764413260
```

A função `decode-universal-time` decodifica o valor de tempo universal indicado mostrando separadamente o conjunto de dados: segundo, minuto, hora, dia, mês, ano, dia da semana, horário de verão e fuso horário. Esta função possui de forma opcional um segundo argumento o qual determina a informação de fuso horário como um número de horas compensadas a partir do meridiano de Greenwich.

```
>>> (decode-universal-time 3764413260)
0
21
11
16
4
2019
1
NIL
3
```

```
>>> (decode-universal-time 3764413260 5)
0
21
9
16
4
2019
1
NIL
5
```

O argumento de fuso horário quando em uso caracteriza-se por ser um valor múltiplo racional de 1/3600 entre -24 e 24 que representa um número de horas compensadas pelo tempo médio de Greenwich. No exemplo anterior o valor **5** estabelece

A função `encode-universal-time` codifica o valor de tempo universal indicado em: segundo, minuto, hora, dia, mês, ano e o uso opcional do fuso horário na forma de tempo decodificado. Não se usa nesta função o dado que representa o horário de verão por ser um dado de tipo lógico e nem a indicação do dia da semana.

```
>>> (encode-universal-time 0 21 11 16 4 2019 3)
3764413260
```

A função `get-internal-run-time` retorna um valor absoluto único serial baseado no formato de hora interna do sistema. O valor de tempo apresentado depende da implementação CL em uso.

```
>>> (get-internal-run-time)
234
```

A função `get-internal-real-time` retorna um valor relativo único serial baseado no formato de hora interna do sistema. O valor de tempo apresentado depende da implementação CL em uso.

```
>>> (get-internal-real-time)
15073340
```

A função `sleep` faz com que a execução de certa tarefa fique inativa um determinado tempo real em segundos, após o que a execução é retornada indicando o valor `NIL`. Esta função opera com valores positivos inteiros ou de ponto flutuante.

```
>>> (sleep 3)
```

```
NIL
```

Além das funções indicadas o recurso de tempo de um ambiente CL possui a constante `internal-time-units-per-second` que indica a medida de unidade de tempo interna por segundo usada. O valor de tempo apresentado depende da implementação CL em uso.

```
>>> internal-time-units-per-second
```

```
1000
```

Como exemplo de uso de recursos de tempo considere uma função que fornecida uma data indique o dia da semana correspondente. Para tanto, siga as instruções.

```
>>> (defun dia-semana (DIA MES ANO)
      (case (nth-value 6
                      (decode-universal-time
                        (encode-universal-time 0 0 0 DIA MES ANO 0)
                        0))
        (0 (format t "segunda-feira~%"))
        (1 (format t "terca-feira~%"))
        (2 (format t "quarta-feira~%"))
        (3 (format t "quinta-feira~%"))
        (4 (format t "sexta-feira~%"))
        (5 (format t "sabado~%"))
        (6 (format t "domingo~%"))))
```

```
DIA-SEMANA
```

```
>>> (dia-semana 26 04 1965)
```

```
segunda-feira
```

```
NIL
```

Ao ser executada a função `dia-semana` com a definição de uma data no formato dia, mês e ano ocorre a apresentação do dia da semana da data informada. Veja

que a parte (`encode-universal-time 0 0 0 DIA MES ANO 0`) efetua a codificação da data informada em um valor universal serial que é então decodificado pela função `decode-universal-time` com argumento 0 de fuso horário. A partir dessas duas ações ocorre a obtenção da série de valores segundo, minuto, hora, dia, mês, ano, dia da semana, horário de verão e fuso horário.

Note que a informação de dia da semana é a sétima informação existente, ou seja, é a informação de número 6, contando a partir de zero. Por esta razão usa-se o valor 6 na indicação de uso da macro `nth-value` que é operada com a sintaxe.

(`nth-value <índice> <formulário>`)

Onde, a indicação **índice** refere-se a indicação do índice de acesso posicional do conteúdo existente no argumentos e **formulário** que indica uma coleção de valores da qual um item é usado.

Observe alguns exemplos de uso simplificado da macro `nth-value`.

