UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA

INTRODUÇÃO À LINGUAGEM DE PROGRAMAÇÃO

COMMON LISP

por

Mara Abel

Porto Alegre, agosto de 1995.

PREFÁCIO

Este texto foi criado para servir de apoio aos iniciantes da linguagem de programação COMMON LISP, especialmente aqueles que estão buscando aprendê-la para desenvolver aplicações de Inteligência Artificial. Apresenta uma visão geral da linguagem, através de suas funções e exemplos, conduzindo o estudante em seu primeiro contato com a linguagem.

O capítulo inicial caracteriza a linguagem, enquanto que os capítulos 2 a 9 selecionam, apresentam e fornecem exemplos das funções mais utilizadas no desenvolvimento de programas em LISP. Essa seleção não esgota, no entanto, o grande número de funções disponíveis na linguagem e que pode ser conhecido através dos manuais das diferentes versões ou de outras obras mais completas. Os capítulos de 10 a 13 introduzem o desenvolvimento de programas, apresentando alguns exemplos típicos de Inteligência Artificial. Cada capítulo inclui exercícios que revisam os tópicos apresentados e propõem novos desafios, cujas soluções são apresentadas no final do texto.

As soluções dos exercícios aqui apresentados, bem como os programas de exemplo devem ser creditados aos alunos do Curso de Bacharelado em Ciência da Computação da Universidade Federal do Rio Grande do Sul, nominalmente: Fábio Korbes, Gabriel Luhers Graça, Luís Alvaro de Lima e Silva e Vinícius Leopoldino do Amaral.

A edição final do texto, verificação dos exemplos, seleção, teste e correção dos programas apresentados foi resultado do paciente trabalho do bolsista de iniciação científica do CNPq, Luís Alvaro de Lima e Silva, a quem registro meu agradecimento.

Mara Abel

Porto Alegre, 25 de agosto de 1995.

ÍNDICE

1. INTRODUÇÃO

A Linguagem LISP (do inglês LISt Processing language) foi criada em 1958, pelo Dr. John McCarthy, um pesquisador do Massachussetts Institute of Technology (MIT - EUA), um dos grandes centros de pesquisa em Inteligência Artificial do mundo. A linguagem foi inicialmente implementada em um computador IBM 704 pelo Grupo de Inteligência Artificial do MIT.
O objetivo primeiro da linguagem foi a *manipulação simbólica*, ou seja, as configurações de bits no computador representam símbolos quaisquer e são tratados como se fossem palavras ou frases. Essa caraterística da linguagem permite a representação de objetos e procedimentos que são inviáveis de serem representados nas linguagens convencionais. Mais particularmente, o interesse do grupo de IA do MIT na época era simplificar o desenvolvimento e experimentações com o sistema *Advice Taker*, que procurava simular o comportamento humano em resposta a sentenças declarativas e imperativas que recebia.
A versão original, bastante simples, de Lisp do MIT foi apresentada pela primeira vez no artigo *Recursive Functions of Symbolic Expressions and their Computation by Machine*, de autoria de John McCarthy e publicado em 1960 na revista *Communications of ACM*. Esta, por sua vez, deu origem a diversos dialetos adaptados as mais variadas necessidades. Surgiram assim o Interlisp, Franzlisp, Maclisp, Zetalisp, Scheme e Le_Lisp, entre outros. Com o objetivo de normalizar a linguagem e aumentar sua portabilidade surgiu, em 1984, o Common Lisp, que reuniu as principais características dos dialetos mais utilizados (Maclisp, Zetalisp, Scheme e Interlisp, principalmente). Sua definição é apresentada no livro *Common Lisp: The Language*, de Guy Steele Jr. *Common Lisp* é uma linguagem bastante completa, contendo diversos recursos inicialmente utilizados apenas em linguagens procedurais, como arranjos e estruturas. Apesar de execrada pelos puristas, a introdução destes recursos é resultado da integração de uma série de alterações contidas nos dialetos Lisp que visavam extender a sua utilização e aumentar sua eficiêcia.
Outro passo importante na evolução do Lisp que segue a tendência de extender o seu âmbito de aplicação para além do domínio de IA, foi a criação do *Common Lisp Object System (CLOS)*, ou seja, a introdução de orientação a objetos no contexto do Lisp. CLOS permite ao programador explorar o potencial de orientação a objetos no desenvolvimento de sistemas complexos e robustos, ao mesmo tempo que dispõe dos recursos de Lisp para a implementação do comportamento e estrutura de dados internas dos objetos. Uma vez que CLOS foi um desenvolvimento incorporado *a posteriori*, não resultando de uma evolução da linguagem em sí, este texto apresentará primeiramente aspectos básicos linguagem Common Lisp para, a seguir, introduzir os conceitos de CLOS em uma seção em separado.

1.1 Características da Linguagem

Existem uma série de particularidades que diferenciam Lisp das linguagens de programação convencionais. Em primeiro lugar, Lisp é uma *linguagem interativa*, ou seja, o usuário dialoga com o sistema fornecendo expressões simbólicas que são avaliadas e cujo resultado é retornado para o usuário1. Nos primeiros ambientes Lisp isto era implementado por meio de um interpretador. Sistemas interpretados são claramente menos eficientes em tempo de execução do que sistemas compilados. Entretanto, o desenvolvimento de programas em um ambiente interpretado é muito

mais simples do que utilizando compiladores. O ambiente interpretado oferece um conjunto dinâmico de recursos ao programador onde funções e estruturas de dados podem ser criadas, eliminadas ou modificadas em tempo de execução, sem a necessidade de se recompilar e/ou religar todo o sistema a cada modificação realizada. Para melhorar a performance dos ambientes Lisp, sistemas atuais *compilam* as funções e formas fornecidas pelo usuário mantendo, entretanto, o seu caráter interativo.

Em termos gerais, podemos resumir as características do Lisp da seguinte maneira:

• Computação mais com expressões simbólicas do que com números ou símbolos aritméticos;

• Estrutura de dados basicamente composta por listas encadeadas internamente e listas de múltiplos níveis externamente;

• Estrutura de controle baseada na composição de funções para formar outras funções mais complexas;

• Processos e problemas descritos através de recursividade;

• Os programas são representados como listas exatamente como os dados o são;

• A função *eval*, escrita no próprio LISP, serve de intérprete para o LISP e cria um ambiente de programação completamente interativo;

• O gerenciamento de espaço na mémoria é feito automaticamente pela linguagem através de alocação dinâmica, e é transparente para o programador.

Existem poucas funções LISP básicas, todas as outras são definidas a partir destas. Este fato, aliado à representação de programas e dados de maneira uniforme, permite facilmente construir sistemas operacionais, compiladores ou interpretadores de texto em LISP.

Os programas compõem-se de módulos independentes que realizam tarefas específicas, praticamente sem interferirem uns nos outros. Essa facilidade de construção e modificação dos programas, aliado às habilidades de processamento simbólico tornou o LISP a linguagem mais utilizada para desenvolvimento de sistemas e ferramentas de Inteligência Artificial.

Uma característica importante de Lisp, herdada por linguagens novas como Java, é o gerenciamento automático de alocação/dealocação de memória. Em Lisp o programador não manipula diretamente ponteiros. Toda manipulação, criação e eliminação de blocos de memória é feita de forma transparente pelo sistema, que controla o acesso as estruturas alocadas e remove-as automaticamente quando não são mais necessárias, utilizando um mecanismo chamado de *coleta de lixo* (do inglês *garbage collection*). Esse mecanismo elimina uma das causas mais comuns de falhas de programa que são os erros de gerenciamento de memória. O preço a pagar neste

1Na verdade, por questões de eficiência computacional, os sistemas Lisp atuais usualmente compilam as
funções Lisp mantendo uma ligação entre a versão compilada e sua versão textual.

caso é uma redução de performance, especialmente nos momentos em que o sistema interrompe o programador para realizar a coleta de lixo.

1.2 Breve Passeio Pela Linguagem

Lisp é uma linguagem *interativa.* O usuário interage continuamente com o ambiente, fornecendo expressões ou *formas* que são avaliadas pelo sistema e cujo resultado é retornado ao usuário. Sob certo ponto de vista, o ambiente Lisp pode ser considerado

uma calculadora simbólica, que recebe uma forma do usuário, avalia-a e retorna o resultado. As formas que o usuário fornece podem ser basicamente *átomos* ou *listas*. Por exemplo:

> 5
5

O símbolo ">" é o *prompt* do sistema Lisp (no Macintosh Common Lisp - MCL - o *prompt* é "?"). Ele indica que o ambiente Lisp está disponível para receber uma forma a ser fornecida pelo usuário. A forma que segue o *prompt* é avaliada pelo sistema, que retorna na linha seguinte o seu resultado. No exemplo acima, um número é fornecido ao sistema, que retorna o próprio número como resultado, indicando a idéia um tanto óbvia de que o valor associado a um número é ele mesmo.

> 'a
a

Neste outro exemplo, a expressão " 'a " é fornecida ao sistema, que retorna o símbolo "a". Símbolos e números são átomos da linguagem. A cada símbolo pode ser associado um valor qualquer. O apóstrofe na frente do símbolo indica ao sistema que ele não deve avalia-lo.

A primeira vista, um programa Lisp parece um amontoado de parênteses ilegível. Isso decorre do fato de programas e estruturas de dados serem organizados em listas e estas serem exibidas ao usuário na forma de uma sequência de elementos delimitada por parênteses. Por exemplo, uma lista de três elementos, a, b e c é apresentada ao usuário como:

(a b c)

Enquanto que um função que retorna a soma de dois parâmetros poderia ser chamada como:

(soma x y)

A interpretação de uma lista como uma estrutura de dados ou como uma chamada de função depende do contexto onde ela ocorre. Isso pode parecer confuso numa primeira abordagem, mas reflete o fato de Lisp apresentar uma sintaxe extremamente simples. Todos os operadores da linguagem são expressos na forma de funções descritas com a mesma sintaxe: uma lista, onde o primeiro símbolo identifica a função e os restantes os parâmetros que ela recebe.

Os elementos da lista são separados por espaços. Além de listas, o outro tipo de dado básico em Lisp é o *átomo*. Um átomo, por sua vez, pode ser um símbolo ou um valor numérico.

2. EXPRESSÕES SIMBÓLICAS (*expressão-s*) ou FORMAS

São as estruturas de dados ou objetos do LISP. Podem ser átomos ou listas. Ao contrário das linguagens convencionais, são os objetos do Lisp que são tipados, não as variáveis. As variáveis podem ser associadas a qualquer objeto, sem que haja inconsistência de tipos.

2.1 Átomos

É dado primitivo do LISP que não pode ser subdividido. Pode ser:

• Numérico: um número qualquer, inteiro, ponto flutuante, octadecimal, hexadecimal, binário e números complexos.

• Simbólicos: são os objetos de dados da linguagem, iniciam com uma letra seguida de quaisquer caracteres (exceto brancos), sem tamanho definido.

Têm o papel de variáveis da linguagem, porém com estrutura complexa. Podem ser vistas como um registros encadeados uns aos outros, contendo:
Printname (ou *pname*): guarda o nome do símbolo como é referenciado pelo usuário. Os nomes dos símbolos são únicos para cada execução. No caso das constantes (números) o nome e valores são iguais, para as variáveis, o nome é aquele definido pelo programador e é igualmente representado internamente.
Valor: funcionam como variáveis "armazenando" valores de qualquer tipo.O valor de uma constante é seu próprio printname. O valor de uma variável é inicialmente (*unbound*). Se durante a execução, a variável receber algum valor através de *setq*, este valor irá substituir o valor anterior (*unbound* ou outro).
Definição de funções: como valor de um símbolo pode ser atribuído uma declaração de função a ser executada quando o símbolo é avaliado. O valor inicial da definição de função de um identificador é (*unbound*) e a associação de uma função com um nome é uma operação semelhante a atribuir um valor a uma variável. Essa característica permite que uma função manipule outras exatamente como qualquer dado.
Lista de propriedades (*plist*): Cada símbolo possui uma lista de informações associadas na forma de pares "propriedade-valor" que podem ser modificados por funções LISP. A propriedade dos pares contém o nome da propriedade atribuída ao objeto e é associado seu valor. Cada um dos campos da lista de propriedades do símbolo pode ser incluído/consultado/alterado. Por exemplo, ao identificador *homem* pode ser associado a lista de propriedades:
((meio terrestre) (alimentação onívoro) (postura bípede))
A lista de propriedades é manipulada através das funções:
(put pname val prop)
(get pname prop)
Inclusão: *(put ’homem ’terrestre ’meio)*
Consulta: *(get 'homem 'meio) ` terrestre*
Remoção: *(remprop ’homem ’meio)*
Existem dois átomos especiais na linguagem que sempre são avaliados para eles mesmos: *NIL* e *T*. O valor *NIL* (também pode ser considerado uma lista vazia) corresponde ao valor booleano falso e o T, ao valor verdadeiro (true).

2.2 Listas

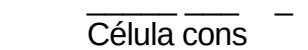
Uma seqüência qualquer de átomos ou outras listas delimitada por parênteses. Pode ser uma lista de dados como os exemplos abaixo, uma chamada de função , uma chamada de macro ou uma forma especial.
(janeiro fevereiro março)
(+ 4 5)
(A (B C) D (E))
( )

2.2.1 Representação Interna de Listas

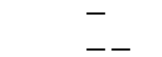
Uma lista, na linguagem LISP, difere dos arrays das linguagens convencionais na forma de seu armazenamento em memória. As listas correspondem a estruturas de dados de *listas simplesmente encadeadas*, onde os elementos da lista têm um formato único e são manipuladas de maneira similar a uma estrutura de pilha, ou seja, as inclusões e exclusões são feitas no início da lista.

