AutoLISP

LISP (list processing) é uma linguagem de programação procedural de alto-nível[1] (dita de 3ª geração) criada por John McCarthy em 1959. É a segunda[2] linguagem, desse tipo, mais antiga ainda utilizada em larga escala.

A linguagem AutoLISP implementada pela empresa Autodesk, em 1982, deriva desta por permitir a manipulação de objetos gráficos no ambiente de um desenho em AutoCAD.

Elementos Básicos da Linguagem

Funções

Programas em AutoLISP são compostos de funções. Uma função em LISP pode, ou não, requerer argumentos, pode invocar outras funções e sempre retorna um resultado. A possibilidade de executar qualquer função diretamente favorece o desenvolvimento (e teste) modular de programas.

Para executar uma função AutoLISP diretamente da linha de comando basta teclar, entre parênteses, o nome da função desejada seguido dos argumentos necessários à sua execução.

Por exemplo: para calcular a soma de dois números inteiros **5** e **9**, faça (deixe um espaço em branco entre cada elemento):

Command: **(+ 5 9)**

14

Command:

obs: por convenção, o que o usuário digita está indicado em negrito.

A função **+** (adição) é uma das funções matemáticas primitivas do AutoLISP. O resultado (neste caso, 14) será apresentado na linha de comando. A expressão acima, delimitada por parênteses constitui uma lista ordenada, onde o primeiro elemento é (em geral) uma função e

[1] Procedural porque o programador especifica, passo a passo, como resolver problemas de um determinado tipo, isto é, a seqüência necessária de instruções para sua solução (em contrapartida existem as linguagens declarativas, ditas de 4ª geração, onde o programador especifica apenas o problema ser resolvido dentro de um certo domínio); de alto-nível porque independe da arquitetura do equipamento utilizado, sendo o esforço maior de programação orientado para a solução do problema (e não para sua implementação em linguagem de máquina).

[2] A linguagem (de 3ª geração) mais antiga ainda em uso é o Fortran, criado a partir de 1954 por John W.Backus e equipe.

os demais (separados por espaços em branco) são os argumentos passados para essa função. Dessa, forma há dois tipos de delimitadores na linguagem:

- **separadores**: representados por espaços em branco, servem para separar os símbolos, unidades moleculares de sintaxe da linguagem (funções, variáveis e constantes) e;
- **agrupadores**: representados por pares de parênteses, servem para delimitar as diferentes regiões de um programa de forma a prover estrutura ao mesmo.

Se um dos argumentos passados para uma função for uma lista, o primeiro item desta lista será considerado também uma função, cujo resultado é argumento da lista que o contém.

Por exemplo, se quisermos calcular a razão entre as somas **6+8** (numerador) e **4+3** (denominador), podemos escrever:

Command: **(/ (+ 6 8) (+ 4 3))**

Inicialmente são avaliados as funções internas, pois sem seus resultados (14 e 7) a divisão (entre 14 e 7) não pode ser executada. A seqüência de cálculo (em itálico), para a avaliação da expressão acima, será :

Command: **(/ (+ 6 8) (+ 4 3))**

(/ 14 (+ 4 3))

(/ 14 7)

2

Command:

Funções Aritméticas

Além das funções **+** (soma) e **/** (divisão), vistas acima, existem outras funções aritméticas, a saber:

- multiplicação:

  Command: **(* 14 7)**

  98

  Command:

- subtração:

  Command: **(- 14 7)**

  7

  Command:

por Dr. Marcelo Eduardo Giacaglia

- incrementa:

  Command: **(1+ 8)**

  9

  Command:

- decrementa:

  Command: **(1- 8)**

  7

  Command:

  obs: 1+ e 1- são os nomes das funções incrementa e decrementa!

- resto divisão:

  Command: **(rem 14 3)**

  2

  Command:

- módulo:

  Command: **(abs (– 7 14))**

  (abs –7)

  7

  Command:

- converte “real” (ponto flutuante) para inteiro:

  Command: **(fix pi)**

  3

  Command:

  obs: pi (3.14159) é uma constante do AutoLISP.

- converte inteiro para “real”:

  Command: **(float 3)**

  3.0

  Command:

- raiz quadrada:

  Command: **(sqrt 2)**

  1.41421

por Dr. Marcelo Eduardo Giacaglia

Command:

- exponenciação:

  Command: **(exp 1)**

  2.71828

  Command:

- logaritmo base natural:

  Command: **(log (exp 1))**

  1.0

  Command:

- potenciação:

  Command: **(expt 2 3)**

  8

  Command: **(expt 8 (/ 1.0 3.0))**

  2.0

  Command:

Na falta de funções que calculem potências ou raízes com expoente diferente de 2 (raiz quadrada), pode-se utilizar a relação:

$$a^x = e^{x \cdot \ln a}$$

Dessa forma para calcular, por exemplo, o cubo de dois (2^3) e a raiz cúbica de 8, as expressões utilizadas seriam:

Command: **(exp (* 3 (log 2)))**

8.0

Command: **(exp (* (/ 1.0 3.0) (log 8)))**

2.0

Command:

Observa-se acima que, no caso da divisão, foram utilizados números reais (com ponto flutuante). Se forem utilizados números inteiros (sem a indicação de ponto flutuante) o resultado da divisão é truncado (para zero, neste exemplo) e a expressão não retorna o valor esperado.

Command: **(exp (* (/ 1 3) (log 8)))**

por Dr. Marcelo Eduardo Giacaglia

(exp (0 (log 8)))*

(exp (0 2.07944))*

(exp 0)

1.0

Command:

Do exemplo acima, percebe-se que o interpretador da linguagem AutoLisp atribui tipos (inteiro, ponto flutuante, etc.) aos dados processados com base na informação disponível a cada avaliação de função. Deve-se, portanto, ter cuidado na montagem das expressões submetidas a ele. Os tipos de dados manipulados pelo AutoLisp são apresentados adiante.

Funções Circulares

Os ângulos são sempre dados em radianos. Pode-se calcular, diretamente, por meio de funções, o seno e o cosseno de ângulos assim como a inversa da tangente (arco-tangente).

A conversão dos ângulos é feita por regra de três, sabendo-se que a 180^{o} correspondem π radianos.

- **seno** de um arco (de 0, 30, 45, 60 e 90 graus):

  Command: **(sin 0)**

  0.0

  Command: **(sin (* pi (/ 30.0 180.0)))**

  0.5

  Command: **(sin (* pi (/ 45.0 180.0)))**

  0.707107

  Command: **(sin (* pi (/ 60.0 180.0)))**

  0.866025

  Command: **(sin (/ pi 2))**

  1.0

  Command:

- **cosseno** de um arco (de 0, 30, 45, 60 e 90 graus):

  Command: **(cos 0)**

  1.0

  Command: **(cos (* pi (/ 30.0 180.0)))**

  0.866025

  Command: **(cos (* pi (/ 45.0 180.0)))**

0.707107

Command: **(cos (* pi (/ 60.0 180.0)))**

0.5

Command: **(cos (/ pi 2.0))**

6.12303e-017

Command:

Observa-se, do exemplo acima, que o resultado do cálculo do cosseno de 90^{o} , o resultado deveria ter sido zero, mas o retornado foi um número positivo, ainda que "muito pequeno" ($6{,}12303 \cdot 10^{-17}$).