```
>>> (nth-value 1 (values 'a 'b 'c 'd 'e))
B

>>> (nth-value 4 (values 1 2 3 4 5))
5

>>> (nth-value 6 (decode-universal-time 3764413260) ; dia da semana
1

>>> (nth-value 5 (decode-universal-time 3764413260) ; ano
2019
```

Após capturar o valor do sétimo item da coleção de valores gerados a partir da data fornecida a função case verifica qual dos valores da data referente ao dia da semana foi retornado e apresenta o extenso do dia correspondente.

4.6 RANDOMIZAÇÃO

Randomização é a ação de realizar algo de forma aleatório. A linguagem CL oferece para a geração de números pseudoaleatórios a função `random` que é usada a partir da seguinte sintaxe.

`(random <limite> [<estado>])`

Onde, a indicação **limite** refere-se a definição do valor numérico positivo inteiro ou de ponto flutuante limite da geração de valores aleatórios de 0 (inclusivo) até limite (exclusivo) e **estado** refere-se a definição opcional do estado de aleatoriedade `make-random-state` com valor T a ser empregado, sendo o padrão por omissão definido junto a variável global interna ***RANDOM-STATE*** que atua como se fosse a definição de uma semente implícita de geração de números aleatórios, pontuando que CL não permite que seja definido de forma explícita um valor de semente para a geração de números aleatórios.

O estado para geração de números aleatórios muda a cada chamada efetuada a função `random` na mesma instância em que o ambiente CL está sendo executado. Se o ambiente for encerrado e inicializado novamente a geração de números aleatórios anteriores se repetirá, comprovando ser uma geração de números pseudoaleatórios. O efeito de repetição de números aleatórios a cada execução do ambiente pode ser neutralizado com o uso do estado (`make-random-state t`).

Observe a seguir a geração de valores aleatórios (em seu sistema os valores podem ser diferentes) sem a definição do estado de geração de aleatoriedade com valores inteiros entre **0** e **9**.

```
>>> (random 10)
```

7

```
>>> (random 10)
```

2

```
>>> (random 10)
```

6

```
>>> (random 10)
```

```
8
```

Saia do ambiente e retorne a ele novamente e repita as quatro etapas anteriores e observe a apresentação dos mesmos valores anteriormente sorteados.

Saia e retorne ao ambiente e veja a geração de valores aleatórios com a definição do estado de geração de aleatoriedade com valores inteiros entre **0** e **9**.

```
>>> (random 10 (make-random-state t))
```

```
3
```

```
>>> (random 10 (make-random-state t))
```

```
4
```

```
>>> (random 10 (make-random-state t))
```

```
4
```

```
>>> (random 10 (make-random-state t))
```

```
2
```

Saia do ambiente e retorne a ele novamente e repita as quatro etapas anteriores e observe a apresentação dos valores anteriormente sorteados dispostos de forma diferente.

Usar a geração de números aleatórios com a definição de estado de aleatoriedade passa a ser mais interessante na ação de simulação de geração de valores aleatórios.

Como comentado a geração de números aleatórios é realizada a partir de zero até o limite exclusivo definido. Caso deseje, por exemplo, gerar números aleatórios entre **1** e **10** use a instrução.

```
>>> (+ 1 (random 10 (make-random-state t)))
```

```
2
```

Em sentido mais amplo pode-se definir uma função que efetue a geração de números aleatórios entre a definição de um valor inicial e final. Assim, observe o código seguinte.