Cada elemento de uma lista compreende um par interno ou célula *cons* (duas posições contíguas de memória) de ponteiros. Uma das posições aponta para o elemento da lista e outra aponta para o próximo elemento da lista. (Por simplificação, diz-se que uma posição "contém" o elemento da lista e outra "aponta" para o próximo).
A lista ***(A B C)*** corresponde a:

A primeira posição do par interno, que normalmente guarda o elemento da lista, é chamada *car* e a segunda posição, que guarda o endereço do próximo elemento da lista, é chamada *cdr*. Como a própria lista é referenciada através do endereço do primeiro elemento, costuma-se referir a *car* como o primeiro elemento da lista e a *cdr* como o restante da lista. O *cdr* do último elemento da lista contém o símbolo / ou *NIL* ou ( ), ou seja, a lista vazia.

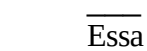


Célula cons
Lista : *L = (A B C D)*
(car L) → *A O átomo A.*
(cdr L) ` → *(B C D) A lista (B C D).*
Para a representação de listas complexas (listas de listas) o *car* do par interno que representa o elemento que é uma lista deve conter o ponteiro para o primeiro elemento - *cons* - par interno desta lista.
(A (B C) D E)



(A B (C (D E)) F (G))

Um par interno pode conter dois átomos, ao invés de um átomo e um ponteiro. Nesse caso, o par é representado pelos dois elementos entre parênteses e separados por um ponto.
(A . B)



Essa poderia ser uma representação coerente para pares de coordenadas cartesianas, resultando em uma economia de memória (uma posição a menos do que uma lista de dois elementos). Na prática, porém, evita-se utilizar pares internos para representar outra coisa senão listas, por uma questão de clareza da programação.

2.3 Exercícios
Desenhe a representação interna das seguintes listas:
1 - *(a b c (d))*
2 - *((a b) (c d) e (f g))*
3 - *(a (b (c (d) e) f) g h)*
4 - *(a b c (d . e) f)*

3. FUNÇÕES PRIMITIVAS

Uma função em LISP corresponde a uma lista cujo *car* é reconhecido pelo interpretador como um símbolo que possui uma definição de função assossiada e o *cdr*, uma lista de argumentos (átomos, listas ou outras funções).
As funções podem conter chamadas embutidas em qualquer nível inclusive recursivamente e sempre retornam um valor na chamada (ou seja, ou valor é devolvido no lugar do nome da função).
Nota: Para facilitar a compreensão do uso de uma função, os argumentos apropriados das funções são apresentados neste texto como:
lst - para lista ou função
atm - para átomo qualquer
exp - para lista ou átomo
O resultado da avaliação da função é dado após o símbolo "→" . Por simplificação, os exemplos incluem as chamadas de funções necessárias para atribuir valores aos átomos e listas, mas somente os resultados da função que está sendo apresentada são mostrados.

3.1 Funções de Seleção em Listas

3.1.1 (car lst)

Devolve o primeiro elemento de uma lista, ou *NIL* se a lista for vazia. Resulta em erro se aplicada a um átomo. Note que, em termos da estrutura interna, o car é o elemento apontado pelo respectivo ponteiro do cons.
(car '(A B C)) `→ A

– – –
—— –
_____
(car '((D E) F G)) `→ (D E)

–
—— –
——
–
___ –
(car (car '((D E) F G))) `→ D
car
car
D E
NIL
F G
(car ( )) `→ NIL

3.1.2 (cdr lst)

Devolve uma lista composta por todos os elementos da lista de entrada menos o *car*. Resulta em erro se aplicada a algum átomo. *car* e *cdr* são funções de consulta, não alteram a lista sobre a qual são aplicadas.
(cdr '(A B C)) →(B C)
A B C
NIL
(cdr '((D E) F G)) →` (F G)
F G
D E
NIL
NIL

(cdr (cdr '((D E) F G) →` (G)
E D
NIL
NIL
G F
cdr
cdr
(cdr (A))` →NIL
(cdr ()) ` →NIL
(car (car (cdr (cdr '(A B ((C) D) E F) →` (C)
Todas as composições de até 4 funções *car* e *cdr* são definidas como funções separadas em Common Lisp. O nome da composição correspondente é obtido simplesmente justapondo-se a segunda letra de cada função na ordem em que aparecem, da esquerda para a direita, delimitando-as com um *c* e e um *r*. Assim, *(car (cdr L)) → "c" + "a" + "d" + "r" →* (*cadr L)*.
Por exemplo, expressões como:
(car (car (car lst)))
(car (cdr (car (cdr lst))))
(cdr (cdr lst))
podem ser escritas respectivamente como:
(caaar lst)
(cadadr lst)
(cddr lst)

3.2 Funções de Construção de Listas

A função mais elementar para construir listas é *cons*. Ela aloca um par de ponteiros (*car . cdr*) e associa seus parâmetros a eles. *List* e *append* podem ser derivadas a partir de *cons*.

3.2.1 (cons expr1 expr2)

É construtor de listas. Retorna uma lista cujo *car* corresponde a *expr1* e o *cdr* a *expr2*. Se *expr1* e *expr2* forem átomos, devolve um par interno (ou célula *cons*) composta pelos dois elementos.
(cons 'A '(B C)) ` →(A B C)
(cons '(A) '(B C)) →((A) B C)
(cons '(A B) 'C))` →((A B) . C)
(cons 'A 'B) ` →(A . B)
(cons 'A NIL)` →(A)

3.2.2 (list expr1 expr2... exprn)

Retorna uma lista formada pelos elementos *expr1* , *expr2* e *exprn* .
(list 'A 'B 'C) ` →(A B C)
(list '(A B) '(B C)) →((A B) (C D))
(list 'A '(B C)) →(A (B C))
(list 'A 'B 'C ( )) →(A B C)
(list 'A) →(A)

3.2.3 (append expr1 expr2... exprn)

Retorna uma lista formada pelos elementos contidos no primeiro nível das listas *expr1 expr2... exprn* . Todos os argumentos com exceção do último devem ser listas. *Append* copia as listas passadas como parâmetro encadeando o campo *cdr* dos respectivos *cons*.
(append '(A B) '(B C)) →(A B C D)

(append ’(A) ’(B C) ( )) →(A B C)
(append nil ’A) →(A)
(append '((A 1) (B 1)) ‘(3 4)) →((A 1) (B 1) 3 4)
(append ‘(1 2) 3) → (1 2 . 3)
(append 'A 'B 'C) →ERRO!

3.2.3 Comparando *cons, list e append*

Para ilustrar as diferenças entre estes três comandos, considere o efeito da aplicação de cada um deles sobre as listas: ‘(Essa não) e ‘(de novo), cuja estrutura é:

!#"$"&% '_(_) *_+ '_)-,.)
(cons ‘(Essa não) ‘(de novo)) →((Essa não) de novo)
/#0&0&1 2_3_4
5_6 2_4-7.4
(list ‘(Essa não) ‘(de novo)) →((Essa não) (de novo))
8#9&9&: ;_<_= >_? ;_=-@.=
(append ‘(Essa não) ‘(de novo))→(Essa não de novo)
!#"$"&% '_(_) *_+ '_)-,.)

Nas ilustrações pode-se ver claramente as formas alternativas como os *conses* são gerados e encadeados. *cons* cria apenas um par de ponteiros. *list* cria um par para cada parâmetro enquanto *append* encadeia os conses sem criar nenhum novo par.

3.3 Funções de Manipulação de Listas

Testam propriedades ou realizam modificações sobre listas. Não aceitam átomos como argumentos.

3.3.1 (length lst)

Retorna o comprimento da lista, ou seja, o número de elementos que existem no primeiro nível da lista.

(length '(A B C)) ` →3
(length '((A B) (B C D)) ` →2
(length (append '(A B) '(B C D))) ` → 5
(length ( )) ` →0

3.3.2 (reverse lst)

Retorna a lista com seus componentes do primeiro nível em ordem inversa.

(reverse '(A B C)) ` →(C B A)
(reverse '((A B) C '(D E))) ` →((D E) C (A B))
(reverse (append '(A B) '(B C D))) ` →(D C B B A)

3.3.3 (subst nova-exp antiga-exp lst)

Substitui todas as ocorrências de *antiga-exp* por *nova-exp* na lista *lst*.

(subst 'A 'B '(A B C)) ` →(A A C)
(subst 'B 'A '(A B C)) ` →(B B C)
*(subst 5 'X '(* (+ X X) (/ 10 X))) ` →(* (+ 5 5) (/ 10 5))*
(subst 'correto 'errado '(O resultado está errado.)) → (O resultado está correto.)

3.3.4 (last lst)

Retorna uma lista contendo o último *cons* da lista dada.

(last '(A B C)) ` →(C)
A B C
NIL
(last '((A B) (C D)) ` →((C D))

A B C D
last
(last ( )) ` →( )
(last ‘(A B C . D)) →(C . D)
A B_C-D
D A B C
3.3.5 (nth n lst)
Retorna o enésimo elemento da lista lst, onde *n* é um inteiro positivo. O primeiro elemento da lista tem índice zero. Indexando-se além do tamanho da lista a função retorna *nil*.
(nth 0 ‘(1 2 3)) →1
(nth 2 ‘(1 2.3)) →Erro! 3 não é lista.
3.3.6 (nthcdr n lst)
Retorna o enésimo *cdr* da lista lst, onde *n* é um inteiro positivo. O índice zero indica a própria lista. Índices além do tamanho da lista geram *nil*, se a lista for própria, senão ocorre erro.
(nthcdr 0 (cons 1 (cons 2 3)))→(1 2 . 3)
(nthcdr 1 (cons 1 (cons 2 3)))→(2 . 3)
(nthcdr 5 (cons 1 (cons 2 3)))→Erro! 3 não é lista.
(nthcdr 5 (1 2 3)) →nil
***3.3.7* (remove *item* lst)**
Remove o elemento *item* da lista *lst*. Se item aparece repetidas vezes, todas as ocorrências são removidas. Retorna a lista resultante. Se o elemento não aparece na lista, retorna-a inalterada.
(remove ‘a ‘(b a n a n a)) →(b n n)
(remove ‘a ‘(1 2)) →(1 2)
3.4 Funções de Atribuição
3.4.1 (quote expr) ou ’expr
A função *quote* (abreviada por um apóstrofe) retorna seu argumento sem avaliação. Permite que um átomo ou lista sejam usados como listas de caracteres ou strings.
(quote A) ou *’A → A*
(quote (car (A B C))) ou *’(car (A B C)) →(car (A B C))*
(quote (quote (A B)) ou *’(quote (A B)) →(quote (A B))*
(car (quote (A B C))) ou *(car ’(A B C)) →A*
3.4.2 (set expr expr)
A função *set* avalia seus dois argumentos e retorna o valor avaliado para *expr*. Porém tem como "efeito colateral" o fato de associar ao resultado da avaliação do primeiro argumento o resultado da avaliação do segundo argumento.
(set (car '(A B C)) (cadr '(1 2 3))) ` →2 A = 2
(set 'L (append '(A B) '(C D E))) ` →(A B C D E)
L = (A B C D E)
(set 'B 'CERTO) ` →CERTO
B = CERTO
3.4.3 (setq atm expr)
A função *setq* é uma combinação das funções *set* e *quote*, ou seja, avalia apenas o seu segundo argumento retornando o resultado da avaliação. Associa, ainda, ao átomo *atm* (normalmente uma variável) o valor de *expr*. LISP não exige declaração prévia de variáveis de qualquer tipo. A simples atribuição de algum valor através de *setq* faz com que o interpretador passe a reconhecê-la como uma variável declarada

(naturalmente a consistência das atribuições deve ser garantida pelo programador). Qualquer variável que for referida ou utilizada sem ter sofrido atribuição prévia resultará em uma mensagem de erro de variável não definida *(unbound variable)*.

(setq A 5) ` →5 A = 5
(setq A '(F G H)) ` →(F G H) A = (F G H)
(setq A (car (cdr '(F G H)))) ` →G A = G
(setq A (quote B)) ` →B A = B (mesmo que *(setq A 'B)*)
(setq B 5) ` →5 B = 5
(quote A) ` →A A = A
(setq A B) ` → 5 A = 5

3.5 Predicados

São funções que testam as propriedades de seus argumentos.

3.5.1 (atom atm)

Testa se *atm* é um átomo, retornando *T* se for e *NIL* no caso contrário. Qualquer parâmetro que não seja uma lista gera valor T, como números, estruturas, arrays e objetos, por exemplo. Note que *nil* é uma exceção pois representa ao mesmo tempo uma lista vazia e o um símbolo.

(atom 'A) →T
(atom NIL) →T
(atom '()) →T
(atom '(A))→F

3.5.2 (boundp atm)

Espera um símbolo com escopo global (tipo *special*) como argumento. Retorna *T* se esse símbolo tiver algum valor associado e *NIL* no caso contrário.

(setq a 1) →5
(boundp 'A) →T
(defun teste (par)
(setq par 1)
(boundp 'par))
(teste 1) →F

3.5.3 (equal exp1 exp2)

Testa se *exp1* e *exp2* tem o mesmo valor associado. Retorna *T* se forem iguais e *NIL* se forem diferentes.

3.5.4 (null exp)

Testa se seu argumento é uma lista vazia. Retorna *T* se for e *NIL* no caso contrário.

3.5.5 (member exp lst)

Testa para ver se *exp* é parte da lista *lst*. Se *exp* for parte da lista, *member* retorna a sublista de *lst* que começa com *exp*. Se não for, retorna *NIL*.

(member 'C '(A B C D)) `→ (C D)

Originalmente, *member* retornava *T* ou *NIL*. Entretanto, julgou-se mais útil retornar uma informação adicional no caso de presença do elemento na lista. Por exemplo, a função

(precede-a X Y lst) verifica se *X* precede *Y* em uma lista *lst*.