- **arco-tangente** (de 0, 1, 0.57735 e 1.73205, em graus):

Command: **(/ (* (atan 0) 180) pi)**

0.0

Command: **(/ (* (atan 1) 180) pi)**

45.0

Command: **(/ (* (atan 0.57735) 180) pi)**

30.0

Command: **(/ (* (atan 1.73205) 180) pi)**

60.0

Command:

Os cálculos devem ser feitos considerando-se os ângulos sempre no primeiro quadrante, isto é, variando de 0 a 90 graus. Para ângulos maiores que 90 graus deve-se observar as relações abaixo:

$\text{sen}(90^{o} + \alpha) = \cos(\alpha)$ e $\cos(90^{o} + \alpha) = -\text{sen}(\alpha)$;

$\text{sen}(180^{o} + \alpha) = -\text{sen}(\alpha)$ e $\cos(180^{o} + \alpha) = -\cos(\alpha)$ e;

$\text{sen}(270^{o} + \alpha) = -\cos(\alpha)$ e $\cos(270^{o} + \alpha) = \text{sen}(\alpha)$.

Dados

Como toda linguagem de programação, o AutoLISP processa dados. Alguns dos tipos de dados primitivos do AutoLISP são encontrados em outras linguagens de programação, outros não. Os tipos de dados primitivos do AutoLISP incluem: constantes, variáveis e listas. Um dado

processado em um programa AutoLISP é também denominado um objeto AutoLISP (ou simplesmente objeto).

As constantes **T** e **NIL** representam, respectivamente, os valores lógicos, Verdadeiro e Falso. A constante **NIL** é também utilizada para representar uma lista vazia.

As variáveis do AutoLISP são rótulos associados internamente (por meio de apontadores) a espaços na memória do computador. Variáveis AutoLisp podem representar valores numéricos, literais (seqüências de caracteres ou strings), constantes, etc.

A atribuição de um valor a uma variável pode ser feita pela função primitiva **setq**, por exemplo,

Command: **(setq A (+ 5 2))**
7
Command:

associa à variável de nome **A** (primeiro argumento passado para a função **setq**) um espaço na memória do computador que guarda o valor **7** (resultante do cálculo passado como segundo argumento para a função **setq**).

A função **setq** permite atribuir valores a mais de uma variável, caso em que cada atribuição forma um par variável-valor. O valor retornado é o da última atribuição de valor executada.

Command: **(setq A 3 B (1- A))**
2
Command:

Outros Tipos de Dados

Além de constantes e variáveis, representando valores inteiros (INT) e ponto flutuante (REAL) e das listas (LIST), indicadas, em detalhe, mais adiante, o AutoLISP permite manipular outros tipos de dados (que podem ser identificados por meio da função **type**), dentre eles:

- símbolo (**SYM**):

Command: **(setq temp 'frio)**
FRIO
Command: **(type temp)**
SYM
Command:

- função (**SUBR**):

  Command: **(type setq)**

  SUBR

  Command:

- cadeia de caracteres (**STR**):

  Command: **(setq frase "vamos tomar um café?")**

  "vamos tomar um café?"

  Command: **(type frase)**

  STR

  Command:

Obs: a constante lógica **T** é do tipo **SYM**, e **NIL** é do tipo **NIL**.

Funções de manipulação de cadeias de caracteres

Cada símbolo que pode ser interpretado por um computador é representado internamente por um código binário composto de certa quantidade fixa de bits (binary digits). O **bit** é a menor unidade de dados de um computador, representando dois estados distintos: **0** ou **1**. Os primeiros microcomputadores utilizavam um código de 7 bits denominado **ASCII** (American Standard Coding for Information Interchange) o que permitia a representação de até (2x2x2x2x2x2x2) = 2^7=128 símbolos. Esta codificação foi logo estendida para 8 bits (256 símbolos) de modo a permitir, principalmente, a representação de caracteres internacionais (com acentuação), mas em diferentes versões, uma para cada contexto de uso.

Dessa forma, o código ASCII da letra "A" é 65, da letra "B" é 66 (65+1) e da letra "Z" é 90 (65+25). Os códigos ASCII dos dígitos "0" a "9" são, respectivamente 48 a 57 (48+9). As letras minúsculas "a" a "z" tem respectivamente os códigos 97 a 122 (97+25).

Outros símbolos, que não os alfanuméricos, também possuem códigos. Por exemplo, os códigos ASCII 33 a 47 representam, respectivamente, **! \ # $ % & ' ( ) * + , - . /**; os códigos 58 a 64 representam, respectivamente, **: ; < = > ? @**; 91 a 96 representam **["] ^ _ `** e ; 123 a 126 representam **{ | } ~**. O espaço em branco é representado pelo código 32. A Tabela 1, indica os caracteres segundo a padronização ASCII.

Tabela 1 - códigos ASCII para caracteres mostrados

		+1	+2	+3	+4	+5	+6	+7	+8	+9
30				!	\	#	$	%	&	‘
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	“	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~			
130										
140						‘	’			
150										
160		¡	¢	£	¤	¥	¦	§	¨	©
170	ª	«	¬	-	®	¯	°	±	²	³
180	´	µ	¶	·	¸	¹	º	»	¼	½
190	¾	¿	À	Á	Â	Ã	Ä	Å	Æ	Ç
200	È	É	Ê	Ë	Ì	Í	Î	Ï	Ð	Ñ
210	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û
220	Ü	Ý	Þ	ß	à	á	â	ã	ä	å
230	æ	ç	è	é	ê	ë	ì	í	î	ï
240	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù
250	ú	û	ü	ý	þ	ÿ				

por Dr. Marcelo Eduardo Giacaglia

Além disso, serve para representar outros símbolos, como os gerados, quando pressionamos as teclas Esc, Enter, Tab, Backspace, respectivamente 27, 13, 9 e 7.

As funções **chr** e **ascii** convertem números inteiros (de 8 bits) em caracteres e vice-versa.

Command: **(chr 64)**
"@"
Command: **(ascii "@")**
64
Command:

Os caracteres indicados na Tabela 1 são os gerados com uso da função **chr**. Pode-se gerar a tabela completa em outras aplicações, teclando **Alt** + **0** + **código ASCII** desejado, por exemplo, no **Bloco de Notas** (editor de texto do Windows). O caractere exibido depende da fonte de texto usada.

Se aplicada a uma cadeia de caracteres, a função **ascii** retorna o código do primeiro apenas.