```
>>> (defun sorteio (INICIO FIM)
      (+ INICIO
          (random (1+ (- FIM INICIO)))
          (make-random-state t))))
```

10

A função sorteio caracteriza-se por ser a definição de uma forma que estabelece uma maneira mais cômoda em realizar a geração de números aleatórios. Nesta função está sendo usada a função de incremento “+” anexa a um valor de incremento como em “1+” para $(1+ (- \text{FIM} \text{ INICIO}))$, gerando o mesmo efeito no uso de $(+ 1 (- \text{FIM} \text{ INICIO}))$.

4.7 FUNÇÃO ANÔNIMA (LAMBDA)

Função anônima ou lambda caracteriza-se por ser uma sub-rotina trivial sem nome de identificação com o objetivo de retornar uma resposta rápida a execução de certa ação operacional de cunho genérico.

A estrutura de definição de uma função lambda (anônima) segue o modelo matemático $\lambda a. (ab)$, onde a função anônima $[\lambda]$ de [a] a qual [.] efetua a ação prevista na operação $[(ab)]$ tem desta ação seu resultado.

Uma função anônima é definida em CL com a macro lambda, ficando a disposição para uso uma única vez na memória a partir ou em conjunto com a execução da função funcall. Após a função anônima ser usada esta é removida automaticamente da memória.

A definição de uma função anônima é realizada pela instrução.

(lambda <argumentos> <corpo>)

Onde, a indicação **argumentos** refere-se a definição da lista de argumentos a ser usada pela função e **corpo** refere-se a estrutura operacional a ser executada a partir dos parâmetros fornecidos.

A chamada e execução de uma função anônima é realizada pela instrução.

(funcall <função> <argumentos>)

Onde, a indicação **função** refere-se a definição da função anônima a ser usada e **argumentos** o conteúdo passado para a execução da função indicada.

Considerando a função $\lambda x.(x + 1)$ que apresenta o sucessor do argumento fornecido, onde o argumento **X** seja **9** tem-se como forma matemática escrita a operação $\lambda x.(x + 1).9$, que efetuado o cálculo para a operação $\lambda x.(9 + 1)$, terá como resultado o valor **10** para $\lambda = 10$ e poderá ser declarada em CL a partir da função Lambda.

Para tanto, execute as instruções a seguir no ambiente CLISP.

```
[n]> (lambda (X) (+ X 1))  
#<FUNCTION :LAMBDA (X) (+ X 1)>
```

Para tanto, execute as instruções a seguir no ambiente SBCL.

```
* (lambda (X) (+ X 1))  
#<FUNCTION (LAMBDA (X)) {10025D002B}>
```

Após a definição da função anônima esta pode ser executada a partir da instrução.

```
>>> (funcall * 9)  
10
```

Na sequência se for solicitada nova execução da instrução (`funcall * 9`) ocorrerá a apresentação de mensagem de erro, pois a função anônima definida não está mais presente na memória. Isto ocorre, pois o argumento definido para uma função lambda permite referência de uso apenas no âmbito do corpo da função, mas nunca fora da função.

A execução de uma função anônima pode ser produzida de uma única vez com o uso conjunto da função `funcall` com a macro `lambda`. Observe o exemplo seguinte que mostra o quadrado do valor **4** definido na execução da chamada da função.

```
>>> (funcall (lambda (X) (expt X 2)) 4)  
16
```

Neste caso o uso da função anônima ficou um pouco mais restritivo por exigir que o valor a ser usado no cálculo seja fornecido em conjunto com a função anônima definida.

A definição de função anônima pode ser realizada e utilizada em conjunto de outras funções, como por exemplo, apresentar o resultado de certa operação. Veja a seguir a definição de função anônima que apresenta o resultado da soma de dois valores.