(defun precede-a (X Y lst)
(member Y (member X lst)))
(precede-a 'Dom 'Pedro '(Dom Juan de Marco)) →(Dom Juan de Marco)
(precede-a 'dura 'agua '(agua mole em pedra dura)) →NIL

3.5.6 (consp elm)

Retorna T se *elm* é um *cons*, F caso contrário. Retorna F se *elm* é uma lista vazia, o

que o diferencia do predicado *listp.*
(consp nil) →F
(consp '(1 . 2)) →T
(consp '(1 2 3)) →T

3.5.7 (listp lst)

Retorna T se *lst* é uma *lista,* F caso contrário. Retorna T se *lst* é uma lista vazia, o que o diferencia do predicado *consp*.
(listp nil) →T
(listp '(1 . 2)) →T
(listp '(1 2 3))→T
(listp 1) →F

3.5.8 Comparadores Numéricos

São predicados que operam sobre números.
(numberp exp)
numberp testa se seu argumento é avaliado para um número, retornando T. Se não for um número, retorna NIL.
(minus exp)
minus espera um argumento numérico retornando seu valor com o sinal invertido.
(zerop exp)
zerop espera um argumento numérico retornando T se esse argumento avaliar para zero, e NIL caso contrário.
(minusp exp)
minusp espera um argumento numérico retornando T se esse argumento avaliar para um argumento avaliar para um número negativo, e NIL se for positivo.
(< num1 num2)
retorna T se num1 for menor do que num2.
(> num1 num2)
retorna T se num1 for maior do que num2.
(<= num1 num2)
retorna T se num1 for menor ou igual a num2.
(>= num1 num2)
retorna T se num1 for maior ou igual a num2.
(/= num1 num2)
retorna T se num1 for diferente de num2.

3.9 Exercícios

Avalie as seguintes expressões:
1 - *(car '(a b c))*
2 - *(cdr '(a b c))*
3 - *(car '((1 2) 3 (4 5)))*
4 - *(cdr '((1 2) (3 4)))*
5 - *(car (car (cdr '(a ((b c) d)))))*
6 - *(cdr (cdr (car '((a b c) d e))))*
7 - *(caddr '(a b (c (d e))))*
8 - *(caaar '(((a) b) c))*
9 - *(cons 'a '(b (c)))*
10 -*(cons '(b (c)) 'a)*
11 -*(list 'a '(b (c)))*
12 -*(list '(b (c)) 'a 'd 'e)*

13 -*(append ’a ’(b (c)))*
14 -*(append ’(a b (c)) ’(e) ’(f))*
15 -*(lenght '(morango laranja maçã uva))*
16 -*(lenght '(morango (laranja maçã uva)))*
17 -*(lenght '(morango (laranja (maçã) uva)))*
18 -*(reverse '(morango laranja (maçã uva)))*
19 -*(reverse (cddr '(morango laranja maçã uva)))*
20 -*(reverse (subst 'lima 'uva (cdr '(morango laranja maçã uva))))*
21 -*(last '(morango laranja (maçã uva)))*
22- *(last ‘(C . D))*
23 - *(last ‘((oi)))*

4. AVALIAÇÃO DE FUNÇÕES

Quando uma expressão simbólica S é digitada no ambiente LISP, ela é lida e avaliada pela função *eval*, segundo o algoritmo:

Se S é um átomo
Então Se S é igual a *T* ou *NIL*
Então retorne *T* ou *NIL*
Senão retorne o valor de S
Senão Se o *car* de S é *quote*
Então Retorne o segundo elemento de S
Senão Se o *car* de S é uma forma especial
(por exemplo: *let, setq, progn*)
Então não avalie os argumento e trate o caso específico
Senão Se o *car* é uma definição de macro
Então expanda a macro
Senão chame *eval* para cada um dos elementos
do *cdr* de S e aplique a função
associada ao *car* de S ao resultado
da avaliação.

Embora *eval* sempre seja acionado quando uma expressão S é fornecida ao LISP, pode ser chamada explicitamente para forçar uma avaliação dupla.

Exemplos:

(setq A ’B) →B
(setq B 'C) ` →C
A ` →B
B ` →C
(eval A) ` →C
(setq A '(car '(1 2 3))) → (car '(1 2 3))
A ` →(car '(1 2 3))
(eval A) ` →1
(setq EQN '(1 2 3)) → (* 1 2 3)*
EQN → (1 2 3)*
(eval EQN) → 6
(setq EQN '((car (cons 1 2)) (cdr (cons 2 3))))*
→ ((car (cons 1 2)) (cdr (cons 2 3)))*
EQN → ((car (cons 1 2)) (cdr (cons 2 3)))*
(eval EQN) → 3

5. DEFINIÇÃO DE FUNÇÕES

Há duas formas de definir uma função de forma a que possa ser chamada e aplicada sobre seus argumentos. Uma forma é associar uma definição de função a um símbolo, que passa a ser considerado o *nome* da função. Outra maneira é definir a função no local onde ela deveria ser chamada, sem dar-lhe nome algum. Na primeira forma, a função é definida através da forma especial *defun*, a segunda através de expressões *lambda*.

5.1 (defun <nome da função> (<lista de parâmetros>) (<descrição das ações>))

defun estabelece uma relação entre o átomo simbólico utilizado como nome da função e uma definição de uma seqüência de ações (na verdade, a função *lambda*, que contém o corpo da função). Essa definição inclui a forma de aplicar essas ações sobre os argumentos.

O nome da função deve ser um átomo. A lista de argumentos pode ter qualquer número de elementos e as ações são descritas como outras chamadas de funções que utilizam ou alteram os argumentos. Uma função pode ser pensada como uma caixa preta que recebe os valores dos seus argumentos como entrada e retorna um outro valor como saída (esse valor é o último valor avaliado e é retornado no ponto da chamada da função).

defun pode ser utilizada para redefinir (ou corrigir) uma função ou macro já existente para que trabalhe da forma desejada pelo programador. Resulta em erro se for utilizada para modificar a definição de uma forma especial.

Exemplo:

Uma função que recebe temperatura em graus Farenheit e retorna a mesma em graus Celsius.

(defun F-para-C (temperatura)
((- temperatura 32) 1.8))*
(F-para-C 80) ` → 26.6

5.2 (lambda (<lista de parâmetros>) (<descrição das ações>))

defun associa funções a um nome simbólico para que elas possam ser posteriormente chamadas pelo programador e executadas diversas vezes. Porém, determinadas funções só necessitam ser aplicadas uma única vez e, portanto, não necessitam serem associadas a um nome. Mais ainda, pode ser considerado um desperdício manter a definição da função disponível, se ela não será posteriormente utilizada. Neste caso, a função pode ser definida através da expressão *lambda*, no local onde ela deveria ser aplicada aos argumentos. A função é definida, aplicada, o último valor avaliado é retornado como seu valor e o corpo da função é descartado.

lambda é mais comumente utilizada no corpo de outras funções, quando é necessário manipular argumentos de uma forma específica, não prevista em outras funções.

Exemplos:

Definir uma função para obter uma divisão inteira de dois números. Os parênteses em negrito delimitam a função, que está exatamente no lugar de uma chamada de função (no *car* de uma lista), cujos argumentos são 17 e 5.

*((**lambda (a b) (round (/ a b))**) 17 5) ` → 3*

Incluir um elemento no final de uma lista:

((lambda (atm lst) (reverse (cons atm (reverse lst)))) *5 '(1 2 3 4))*
` (1 2 3 4 5)

5.3 Exercícios

1 - Defina uma função para selecionar o segundo elemento de uma lista qualquer.

2 - Defina uma função que retorna o elemento seguinte a um dado elemento de uma lista.
3 - Defina uma função que substitui todas as ocorrências do último elemento de uma lista por um outro átomo fornecido.
4 - Defina *calcula* que recebe como argumento uma lista de três números. Soma o segundo ao terceiro e multiplica pelo primeiro, retornando o resultado.
5 - Defina a função *extrai*, com dois argumentos, um átomo e uma lista, para retirar a primeira ocorrência do átomo na lista, retornando a lista modificada.

6. FUNÇÕES DE TESTE

São funções que definem os testes lógicos no corpo de outras funções de forma a alterar o fluxo do programa.

6.1 (cond
(teste1 ação1 ação2 ...)
(teste2 ação1 ação2 ...)
(teste3 ação1 ação2 ...)
...)

cond testa o primeiro argumento de cada lista, avalia (se for uma função, executa) até encontrar o primeiro valor não-*NIL*. Neste caso, executa o resto das ações da lista e retorna o último valor avaliado. Por exemplo, seja uma função para testar se um determinado elemento está em uma lista. Se estiver, retorna a lista original, senão inclui o elemento na lista e retorna a lista modificada.

```
(defun testalista (atm lst)
(cond
( ( member atm lst) lst )
( (cons atm lst))))
```

6.2 (and (função1 arg arg)
(função2 arg arg)
...)

and avalia cada um de seus argumentos (funções ou variáveis) até que um dos valores seja *NIL*. Se um dos valores for *NIL*, *and* retorna *NIL* sem avaliar os outros argumentos. Se nenhum for *NIL*, retorna o último argumento avaliado.

```
(and T T NIL) ` NIL
(setq FRUTAS '( maçã laranja amora limão))
(and (member 'maçã FRUTAS) (member 'amora FRUTAS))
` (amora limão)
```

6.3 (or (função1 arg arg)
(função2 arg arg)
...)

OR avalia seus argumentos (funções ou variáveis) em seqüência até que um deles retorne um valor não-*NIL*. *or* retorna este valor sem avaliar os outros argumentos. Se todos os argumentos forem *NIL*, retorna *NIL* como valor de *or*.

```
(or T T NIL) ` T
(defun sinaleira (atm)
(or (if (equal atm ’vermelho) ’Pare!)
(if (equal atm ‘amarelo) 'Atenção!)
(if (equal atm ‘verde) 'Siga!)))
>(sinaleira ‘verde)
Siga!
```

6.4 (not expr)

not retorna *T* se o resultado da avaliação de *expr* for *NIL* e *NIL* se for outro valor.
(not (atom '(A B C))) ` T
(not (atom 'B)) ` NIL

6.5 Exercícios

1 - Escreva uma função *multiplica*, com dois argumentos, um número e uma lista de números. Garanta que todos os argumentos são numéricos e multiplique o número por cada um dos elementos da lista. Retorne a nova lista como resultado.

7. FUNÇÕES RECURSIVAS

Funções recursivas são aquelas definidas em termos dela mesma. Definições nãorecursivas
descrevem operações mais simples sobre o conjunto de dados.
Programação recursiva aplica a mesma operação sobre dados cada vez mais simples.
Uma definição recursiva é composta de duas partes:
• a definição da operação sobre o dado mais simples, feita de uma maneira não-recursiva (condição de fim da recursividade);
• a definição da operação sobre os dados mais complicados feita a partir da operação sobre o termo mais simples. É a porção recursiva da definição.
Exemplo 1:
a) fatorial de 1 é 1;
b) fatorial de n é n * fatorial(n - 1).

```
(defun fatorial (n)
(cond
( (eq n 1) 1)
( T (* n (fatorial (sub1 n))) )) )
```

Exemplo 2:
A função de *delay* no LISP.

```
(defun pausa (num)
(cond
( (eq num 0) T)
( (pausa (sub1 num)) )) )
```

Exemplo 3:
Se um elemento é membro de uma lista.

```
(defun membro (elem lst)
(cond
( (eq lst NIL) NIL )
( (equal elem (car lst)) T)
( T (membro elem (cdr lst)) )) )
```

Exemplo 4:
Leitura de arquivo recursiva.
Chamada por *(learq ( ) arqinput)* .

```
(defun learq (lst arq)
(let token (READ-PRESERVING-WHITESPACE arq))
(cond
( (eq token 'FIM) (reverse lst) )
( T (learq (cond token lst) arq) )) ))
```

Exemplo 5:
Recursão dupla na função que conta os átomos de uma lista.

```
(defun contatomo (lst)
```

(cond
((null lst) 0)
((atom lst) 1)
(T (plus (contatomo (car lst) (contatomo (cdr lst))))))
Exemplo 6:
Função filtra-impar, que retira números ímpares de uma lista.
(defun filtra-impar (lst)
(cond
((null lst) nil)
((oddp (car lst)) (filtra-impar (cdr lst)))
(t (cons (car lst) (filtra-impar (cdr lst))))))

7.1 Exercícios

1 - Defina a função *maior* que retorna o maior elemento de uma lista númérica.
2 - Redefina a função acima para *extraitudo*, que retira todas as ocorrências do átomo.
3 - Escreva uma função para concatenar ordenadamente duas listas já ordenadas.
4 - Defina uma função, com dois argumentos, um número *n* e uma lista, que retorna o *n-ésimo* elemento da lista.

8. APLICAÇÃO DE FUNÇÕES

São funções que aplicam outras funções sobre seus argumentos sucessivas vezes e de diferentes formas. A função a ser aplicada deve ser um símbolo que tem uma definição de função associada por *defun* ou uma função compilada. Não pode ser uma macro ou forma especial.