Command: **(ascii "abcd")**
97
Command:

As funções **strcat** e **substr** permitem, respectivamente, concatenar cadeias de caracteres e copiar partes de cadeias de caracteres.

Command: **(setq C (strcat "abcd" "efg"))**
"abcdefg"
Command: **(substr C 1 4)**
"abcd"
Command: **(substr C 5 3)**
"efg"
Command:

Nota-se que, na função **substr**, os argumentos correspondem, na ordem, à cadeia de onde é feita a cópia, a posição do primeiro caractere a ser extraído e o número de caracteres a serem copiados.

Se o último argumento for omitido, todos os caracteres a partir do início (indicado pelo primeiro argumento) serão copiados. Se o primeiro argumento for maior que o número de caracteres da cadeia a função retorna "" (cadeia vazia). O número de caracteres de uma cadeia é dado pela função **strlen**.

por Dr. Marcelo Eduardo Giacaglia

Command: **(substr C 1)**

"abcdefg"

Command: **(substr C 8)**

""

Command: **(strlen C)**

7

Command: **(strlen (substr C 8))**

0

Command:

Números inteiros e reais

Números inteiros do AutoLisp constituem representações de inteiros matemáticos, mas limitados ao intervalo –2.147.483.648 a 2.147.483.647. A limitação é devida ao tamanho da memória do computador que é alocado para representação de cada inteiro, igual a 32 bits.

A representação dos números reais é mais complexa. Em computadores do tipo PC é comum usar dois tipos de números reais, ou ditos de ponto flutuante: ponto flutuante curto e ponto flutuante longo, ambos definidos pela norma ANSI/IEEE 754, especialmente por causa da estrutura do co-processador numérico: originalmente o chip Intel 8087 dos PCs XT (8088), depois o 80287 dos PCs AT (80286), 80387 dos i386 e, integrado ao principal, a partir do i486.

Ponto flutuante curto: de 32 bits, sendo o 1^{o} para o sinal (0 = positivo, 1 = negativo), os 8 bits seguintes para o expoente (inclusive o sinal) e os 23 bits finais para a mantissa (precisão).

O número tem a forma binária (interna):

seeeeeeeemmmmmmmmmmmmmmmmmmmmm

ou, simplificadamente: sEM

- para $0 < E < 255$ (2^{8}-1 ou 11111111 em binário) temos:

  $V = (-1)^{s} . 2^{(E-127)} . (1,M)$

- para E = 0 temos:

  $V = (-1)^{s} . 2^{(-126)} . (1,M)$

- para E = 0, s = 0 e M = 0 temos:

  V = 0,0

por Dr. Marcelo Eduardo Giacaglia

O número de ponto flutuante curto possui mantissa (M) variando de 0 a 2^{23} (0 a 0,8388608). Seu expoente (E) varia de $2^{–126}$ a 2^{128}, ou seja, aproximadamente de 1,175494 . $10^{(-38)}$ a 3,402824 . $10^{(+38)}$.

Ponto flutuante longo: de 64 bits, sendo o 1º para o sinal (0 = positivo, 1 = negativo), os 11 bits seguintes para o expoente (inclusive o sinal) e os 52 bits finais para a mantissa (precisão).

O número tem a forma binária (interna):

seeeeeeeeeeemmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm

ou, simplificadamente: sEM

- para 0 < E < 2047 (2^{11}-1) temos:

  $V = (-1)^{s} . 2^{(E-1023)} . (1,M)$

- para E = 0 temos:

  $V = (-1)^{s} . 2^{(-1022)} . (1,M)$

- para E = 0, s = 0 e M =0 temos:

  V = 0,0

O número de ponto flutuante longo possui mantissa (M) variando de 0 a 2^{52} (0 a 0,4503599627370496). Seu expoente (E) varia de 2–1022 a 21024, ou seja, aproximadamente de 2,225074 . $10^{(-308)}$ a 1,797693 . $10^{(+308)}$.

O número de ponto flutuante do AutoLisp/AutoCAD possui 32 bits de tamanho, mas, ao que tudo indica, seu expoente possui 11 bits (e não 8). Dessa forma, se sua representação interna for semelhante a da IEEE, restariam 20 bits para a mantissa, resultando em até 2^{20}, ou seja, 0,1048576.

Listas

Uma lista é uma seqüência ordenada de zero ou mais objetos AutoLISP, incluindo outras listas.

Todos os objetos AutoLisp exceto as listas são chamados de átomos. O objeto **NIL**, apesar de poder representar listas vazias, é também considerado como sendo um átomo.

Internamente, uma lista é uma corrente de objetos denominados células. Cada célula contém dois apontadores: **CAR** e **CDR** (essas denominações[3] são remanescentes da primeira implantação da linguagem LISP). Em uma lista simples, o CAR aponta para um objeto e o elemento CDR aponta para a próxima célula dessa lista.

Por exemplo, a lista simples (+ 2 5), pode ser representada graficamente pela Figura 1. Como podemos observar, em uma lista simples, o último CDR aponta para a constante **NIL**.

Uma lista, de maior complexidade, como, por exemplo, (+ 2 (* 3 5)), pode ser representada graficamente pela Figura 2.

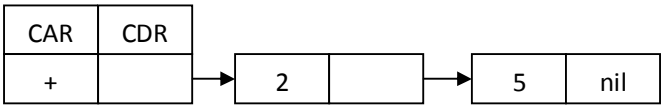

Figura 1 – representação gráfica da lista (+ 2 5)

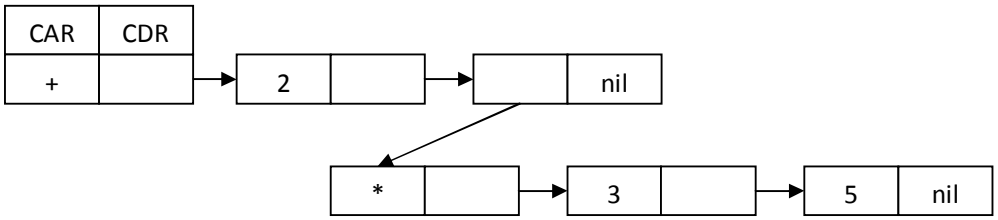

Figura 2 – representação gráfica da lista (+ 2 (* 3 5))

Para poder passar uma lista como argumento de uma função, sem que essa lista seja tratada como uma chamada de função, em si, deve-se precedê-la com uma apóstrofe (ou quote), por exemplo,

Command: **(setq B '(+ 2 3))**

(+ 2 3)

Command:

ou

[3] CAR – contents of address register e CDR – contents of decrement register, correspondem a operações em linguagem assembler do primeiro computador onde a linguagem Lisp foi implementada, um IBM-704.

Command: **(setq B (quote (+ 2 3)))**

(+ 2 3)

Command:

retorna e associa à variável **B** a lista (+ 2 3)

Outra forma de se obter o mesmo resultado é por meio da função **list**, conforme indicado abaixo.