```
>>> (write ((lambda (X Y) (+ X Y)) 3 7))  
10  
10
```

Ao ser executado o código anterior ocorre a apresentação do resultado calculado pela função anônima.

O uso de funções anônimas, nesta obra, será visto na parte que se refere ao uso da linguagem CL na forma funcional.

4.8 TIPO DE DADO TABELA DE SÍMBOLO (HASH)

As tabelas hash (**hashables**) ou tabelas de símbolos são estruturas usadas para o gerenciamento de coleção de dados esparsos (dados que não seguem uma ordem predefinida) de propósito geral definidos com pares de elementos compostos de chaves e valores organizados com base na definição da chave em si.

No uso de tabelas hash é possível criar e excluir entradas, além de efetuar a localização de conteúdo associado a certa chave. A localização de elementos é realizada de forma mais rápida do que as ações efetuadas com listas ou matrizes, pois se utiliza da chave para acessar os elementos na coleção.

Tabelas hash associam um valor a uma chave, de modo que, se houver a tentativa de adicionar novo valor a uma chave existente ocorrerá a substituição natural do valor existente pelo novo valor.

Tabelas hash são criadas a partir do uso da função `make-hash-table` que opera a partir da sintaxe.

```
(make-hash-table [<chave> <teste> <tamanho> <aumentar> <limite>])
```

Onde, a indicação **chave** refere-se a definição de uma chave para a tabela, **teste** define a maneira pela qual as chaves são comparadas, **tamanho** define o tamanho inicial da tabela a partir da indicação de um valor numérico inteiro maior que zero, **aumentar** define o valor do quanto aumentar o tamanho da tabela quando ficar

cheia e **limite** que determina o tamanho que a tabela deve possuir antes de ficar cheia. É oportuno pontuar que a função `make-hash-table` pode ser executada sem a indicação de argumentos, os quais são aqui omitidos por estarem fora do escopo deste trabalho.

Observe em seguida a instrução para a definição de uma tabela de símbolo (hash) chamada ***ALUNO***, definida como sendo uma variável global.

```
>>> (defvar *ALUNO* (make-hash-table))  
*ALUNO*
```

Observe como a criação da tabela hash é identificada no programa CLISP.

```
[n]> *ALUNO*  
#S(HASH-TABLE :TEST FASTHASH-EQL)
```

Observe como a criação da tabela hash é identificada no programa SBCL.

```
* *ALUNO*  
#<HASH-TABLE :TEST EQL :COUNT 0 {1002576DF3}>
```

A partir da definição da tabela hash ***ALUNO*** é possível adicionar elementos e recuperá-los para apresentação, por exemplo, por meio de uso da função `gethash` que opera a partir da sintaxe.

```
(gethash <chave> <tabela> [<padrão>])
```

Onde, a indicação **chave** refere-se a definição de uma chave existente na tabela, **tabela** refere-se a definição da tabela a ser pesquisada e padrão a definição de um valor opcional a ser retornado.

Para fazer uso da função `gethash` na ação de adição de elementos em uma tabela usa-se em conjunto a função `setf`. Assim, observe as instruções a seguir que adicionam os nomes dos alunos a tabela ***ALUNO***.

```
>>> (setf (gethash '001 *ALUNO*) '(Augusto))  
(AUGUSTO)
```

```
>>> (setf (gethash '002 *ALUNO*) '(Sandra))  
(SANDRA)
```

Observe os detalhes indicados para a tabela ***ALUNO*** no programa CLISP.

```
[n]> *ALUNO*
#S(HASH-TABLE :TEST FASTHASH-EQL (2 . (SANDRA)) (1 . (AUGUSTO)))
```

Observe os detalhes indicados para a tabela ***ALUNO*** no programa SBCL.

```
* *ALUNO*
#<HASH-TABLE :TEST EQL :COUNT 2 {1002576DF3}>
```

Para apresentar especificamente um dos elementos cadastrados na tabela hash pode-se fazer uso de uma das funções de saída conhecidas, como indicado a seguir.

```
>>> (princ (gethash '002 *ALUNO*))
(SANDRA)
(SANDRA)
```

Como experimento de nova entrada de dado na tabela e apresentação do conteúdo existente execute em seguida as instruções.