8.1 (function função) ou #' função

Retorna a interpretação funcional de *função,* que deve ser um símbolo funcional ou uma função *lambda*. A interpretação funcional corresponde à seqüência de ações que a função espera ver executada e a forma como atua sobre seus argumentos.
A função abaixo (Steele, 1990) associa uma lista com duas definições de funções ao símbolo duas-func. A primeira função não tem argumentos e retorna o valor do argumento de *duas-func*. A segunda tem um argumento, que é utilizado para alterar o valor do argumento de *duas-func*, através de *setq*.
(defun duas-func (x)
(list (function (lambda ( ) x))
(function (lambda (y) (setq x y)

8.2 (apply função (args))

Aplica *função* sobre seus argumentos.
Exemplos:
(setq f '+)
(apply f '(1 3)) ` → 4
(apply 'cons '((+ 5 4) 8) ` → ((+ 5 4) . 8)
*(apply #' * (2 4 5)) ` → 40*
(apply #' (lambda (a b)
(cond ((numberp a) (cons a b))))
'(5 (1 3 5 6))) ` → (5 1 3 5 6)

8.3 (funcall função (args))

Aplica a *função* aos ser argumentos de modo similar a *apply*, exceto que realmente considera seu argumento uma função, sem fazer qualquer interpretação adicional.
(cons 1 2)
(setq cons #' +)
(funcall cons 1 2) → 3

8.4 (mapcar função lst1 ... lst5)
Aplica *função* ao *car* das sucessivas caudas das listas, retornando uma lista com os valores retornados por *função* .
(mapcar 'cons '(A B C) '(X Y Z)) ` → ((A . X) (B . Y) (C . Z))
(mapcar 'sqrt '(1 4 9 25)) ` → (1 2 3 5) ;função raiz quadrada
8.5 (maplist função lst1 ... lst5)
Aplica *função* às sucessivas caudas das listas retornando a lista dos resultados retornados pela função.
(maplist #' (lambda (x) (cons 'Lista: x)) '(A B C))
` → ((Lista: A B C) (Lista: B C) (Lista: C))
8.6 (mapc função lst1 ... lst5)
Aplica *função* ao *car* das sucessivas caudas das listas, como *mapcar* faz, mas não acumula o resultado, retornando *NIL*. É utilizado quando o que interessa da função é o efeito colateral e não o resultado da aplicação da função em si.
(mapc 'print '(A B C)) ` → NIL
>A
B
C
8.7 (mapl função lst1 ... lst5)
Funciona como *mapc*, só que aplica a função às sucessivas caudas da lista.
(mapl 'print '(A B C) '(C D)) `→ NIL
>(A B C)
(B C)
(D)

9. FUNÇÕES DE CONTROLE
São funções que permitem controlar o fluxo do programa, a forma como os objetos são acessados e quais valores devem ser retornados. É utilizado para definir blocos de programa e reiterações (*loops*)
9.1 progn
(progn
(função1 arg arg)
(função2 arg arg))
progn avalia suas expressões sequencialmente, retornando o valor da última função avaliada. As expressões embutidas no *progn*, exceto a última, são avaliadas pelo seu efeito colateral (I/O, alteração de valores de variáveis, etc), uma vez que seu valor de retorno é perdido. A última expressão avaliada é a única que retorna seu valor, seja ele um átomo, uma lista ou múltiplos valores.
progn constrói uma estrutura de programa semelhante ao bloco do comando *beginend* do Pascal e é preservada no LISP por razões históricas. Muitas outras construções podem fazer o sequenciamento ou construir laços no lugar de *progn*.
(setq lst '(A B C))
(progn
(car lst)
(rplaca lst 'E)) ;rplaca substitui o car do seu primeiro
;argumento por seu segundo argumento
` (E B C)
9.2 prog1
(prog1
(função1 arg arg)

(função2 arg arg))
Faz o mesmo que *progn* mas retorna a avaliação da sua primeira expressão, seja um átomo ou uma lista. No caso de valores múltiplos, *prog1* retorna apenas o primeiro. Utilizado quando se quer executar uma série de funções com efeitos colaterais, mas preservando o valor anterior.
(prog1
(car lst)
(rplaca lst 'E)) ; rplaca substitui o car do seu primeiro
; argumento por seu segundo argumento
` A
E ` (E B C)
9.3 prog2
(prog2
(função1 arg arg)
(função2 arg arg))
Faz o mesmo que *prog1*, mas retorna o valor da sua segunda expressão.
9.4 let
(let ((atm1 valor1) ; variável e valor
(atm2 valor2)
...)
declaração ;declarações para escopo local de variáveis ou funções
(função1 ...) ;seqüência de ações a ser executada por *let*
(função2 ...)
...)
let permite atribuir valores para variáveis que só terão validade localmente. Primeiro avalia as expressões *valor1, valor2* nesta ordem. Depois, esses valores são ligados ("atribuídos") a *atm1, atm2* em paralelo. Em *declaração* pode ser utilizada a função *declare* para definir comportamentos específicos de variáveis e funções (restrições de tipos, comportamento de funções, etc). As variáveis terão esses valores localmente, até o final da execução de *let*. A seguir, são avaliadas as funções do corpo de *let* em ordem. Todos os valores são descartados, exceto o último, que é retornado como valor de *let*.
let pode ser utilizado para inicializar variáveis que serão utilizadas posteriormente no corpo da função, por uma questão de estilo e correção. Neste caso, os valores dos argumentos podem ser omitidos.
(let ((atm1)
(atm2)
....))
Ou seja, *atm1* e *atm2* passam a valer *NIL* .
Exemplo:
(defun escopo (x) ;escopo tem um argumento
(print x) ;mostra valor do parâmetro
(let ((x 5) ;vincula 5 a x
(w 6)) ;vincula 6 a w
(print (+ x w)) ;mostra a soma de x e w
(print x)) ; mostra x no escopo de let
(print x)) ; mostra x fora do escopo de let
?*(escopo 2)*
Output: 2
11

5
2 ;valor retornado
9.5 let*
(let* ((atm1 valor1)
(atm2 valor2)
...)
declaração ;declarações para escopo local de variáveis ou funções
(função1 ...) ;seqüência de ações a ser executada por *let**
(função2 ...)
...)
*let** funciona como *let*, mas as ligações das variáveis são feitas uma após a outra e não em paralelo, como em *let*. Isso permite que a expressão de atribuição de uma variável referencie as anteriores.
(let ((x 4)*
(y (+ x 12))
(sqrt (/ y x))) ` → 2 ;raiz quadrada de 16/4
9.6 block / return-from
; permitem construir blocos de programa com
; pontos definidos de transferência de controle.
(block nome ;símbolo que nomeia o bloco, não é avaliado
(função1 ...)
(função2 ...)
...
(return-from nome resultado));encerra o bloco e retorna *resultado*
block executa suas funções sequencialmente e retorna o último valor avaliado. Se houver, dentro do bloco, um *return-from* com o mesmo nome do bloco, a execução pára e o valor associado a *return-from* é retornado como o valor do bloco (se não houver valor, retorna *NIL*). Normalmente, *return-from* é associado a alguma função condicional como *if* ou *cond*, para decidir o encerramento do bloco ou não.
Exemplo:
block define um bloco de execução se todos os elementos de uma lista são números. Encerra a execução se encontrar algum valor não numérico.
(block testa-num
(mapcar # '(lambda (x) ;função a ser aplicada sobre lista por mapcar
(cond ((numberp x) (print x))
((return-from testa-num 'nao-numerico)))
lista))
Se *block* não utilizar nome para a estrutura, a saída do bloco deve ser sinalizada com *return*. *return* é uma macro que funciona exatamente como *return-from*, mas não exige nome para o bloco. Irá terminar a execução do bloco onde estiver embutida.
9.7 loop
(loop
(função1 ...)
(função2 ...))
Quando o corpo de *loop* é constituido apenas por chamadas de funções, ele constrói ciclos de execução infinita dentro do programa. Avalia seus argumentos sequencialmente; quando o último argumento for avaliado, retorna para avaliar o primeiro e assim por diante, sem retornar nenhum valor. A execução deve ser encerrada explicitamente utilizando *return*, que poderá especificar um resultado a ser retornado. O exemplo a seguir lê repetidamente a entrada e imprime seu valor, até ser

digitado o símbolo “Fim”.
(loop
(let ((x (read)))
(format t "~A~%" x)
(if (eq x ’Fim) (return x))))
> 1
1
> seis
seis
> fim
fim
fim
Formas especiais de *loop*: existe uma infinidade de combinações de parâmetros que podem ser fornecidos no corpo de *loop*. É provavelmente um dos mais obscuros comandos de LISP. A título de ilustração, considere os dois exemplos a seguir, que utilizam uma variação do comando *for*. Uma descrição completa do comando pode ser encontrada no livro de Guy Steele Jr.
(loop for x from 0 to 9
do (princ x))
>0123456789
NIL
A variável x é sucessivamente ligada aos valores de 0 a 9 no corpo de *loop*. Após o comando *do* podem aparecer uma sequência de comandos. De forma similar, a variável x pode ser associada a uma sequência de elementos em uma lista:
(loop for x in ‘(g a t o)
do (princ x))
>gato
NIL
Algumas formas de interesse prático são o *loop until* e o *loop while*, que mimetizam os respectivos comandos das linguagens procedurais.
(loop while <cond> do
<corpo>)
(loop until <cond> do
<corpo>)
loop while executa o corpo do laço enquanto <*cond*> for verdadeira. *loop until* executa o corpo do laço enquanto <*cond*> for falsa.
9.8 dotimes
(dotimes (i n valor)
(função1 ...)
(função1 ...)
...)
Dotimes executa um laço *n* vezes, variando i de 0 a *n*-1. O valor retornado pode ser indicado opcionalmente na lista de parâmetros. Se nada for indicado, retorna *NIL*.
? (dotimes (i 10 ’Terminou!)
(format t "~d + 1 = ~d~%" i (1+ i)))
0 + 1 = 1
1 + 1 = 2
2 + 1 = 3
3 + 1 = 4
4 + 1 = 5

5 + 1 = 6
6 + 1 = 7
7 + 1 = 8
8 + 1 = 9
9 + 1 = 10
TERMINOU!
?

9.9 dolist

(dolist (item lst valor)
(função1 ...)
(função1 ...)
...)

Realiza uma iteração associando uma variável aos elementos de uma lista. Apenas os elementos do primeiro nível da lista são associados a variável no corpo do laço. De forma similar a dotimes, retorna o valor especificado na lista de parâmetros ou *NIL*, por default.

? (dolist (v '(l i s t a) 'Listou!)
(if (equal v 'l) (setq v 'P))
(format t "~A " (string-downcase v)))
P I S T A
LISTOU!

9.10 do

(do ((atm1 valor-inicial1 passo1) ;modifica *atmj* a partir
(atm2 valor-inicial2 passo2)) ;*valor-inicialj* com o *passoj*
...
(teste-fim result) ;avalia *teste-fim*, retorna *result* se V
(declaração) ;declara tipos de variaveis locais, se quiser
(função1 ...) ;corpo da função *do*
(função ...))

do avalia suas variáveis *atm1, atm2, ...* , e associa a elas, em paralelo, os respectivos valores iniciais. Depois, avalia o teste de fim. Se for diferente de *NIL*, avalia a expressão fornecida como *result* e retorna seu valor como valor de *do*. Senão, executa as funções colocadas no corpo de *do*. Depois disso, modifica suas variáveis *atm1, atm2, ...* , da forma como for definido em *passo1, passo2, ...* , avalia *teste-fim* novamente e, se não for verdadeiro, executa o corpo da função novamente. O ciclo repete até que *teste-fim* forneça um valor não-*NIL*. *do* encerra e retorna o valor avaliado em *result*.

Se não fornecido valor para *passo*, *do* irá associar valores às variáveis como *let*. Se o *teste-fim* for igual a *NIL*, *do* irá executar infinitamente, a não ser que seja encerrado com *return-from* ou *return*, por exemplo.

(do ((i 0 (+ i 1)) ;incrementa i = 0 com passo 1
(n (length vetor))) ;associa a n o tamanho de um vetor
((= i n) vetor) ;executa n vezes, retorna vetor
(cond
((null (nth i vetor))) ; posição i do vetor é igual NIL?
((setf (nth i vetor) 0)))) ; coloque zero na posição
> (0 0 0)

9.11 do*

*do** funciona exatamente como *do*, exceto que avalia e associa valores às suas variáveis sequencialmente, usando *let** , permitindo que uma variável faça referência

ao valor da anterior.
(do ((x 1 (+ x 1))*
(y x x))
((> x 5))
(format t "(~A ~A)" x y))
> (1 1) (2 2) (3 3) (4 4) (5 5)
NIL

10. ESTRUTURAS DE DADOS

Common Lisp oferece uma série de estruturas de dados usualmente disponíveis em linguagens procedurais. Apesar de não serem tão versáteis como listas, estruturas específicas são implementadas de forma eficiente, minimizando espaço de memória e tempo de acesso.

10.1 Arranjos

Arranjos (*arrays*, em inglês), são estruturas multidimencionais cujo acesso é indexado por números inteiros. Nos arranjos os elementos são armazenos adjacentes na memória, eliminando a necessidade de ponteiros para encadeamento dos dados. Desta forma, um arranjo é mais econômico do um conjunto de listas. Por outro lado, arranjos são mais rígidos em termos de expansão ou redução de suas dimensões. Suas dimensões devem ser pré-definidas no momento de sua criação. A primitiva *makearray* cria um arranjo multimensional:

> (setf arranjo (make-array '(2 3) :initial-element 7))
#2a((7 7 7) (7 7 7))

As dimensões do arranjo são o único parâmetro obrigatório para a função *make-array*. São fornecidas em uma lista cujo comprimento indica o número de dimensões do arranjo. Assim, o exemplo acima cria um arranjo bidimencional, sendo a primeira dimensão de amplitude 2 e a segunda de amplitude 3. A palavra chave *:initial-element* permite especificar qual o elemento initial em *todas* as entradas do arranjo. O valor *default* depende do sistema. O Macintosh Common Lisp coloca 0 (zero) como valor inicial por *default*.