Command: **(setq B (list + 2 3))**

(+ 2 3)

Command:

Se posteriormente quisermos processar a lista contida na variável **B**, podemos utilizar a função primitiva **eval**,

Command: **(eval B)**

5

Command:

Se o primeiro elemento da lista não for uma função, é possível aplicar a função desejada às suas células por meio da primitiva **apply**.

Command: **(setq C (list 2 3))**

(2 3)

Command: **(apply '+ C)**

5

Command:

Navegando em Listas

Os nomes **CAR** e **CDR** são também usados para designar funções primitivas do AutoLISP.

A função **car** retorna o valor do objeto apontado pelo primeiro **car** (da primeira célula) de uma lista. A função **cdr** retorna uma lista, formada a partir da original, onde o primeiro elemento é removido.

Por exemplo, seja **A**, a lista de números inteiros: 1, 2 e 3, criada pela expressão,

Command: **(setq A '(1 2 3))**

(1 2 3)

Command:

As expressões,	Retornam os valores:
(car A)	1
(setq B (cdr A))	(2 3)
(car B)	2
(setq C (cdr B))	(3)
(car C)	3
(setq D (cdr C))	NIL (uma lista vazia)

Como se pode notar, a aplicação consecutiva das funções **car** e **cdr** permite visitar, da primeira à última, todas as células de uma lista. A navegação na direção contrária não é tão fácil.

O acesso a uma dada célula pode ser feito diretamente por meio da função **nth**, caso em que se fornece a ordem da célula desejada. À primeira célula corresponde a ordem **0** (zero) e à última corresponde ordem igual ao número de células da lista menos um. Além disso, a função **last** retorna a última célula, a função **length** retorna o número de células da lista e a função **reverse** retorna uma lista com suas células na ordem inversa da lista original.

Command: **(setq A '(1 2 3))**
(1 2 3)
Command: **(nth 0 A)**
1
Command: **(nth 1 A)**
2
Command: **(nth 2 A)**
3
Command: **(nth 3 A)**
nil
Command: **(length A)**
3
Command: **(last A)**
3
Command: **(setq B (reverse A))**
(3 2 1)

Igualdade

Existem dois tipos de noções igualdade em AutoLISP. A primeira é a igualdade de objeto, a segunda é a igualdade de valor.

A função **eq** permite o teste de igualdade de objeto, isto é, se duas variáveis apontam para um mesmo objeto (ou se tratam-se das mesmas constantes). Por exemplo, as seguintes expressões (de teste de igualdade de objeto) retornam **T** (Verdadeiro):

Command: **(setq A 100000)**
100000
Command: **(setq B 100000)**
100000
Command: **(eq A B)**
T
Command: **(eq 100000 (+ 1 99999))**
T
Command: **(eq "A" "A")**
T

Command:

No teste de igualdade (de objeto), abaixo, a função eq retorna **NIL** (falso).

Command: **(eq '(1 2 3) '(1 2 3))**
nil
Command: **(setq A '(1 2 3))**
(1 2 3)
Command: **(setq B '(1 2 3))**
(1 2 3)
Command: **(eq A B)**
nil
Command:

Para efetuar teste de igualdade de valor, utiliza-se a função **equal**.

Predicados

Além de **equal** e **eq**, existem outras funções de predicado que retornam **T** ou **nil**, conforme indicado a seguir.

- comparação numérica: <, <=, = , >=, > e /= (diferente)

  Command: **(> (* 2 3) (+ 1 4))**

  T

  Command:

- comparação numérica: **zerop** (= zero), **minusp** (< zero) e **numberp** (é número?)

  Command: **(setq B –1)**

  -1

  Command: **(numberp B)**

  T

  Command: **(zerop B)**

  Nil

  Command: **(minusp B)**

  T

  Command:

- verificação do tipo de objeto: **atom** (para átomos), **listp** (para listas) e **null** (para nil)

  Command: **(setq A (list 1 2 3))**

  (1 2 3)

  Command: **(atom A)**

  nil

  Command: **(listp A)**

  T

  Command: **(atom (car A))**

  T

  Command: **(listp (cdr A))**

  T

  Command: **(null (cadddr A))**

  T

  Command:

  obs: (caddr A) equivale a (car (cdr (cdr (cdr A))))

- mais de uma condição: **and**, **or** e **not**

  Command: **(and (atom A) (listp A))**

  nil

Command: **(or (atom A) (listp A))**

T

Command: **(not T)**

nil

Command:

Listas Impróprias

Uma Lista Imprópria é uma Lista AutoLisp na qual o último CDR aponta para um objeto diferente do objeto NIL.

As listas impróprias são declaradas de maneira similar às listas simples. Seus objetos componentes são escritos entre parênteses e separados entre si por espaços.

Entretanto, após o penúltimo elemento da lista e antes do último, insere-se um •(ponto).

As listas impróprias são pares de átomos separados por um ponto, onde o átomo à direita não é NIL.

Por exemplo, a lista imprópria (70 • 16), pode ser representada graficamente pela Figura 3.

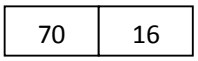

Figura 3 – representação gráfica da Lista Imprópria (70 • 16)

Para se criar uma Lista Imprópria como a da Figura 3, pode-se utilizar a função **cons**, como na expressão:

Command: **(cons 70 16)**

(70 • 16)

Command:

O uso da notação “de ponto” serve tanto para listas simples como para listas impróprias. Por exemplo:

- a lista, em notação “de ponto” (5 • NIL) é equivalente à lista simples (5).

  Command: **(setq A (cons 5 nil))**

  (5)

  Command:

- a lista, em notação “de ponto” (+• (2 • (5 • NIL))) é equivalente à lista simples (+ 2 5) representada internamente como indicado na Figura 1.

Command: **(cons '+ (cons 2 (cons 5 nil)))**

(+ 2 5)

Command:

Outras funções de manipulação de listas

Para concatenar listas pode-se utilizar a função **append** :

Command: **(setq L (append '(1) '(2) '(3)))**

(1 2 3)

Command: **(setq M (append '(0) L '(4)))**

(0 1 2 3 4)

Command:

Nota-se que a função **append** forma a lista com os argumentos das listas concatenadas. Se fosse utilizada a função **list**, a nova lista seria formada com as listas passadas como argumentos:

Command: **(setq L (list '(1) '(2) '(3)))**

(1 2 3)

Command: **(setq M (list '(0) L '(4)))**

((0) ((1) (2) (3)) (4))

Command:

A função **member** busca por uma célula na lista com valor igual ao da expressão fornecida como parâmetro. Se encontrar, retorna uma lista com as células de ordem maior ou igual que a da encontrada, caso contrário retorna **nil**.