```
>>> (princ (gethash '001 *ALUNO*))
(AUGUSTO)
(AUGUSTO)

>>> (setf (gethash '001 *ALUNO*) '(Audrey))
(AUDREY)
```

```
>>> (princ (gethash '001 *ALUNO*))
(AUDREY)
(AUDREY)
```

```
>>> (princ (gethash '002 *ALUNO*))
(SANDRA)
(SANDRA)
```

Observe que na sequência anterior de instruções ao se definir com a chave **001** o valor **AUDREY** este foi sobreposto ao valor **AUGUSTO** anterior. É importante tomar cuidado com este tipo de operação.

Para realizar a apresentação dos elementos de uma tabela de hash pode-se usar a função `maphash` que possui a sintaxe.

(maphash <função> <tabela>)

Onde, a indicação **função** refere-se a uma função aplicada sobre os elementos da tabela em uso e **tabela** a indicação da tabela hash a ser usada.

Observe a instrução a seguir para apresentação do conteúdo da tabela ***ALUNO*** no programa CLISP.

```
[n]> (maphash (lambda (CHAVE VALOR)
                         (format t "~a: ~a~%" CHAVE VALOR)) *ALUNO*)
2: (SANDRA)
1: (AUDREY)
NIL
```

Observe a instrução a seguir para apresentação do conteúdo da tabela ***ALUNO*** no programa SBCL.

```
* (maphash (lambda (CHAVE VALOR)
                  (format t "~a: ~a~%" CHAVE VALOR)) *ALUNO*)
1: (AUDREY)
2: (SANDRA)
NIL
```

O uso da função `maphash` anterior tem como primeiro argumento a definição da função `lambda (CHAVE VALOR)` que é aplicada sobre a função `format` que apresenta o conteúdo indicado por `CHAVE` e `VALOR`. A função `maphash` efetua uma ação de iteração sobre todos os elementos da tabela em uso.

Outra maneira de obter o conteúdo de uma tabela hash é fazendo uso da macro `loop for ... do` a partir da instrução.

```
>>> (loop for C being the hash-keys in *ALUNO* using (hash-value V)
          do (format t "~a: ~a~%" C V))
```

Ao ser executada a instrução anterior o resultado apresentado será o mesmo gerado pela função `maphash`. Observe que a definição da variável local **C** se refere ao uso da chave e a definição da variável local **V** se refere ao uso do valor existente para cada elemento da tabela ***ALUNO***.

Para “pegar” a chave com o laço `loop` e associá-la a variável **C** está sendo usada a ação “`C being the hash-keys in`” que atribui o valor da chave (`hash-keys`) a variável **C** da tabela indicada e em seguida por meio da ação “`using (hash-value V)`” pega o valor correspondente (`hash-value`) para a variável **V** associando **C** (chave) e **V** (valor) e definindo assim o elemento a ser utilizado. Desta forma `loop` percorre toda a tabela e permite apresentar os elementos existentes em ***VALORES*** por meio da função `format`.

A expressão de iteração definida com `loop for ... do` pode ser usada a partir de quatro variações.

1. CHAVE being each hash-key of <tabela>
2. CHAVE being the hash-keys in <tabela> using (hash-value VALOR)
3. VALOR being each hash-value of <tabela> using (hash-key CHAVE)
4. VALOR being the hash-values in <tabela>

Os termos `hash-key/hash-keys` e `hash-value/hash-values` (atente para a forma no plural e no singular pois atuam de forma diferente na ação) são usados respectivamente para identificar em uma tabela hash a chave e o valor de um elemento. O termo `each ... of` refere-se a ação de “pegar” cada chave do elemento se usado `hash-key` ou os valores dos elementos se usado `hash-value` e realizar alguma ação com o conteúdo existente na tabela. O termo `the ... in` refere-se a ação de “pegar” as chaves dos elementos se usado `hash-keys` ou o valor do elemento se usado `hash-values` e realizar alguma ação sobre o conteúdo da tabela. O termo `using` coloca em uso para a operação a chave (`hash-key`) ou o valor (`hash-value`) quando respectivamente está em uso `being each hash-values` e `being the hash-keys in`.

A título de ilustração observe a seguir exemplos de uso das formas de iteração **1, 3** e **4**, sendo que a forma **2** já fora apresentada.