A variável do sistema **print-array** controla a forma como o array é exibido. No exemplo acima, o valor de **print-array** é t. Colocando **print-array** em *nil*, ele é exibido de outra forma:

> (setf arranjo (make-array '(2 3) :initial-element 7))
#<ARRAY 2x3, simple>

Para especificar o conteúdo do arranjo elemento por elemento utiliza-se a palavra chave *:initial-contents*. Os valores iniciais dos elementos são então apresentados na forma de listas, uma para cada dimensão.

> (setf arranjo (make-array '(2 3) :initial-contents '((7 8 9) (a b nil))))
#2a((7 8 9) (A B NIL))

Para ler o valor de uma entrada de um arranjo é utilizada a função *aref*. Como tudo em Lisp, o acesso a um elemento de um arranjo se dá também através de uma função. As dimensões do arranjo começam por 0:

> (aref arranjo 0 0)
7

A função *aref* funciona como localizador. Quando utilizada com *setf*, permite alterar o valor de uma entrada do arranjo.

> (setf (aref arranjo 0 0) nil)
nil

Existem predicados específicos que permitem obter informações relativas a um dado

arranjo. A seguir descreve-se alguns predicados mais utilizados:
(array-rank arranjo)
Retorna o número de dimensões do array. Por exemplo:
> (array-rank arranjo)
2
(array-dimensions arranjo)
Retorna uma lista com as dimensões do arranjo.
>(array-dimensions arranjo)
(2 3)
(array-total-size arranjo)
Retorna o número total de elementos do arranjo, dado pelo produto de suas dimensões.
>(array-total-size arranjo)
6
Arranjos de uma dimensão apenas podem ser criados tanto com a função *make-array* quanto com a função *vector*. No caso de usar *make-array*, pode-se especificar diretamente a dimensão do vetor através de um inteiro (e não uma lista com um inteiro):
>(setf vetor (make-array 5 :initial-element nil)
#(NIL NIL NIL NIL NIL)
>(vector nil nil nil nil nil)
#(NIL NIL NIL NIL NIL)
>(vector "a" 'b 33)
#("a" B 33)
A função *vector* não oferece as opções de iniciação de *make-array*, sendo utilizada apenas quando o conteúdo inicial do vetor é conhecido. Para acessar dados em um vetor, utiliza-se a primitiva *svref*, mais eficiente do que *aref*.
> (svref vetor 0)
nil
> (setf (svref vetor 0) 'Ananda)
ANANDA

10.2 Strings

Um *string* é um tipo particular de vetor: um vetor de caracteres. É importante esta particularização pois permite uma representação mais eficiente em termos de espaço de memória. Cada elemento do *string* ocupa apenas o espaço de um caracter. *Strings* constantes são representados da maneira usual, entre aspas, como "um string", e caracteres individuais são indicados na forma #\<char>. Por exemplo, o caracter *a* é indicado por #\a. As funções que manipulam *strings* começam com o prefixo "string". Funções não-destrutivas aceitam um símbolo como parâmetro. Neste caso, o *printname* do símbolo é utilizado pela função.
O acesso a um caracter de um string é realizado através da função *char*.
? (char "Pindamonhangaba" 3)
#\d
A alteração de caracteres de um *string* pode ser realizada combinando-se *setf* com *char*.
? (setq s "Pindamonhangaba")
"Pindamonhangaba"
? (setf (char s 3) #\t)
#\t
? s

"Pintamonhangaba"
Existem diversas funções que permitem comparações entre *strings*.
(string= string1 string2 &key :start1 :end1 :start2 :end2)
string= compara dois strings retornando t se ambos são iguais. Optionalmente podese especificar um subconjunto de caracteres do *string* para comparação através das palavras chave *start* e *end*. O substring começa em *start* e termina no caracter anterior a *end*.
? (string= "porto alegre" "alegre" :start1 6 :end1 12)
T
? (string= "Porto Alegre" "porto alegre")
NIL
No caso de se desejar uma comparação independente de maiúscula, existe a função *string-equal*:
? (string-equal "Porto Alegre" "porto alegre")
12
Neste caso, a função retorna o tamanho do *string* caso sejam idênticos, *nil* caso contrário.
Comparações lexicográficas são aquelas que testam os códigos dos caracteres de dois *strings*. Um caracter é maior do que outro se seu código (ASCII, por exemplo) é maior do que o código do outro. Isso permite que *strings* sejam ordenados, como mostra o exemplo a seguir.
? (sort "elbow" #'char<)
"below"
Sort é uma função Lisp que ordena sequências, recebendo como parâmetro uma função de comparação. *char=*, *char<* , *char<=*, *char>*, *char>=* e *char/=* funcionam como os respectivos comparadores aritméticos, porém aplicados aos códigos dos caracteres.
? (char> #\a #\A)
T
? (char> #\B #\b)
NIL
? (char> #\a #\b)
NIL
? (char> #\c #\b)
T
Comparações entre strings testam sucessivamente os caracteres na ordem em que aparecem (da esquerda para a direita na tela). O resultado é determinado pelo primeiro caracter que difere nos *strings*.
(string< string1 string2 &key :start1 :end1 :start2 :end2)
(string<= string1 string2 &key :start1 :end1 :start2 :end2)
(string> string1 string2 &key :start1 :end1 :start2 :end2)
(string>= string1 string2 &key :start1 :end1 :start2 :end2)
(string/= string1 string2 &key :start1 :end1 :start2 :end2)
? (string< "abcde" "abcdf")
4
? (string>= "qwerty" "qwerty")
6
? (string< "abcdefghij" "abcdf")
4
? (string> "a" "QWERTY")

0
Todas os exemplos acima retornam *T*. Note que o tamanho do *string* não influencia o resultado da comparação.
Outras funções interessantes para manipulação de strings são aquelas relativas a capitalização de letras.
(string-upcase string &key :start :end)
(string-downcase string &key :start :end)
(string-capitalize string &key :start :end)
? (string-upcase "Tudo maiuscula")
"TUDO MAIUSCULA"
? (string-downcase "TUDO MINUSCULA")
"tudo minuscula"
? (string-capitalize "dr. afranio barata")
"Dr. Afranio Barata"
Existem diversas alternativas para a criação de *strings*. A função *string* retorna o *string* associado a seu parâmetro, que pode ser um símbolo, um caracter ou até mesmo um *string*.
? (string 'aba)
"ABA"
? (string #\a)
"a"
? (string "asdf")
"asdf"
Para obtenção de *strings* a partir de outros tipos de dados a função *format* é a mais indicada.
? (setf data (format nil "Hoje eh ~D de Maio de ~D" 12 1997))
"Hoje eh 12 de Maio de 1997"
Por fim, a função *make-string* pode ser utilizada para criar um string vazio com determinado tamanho.
(make-string size &key initial-element)
? (make-string 10)
" "

10.3 Sequências

Listas, vetores e strings são estruturas lineares compostas por sequências de elementos. Apesar de diferirem com relação a sua organização e armazenamento, existe uma série funções que são comuns a essas estruturas. A maior parte destas funções utiliza algumas das seguintes palavras chave:
Palavra Chave Propósito Default
:key função aplicada a cada elemento identidade
:test função de teste para comparação eql
:from-end se verdadeiro, processa de trás para
frente
nil
:start início da sequência 0
:end fim da sequência nil
Veremos a seguir algumas funções que podem ser aplicadas a sequências genéricas. Muitas delas já foram vistas com listas. A diferença básica é que estas funções trabalham com qualquer tipo de sequência.
(elt seq pos)
Retorna o *pos*-ésimo elemento da sequência.

? (elt "LispMania" 4)
#\M
? (elt '(L i s p e i r o) 2)
S
? (elt (vector 1 2 3 4) 3)
4
(length seq) : Retorna o tamanho da sequência.
? (length "barranquilos")
12
(remove item seq) : Remove o item da sequência.
? (remove '3 (vector 1 2 3 4))
#(1 2 4)
(position item seq) : retorna a posição do item na sequência.

11. ESTILO E DOCUMENTAÇÃO

Programas em LISP são construídos como uma seqüência de definições de funções que são chamadas por um programa principal.
Funções em LISP são:

- curtas e referem-se **uma** tarefa específica. Por exemplo, recebem o parâmetro, processam o parâmetro e retornam o valor modificado como saída;
- contém todos os controles de erros previsíveis. Por exemplo, a função *processa-entrada* abaixo verifica se realmente o símbolo *palavra* recebeu algum valor, antes de incluir esse valor na lista de propriedades;

(defun processa-entrada (palavra)
;; Processa palavras lidas na entrada.
(cond ((boundp (quote palavra)) ;; Testa se a entrada foi lida
(putprop 'arquivo palavra 'nome)))) ;; Inclui na lista de prop.

- possuem o mínimo possível de efeitos colaterais, por conter toda a informação necessária para a execução e por criar e atribuir variáveis locais com *let* (a função *setq* cria variáveis "globais" que permanecem após o fim da execução da função);
- são documentadas: a linha seguinte à definição da função contém uma breve descrição do que ela faz; testes e comandos mais sofisticados são documentados no lado direito do comando (comentários são incluídos após o ponto-e-virgula);
- funções muito complexas são substituídas por macros.

12. ENTRADA E SAÍDA

As entrada e saídas são feitas a partir do dispositivo default de entrada e ou saída, referido tambem como a variável ***terminal-io*,** ou a qualquer dispositivo (*streams*) definido pelo usuário: impressora, mouse, arquivos. Se os comandos de entrada e saída forem dados sem especificação do dispositivo, a leitura será feita do teclado e a saída será no monitor. Além dos comandos aqui descritos, que são os mais utilizados, existem inúmeras formas de I/O formatado. Referencie o manual da linguagem que estiver utilizando.

• **Entrada:**

(read stream)
lê um objeto LISP.
(read-delimited-list stream)
lê uma expressão LISP inteira do dispositivo ou arquivo.

(read-line stream)
lê uma linha do arquivo (até o *line feed*).
(read-char stream)
lê um caracter do arquivo.
(peek-char stream)
lê um caracter sem removê-lo do buffer de entrada.
(clear-input stream)
limpa o buffer de entrada.
(read-from-string string)
lê um string e gera o objeto LISP correspondente.
(y-or-n-p string-de-pergunta)
apresenta a pergunta na tela e retorna *T* se o usuário responder *Y* ou *Space* e *NIL* se responder *N* ou *Backspace*.

• **Saída:**

(terpri stream1)
manda uma nova linha para o arquivo.
(princ expr stream1)
escreve *expr* sem os " " dos strings e dos caracteres especiais a partir da posição corrente do cursor.
(print expr stream1)
escreve *expr* em nova linha.
(pprint expr stream)
escreve *expr* em uma nova linha em *pretty-print*, ou seja, de forma identada.
(prin1-to-string expr)
retorna o átomo como um string.
(princ-to-string expr)
retorna o átomo como um string sem caracteres especiais.
(flatsize expr)
retorna o número de caracteres necessários para gerar o string de *expr*.
(flatc expr)
retorna o número de caracteres necessários para gerar o string de *expr* sem caracteres especiais.
(beep)
soa um beep.

12.1 Exercícios

Programa Mutação

Faça um programa para concatenar nomes de animais a partir de suas partes comuns, criando novos animais. Por exemplo:

vaca + cavalo = vacavalo
;reconhece intersecção, corta repetição e concatena
tartaruga + galinha = tartarugalinha
paca + aranha = pacaranha
tubarão + tatu = tatubarão
lobo + cavalo = cavalobo ;concatena no ordem possível

O sistema deve receber uma lista qualquer de animais como entrada e retornar todas as concatenações possíveis de “novos” animais. A concatenação pode se dar por 1, 2, 3 ou mais letras.

Lista de animais para teste:

(lobo cavalo vaca galinha tartaruga tatu tubarão tucano foca)

13. LISTAS DE PROPRIEDADES

13.1 Busca em Árvores Binárias

Árvores binárias podem ser representadas como listas LISP e percorridas recursivamente. A representação é no formato:

(raiz esq dir)

onde **raiz** é o átomo que representa o conteúdo da raiz da árvore e **esq** e **dir** são listas com o mesmo formato da árvore inicial. As folhas são representadas como listas contendo só a raiz.

Neste formato, a busca em profundidade na árvore é facilmente implementada, uma vez que a lista comporta-se como uma pilha.

```
;Busca em profundidade em árvore
;Representação: (raiz (esquerda (direita))
;(setq arv '(a (b (c) (d)) (e () (f))) )
a
b
c d
e
f
(defun depth-first (arvore fim)
(cond ((null arvore) nil)
((equal fim (car arvore)) t)
((depth-first (cadr arvore)) fim)
((depth-first (caddr arvore)) fim)
(t nil)))
```

A busca em largura, no entanto, fica pouco otimizada, necessitando simular uma estrutura de fila para armazenar os nodos a serem percorridos na árvore.

Uma forma mais elegante e geral de representar árvores é utilizando **Listas de Propriedades** (ou **plist**), uma importante estrutura de dados do LISP.

Listas de propriedades são associadas aos símbolos e têm o formato:

(propriedade1 valor1 propriedade2 valor2 ...)

onde *propriedaden* é um átomo único na lista e *valorn* pode ser um átomo ou uma lista. As listas de propriedades são criadas juntamente com o símbolo aos quais estão associadas, inicialmente como listas vazias. As funções de manipulação de listas de propriedades são:

(symb-plist symb) ` plist

Retorna a lista que contém os pares de propriedades de *simb*.

(get simb prop) ` valor

Retorna o valor associado a propriedade *prop* do símbolo *simb,* ou *NIL* se não houver propriedade definida.

(setf (get simb prop) 'novovalor) ` novovalor

Associa *novovalor* como o valor da propriedade *prop* de *simb*.

Obs.: Muitas versões do LISP implementam essa combinação de funções através da função *putprop*.

(remprop simb prop)

Remove a propriedade *prop* e o *valor* associado da *plist* de *simb*.