Command: **(setq A '(1 2 3 4 5 6 7 8 9))**

(1 2 3 4 5 6 7 8 9)

Command: **(member 2 A)**

(2 3 4 5 6 7 8 9)

Command: **(member 9 A)**

(9)

Command: **(member 0 A)**

Nil

Command:

A função **mapcar** aplica a função dada pelo 1º argumento a cada elemento da lista, passada como 2º argumento, resultando numa lista.

Command: **(setq L1 (mapcar 'chr '(65 117 116 111 76 73 83 80)))**
("A" "u" "t" "o" "L" "I" "S" "P")
Command:

Se a função, correspondente ao 1º argumento, requer mais de um argumento, a cada argumento adicional deve corresponder também uma lista adicional, passada como argumento para a função **mapcar**.

Command: **(setq L2 '(1 2 3 4 5 6 7 8 9))**
(1 2 3 4 5 6 7 8 9)
Command: **(mapcar '+ L2 L2)**
(2 4 6 8 10 12 14 16 18)
Command:

Identificadores

Identificadores são objetos LISP que possuem:

- Nome;
- Lista de Propriedades.

Em AutoLISP atribuem-se nomes a certas constantes e também para a classe de objetos denominados identificadores (objetos gráficos do AutoCAD), apenas a esses.

Além disso, cada Identificador possui um conjunto de informações denominado **Lista de Propriedades** (**LP**).

O CAR de um PV de uma LP aponta para (contém) o nome de uma propriedade e o CDR desse PV aponta para (contém) o valor associado àquela propriedade. Isto significa que um PV é na verdade uma Lista Imprópria com dois elementos.

A função **entget** extrai a LP de um Identificador - comumente um objeto gráfico do AutoCAD - retornando uma **Lista de Pares Associados Propriedade-Valor** (**LPA**).

Pode-se manipular uma LPA por meio das seguintes funções:

- **assoc** : recebe como argumentos o nome de uma propriedade e a LPA na qual ela se encontra e retorna, caso encontre, o PV com a propriedade e o valor associado a ela;

- **subst** : recebe como argumentos o novo PV, o PV a ser substituído e a LPA aonde a substituição deve ocorrer.

A função **entmod** permite atualizar a LP de um Identificador (objeto gráfico do AutoCAD) a partir de uma LPA (em geral extraída anteriormente desse mesmo objeto).

Para que os conceitos apresentados acima fiquem claros, vamos ver alguns exemplos de uso das funções de manipulação de LPs e LPAs:

- Seleção de um objeto AutoCAD e extração de sua LP - a função **entsel** permite a seleção de um único objeto, por exemplo, por meio do click do mouse sobre o objeto como uma linha (line), e retorna um PV onde o CAR aponta para o Nome do objeto e o CDR aponta para as coordenadas do ponto do objeto selecionado:

  Command: **(setq L1 (entget (car (entsel))))**

  retorna uma LPA semelhante a:

  ((-1 • <Entity name: 21e0540>) (0 • "LINE") (5 • "28") (100 • "AcDbEntity") (67 • 0) (8 • "0") (100 • "AcDbLine") (10 8.20734 5.78385 0.0) (11 14.0677 7.9585 0.0) (210 0.0 0.0 1.0))

  onde as propriedades:

  - -1 : indica o Nome do objeto gráfico AutoCAD (Identificador);
  - 0 : indica o tipo do objeto;
  - 5 : indica o Handle do objeto;
  - 67 : indica se o objeto está no Model Space ou no Paper Space;
  - 8 : indica que a Layer aonde se encontra o objeto;
  - 10 : indica as coordenadas do ponto extremo inicial do objeto do tipo line;
  - 11 : indica as coordenadas do ponto extremo final do objeto do tipo line;
  - 210 : indica o plano, no espaço 3D, no qual o objeto se encontra, por meio de um vetor unitário normal a esse plano.

- Alterar a **Color** (Cor) do objeto selecionado para **1** (Vermelho) - no exemplo acima a LPA não possui PV indicativo de Cor (propriedade **62**), indicando que ela é a do **Layer** aonde se encontra o objeto (byLayer), caso em que há necessidade de acrescentar tal informação:

  (setq NOVACOR (cons 62 1))

(setq L1 (append L1 (list NOVACOR))

(entmod L1)

- Alterar o ponto de origem do objeto do tipo **line** para as coordenadas (5.0 5.0 0.0):

(setq NOVAORI '(10 5.0 5.0 0.0))

(setq L1 (subst NOVAORI (assoc 10 L1) L1))

(entmod L1)

- Transportar o objeto do Model Space para o Paper Space !

(setq L1 (subst (cons 67 1) (assoc 67 L1) L1))

(entmod L1)

Programação em AutoLisp

Um programa, escrito em linguagem procedural de alto-nível, como o AutoLisp, é constituído por uma seqüência de instruções. As instruções são organizadas, como descritas anteriormente, por meio de delimitadores: parênteses e espaços em branco.

A seqüência normal de processamento das instruções se dá da esquerda para a direita e do nível interno de parênteses para o externo. Entretanto, comumente, existe a necessidade de desviar o fluxo de execução, dado um contexto, expresso por uma condição e/ou de repetir uma seqüência de instruções até que esta condição seja satisfeita.

Estrutura de Seqüenciamento e Controle da Execução

As estruturas de seqüenciamento e controle da execução das instruções de um programa podem ser classificados em:

- **Seqüencial** – fluxo normal de execução, da esquerda para a direita e, no caso de haver expressões entre parênteses, de dentro para fora;
- **Condicional** – execução condicional, isto é, de uma instrução (ou conjunto de instruções[4]) ou outra (ou outro conjunto de instruções) conforme uma condição;
- **de Repetição** – execução repetida de um conjunto de instruções, até satisfazer uma determinada condição;

[4] Por meio da função **progn**, descrita abaixo

Condicional

Duas funções **if** e **cond** podem ser utilizadas para se executar, ou não, partes de um programa, mediante uma condição ou conjunto de condições.

- função **if** <condição> <instrução> [<instrução>]

A título de exemplo, vamos supor que se deseja calcular o módulo de um número, sem o uso da função **abs**. Para tanto, o número fornecido é devolvido como resultado caso seja maior do que zero, e seu sinal invertido caso contrário.

Command: **(setq A 3)**
Command: **(if (> A 0) (+ A) (- A))**
3
Command:

A função **if** aceita 2 a 3 argumentos, sendo o primeiro uma expressão condicional, no exemplo (> A 0). O segundo argumento corresponde ao que deve ser executado caso a condição resulte verdadeira (**T**). O terceiro argumento, opcional, corresponde ao que deve ser executado caso a condição resulte falsa (**nil**). Se, no exemplo acima, o terceiro argumento for omitido o resultado para números negativos será **nil**.

Command: **(setq A -2)**
-2
Command: **(if (> A 0) (+ A))**
nil
Command:

- função **cond** ((<condição-1> <instrução-1>) [...(<condição-i> <instrução-i>)

...])

