
2

Recursos básicos

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

2.1 AÇÕES INTERATIVAS

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

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

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

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

```
>>> 26
```

```
26
```

```
>>> 3.14
```

```
3.14
```

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

```
"A"
```

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

```
"LISP"
```

```
>>> 'a
```

```
A
```

```
>>> 'Lisp
```

```
LISP
```

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

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

```
T
```

Veja uma ocorrência de erro no programa CLISP.

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

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

Veja uma ocorrência de erro no programa SBCL.

```
* (atom X)
```

```
debugger invoked on a UNBOUND-VARIABLE in thread
#<THREAD "main thread" RUNNING {10012E0613}>:
  The variable X is unbound.
```

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

```
restarts (invokable by number or by possibly-abbreviated name):
 0: [CONTINUE] Retry using X.
 1: [USE-VALUE] Use specified value.
 2: [STORE-VALUE] Set specified value and use it.
 3: [ABORT] Exit debugger, returning to top level.
```

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

```
0]
```

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

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

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

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

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

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

```
120
```

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

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

```
>>> ()
```

```
NIL
```

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

```
NIL
```

```
>>> '(alo mundo)
```

```
(ALO MUNDO)
```

```
>>> '(a b c d e)
```

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

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

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

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

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

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

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

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

2.2 TIPO DE DADO NUMÉRICO

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
7.6295s-6
```

```
7.6295s-6
```

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

```
5.4210108624275221706L-20
```

```
5.4210108624275221706L-20
```

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

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

```
5.960465e-8
```

```
5.960465e-8
```

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

```
1.1102230246251568d-16
```

```
1.1102230246251568d-16
```

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

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

```
5.960465E-8
```

```
5.960465E-8
```

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

```
1.1102230246251568d-16
```

```
1.1102230246251568d-16
```

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

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

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

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

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

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

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

Observe o uso da função complex no programa CLISP.

```
[n]> (complex(+ 2.5 1.5) 7)  
#C(4.0 7)
```

Observe o uso da função complex no programa SBCL.

```
* (complex(+ 2.5 1.5) 7)  
#C(4.0 7.0)
```

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

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

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

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

```
1
```

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

```
2
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3 IDENTIFICAÇÃO DE TIPOS DE DADOS

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

```
>>> (type-of 1)  
BIT  
  
>>> (type-of 1.5)  
SINGLE-FLOAT  
  
>>> (type-of 1.5d0)  
DOUBLE-FLOAT  
  
>>> (type-of 2/3)  
RATIO
```

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

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

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

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

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

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

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

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

```
* (type-of -3)
FIXNUM
```

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

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

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

2.4 FUNCIONALIDADES MATEMÁTICAS

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

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

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

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

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

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

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

```
6
```

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

```
4
```

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

```
3.1415927
```

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

```
-1.5707964
```

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

```
3.0
```

```
>>> (log 100.0 10) ; logaritmo na base 10  
2.0  
  
>>> (log 2.0) ; logaritmo neperiano  
0.6931472  
  
>>> (log -1.0) ; logaritmo neperiano  
#C(0.0 3.1415927)  
  
>>> (mod 8 3)  
2  
  
>>> (rem 8 3)  
2  
  
>>> (random 3) ; retorna 0, 1 ou 2  
0  
  
>>> (+ (random 3) 1) ; retorna 1, 2 ou 3  
3  
  
>>> (signum 9) ; retorna 1 se valor maior que zero  
1  
  
>>> (signum -9) ; retorna -1 se valor menor que zero  
-1  
  
>>> (signum 0) ; retorna 0 se valor igual a zero  
0
```

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

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

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

```
2 ;
```

```
0.5
```

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

```
3 ;
```

```
-0.4000001
```

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

```
1 ;
```

```
0.100000024
```

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

```
1 ;
```

```
0.9
```

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

```
2 ;
```

```
-0.9
```

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

```
2 ;
```

```
-0.100000024
```