Exemplo:

Para a árvore acima.

(setf (get ’arvore ’raiz) ’a)

```
(setf (get ’a ’esq) ’b)
(setf (get ’a ’dir) ’e)
(setf (get ’b ’esq) ’c)
(setf (get ’b ’dir) ’d)
(setf (get ’e ’dir) ’f)
(defun busca-df (arv nodo)
; Busca em profundidade pelo nodo
; Na chamada: (busca-df (get ’arvore ’raiz) ’c)
(cond ((null arv) nil)
((equal arv nodo) T)
((busca-df (get arv ’esq) nodo)
((busca-df (get arv ’dir) nodo) ))
(defun busca-bf (arv nodo )
; Busca em largura pelo nodo
; Na chamada (busca-bf ‘arvore 'c )
(do ((fila (list (get arv 'raiz)) ;cria fila e inicializa com a raiz da
arvore
(append (cdr fila) ;step para o do
(list (get (car fila) 'esq)
(get (car fila) 'dir)))))
( (null fila) ) ;condicao de fim do loop
(if (equal (car fila) nodo) (return ‘Achei)) ;sinaliza se encontrou o
nodo
(print (car fila))))
```

13.2 Exercícios

1 - Construa a função *(insere arvore novonodo nodo filho)* para incluir um novo nodo na árvore ou retornar *NIL* se o lugar estiver ocupado.
Chamada: *(insere ’arvore ’G ’F ’esq)*
2 - Modifique a forma de representação e as funções de busca em árvore para que funcionem para grafos.
3 - Faça uma função que construa uma árvore em depth-first e pré-dir a partir de símbolos não numéricos lidos do teclado.

14. PROCESSAMENTO DE ARQUIVOS

LISP necessita de nomes para referenciar arquivos. O problema é que esses nomes tendem a ser dependentes do sistema operacional do ambiente onde eles são criados. *Common Lisp* propõe uma forma uniforme de tratar arquivos externos independentemente do ambiente onde o programa está sendo executado. Há duas maneiras de tratar arquivos em *Common Lisp*:
namestrings : strings com o nome do arquivo e que são dependentes da implementação.
Por exemplo:
"editor.lsp" ou *"c:/gcl/editor.lsp"* para o DOS.
pathnames : tipos abstratos de dados que representam os nomes dos arquivos de forma independente do ambiente. São especialmente necessários em ambientes de rede para acessar arquivos de diferentes plataformas.

14.1 Pathnames

São compostos por seis componentes:
host: nome da máquina ou ambiente onde estão os arquivos.
device: disco ou outro equipamento físico ou lógico onde estão os arquivos.

directory: diretorio do arquivo.
name: nome do arquivo.
type: tipo do arquivo ou extensão.
version: número da versão daquele arquivo.
O pathname corresponde a uma forma de acessar o arquivo e não, necessariamente, a um arquivo existente particular. No momento de definir o pathname, ele deve ser associado a um símbolo para ser posteriomente utilizado. Existem diversas funções para manipulação de pathnames, referencie o manual para aquelas que não estão citadas aqui.
pathname : retorna um pathname a partir de um namestring.
(pathname "c:\\gcl43\\teste.lsp) ` #.(pathname "c:\\gcl43\\teste.lsp")
Obs.: As barras devem ser duplas para que sejam mantidas. Ver Anexo I.
make-pathname : cria um pathname segundo o caminho especificado.
(setq path (make-pathname :device "C"
:directory "gcl43" "dados" :name "teste"
:type "lsp")) `#.(pathname "c:gcl43\\dados\\teste.lsp")
pathnamep : testa se um objeto é um pathname.
(pathnamep path) ` T
namestring : retorna o string com o nome do pathname.
(namestring pathname) ` "c:gcl43\\dados\\teste.lsp"

14.2 Abrindo e Fechando Arquivos

open nome-arquivo :direction :element-type :if exists :if-does-not-exist
Abre um arquivo determinando seu tipo e uso.
nome-arquivo pode ser um string, stream ou um pathname. É o único componente obrigatório do comando.
:direction pode ser :input, :output, :io, :probe (esse último não abre realmente o arquivo, só teste se está lá).
:element-type define se o arquivo deve ser lido como caracteres, bytes, bits, etc..
:if-exists define ações para output se o arquivo já existir, como sinalizar um erro, abrir uma nova versão, trocar o nome, escrever em cima, fazer append, etc.
:if-does-not-exist define ações para input se o arquivo não existir, como sinalizar erro ou criar um novo arquivo
Exemplo:
(setq arq (open "c:\\gcl43\\doenças.lsp" :direction :input :if-does-not-exist :error)) `
#<File input stream c:\gcl43\doencas.lsp>
Depois de aberto o arquivo pode ser acessado com os comandos de entrada e saida e posteriormente fechado:

```
(defun learq (lst arq)
(setq palavra (read arq) ;lê um símbolo do stream
(cond ( (equal palavra 'fim) ;a palavra fim sinaliza o fim do arquivo
(reverse lst) ) ;inverte a lista para colocar o texto na ordem
( T (learq (cons palavra lst) arq) )) )
(defun inicio ()
(setq arq (open "c:\\gcl43\\doencas.lsp" :direction :input
:if-does-not-exist :error))
(learq arq ( ))
(close arq ( ) ))
```

with-open-file stream nome-arquivo :direction :element-type
:if-exists :if-does-not-exist <corpo da função>
Abre o arquivo *nome-arquivo*, associa ao s*tream* com as demais especificações como em open e executa o corpo da função, que normalmente são operações sobre o arquivo. Ao encerrar a função o arquivo é fechado.
Exemplo:
(with-open-file (arq "c:\\gcl43\\doenças.lsp" :direction :input :if-does-not-exist :error)
(do ((palavra (read arq))
(and (equal palavra 'fim) (reverse lst)
(setq lst (cons palavra lst))))

14.3 Construindo Estruturas a Partir de Arquivos

Para muitas aplicações de Inteligência Artificial, o passo inicial para construir sistemas (especialistas ou de processamento de linguagem natural) é a leitura e interpretação de arquivos semi-estruturados. A partir deles são construídas estruturas internas de representação da informação, utilizando listas ou listas de propriedades.
Utilizaremos aqui como exemplo um sistema especialista para aconselhar qual o melhor vinho para acompanhar uma refeição, segundo os pratos e sabores dessa refeição. O banco de conhecimento utilizado é o da versão de demonstração do sistema PSE, desenvolvido pela EMBRAPA (veja exercício 12, no Anexo II).
Para ler arquivos de texto livre, muitas vezes é necessário mudar as definições associadas aos caracteres especiais de forma a impedir erros durante o processamento do arquivo. No exemplo abaixo, o caracter "," "(vírgula), que tem associado no LISP uma chamada de macro foi redefinido para a mesma definição de "-" (hífen), considerado um caracter comum. Para isso, a tabela de caracteres da linguagem é salva e a definição é modificada antes do processamento do arquivo. No final do processamento, a tabela original é restaurada. (Ver capítulo INPUT/OUTPUT no manual da linguagem).
(defun inicialset () ;; antes de processar arquivo
(copy-readtable ;; faça uma cópia de segurança da tabela de caracteres
(set-syntax-from-char #\, #\-))
;; faça a sintaxe de "," ser igual a de "-"
(defun reseta-sistema ;; antes de encerrar programa
*(setq *readtable* (copy-readtable NIL)))* ;; retorna a tabela original
A função de leitura e contrução das estruturas propriamente ditas abre o arquivo de texto e, a medida que ele vai sendo lido, palavra por palavra, constrói a estrutura a partir das palavras-chave e informações do arquivo.
Exemplo - funções para ler, interpretar e gerar a representação interna de um arquivo de texto contendo regras de produção, no formato:
SE <condição>
E <condição>
...
ENTÃO <conclusão>
(defun learq (arquivo)
;; arquivo deve ser um string com nome do arquivo texto
(with-open-file
(narq arquivo :direction :input :if-does-not-exist :error)
;;abre o arquivo
(loop
(let ((palavra (read narq))) ;; loop de leitura do arquivo

(print palavra)
(cond ((eq palavra ’FIM)
(return-from learq nil))
((eq palavra ’REGRA)
(setq regra (gensym "Regra")) ; cria um simbolo Regra1
(setq lregras (cons regra lregras)) ; inclui regra na lista de regras
;;ou (setq regra (make-symbol (string-append "Regra" (string(read narq)))))
((or (equal palavra ’SE) (equal palavra ’E)))
(setq atributo (pega-atributo (read-line narq)))
(if ((not (member atributo latrib))
(cons atributo latrib)))
(putprop regra atributo ;; coloca na plist do símb
(pega-valor (read-line narq)))
((equal palavra 'ENTAO)
(putprop regra 'conclusao (read-line narq)))))))))

A palavra-chave *PERGUNTA* no arquivo texto indica para cada atributo pesquisado, qual a pergunta que deve ser feita ao usuário de modo a obter o valor daquele atributo, na opinião do usuário. É utilizada para construir a interface do sistema.

No exemplo:

PERGUNTA molho = tomate, creme, apimentado, agridoce, tartaro [multiplo]
Que tipo de molho e’ usado na refeicao ?

PERGUNTA é a palavra-chave que indica que o atributo será lido a seguir. O nome do atributo é dado a seguir (molho), seguido do sinal de igualdade e da lista de valores que aquele atributo pode assumir (tomate, creme, apimentado, agridoce, tartaro).

Entre colchetes é dada a forma como aquela pergunta deve ser tratada. As perguntas com "*múltiplo*" permitem que o usuário escolha mais de um valor para o atributo, ao contrário de "*único*", onde só um valor pode ser utilizado. As perguntas com "*probabilístico*" permitem que o usuário associe um percentual ao valor do atributo que indica o índice de significância daquele valor em relação à conclusão. "*Determinístico*", ao contrário indica que o valor é ou não escolhido.

Na linha seguinte é dada a pergunta na forma como será apresentada ao usuário (*Que tipo de molho e’ usado na refeicao ?*).

Para que sejam processadas também as perguntas no sistema, é necessário modificar a função *learq* e definir uma nova função *lepergunta* que irá associar as perguntas aos atributos.

(defun learq2 (arquivo)
;; função learq modificada para ler e associar perguntas
(with-open-file
(narq arquivo :direction :input :if-does-not-exist :error)
;; abre o arquivo
(loop
(let ((palavra (read narq))) ;; loop de leitura do narq
(print palavra)
(cond ((eq palavra 'FIM)
(return-from learq nil))
((eq palavra 'REGRA)
(setq regra (gensym "Regra"))) ;; cria um simbolo Regra1
;; ou (setq regra (make-symbol (string-append "Regra" (string(read narq)))))

((or (equal palavra 'SE) (equal palavra 'E)))
(putprop regra (pega-atributo (read-line narq))
;; coloca na plist do simb
(pega-valor (read-line narq))))
((equal palavra 'ENTAO)
(putprop regra 'conclusao (read-line narq)))
((equal palavra 'PERGUNTA)
(lepergunta (narq)))))))
(defun lepergunta (arquivo)
(let ((atributo (read arquivo))
(valores (read-line arquivo))
(pergunta (read-line arquivo)))
(putprop atributo 'valor pega-valores)
(putprop atributo 'pergunta pergunta))

Na hora de solicitar ao usuário o valor de determinado atributo, basta buscar na *plist* do atributo pela pergunta e respostas possíveis associadas.

(defun fazpergunta (atributo valor)
(cond ((null (get atributo 'pergunta))
;; existe pergunta associada?
(princ atributo) (princ "= ")
(princ valor) (princ "?") (terpri))
;; se não, mostre o atributo e valor
((escreveperg (get atributo 'pergunta))
;; escreve a pergunta na tela
(setq resposta (read)) ;; le a resposta
(processa resposta atributo)))) ;; trata resposta

A função *escreveperg* apresenta a pergunta no monitor sem parênteses.

(defun escreveperg (listaperg)
(cond ((null listaperg))
((princ (car listaperg)) ;; escreve uma palavra por vez
(princ " ") ;; com brancos no meio
(escreveperg (cdr listaperg)))))

O tratamento dos textos de explicação associados às perguntas ou às conclusões podem ser tratados de maneira similar.

14.4 - Raciocinando em Lisp

Uma vez construída a forma de representação interna das regras há duas formas de raciocinar sobre elas: o raciocínio para frente ou por dados (ou *forward chaining*) e o raciocínio para trás ou por hipóteses (ou *backward chaining*). Eles diferem na forma como interpretam as regras já representadas.

No **raciocínio para frente**, as regras são testadas na ordem em que elas ocorrem. Qualquer regra que possa ser disparada é considerada "candidata". Se forem comprovadas suas premissas, a conclusão associada é incluída na lista dos fatos comprovados, até que não haja mais nenhuma regra a ser disparada. A conclusão é a própria lista de fatos gerada.

Inicialmente, é necessário criar uma lista global de fatos:

(setq listafatos ( ))

A seguir, definir as funções de incluir e pesquisar nesta lista: *memorize* e *relembre*.

(defun memorize (fato listafatos)
(cond ((member fato listafatos)) ;; já existe?
((setq listafatos (cons fatos listafatos))))) ;; se não, inclua!

(defun relembre (fato listafatos)
(cond ((member fato listafatos) T)
;; se o fato estiver na lista, retorne TRUE
(T NIL))) ;; se não, retorne falso

Além da lista de fatos, diversas listas auxiliares podem ser criadas para auxiliar a condução do raciocínio:

• lista dos atributos a serem valorados: definida no momento da criação da estrutura de representação interna (função *learq*, do exemplo);
• lista das regras: idem, facilita a busca pelas regras a serem disparadas;
• lista dos atributos já verificados: construída durante o processo de inferência;
• lista das regras já disparadas: idem.