```
>>> (loop for C being each hash-key of *ALUNO*
          do (format t "~a~%" C))
```

O trecho anterior faz a apresentação apenas das chaves dos elementos existentes na tabela ***ALUNO***.

```
>>> (loop for V being the hash-values in *ALUNO*
          do (format t "~a~%" V))
```

O trecho anterior faz a apresentação apenas dos valores dos elementos existentes na tabela ***ALUNO***.

```
>>> (loop for V being each hash-value of *ALUNO* using (hash-key C) do
          (format t "~a: ~a~%" C V))
```

O trecho anterior faz a apresentação das chaves e valores dos elementos existentes na tabela ***ALUNO***.

OBS: *Usar os termos hash-key, hash-keys, hash-value ou hash-values no singular ou plural pode não interferir a resposta da ação desejada, mas dependendo do interpretador de CL em uso será apresentada mensagem de advertência informado para usar o termo adequado.*

A quantidade de elementos armazenados em uma tabela hash pode ser obtida com o uso da função hash-table-count a partir da sintaxe.

(hash-table-count <tabela>)

Onde, a indicação **tabela** refere-se a definição da tabela hash a ser usada para ter a quantidade de elementos existentes.

Veja em seguida a instrução para saber a quantidade de elementos armazenados na tabela ***ALUNO***.

```
>>> (hash-table-count *ALUNO*)
```

2

Os elementos definidos em uma tabela hash podem ser removidos com o uso da função remhash que possui a sintaxe.

(remhash <chave> <tabela>)

Onde, a indicação **chave** refere-se a definição de uma chave existente na tabela e **tabela** refere-se a definição da tabela a ser pesquisada.

Observe a seguir a instrução para remover o elemento de chave **001** da tabela ***ALUNO*** com a função remhash.

```
>>> (remhash 001 *ALUNO*)  
T  
  
>>> (maphash (lambda (CHAVE VALOR)  
                    (format t "~a: ~a~%" CHAVE VALOR)) *ALUNO*)  
2: (SANDRA)  
NIL
```

Diferentemente das outras funções de gerenciamento de tabela hash a função remhash não usa o sinal de apóstrofo antes do valor da chave. Note ainda que após a remoção da chave **001** ficou definido na tabela apenas o elemento correspondente a chave **002**.

Para a remoção de elementos de uma matriz tem-se também a função clrhash que efetua a remoção de todos os elementos definidos para uma tabela operada a partir da sintaxe.

(clrhash <tabela>)

Onde, a indicação **tabela** refere-se a definição da tabela hash a ter todos os elementos removidos, deixando a tabela vazia.

Veja o resultado da execução da função clrhash executado no programa CLISP.

```
[n]> (clrhash *ALUNO*)  
#S(HASH-TABLE :TEST FASTHASH-EQL)
```

Veja o resultado da execução da função clrhash executado no programa SBCL.

```
* (clrhash *ALUNO*)  
#<HASH-TABLE :TEST EQL :COUNT 0 {1002576DF3}>
```

Na sequência se solicitada a apresentação dos elementos existentes na tabela ter-se-á a resposta NIL.

```
>>> (maphash (lambda (CHAVE VALOR)
                      (format t "~a: ~a~%" CHAVE VALOR)) *ALUNO*)
NIL
```

Como demonstração mais acentuada do gerenciamento de elementos em tabelas hash considere a criação de uma tabela chamada ***VALORES*** formada por um conjunto de elementos configurados com chaves alfabéticas e valores numéricos inteiros. Observe as instruções seguintes.