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

```
0 ;
```

```
0.3
```

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

```
0 ;
```

```
-0.3
```

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

```
4.0
```

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

* (**atan** -1)

-0.7853982

* (**round** 2.5)

2

0.5

* (**round** 2.6)

3

-0.4000001

* (**floor** 1.1)

1

0.100000024

* (**floor** 1.9)

1

0.9

* (**ceiling** 1.1)

2

-0.9

* (**ceiling** 1.9)

2

-0.100000024

* (**truncate** 0.3)

0

0.3

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

```
0
```

```
-0.3
```

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

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

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

2.5 VARIÁVEIS E CONSTANTES

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

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

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

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

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

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

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

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

(`defvar` *<variável>* [<valor>])

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

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

(`makunbound` '<variável>')

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

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

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

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

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

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

```
>>> *x*
9
```

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

```
>>> *x*
9
```

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

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

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

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

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

```
>>> *a*
1
```

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

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

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

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

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

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

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

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

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

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

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

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

```
* (defvar *c*)
```

```
*C*
```

```
* *C*
```

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

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

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

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

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

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

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

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

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

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

```
0]
```

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

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

```
(set '*<variável>* <valor>)
```

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

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

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

```
>>> *c*  
3
```

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

(quote *<variável>*)

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

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

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

(setq *<variável>* <valor>)

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

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

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

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

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

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

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

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

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

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

```
*** - EVAL: undefined function SETD  
The following restarts are available:  
USE-VALUE      :R1      Input a value to be used instead of (FDEFINITION 'SETD).  
RETRY          :R2      Retry  
STORE-VALUE    :R3      Input a new value for (FDEFINITION 'SETD).  
ABORT          :R4      Abort main loop
```

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

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

```
debugger invoked on a UNBOUND-VARIABLE in thread  
#<THREAD "main thread" RUNNING {10012E0613}>:  
  The variable *D* is unbound.
```

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

```
restarts (invokable by number or by possibly-abbreviated name):
 0: [CONTINUE    ] Retry using *D*.
 1: [USE-VALUE   ] Use specified value.
 2: [STORE-VALUE] Set specified value and use it.
 3: [ABORT       ] Exit debugger, returning to top level.
```

```
(SB-INT:SIMPLE-EVAL-IN-LEXENV *D* #<NULL-LEXENV>)
0]
```

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

Observe o uso da função macroexpand no programa CLISP.

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

Observe o uso da função macroexpand no programa SBCL.

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

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

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

```
(let <variável> <valor>)
```

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

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

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

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

```
3
```

```
[n]> e
```

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

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

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

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

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

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

```
* (let e 3)
; in: LET E
;     (LET E
;       3)
;
; caught ERROR:
;   Malformed LET bindings: E.
;
; compilation unit finished
; caught 1 ERROR condition
```

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

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

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

Form:

```
(LET E  
 3)
```

Compile-time error:

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

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

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

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

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

```
source: (LET E  
 3)  
0]
```

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

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

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

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

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

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

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

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

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

```
* pi  
3.141592653589793d0
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
>>> (defconstant m_e 2.718281828459045235360287471352662497757)  
M_E
```

```
>>> m_e  
2.7182817
```

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

```
[n]> (coerce m_e 'short-float)  
2.7183s0
```

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

```
2.7182817459106445313L0
```

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

```
2.7182817
```

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

```
2.7182817459106445d0
```

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

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

```
2.7182817
```

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

```
2.7182817459106445d0
```

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

```
2.7182817
```

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

```
2.7182817459106445d0
```

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

2.6 FORMATAÇÃO NUMÉRICA

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

>>> (format t "~,'.,:d" 123456789) ; ",,'.,4:d" troca "," por "."
123.456.789
NIL
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Observe os exemplos de formatação numérica no programa CLISP.

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

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

Observe os exemplos de formatação numérica no programa SBCL.

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

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

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