Verifica, na ordem, as condições 1, ..., i, ... A primeira das condições satisfeita resulta na execução da instrução correspondente e apenas ela, desconsiderando as condições posteriores.

Para executar uma instrução caso nenhuma das condições seja satisfeita pode-se acrescentar o par (T <instrução-n>) ao fim da lista. Por exemplo, a instrução abaixo retorna o próprio número (variável B) se ele for Real; converte para Real se ele for Inteiro ou; retorna 0.0, caso B seja de qualquer outro tipo.

Command: **(setq B "A")**

"A"

Command: **(cond ((eq (type B) 'INT) (float B)) ((eq (type B) 'REAL) B) (T 0.0))**

0.0

Command:

de Repetição

Duas funções **repeat** e **while** podem ser utilizadas para se obter repetição de uma instrução. A função **repeat** executa a instrução um número fixo (dado) de vezes, enquanto que a função **while** executa enquanto determinada condição seja verdadeira.

Como demonstração do uso de ambas, escolheu-se o cálculo do fatorial de um número, por exemplo **4!**:

- função **repeat** <número de vezes> <instrução a repetir>

  Command: **(setq n 4 r 1 i 0)**

  0

  Command: **(repeat n (setq i (1+ i) r (* r i)))**

  24

  Command:

- função **while** <condição> <instrução a repetir>

  Command: **(setq n 4 r n)**

  4

  Command: **(while (> n 1) (setq n (1- n) r (* r n)))**

  24

  Command:

Reutilização de Programas

Em todos os exemplos apresentados acima, percebe-se que toda vez que há necessidade de resolver um problema de um dos tipos já resolvidos anteriormente (por meio de um programa) deve-se de reescrever tal programa substituindo-se apenas os valores fornecidos.

Tal procedimento, pouco eficiente nos casos simples apresentados anteriormente, é na prática (programas de maior tamanho) também inviável. Salvo os casos triviais como, por exemplo, o da soma de dois números, os programas devem poder ser escritos (com editor de textos

simples[5]), armazenados para uso futuro, carregados na memória do computador e utilizados tantas vezes quanto se desejar (alterando-se apenas os dados de entrada a cada vez).

A carga de um programa pode ser feita por meio do comando **appload** do AutoCAD (quando previamente escrito com um editor de texto simples) ou diretamente pela linha de comando (menos eficiente):

Command: **(defun c:fatorial (/ n r) (setq n (getint "Número:") r n) (while (> n 1) (setq n (1- n) r (* r n))))**
C:FATORIAL

Pode-se verificar se o programa realmente foi carregado na memória do computador, por exemplo, por meio da função **type**.

Command: **(type fatorial)**
SUBR

Depois, executar quantas vezes quiser, até encerrar a aplicação AutoCAD:

Command: **fatorial**
Número: **4**
24
Command: **fatorial**
Número: **5**
120
Command:

No exemplo anterior foi feito uso da função **getint**. Tal função tem por finalidade solicitar, ao usuário, a entrada de um número inteiro para que possa ser atribuído a uma variável (pela função **setq**). O argumento passado para a função **getint** corresponde ao que é indicado na linha de comando para o usuário, quando o valor lhe é solicitado (**prompt**).

A função **defun** tem por finalidade indicar que se trata de um programa a ser armazenado na memória do computador.

O símbolo **C:** antes do nome da função indica que se trata de um programa que pode ser chamado diretamente pelo usuário. Na sua falta a função é considerada interna e acessível por outras funções, ou quando colocada entre parênteses.

[5] Simples significa, neste caso, sem necessidade de recursos elaborados de formatação, o que pode até resultar em erro, devido à existência de caracteres especiais, invisíveis na tela, mas inseridos no texto digitado.

Command: **(defun soma (/ a b) (setq a (getint "A:") b (getint "B:")) (+ a b))**

SOMA

Command: **(type soma)**

SUBR

Command: **soma**

Unknown command "SOMA". Press F1 for help.

Command: **(soma)**

A: **4**

B: **3**

7

Command:

O nome da função é sucedido por uma lista de parâmetros e/ou variáveis locais, que também pode ser vazia. Os parâmetros, se existirem, vêm em primeiro lugar, as variáveis locais, se existirem, vêm em seguida após o símbolo **/**.

Os parâmetros indicam que a função chamada deve ser acompanhada dos argumentos correspondentes. As variáveis locais são utilizadas pela função apenas durante sua execução, sendo descartadas ao término do processamento.

Qualquer outra variável utilizada, que não foi declarada como parâmetro ou de escopo local, é considerada de escopo global. Isto significa que a memória alocada para ela não é liberada após o término do processamento da função. Dessa forma, a variável (e o valor associado a ela) pode ser utilizada mais tarde por outra função. Mais de duas funções fazendo uso (recuperando e alterando o valor) de variáveis globais de mesmo nome podem não trazer os resultados esperados (efeito colateral).

Após a lista de possíveis parâmetros e argumentos vem o corpo da função, a seqüência de instruções requeridas para a solução do problema delegado a ela. O último valor calculado é retornado para quem chamou a função.

Funções para entrada de dados pelo usuário

Além da função **getint**, existem outras, utilizadas de acordo com o tipo de dado requerido, a saber:

- **getreal** : solicita um número real (de ponto flutuante). O ponto pode ser omitido.

  Command: **(setq r (getreal "R:"))**

  R: **1**

1.0

Command:

- **getstring** : solicita uma cadeia de caracteres. A função aceita o parâmetro **T** (além do prompt) para permitir a digitação de mais de uma palavra, separadas por espaços em branco.

  Command: **(setq s (getstring T "Digite seu nome:"))**

  Digite seu nome: **José da Silva**

  "José da Silva"

  Command:

- **getangle** : solicita um vetor, pela indicação de dois pontos. Retorna o ângulo, medido em radianos, entre a projeção do vetor fornecido no plano XY e a direção X da UCS. As coordenadas podem tanto ser digitadas como entradas diretamente na área de desenho com o apontador.

  Command: **(setq a (getangle "First point: "))**

  First point: **10,10**

  Specify second point: **@0,10,10**

  1.5708

  Command:

  Pode-se opcionalmente fornecer como parâmetro as coordenadas do primeiro ponto.

  Command: **(setq p1 '(0 0 0) a (getangle p1 "Indique a direção: "))**

  Indique a direção: **@-10,0,10**

  3.14159

  Command:

Observa-se, do exemplo acima, que as coordenadas de um ponto podem ser fornecidas sob a forma de uma lista de 2 até 3 números, representando respectivamente os deslocamentos nas direções x, y e (opcionalmente) z.

- **getcorner** : dado a coordenada de um ponto (como 1^{o} argumento), solicita a entrada de outro ponto, indicado visualmente por uma janela na área de desenho.

  Command: **(setq c (getcorner '(10 10) "Indique o outro canto: "))**

  Indique o outro canto: **@10,10**

  (20.0 20.0 0.0)

Command:

- **getdist** : solicita um vetor, pela indicação de dois pontos. Retorna a distância, as coordenadas podem tanto ser digitadas como entradas diretamente na área de desenho com o apontador.