No raciocínio para a frente, é contruído um loop que processa as regras a partir da primeira. O sistema tenta comprovar as regras com os atributos que estiverem na lista de fatos. Se os valores dos atributos solicitados pela regra ainda não tiverem sido incluído na lista, e existir pergunta associada a ele, o sistema pergunta ao usuário esses valores, incluindo-os na lista de fatos. Se a regra for validada, sua conclusão também é incluída na lista de fatos.

(defun forward ( )
< Repete seqüência abaixo até que
todos os atributos sejam valorados >
< pega atributos da proxima regra >
< Está na lista de fatos? >
< Se sim, inclui conclusão na lista de fatos >
< Se não, Existe pergunta associada?
< Se sim, pergunta ao usuário,
inclui na lista de fatos,
inclui regra na lista de regras disparadas >
< Se validar regra,
inclui conclusão na lista de fatos >
< Se não, descarta regra e repete o loop >

O loop pode ser repetido até que todas as regras tenham sido disparadas, nesse caso o sistema pode dar mais de um vinho como resposta. Ou até que o primeiro valor do atributo *VINHO* seja determinado, neste caso o sistema dá uma única resposta. Pode ser apresentado como conclusão apenas os valores associados a *VINHO*, ou toda lista de fatos, que contém as conclusões parciais.

No **raciocínio para trás**, junto com as regras é fornecida qual é a conclusão que deve ser pesquisada (no exemplo do trabalho 4, a palavra-chave *RESOLVE* indica qual objeto deve ser qualificado, ou seja, *VINHO*). As regras encontradas que possuirem conclusões que refiram-se ao objeto são consideradas a hipóteses do sistema e são verificadas primeiro. Para comprovar as premissas, busca-se igualmente quais são as regras que possuem o objeto daquela premissa como conclusão, ou se existe pergunta associada, e assim recursivamente. As premissas comprovadas são incluídas na lista de fatos comprovados. Se todas as premissas forem comprovadas, a conclusão do sistema será a conclusão da primeira regra selecionada.

Nessa forma de raciocínio, a inferência é mais dirigida, sendo disparadas menos regras para atingir o objetivo. O objetivo deve ser identificado na carga do arquivo (na função *learq*) associando o valor lido após a palavra-chave *RESOLVE*, a uma variável global.

(defun backward ( )

< Repete a seqüência abaixo até encontrar
o valor (ou valores) do atributo VINHO >
< pega atributos da primeira regra que
contiver conclusão com atributo VINHO >
< Está na lista de fatos? >
< Se sim, inclui conclusão na lista de fatos >
< Se não, Existe pergunta associada? >
< Se sim, pergunta ao usuário,
inclui na lista de fatos >
< Se não, busca regra cuja conclusão
incluir o atributo buscado e repete o loop >

14.5 Exercícios

1 - Construa um Sistema Especialista de árvore de discriminação para diagnosticar doenças comuns *(gripe, gastrite, amigdalite).* Associe explicações às perguntas. Utilize as palavras chaves *banco, primeiro, nodo, perg, expl, sim, nao, conclusao* e *fim* para interpretar o banco de conhecimento.

Estrutura do Banco

banco DOENCA
primeiro UM
nodo UM
perg Voce esta com febre?
expl Sua temperatura e superior a 38 graus?
sim GRIPE
nao GASTRITE
nodo GRIPE
perg Voce esta com dor de cabeca?
expl Sente-se com os olhos pesados e cansados?
sim ULTIMOGRIPE
nao OUTRONODO
nodo GASTRITE
perg Voce tem dor de estomago?
expl Sente alguma coisa queimando dentro do seu estomago?
sim ULTIMOGASTRITE
nao OUTRONODO
nodo ULTIMOGRIPE
conclusao Voce esta com gripe. Tome analgesicos - muito liquido e repouso.
nodo ULTIMOGASTRITE
conclusao Va ao medico.
nodo OUTRONODO
conclusao Nao foi diagnosticada nenhuma doenca.
fim

2 - Defina um programa para interpretar e processar o banco de conhecimentos do sistema especialista PSE (EMBRAPA) para aconselhar o melhor vinho para uma refeição.

Estrutura do Banco

REGRA 1
SE tem molho
E molho apimentado
ENTAO corpo encorpado
REGRA 2

SE sabor delicado
ENTAO corpo leve
REGRA 3
SE sabor medio
ENTAO corpo medio
REGRA 4
SE sabor forte
ENTAO corpo encorpado
REGRA 5
SE tem molho
E molho creme
ENTAO corpo encorpado
REGRA 6
SE prato principal carne
E nao tem vitela
ENTAO cor tinto
REGRA 7
SE prato principal aves
E nao tem peru
ENTAO cor branco
REGRA 8
SE prato principal peixe
ENTAO cor branco
REGRA 9
SE prato principal peixe
E tem molho
E molho tomate
ENTAO cor tinto
REGRA 10
SE prato principal aves
E tem peru
ENTAO cor tinto
REGRA 11
SE prato principal nao sabe
E tem molho
E molho creme
ENTAO cor branco
PERGUNTA tem molho
Tem molho na refeicao ?
PERGUNTA tem peru
A refeicao inclui carne de peru ?
PERGUNTA tem vitela
A refeicao inclui carne de vitela ?
PERGUNTA prato principal
Qual eh o prato principal da refeicao (carne, peixe, aves) ?
PERGUNTA corpo
Voce geralmente prefere vinho leve ou encorpado ?
PERGUNTA cor
Voce geralmente prefere vinho tinto ou branco ?
PERGUNTA suavidade

Voce geralmente prefere vinho seco ou suave ?
PERGUNTA molho
Que tipo de molho é usado na refeicao (creme, tomate, apimentado, agridoce, tartaro) ?
PERGUNTA sabor
Qual o sabor da refeicao (delicado, medio, forte) ?
FIM

15. ORIENTAÇÃO A OBJETOS EM COMMON LISP

O modelo tradicional de orientação a objetos pressupõe que os objetos:
• tem propriedades (ou slots, em CLOS)
• respondem a mensagens (métodos)
• herdam propriedades e métodos de suas superclasses
Implementação de OO em Common Lisp:
• objetos e classes são representados por hash-tables
• propriedades e métodos são entradas das hash-tables
Ex:
• Definindo o objeto circulo:
(setf circ (make-hash-table))
• Criando a propriedade cor:
(setf (gethash 'cor circ) 'preto)
• Acesso a propriedade cor:
(gethash 'cor circ) -> PRETO
• definindo a função area:
(setf (gethash 'area circ)
#'(lambda (x) (pi (expt (rget 'raio x) 2))))*
• Executando a função area:
(funcall (gethash 'area circ) circ)
Utilizando-se uma sintaxe similar a Smaltalk, podemos definir a passagem de mensagem a objetos atraves do método tell:
(defun tell (obj msg &rest args)
(apply (get msg obj) obj args))
Ex: completando definição de círculo:
(setf (gethash 'x circ) 0
(gethash 'y circ) 0
(gethash 'raio circ) 1)
Chamando o método area:
(tell circ 'area) -> 3.1415...
Para acessar as propriedades de um objeto sem utilizar-se os comandos específicos para hash-table, define-se o metodo *rget*:
(defun rget (prop obj)
(gethash prop obj))
Ex:
(rget 'raio circ) -> 1

15.1 Herança

Cria-se uma propriedade :*parent* nas *hash-tables* que indicam a superclasse
Ex:
(setf classe-geometria (make-hash-table)
(gethash 'cor classe-geometria) 'preto)
(setf classe-circulo (make-hash-table)

(gethash :parent classe-circulo) geometria)
classe-circulo é descendente de classe-geometria.
Para simplificar o tratamento de classes e objetos, define-se um objeto como uma instância de uma classe através do mesmo procedimento:
(setf (gethash :parent circ) classe-circulo)
Deve-se modificar a função *rget* de forma que ela busque propriedades e funções também nas superclasses:
(defun rget (prop obj)
(multiple-value-bind (val in) (gethash prop obj)
(if in
(values val in)
(let ((par (gethash :parent obj)))
(and par (rget prop par))))))
assim como o método *tell*:
(defun tell (obj msg &rest args)
(apply (rget msg obj) obj args))
A criação de objetos pode ser implementada por uma função instancia, para abstrair a utilização de *hash-table* do usuário:
(defun instancia (classe)
(let ((obj (make-hash-table)))
(setf (gethash :parent obj) classe)
obj)
)
Exemplo:
(setf classe-geometria (make-hash-table)
(gethash 'x classe-geometria) 0
(gethash 'y classe-geometria) 0
(gethash 'cor classe-geometria) 'preto)
(*setf classe-ponto (instancia classe-geometria))*
(setf (gethash 'print classe-geometria)
#'(lambda (obj) (format t "~%X: ~d Y: ~d" (rget 'x obj) (rget 'y obj))))
(setf p1 (instancia classe-ponto))
(tell p1 'print)
?
X: 0 Y: 0
NIL
Para alterar as propriedades de um objeto, deve-se definir uma função que cria a propriedade no âmbito do objeto:
(defun (setf rget) (val prop obj)
(setf (gethash prop obj) val))
(setf p2 (instancia classe-ponto))
(rget 'x p2) -> 0
(setf (rget 'x p2) 10) -> 10
(rget 'x p2) -> 10
(rget 'x p1) -> 0

16. REFERÊNCIAS

ASSOCIATION OF LISP USERS. CMU Artificial Intelligence Repository, Brad Miller, University of Rochester CS Dept. . Home Page: http://www.cs.rochester.edu/users/staff/miller/alu.html

STEELE, G. L. S. Common Lisp the Language, 2nd Edition. Home Page: http://www.cs.cmu.edu/Web/Groups/AI/html/cltl/cltl2.html
STEELE Jr, G. L. S. Common LISP. The Language. Bedford, Mass, 1990.
WINSTON Patrick H. e HORN, Berthold K. P. LISP. Reading, Mass, Addison Wesley, 1981.
GRAHAN, P. ANSI Common Lisp. Prentice Hall Inc. Englewood Cliffs, New Jersey, 1996.

ANEXO I - CARACTERES RESERVADOS

(Abre uma lista.
) Fecha uma lista.
' Apóstrofe. Abreviação da função *quote.*
; Comentários. Ignora o resto da linha.
\ Barra invertida. Considera o próximo dígito como um caracter, sem significado especial. Para utilizar na notação de diretório é necessário incluir duas barras (a primeira "avisando" que a segunda é efetivamente uma barra).
" Aspas duplas. Delimitam strings.
| Barra vertical.
Delimitam símbolos que incluem caracteres especiais. | d'água |
Sinaliza outras bases numéricas.
#o 105 - 105 na base octal
#x 105 - 105 na base hexadecimal
#b 1011 - 1011 na base binária
#L - caracter objeto de L.
#(a b c) - vetor de três elementos : a, b e c.
#'fn - equivale a (*function* fn)
` Acento grave. Utilizado para constuir tabelas que incluem vírgulas e devem ser montadas.
, Vírgula. Utilizadas em tabelas.
: Dois pontos. Sinaliza palavras-chave, ou símbolos que são avaliados para eles mesmos. *:reset* Pode indicar também a que pacote refere-se o símbolo. *network:reset*
***** Asteriscos não são caracteres reservados, mas normalmente são utilizados, por clareza, como delimitadores de variáveis globais. **variavel**
Os demais caracteres são de uso livre para o usuário.

ANEXO II - RESPOSTAS DOS EXERCÍCIOSANEXO II - RESPOSTAS DOS EXERCÍCIOS

EXERCÍCIOS 3.9

1 - *a* **12 -** *((b (c)) a d e)*
2 - *(b c)* **13 -** Erro!
3 - *(1 2)* **14 -** *(a b (c) e f)*
4 - *((3 4))* **15 -** *4*
5 - *b* **16 -** *2*
6 - *(c)* **17 -** *2*
7 - *(c (d e))* **18 -** *((maçã uva) laranja morango)*
8 - *a* **19 -** *(uva maçã)*
9 - *(a b (c))* **20 -** *(lima maçã laranja)*
10 - (*(b (c))* **.** *a)* **21 -** (*(maçã uva))*
11 - *(a (b (c)))*

EXERCÍCIOS 5.3

1 - *(defun segundo (lista)*

(cadr lista))
2 - *(defun seguinte (atomo lista)*
(cadr (member atomo lista)))
3 - *(defun troca_ultimo (atomo lista)*
(subst (last lista) atomo lista))
4 - *(defun calcula (L)*
;;;soma o segundo elemento da lista ao terceiro e multiplica pelo primeiro
(cond
((equal (length L) 3) ((car L) (+ (cadr L) (caddr L))))))*
5 - *;Retira a primeira ocorrencia do atomo da lista.*
(defun extrai(atm lst)
(cond
((null lst) lst)
((equal atm (car lst)) (cdr lst))
(T (cons (car lst) (extrai atm (cdr lst))))))

EXERCÍCIOS 6.5

1 - *;entrada: um numero e uma lista de numeros.*
;saida: a lista multiplicada pelo numero.
;Multiplica a lista recursivamente.
(defun mul_rec(exp lst)
(cond
((null lst) lst)
(T (cons ((car lst) exp)*
(mul_rec exp (cdr lst))))))
;---
;retorna TRUE se a lista for numerica e
;NIL caso exista pelo menos um atomo nao numerico.
(defun isdigits(lst)
(cond
((null lst) T)
((numberp (car lst)) (isdigits (cdr lst)))))
;---
;Chama funcao para multiplicar uma lista por um numero.
;Realiza testes para verificar se os argumentos sao numericos.
;Qualquer incorrecao nos argumentos a funcao retornara NIL.
(defun mult(exp lst)
(cond
((and (numberp exp) (isdigits lst))
(mul_rec exp lst))))
;---
;Chama funcao para multiplicar uma lista por um numero.
;Realiza testes para verificar se os argumentos sao numericos.
;Qualquer incorrecao nos argumentos a funcao retornara NIL.
;Implementacao nao recursiva!!
(defun mult2(exp lst)
(cond
((and (numberp exp) (isdigits lst))
(mapcar #'(lambda(y) (exp y)) lst))))*