```
>>> (defvar *VALORES* (make-hash-table :size 6))
>>> (setf (gethash 'A *VALORES*) 1)
>>> (setf (gethash 'B *VALORES*) 2)
>>> (setf (gethash 'C *VALORES*) 3)
>>> (setf (gethash 'D *VALORES*) 4)
>>> (setf (gethash 'E *VALORES*) 5)
>>> (setf (gethash 'F *VALORES*) 6)
```

A partir dos valores definidos considere realizar uma operação de atualização nos valores dos elementos existentes. A instrução que fará esta operação deve considerar que sendo um valor ímpar este deverá ser retirado da tabela e caso contrário deve o valor ser elevado ao cubo.

```
>>> (maphash (lambda (CHAVE VALOR)
                      (if (oddp VALOR)
                          (remhash CHAVE *VALORES*)
                          (setf (gethash CHAVE *VALORES*) (expt VALOR 3))))
                  *VALORES*)
NIL
```

Em seguida a fim de visualizar os valores que ficaram na tabela elevados ao cubo execute a instrução.

```
>>> (maphash (lambda (CHAVE VALOR) (format t "~a: ~a~%" CHAVE VALOR))
                  *VALORES*)
```

Serão então apresentados os elementos **B** com valor **8**, **D** com valor **64** e **F** com valor **216**.

Para verificar se dada variável definida em memória é de fato uma tabela de hash pode-se fazer uso da função `hash-table-p` que retorna T se a verificação for verdadeira ou NIL se a verificação for falsa.

A função `hash-table-p` é usada a partir da sintaxe.

```
(hash-table-p <variável>)
```

Onde, a indicação **variável** refere-se a definição de uma variável a ser verificada.

Observe a instrução a seguir.

```
>>> (hash-table-p *VALORES*)
```

```
T
```

Uma ação que pode ser interessante ou até mesmo necessária é criar um mecanismo que venha a popular uma tabela hash de forma iterativa. Assim sendo, observe as instruções seguintes que criam uma tabela chamada ***TAB-X1*** e efetua a definição de quatro chaves e seus respectivos valores.

```
>>> (defvar *TAB-X1* (make-hash-table))
```

```
*TAB-X1*
```

```
>>> (loop for (CHAVE VALOR) on
```

```
      '(um "um" dois "dois" tres "tres" quatro "quatro") by #'cddr
```

```
      do (setf (gethash CHAVE *TAB-X1*) VALOR))
```

```
NIL
```

```
>>> (loop for V being each hash-value of *TAB-X1* using (hash-key C)
```

```
      do (format t "~a: ~a~%" C V))
```

Veja que na apresentação do conteúdo as chaves são expressas em formato maiúsculo e os valores que foram definidos entre aspas inglesas ficam mantidos em caracteres minúsculos.

4.9 TIPO DE DADO ESTRUTURA

O tipo de dado estrutura permite que sejam criados conjuntos de dados formados a partir de dados de tipos diferentes para representarem registros definidos a partir da macro `defstruct` que é operada a partir da sintaxe simplista e resumida.

```
(defstruct <nome> (<campo1 [<opção1>]) [(<campoN [<opçãoN>])])>
```

Onde, a indicação **nome** refere-se ao nome de identificação da estrutura; as indicações **campo1** e **campoN** referem-se ao conjunto de elementos informativos que formam a base do registro a partir do uso opcional da definição de opções **opção1** e **opçãoN** de configuração que podem conter a definição de valores iniciais de cada campo, do tipo de dado a ser aceito por cada campo, além de outras possibilidades.

Considere a definição de uma estrutura que modele um conjunto de dados para o gerenciamento de alunos. Para tanto, é necessário indicar um campo para o nome do aluno, um campo para registro das notas e outro campo para o armazenamento da média. Para tanto, siga as instruções.