  Command: **(setq d (getdist "First point: "))**
  First point: **0,0,0**
  Specify second point: **10,10,10**
  17.3205
  Command:

  Pode-se opcionalmente fornecer como parâmetro as coordenadas do primeiro ponto.

  Command: **(setq p2 '(0 0 0) d (getdist p2 "Indique o outro pto.: "))**
  Indique o outro pto.: 10,10,10
  17.3205
  Command:

- **getpoint** : solicita as coordenadas de um ponto. A função aceita como parâmetro (opcional) as coordenadas de um ponto inicial (além do prompt) a partir do traça um segmento de reta que acompanha o movimento do apontador.

  Command: **(setq p3 (getpoint '(10 10 10) "Novo pto.:"))**
  Novo pto.: **@**
  (10.0 10.0 10.0)
  Command:

- **getkword** e **initget**: a função **getkword** solicita uma cadeia de caracteres dentre as previamente definidas pela função **initget** chamada imediatamente antes.

  Command: **(initget "Sim Não Talvez")**
  nil
  Command: **(setq op (getkword "Opção [Sim/Não/Talvez]: "))**
  Opção [Sim/Não/Talvez]: **Certamente**
  Invalid option keyword.
  Opção [Sim/Não/Talvez]: **Sim**
  "Sim"
  Command:

No exemplo anterior, se o usuário fornecer uma entrada nula (pressionar a tecla Enter) a função retorna nil. Pode-se tornar obrigatória a entrada de um valor não nulo por meio de um parâmetro (1) opcional.

Command: **(initget 1 “Sim Não Talvez”)**
nil
Command: **(setq op (getkword "Opção [Sim/Não/Talvez]: "))**
Opção [Sim/Não/Talvez]:
Invalid option keyword.
Opção [Sim/Não/Talvez]: **Talvez**
“Talvez”
Command:

Tal parâmetro permite também outras formas de controle, em função do valor passado:

- **1** : não aceita entrada nula;
- **2** : não aceita valor zero (initget pode ser utilizado em todas funções de entrada de dados pelo usuário, aonde cabível);
- **4** : não aceita valor negativo;
- **64**: descarta deslocamento na direção z (em getdist);
- para combinar dois ou mais controles basta somar seus números (todos são potências de dois e dessa forma geram combinações unívocas). Por exemplo, o **7** como argumento (1+2+4) indica que a entrada não pode ser nula, nem zero ou negativa.

Outras formas de repetição

Além das funções de repetição por um número fixo de vezes e condicional, respectivamente **repeat** e **while**, indicadas acima, pode-se fazer uso de outros recursos, conforme a necessidade.

- **Recursão** : a estrutura da linguagem LISP favorece a implantação de algoritmos que fazem uso da recursão. Por exemplo, o cálculo do fatorial de um número pode ser implementado da forma indicada a seguir.

```
(defun c:fatorial ( / N)
        (setq N (getreal "Número:"))
        (fator N)
)
(defun fator (N / )
        (if (> N 1) (* N (fator (- N 1))) 1.0 )
)
```

A função fatorial, definida pelo usuário, pede que se forneça um número e em seguida mostra o resultado da aplicação da função interna fator, também definida pelo usuário.

A função fator recebe o argumento N; verifica se é maior do que 1 (um), caso em que faz a multiplicação do argumento N recebido pelo fatorial de (N - 1) chamando ela mesma (aqui está a recursão!) com argumento (N - 1) ou; retorna 1.0 (um) caso contrário (N<1).

Para escrever tais funções, utilize um editor de textos simples, digite o programa acima e salve em disco rígido (HD) como **fatorial.lsp**; para carregá-la utilize o comando **appload** (indique o arquivo fatorial, recém gravado) e; para executá-la basta digitar seu nome, fatorial, na linha de comando.

Alternativamente (com a única finalidade de agilizar a verificação deste exemplo), pode-se digitar o texto acima, linha a linha, na linha de comando do AutoCAD.

```
Command: (defun c:fatorial ( / N)
(_> (setq N (getreal "Número:"))
(_> (fator N)
(_> )
C:FATORIAL
Command: (defun fator (N / )
(_> (if (> N 1) (* N (fator (- N 1))) 1.0 )
(_> )
FATOR
Command: fatorial
Número:3
6.0
```

por Dr. Marcelo Eduardo Giacaglia

Command:

Nota-se, no exemplo acima, que o usuário pode utilizar diretamente a função interna fator para obter o mesmo resultado, caso em que se deve colocá-la entre parênteses e fornecer, como argumento, o número cujo fatorial se deseja calcular.

Command: **(fator 3)**

6.0

Command:

- Função **progn** : permite a execução de várias funções, indicadas como seus argumentos, mas retorna apenas o resultado da última. Isso corresponde, na prática, a escrever um programa dentro de outro. É utilizada, em especial, quando se deseja executar mais de uma instrução para uma dada condição em uma função if.

  Command: **(progn (setq A 1) (+ A 1) (+ A 2))**

  3

  Command:

- Função **foreach** : permite repetir a execução de uma função a cada célula de uma lista, mas retorna apenas o resultado da última.

  Command: **(foreach n (list 1 2 3) (expt n 2))**

  9

  Command:

Extended data

Xdata

A função **xdata** possibilita o registro de um pacote de dados, denominado **application**, sua associação com um objeto gráfico do AutoCAD e um conjunto de tipos de dados e valores correspondentes.

Por exemplo, pode-se associar a uma **line** existente, os dados (applications) MATERIAL e BITOLA, associar, respectivamente, os tipos string (STr) e real (Real), e atribuir-lhes, respectivamente, os valores “cobre” e 2.5 (mm^2).

Command: **xdata**

Initializing...

Select object:

Enter application name: **material**

MATERIAL new application.

Enter an option

[3Real/DIR/DISP/DIST/Hand/Int/LAyer/LOng/Pos/Real/SCale/STr/eXit] <eXit>: **st**

Enter ASCII string: **cobre**

Enter an option

[3Real/DIR/DISP/DIST/Hand/Int/LAyer/LOng/Pos/Real/SCale/STr/eXit] <eXit>:

New xdata appended.

Command: **xdata**

Select object:

Enter application name: **bitola**

BITOLA new application.

Object has 14 bytes of Xdata - new Xdata will be appended.

Enter an option

[3Real/DIR/DISP/DIST/Hand/Int/LAyer/LOng/Pos/Real/SCale/STr/eXit] <eXit>: **r**

Enter real number: **2.5**

Enter an option

[3Real/DIR/DISP/DIST/Hand/Int/LAyer/LOng/Pos/Real/SCale/STr/eXit] <eXit>:

New xdata appended.