EXERCÍCIOS 7.1

1 - *(defun Maior (LstNum)*

```
(AuxMaior (cdr LstNum) (car LstNum)) )
;-----------------------------------------------------------------------------------------
(defun AuxMaior(LstNum NumMax)
(cond
((null LstNum) NumMax)
((> (car LstNum) NumMax) (AuxMaior (cdr LstNum) (car LstNum)))
(T (AuxMaior (cdr LstNum) NumMax)) ))
```
2 -
```
(defun ExtraiTudo (atm Lst)
(cond
((null Lst) NIL)
((equal atm (car Lst)) (ExtraiTudo atm (cdr Lst)))
(T (cons (car Lst) (ExtraiTudo atm (cdr Lst)))) ))
```
3 -
```
;Concatena ordenadamente duas listas ordenadas de caracteres.
(defun concat(lst1 lst2)
(cond
((null lst1) lst2)
((null lst2) lst1)
((char< (character(car lst1)) (character(car lst2)))
(cons (car lst1) (concat (cdr lst1) lst2)))
((cons (car lst2) (concat lst1 (cdr lst2))))))
;-----------------------------------------------------------------------------------------
;Concatena ordenadamente duas listas ordenadas de valores numericos.
(defun concat2(lst1 lst2)
(cond
((null lst1) lst2)
((null lst2) lst1)
((< (car lst1) (car lst2))
(cons (car lst1) (concat (cdr lst1) lst2)))
((cons (car lst2) (concat lst1 (cdr lst2))))))
```
4 -
```
(defun getn (N L)
;;;retira o n-esimo elemento da lista L
(cond
((null L)())
((equal 1 N) (car L))
((not(equal 1 N)) (getn (- N 1) (cdr L)))))
```
EXERCÍCIOS 12.1

1 -
```
(defun le_lista ()
;;;le uma lista de strings usando "le_nome"
(setq l (le_nome))
(cond
((equal l #\Newline) ())
((not(equal l #\Newline)) (cons l (le_lista)))))
;-----------------------------------------------------------------------------------------
(defun le_nome ()
;;;le um string
(setq c (read-char))
(cond
((equal c #\Space) () )
((equal c #\Newline) #\Newline)
((not(and
```

```
(equal c #\Space)
(equal c #\Newline))) (cons c (le_nome)))))
;-----------------------------------------------------------------------------------
(defun mutantes ()
;;; basicamente, faz muito pouco !!
(permut(le_lista)))
;-----------------------------------------------------------------------------------
(defun permut (l)
(cond
((null l) ())
((not(null l)) (comp (car l) (cdr l)) (permut (cdr l)))))
;-----------------------------------------------------------------------------------
(defun comp (str l)
(cond
((null l) ())
((not(null l)) (mostra (junta str (car l) (interseccao str (car l))))
(mostra (junta (car l) str (interseccao (car l) str)))
(comp str (cdr l)))))
;-----------------------------------------------------------------------------------
(defun junta ( l1 l2 i)
(cond
((null i) ())
((not(equal l1 i))
(cons (car l1) (junta (cdr l1) l2 i)))
((equal l1 i) l2)))
)
;-----------------------------------------------------------------------------------
(defun interseccao ( l1 l2 )
(cond
((null l1) ())
((null l2) ())
((equal l1 l2) l1)
((not(equal l1 l2))
(interseccao (cdr l1) (nbutlast l2)))))
;-----------------------------------------------------------------------------------
(defun mostra ( l )
(cond
((null l) (princ #\newline) () )
((not(null l)) (princ (car l)) (mostra (cdr L)))))
```

EXERCÍCIOS 13.2

1 -
```
(defun insere (arvore novonodo nodo filho)
(cond ((null arvore) nil)
((equal arvore nodo) (cond ((get arvore filho) nil)
((putprop arvore novonodo filho))))
((insere (get arvore 'esq) novonodo nodo filho))
((insere (get arvore 'dir) novonodo nodo filho)) ))
```

2 -
```
;;representacao dos grafos NODO -> ( LIGA (A B C) )
;;diff -> tudo que tem em l1 mas nao em l2.
(defun diff ( l1 l2 )
(cond ((null l1) nil)
```

```
((member (car l1) l2) (diff (cdr l1) l2))
((cons (car l1) (diff (cdr l1) l2)) )) )
;-----------------------------------------------------------------------------------------
;;net-bf
;;pesquisa um nodo e coloca-o numa lista
;;em seguida adiciona a fila de nodos a serem pesquisados todos
;;os nodos que nao estao na lista de nodos pesquisados
(defun net-bf (grafo nodo)
(let ((lista nil)
(fila (get grafo 'liga)))
(loop
(cond ((equal (car fila) nodo) (return T))
((null fila) (return nil))
(T (setq lista (append lista (list(car fila)) ))
(setq fila (append (cdr fila)
(diff (get (car fila) 'liga) lista)) )) )) ))
;-----------------------------------------------------------------------------------------
;;net-df
;;faz o mesmo que net-bf mas com uma pilha e nao uma fila !
(defun net-df (grafo nodo)
(let ((lista nil)
(pilha (get grafo 'liga)))
(loop
(cond ((equal (car pilha) nodo) (return T))
((null pilha) (return nil))
(T (setq lista (append lista (list (car pilha)) ))
(setq pilha (append (diff (get (car pilha) 'liga) lista)
(cdr pilha)) )) )) ))
```

EXERCÍCIOS 14.5

1 -

```
;;;learq le um banco de conhecimentos, interpreta as
;;;palavras chaves desse banco, e gera uma arvore com
;;;o conhecimento lido.
(defun learq()
(setq pal)
(with-open-file (arq "b:\\doencas.txt" :direction :input
:if-does-not-exist :error)
(do ((pal (read arq)(setf pal (read arq))))
((equal pal 'fim))
(cond
((equal pal 'banco)
(setf banco (read arq)))
;nome do banco de conhecimento
((equal pal 'primeiro)
(putprop banco (read arq) 'primeiro))
;indicacao para o primeiro nodo da lista
((equal pal 'nodo)
(setf simbolo (read arq)))
;nome do nodo
((equal pal 'perg)
(putprop simbolo (read-line arq) 'perg))
```

```
;pergunta associada ao nodo
((equal pal 'expl)
(putprop simbolo (read-line arq) 'expl))
;explicacao associada a pergunta
((equal pal 'sim)
(putprop simbolo (read arq) 'sim))
;elo para novo nodo
;resposta a pergunta foi sim
((equal pal 'nao)
(putprop simbolo (read arq) 'nao))
;elo para novo nodo
;resposta a pergunta foi nao
((equal pal 'conclusao)
(putprop simbolo (read-line arq) 'conclusao))))
;conclusao associada ao sistema
(close arq)))
;-----------------------------------------------------------------------------------------
;;;Rotina principal do sistema de diagnostico
(defun principal()
(setq banco)
(setq simbolo)
(learq) ;le o banco de conhecimento
(diag (get banco 'primeiro))) ;faz o diagnostico da doenca
;-----------------------------------------------------------------------------------------
;;;diag percorre a lista de conhecimento ate chegar a
;;;algum diagnostico.
;;;Parametro:
;;; simbolo: nodo de conhecimento
(defun diag(simbolo)
(cond
((not (null (get simbolo 'conclusao)))
(get simbolo 'conclusao))
;ultimo nodo - apresenta conclusao
(T (print(get simbolo 'perg))
;mostra pergunta associado ao nodo
(print(get simbolo 'expl))
;mostra explicacao associada a pergunta
(cond
((yes-or-no-p) (diag (get simbolo 'sim)))
;passa para novo nodo com resp. sim
(T(diag (get simbolo 'nao)))))))
;passa para novo nodo com resp. nao
```

2 - *(defun vinho ()*

```
(close-all-files)
(setq arquivo (open "vinho"))
(le-arquivo arquivo)
(forward))
;-----------------------------------------------------------------------------------------
(defun member (e l)
;;um elemento e eh membro da lista l se for igual a um de seus
```

```
;;elementos ou se um de seus elementos for substring de e.
(cond ((null l) NIL)
((and (equal e (car l))
(if (stringp e) (string-search* (car l) e) T )) T)
( T (member e (cdr l)) )) )
;-----------------------------------------------------------------------------------
(defun inclui (e l)
;;devolve uma lista composta pela lista l e o elemento e se este
;;ja nao pertencer a lista
(if (not (null e))
(cond ((not(member e l)) (cons e l))
( T l )) l ))
;-----------------------------------------------------------------------------------
(defun retira (e l)
;;devolve uma lista l sem o elemento e se este for igual a um de seus
;;elementos ou um destes for substring de e.
(if (not (null e))
(cond ((null l) ())
((or (equal (car l) e)
(if (stringp e) (string-search* (car l) e)))
(retira e (cdr l)) )
( T (cons (car l) (retira e (cdr l)) )) ) l ))
;-----------------------------------------------------------------------------------
(defun getatrib (regra arquivo)
;;le uma atributo de uma regra, adiciona este a lista de atributos
;;da regra e a lista de todos os atributos ( latributos ).
(let ((atributo (read-line arquivo)))
(set regra (cons atributo (eval regra)) )
(setq latributos (cons atributo latributos)) ))
;-----------------------------------------------------------------------------------
(defun le-arquivo (arquivo)
(setq lregras ())
(setq latributos ())
(setq lperguntas ())
(loop
(let ((palavra (read arquivo)))
(cond
((equal palavra 'FIM)
(return))
((equal palavra 'REGRA)
(setq regra (make-symbol (string-append "regra" (read-line arquivo)) ))
(set regra ())
(setq lregras (cons regra lregras)) )
((or (equal palavra 'SE) (equal palavra 'E))
(getatrib regra arquivo))
((equal palavra 'ENTAO)
(putprop regra (read-line arquivo) 'conclusao ))
((equal palavra 'PERGUNTA)
(le-pergunta arquivo)) )) ))
;-----------------------------------------------------------------------------------
```

```
(defun le-pergunta (arquivo)
;;le um atributo e um string associado a uma pergunta e coloca-os
;;na lista de perguntas
(let ((atributo (read-line arquivo))
(pergunta (read-line arquivo)) )
(setq lperguntas (cons (list atributo pergunta) lperguntas)) ))
;---------------------------------------------------------------------------------------
(defun pergunta (l)
;;recebe ums lista composta de um atributo e um string associado
;;a ele, pergunta e devolve a resposta
(if (not (null l))
(let ((atributo (car l))
(pergunta (cadr l)) )
(cond ((string-search* "tem" atributo)
(terpri) (princ pergunta) (terpri)
(cond ((y-or-n-p) atributo)
( T (setq latributos (tira-atrib atributo latributos))
(string-append "nao " atributo)) ))
( T
(terpri) (print pergunta) (terpri)
(string-append atributo " " (read-line)) )) )) )
;---------------------------------------------------------------------------------------
(defun tira-atrib (a l)
;;se uma pergunta do tipo "tem molho ?" teve resposta negativa, esta funcao
;;retira todas as demais perguntas relativas a este atributo
(cond ((null l) ())
((string-search* a (string-append "tem " (car l)) )
(tira-atrib a (cdr l)) )
( T (cons (car l) (tira-atrib a (cdr l)) )) ))
;---------------------------------------------------------------------------------------
(defun busca-pergunta (a l)
;;busca uma pergunta associada ao atributo a na lista l
;;neste programa l sera sempre lperguntas
(if (not (null l))
(let ((atributo (car (car l)) )
(pergunta (cadr l)) )
(cond ((and (not(null(string-search* atributo a)) )
(if (string-search* "tem" a) (string-search* "tem" atributo) T))
(car l))
((busca-pergunta a (cdr l)) )) )) )
;---------------------------------------------------------------------------------------
(defun valida (regra)
;;busca pelos atributos de uma regra na lista de fatos
;;se um deles nao estiver presente, retorna NIL senao retorna T
(do ((atributos regra (cdr atributos)))
((null atributos) T)
(if (not (member (car atributos) fatos)) (return nil)) ))
;---------------------------------------------------------------------------------------
(defun forward()
(let ((resposta () )
```

```
(fatos () )
(atributos () ))
;;repete ate que esgotar todas as regras ou ateh que os atributos
;;sejam todos valorados
(do ((regra (eval (car lregras)) (eval (car lregras)) ))
((or (null latributos)
(null lregras)) resposta)
;;pega os atributos de uma regra e busca as perguntas associadas
;;a cada um se ja nao tiver sido valorado
;;cada atributo valorado e retirado de latributos
;;cada pergunta e retirada de lperguntas se ao ser utilizada
;;cada resposta e incluida na lista de fatos
(do ((atributos (reverse regra) (cdr atributos)) )
((null atributos))
(setq atrib (car atributos))
(cond ((member atrib latributos)
(setq latributos (retira atrib latributos))
(cond ((not (member atrib fatos))
(setq perg (busca-pergunta (car atributos) lperguntas))
(setq fatos (inclui (pergunta perg) fatos))
(setq lperguntas (retira perg lperguntas)) )) )) )
;;se todos as condicoes da regra estiverem na lista de fatos
;;a sua conclusao e incluida na lista de respostas
(cond ((valida regra)
(setq resposta (inclui(get(car lregras) 'conclusao) resposta)) ))
(setq lregras (cdr lregras)) )) )
```