```
>>> (defstruct CADALUNO  
      (nome "" :type string)  
      (notas (make-array 4 :element-type 'single-float))  
      (media 0.0 :type single-float))
```

Veja que a definição da estrutura **CADALUNO** é configurada com o campo `nome` sem a indicação de nenhum conteúdo inicial ("") do tipo cadeia a partir da indicação `:type string`. O campo `notas` é configurado como uma matriz de uma dimensão para quatro notas bimestrais `make-array 4` com elementos do tipo ponto flutuante simples a partir da indicação `:element-type 'single-float`. Por último o campo `media` é também inicializado com valor zero definido para uso de valor de ponto flutuante simples.

Com a definição da estrutura **CADALUNO** são implicitamente definidas de forma automática **funções de acesso** aos campos existentes, sendo representados por: `cadaluno-nome`; `cadaluno-notas` e `cadaluno-media`, os quais receberão um argumento como componente da representação de seu dado informativo.

Assim que um tipo estrutura está definido na memória é possível criar variáveis que farão uso dessa estrutura. Para tanto, execute o uso da função `defvar` para definir

a variável e use a função setf para inserir dados na variável criada. Assim sendo, siga as instruções seguintes.

```
>>> (defvar *ALUNO*)  
*ALUNO*  
  
>>> (setf *ALUNO* (make-cadaluno  
:nome "Jose"  
:notas #(2.5 7.5 8.5 4.5)))  
#S(CADALUNO :NOME "Jose" :NOTAS #(2.5 7.5 8.5 4.5) :MEDIA 0.0)
```

Note o uso da função construtora implícita make+<nome da estrutura>, neste caso, make-cadaluno que tem por finalidade modelar a variável global ***ALUNO*** com os campos estabelecidos para a estrutura **CADALUNO**. Observe que a função make-cadaluno quando invocada, cria na variável em uso a estrutura de dados adequada para uso com as funções de acesso relacionadas aos campos estabelecidos, ou seja, cria uma instância com o construtor definido. Uma estrutura definida é identificada com o prefixo “#S” podendo ser usada para operações de leitura e escrita como será demonstrada.

Observe que ao se definir os dados para a variável estruturada está sendo omitido o valor da média, pois este será internamente atribuído a zero automaticamente pela simples omissão de seu dado.

Para visualizar o conteúdo total da variável execute a instrução.

```
>>> *ALUNO*  
#S(CADALUNO :NOME "Jose" :NOTAS #(2.5 7.5 8.5 4.5) :MEDIA 0.0)
```

Veja que foram indicados o nome do aluno e informada as quatro notas bimestrais e deixado com valor zero o campo média, pois esta ação será realizada a partir do cálculo da média sobre as notas existentes. Para tanto, execute a instrução.

```
>>> (setf (cadaluno-media *ALUNO*)  
(/ (apply '+ (coerce (cadaluno-notas *ALUNO*) 'list)) 4))  
5.75
```

Para o cálculo a função setf atribui no campo media da estrutura cadaluno-media o valor da média calculada. Para calcular a média é feita primeiramente a conver-

são da matriz em lista (`(coerce (cadaluno-notas *aluno*) 'list)`). Assim que os dados da matriz são convertidos como lista a função `apply` efetua a soma dos elementos da lista e em seguida é dividido por quatro.

A partir da definição dos dados de uma estrutura é possível apresentá-los separadamente ou juntos a partir das instruções.

```
>>> (cadaluno-nome *ALUNO*)
Jose

>>> (cadaluno-notas *ALUNO*)
#(2.5 7.5 8.5 4.5)

>>> (cadaluno-media *ALUNO*)
(cadaluno-media *aluno*)

>>> *ALUNO*
#S(CADALUNO :NOME "Jose" :NOTAS #(2.5 7.5 8.5 4.5) :MEDIA 5.75)
```

Para definição de mais alunos basta definir novas variáveis globais estruturadas e replicar as ações comentadas anteriormente sobre a “nova” variável.

Desejando maior dinamismo na operação você poderá definir funções que automatizem os passos mostrados anteriormente.