Command:

Xdlist

A função **xdlist,** seguida da seleção de um objeto gráfico e da seleção de um, ou todos, **application**s (<*>) desse objeto, exibe seus tipos e valores cadastrados.

No exemplo anterior, se for selecionada a **line**, com a opção "todos" (<*>), serão exibidos o MATERIAL "cobre" e a BITOLA de 2.5 (mm^2).

Command: **xdlist**

Select object:

Enter application name <*>:

* Registered Application Name: MATERIAL

* Code 1002, Starting or ending brace: {

* Code 1000, ASCII string: cobre

* Code 1002, Starting or ending brace: }

* Registered Application Name: BITOLA

* Code 1002, Starting or ending brace: {

* Code 1040, Real number: 2.5

* Code 1002, Starting or ending brace: }

Object has 16353 bytes of Xdata space available.

Command:

Applications registrados em um desenho

Para a manipulação desses dados são necessários programas, como os ilustrados a seguir.

A função **tblnext**,com os argumentos **"APPID"** e **'T**, retorna uma lista (atribuída a **id1**) contendo o 1º **application** registrado, no caso "ACAD".

Command: **(setq id1 (tblnext "APPID" 'T))**

((10 **.** "APPID") (2 **.** "ACAD") (70 **.** 0))

Command:

A mesma função, sem o argumento **'T**, retorna uma lista (atribuída a **id1**) com o próximo **application** registrado, no caso "ACAD_PSEXT".

Command: **(setq id1 (tblnext "APPID"))**

((10 **.** "APPID") (2 **.** "ACAD_PSEXT") (70 **.** 0))

Command:

Executando a função acima, mais duas vezes, tem-se para os exemplos acima, os **application**s registrados pelo usuário. Pode-se aproveitar para reorganizar seus nomes numa nova lista (por exemplo, **lista_app**).

Command: **(setq id1 (tblnext "APPID"))**

((10 **.** "APPID") (2 **.** "MATERIAL") (70 **.**0))

Command: **(setq lista_app (list (cdr (assoc 2 id1))))**

("MATERIAL")

Command: **(setq id1 (tblnext "APPID"))**

((10 **.** "APPID") (2 **.** "BITOLA") (70 **.** 0))

Command: **(setq lista_app (append lista_app (list (cdr (assoc 2 id1)))))**

("MATERIAL" "BITOLA")

Command:

Se houvessem mais applications, eles poderiam ser adicionados à lista **lista_app**, se repetindo os dois últimos comandos acima.

A extração dos applications da nova lista **lista_app** pode ser feito de diversas formas, segundo a necessidade, por exemplo:

Command: **(car lista_app)**

por Dr. Marcelo Eduardo Giacaglia

“MATERIAL”

Command: **(cdr lista_app)**

(“BITOLA”)

Command: **(cadr lista_app)**

“BITOLA”

Command: **(length lista_app)**

2

Command: **(nth 0 lista_app)**

“MATERIAL”

Command: **(nth 1 lista_app)**

“BITOLA”

Command: **(setq i 0 j (length lista_app))**

2

Command: **(repeat j (progn (princ (nth i lista_app)) (print) (setq i (1+ i)) (princ)))**

MATERIAL

BITOLA

Command:

Manipulação de extended data de um desenho

Nos exemplos anteriores, da adição de extended data a uma line, além da função **xdlist**, pode-se buscar acessar esses dados por meio de outras funções AutoLISP. A execução das funções:

Command: **(setq linha (car (entsel)))**

Select object: <Entity name: 7ef59f68>

Command:

pede que o usuário selecione um objeto, no caso a **line**, da qual e retorna o nome (atribuído à **linha**).

Cabe notar que, mesmo que se peça para exibir a especificação da line, sua extended data não é mostrada.

Command: **(entget linha)**

((-1 . <Entity name: 7ef59f68>) (0 . "LINE") (330 . <Entity name: 7ef59cf8>) (5 . "E5") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "0") (100 . "AcDbLine") (10 317.689 224.189 0.0) (11 937.502 744.702 0.0) (210 0.0 0.0 1.0))

Para mostrar extended data há necessidade de incluir um argumento adicional, a lista de applications cujos valores se deseja verificar.

Command: **(entget linha '("MATERIAL"))**

(... (-3 ("MATERIAL" (1002 . "{") (1000 . "cobre") (1002 . "}"))))

Command: **(entget linha '("BITOLA"))**

(... (-3 ("BITOLA" (1002 . "{") (1040 . 2.5) (1002 . "}"))))

Command: **(entget linha lista_app)**

(... (-3 ("MATERIAL" (1002 . "{") (1000 . "cobre") (1002 . "}")) ("BITOLA" (1002 . "{") (1040 . 2.5) (1002 . "}"))))

Command:

Casos de Uso de Extended Data

- **Associative hatch**

O application "ACAD" é usado para indicar que determinado objeto possui associative hatch. Quando se aplica as funções:

Command: **(entget (car (entsel)))**

a um objeto com hachura associativa, sua lista de pares propriedade-valor exibe também algo como:

(... (102 **.** "{ACAD_REACTORS") (330 **.** <Entity name: 9ed56f21 >) (102 **.** "}") ...)

onde **Entity name:** se refere ao nome da hachura em questão.

- **Topologia de redes**

Um uso possível em aplicações de análise de redes é a definição explícita de relações topológicas entre seus arcos e nós.

Um arco orientado (representado por uma linha, por exemplo, line, polyline ou spline) possui origem num Nó e destino em outro Nó. Um nó (representado por figura qualquer, por exemplo, um circle ou rectangle) pode estar conectado a outros nós, por meio de um ou mais arcos (de entrada ou saída).

Um arco possui como extended data os handles dos nós inicial e final dos seus nós, enquanto que um nó possui os handles dos arcos de entram e saem dele.

Command: **xdata**

... Enter application name: **inicio**

... [3Real/DIR/DISP/DIST/Hand/Int/LAyer/LOng/Pos/Real/SCale/STr/eXit] <eXit>: h

Enter database handle: **"5b2"**

...

Command: **xdata**

... Enter application name: **termino**

... [3Real/DIR/DISP/DIST/Hand/Int/LAyer/LOng/Pos/Real/SCale/STr/eXit] <eXit>: h

Enter database handle: **“5d4”**

...

Command: **xdata**

... Enter application name: **entra**

... [3Real/DIR/DISP/DIST/Hand/Int/LAyer/LOng/Pos/Real/SCale/STr/eXit] <eXit>: h

Enter database handle: **“4ef”**

...

Command: **xdata**

... Enter application name: **sai**

... [3Real/DIR/DISP/DIST/Hand/Int/LAyer/LOng/Pos/Real/SCale/STr/eXit] <eXit>: h

Enter database handle: **“50c”**

...

Obs: a função **handent** recebe como argumento um **handle** e retorna o correspondente **Entity Name**.

Command: **(handent “e5”)**

<Entity Name: 7ef59f68>

Command: