LISP 1.5 E UMA IMPLEMENTAÇÃO

NO SISTEMA B-6700

SIANG WUN SONG

DISSERTAÇÃO APRESENTADA AO

INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

DA

UNIVERSIDADE DE SÃO PAULO

PARA OBTENÇÃO DO GRAU DE MESTRE

EM

MATEMÁTICA APLICADA

ORIENTADOR:

Prof.Dr. VALDEMAR WAINGORT SETZER

-São Paulo, agosto de 1975 -

A meus pais

e

minha esposa

AGRADECIMENTOS

ao Prof. Dr. Valdemar Waigort Setzer, pela valiosa orientação e incentivos,

ao colega Prof. Jorge Stolfi, pelas sugestões e críticas construtivas,

aos colegas e funcionários do Departamento de Matemática Aplicada,

ao Sr. João Baptista Esteves de Oliveira, pela excelente datilografia

e ao Sr. Armando Garcia Segura, pelo esmerado serviço gráfico.

INTRODUÇÃO

O presente trabalho é constituído de quatro capí tulos. No Capítulo I, é feita uma introdução à linguagem LISP. A sintaxe da meta-linguagem LISP é apresentada e, para dar a sua semântica, uma meta-função chamada val é definida. A transformação de m-expressões para S-expressões é feita por uma meta-função chamada tran. Desta maneira, pude mos desenvolver uma descrição funcional, original, da linguagem LISP.

No Capítulo II, o interpretador LISP é descrito de uma maneira geral, sem detalhes de implementações particula res. É definida a função universal evalquote e estabelecida a relação entre esta e a meta-função val. É também discutido o problema de "garbage collection", ou "recuperação de cé lulas inativas", em sistemas de computação com memória virtual.

No Capítulo III, é descrita uma implementação particular, feita por nós em ALGOL no sistema B-6700. É feita uma descrição detalhada do algoritmo de recuperação de célu las inativas. Diversos outros tipos de algoritmos de recupe ração são comentados.

No Capítulo IV, é descrita uma outra implementação no B-6700, feita por M. Magidin e R. Segovia. São feitas algumas comparações entre as duas implementações, assim como diversos comentários e críticas sobre o trabalho apresentado.

Note-se que esteve sempre presente na redação desta dissertação a possível utilização didática de certos capítulos da mesma, notadamente os Capítulos I e II. No entanto, deve-se observar que supomos um nível de alunos de graduação ou pós-graduação com boa formação matemática. Dessa maneira, o formalismo do Capítulo I é, em nossa opinião, adequado ao ensino da linguagem LISP para esse tipo de alunos.

A B S T R A C T

The LISP 1.5 language is defined in a formal way developed by the author. The syntax of the meta-language LISP is presented and, in order to give its semantics, a meta-function called *val* is defined. The transformation of m-expressions into S-expressions is performed by a meta-function called *tran*. To describe the LISP interpreter, the universal function evalquote is defined and its relation to the meta-function *val* is established. The problem of garbage collection in virtual memory computer systems is discussed and comments concerning various kinds of garbage collection algorithms are made. An interpreter implemented in ALGOL on the B-6700 computer is described. A comparison with the Magidin-Segovia implementation on the B-6700 is also presented.

Key words and phrases: LISP, list processing, lambda calculus, garbage collection, interpreters.

ÍNDICE

Capítulo I - Introdução à linguagem LISP

1. S-EXPRESSÕES 1
   1.1 - Átomos, S-expressões, listas 1
   1.2 - Representação gráfica de S-expressões. . 3
2. FUNÇÕES E FORMAS 5
   2.1 - Meta-linguagem: m-expressões 5
   2.2 - Funções e predicados elementares 7
   2.3 - Funções e predicados aritméticos 10
   2.4 - Ocorrências livre e ligada de um identificador. Operador de substituição. . . . 13
   2.5 - Avaliação de funções e formas. 16
   2.6 - Algumas funções definidas recursivamente 22
   2.7 - Funções lógicas. 25
3. TRANSFORMAÇÃO DE m-EXPRESSÕES PARA S-EXPRESSÕES 26

Capítulo II - Descrição do Sistema LISP

1. SISTEMA LISP 28
2. ESPAÇO LIVRE E SUA ORGANIZAÇÃO 28
   2.1 - Célula 28
   2.2 - Espaço livre 30
   2.3 - Organização do espaço livre. 30
3. REPRESENTAÇÕES INTERNAS. 32
   3.1 - Representação interna de átomos não numéricos. 32
   3.2 - Lista dos símbolos atômicos. 39
   3.3 - Função para busca de propriedades: função get. 40

3.4 - Representação interna de S-expressões. . 41

4. PSEUDO-FUNÇÕES 43
4.1 - Pseudo-funções RPLACA e RPLACD 44
4.2 - Pseudo-função NCONC. 44
4.3 - Pseudo-funções CSET e CSETQ. 44
4.4 - Pseudo-funções READ e PRINT. 45
4.5 - Pseudo-função DEFINE 46

5. INTERPRETADOR. 47
5.1 - Função universal evalquote 47
5.2 - Exemplos de interpretação. 52
5.3 - Lista de associação. 54
5.4 - Argumentos funcionais. 57

6. AVALIAÇÃO DA FORMA ESPECIAL PROG 59

7. RECUPERADOR DE CÉLULAS INATIVAS. 62

Capítulo III - Implementação do Sistema LISP

1. ORGANIZAÇÃO ESQUEMÁTICA DO SISTEMA LISP. . . . 66

2. ORGANIZAÇÃO DE DADOS 66
2.1 - Célula 66
2.2 - Espaço livre 69
2.3 - Lista dos símbolos atômicos. 71
2.4 - Representação de números 72

3. ENTRADA E SAÍDA. 73
3.1 - Átomos especiais 73
3.2 - Criação de p-listas. 74
3.3 - Procedimentos de entrada 77
3.4 - Procedimentos de saída 79

4. FUNÇÕES COMPOSTAS DE CAR'S E CDR'S 81

5. FUNÇÕES DO SISTEMA LISP. 82

6. INTERPRETADOR. 84

7. RECUPERADOR DE CÉLULAS INATIVAS. 86
7.1 - Pilha de argumentos save 86
7.2 - Uso da pilha save. 86
7.3 - Alguns exemplos. 88
7.4 - Finalidade da pilha save 90
7.5 - Descrição do algoritmo 90
7.6 - Comentários sobre o algoritmo. 96
7.7 - Procedimento garbage 98

8. CARTÕES DE CONTROLE. 101

9. SUPERVISOR 104

10. RESUMO DAS PARTICULARIDADES DA IMPLEMENTAÇÃO . 105

Capítulo IV - Uma outra implementação e comentários

1. IMPLEMENTAÇÃO MAGIDIN-SEGOVIA. 107
1.1 - Organização de dados 107
1.2 - Recuperador de células inativas. 110
1.3 - Procedimentos de entrada e saída 113
1.4 - Procedimento de avaliação 114

2. ALGUMAS COMPARAÇÕES ENTRE AS DUAS IMPLEMENTAÇÕES 115
2.1 - Representação de átomos. 115
2.2 - Abreviatura de car's e cdr's 116
2.3 - Número de argumentos 116
2.4 - Ligação de valores aos átomos. 116
2.5 - Recuperação de células inativas. 117

3. COMENTÁRIOS E CRÍTICAS 118
3.1 - Eliminação da notação do par com ponto . 118
3.2 - Meta-função val. 119
3.3 - Uso do ALGOL na implementação. 121
3.4 - O interpretador. 122

Apêndice I. 124

Apêndice II 127

Referências Bibliográficas. 129

Referências Bibliográficas Adicionais 131

- □ -

CAPÍTULO I

INTRODUÇÃO À LINGUAGEM LISP

1. S-EXPRESSÕES

1.1 - Átomos, S-expressões, listas

Um conceito importante no sistema LISP é o de expressões simbólicas, ou S-expressões, cuja definição envolve o conceito de átomo. O conceito de átomo será considerado primitivo. Vamos supor a existência de um conjunto A de átomos e de um conjunto N, subconjunto de A, chamado conjunto de átomos numéricos. Suporemos que a cada elemento de N pode-se associar um número dentro de certos limites de grandeza e com um certo número fixo de algarismos.

Os átomos não numéricos, de A - N, serão representados por uma cadeia de letras maiúsculas e algarismos, iniciando-se a cadeia por uma letra maiúscula. Os átomos serão também chamados de símbolos atômicos.

Exemplos de representações de átomos:

1234

-25

1.5

1.5E2

1.5E-2

ATOMO

H2S04

ATOMOMUITOMUITOLONGO

Suporemos ainda a existência dos átomos não numéricos T e NIL, pertencentes a A - N.

DEFINIÇÃO 1.1.1 - Define-se o *conjunto das S-expressões* S como sendo o menor conjunto tal que:

a) um átomo ∈ S

b) Se u ∈ S e v ∈ S então (u . v) ∈ S.

O par (u . v) é denominado de *par com ponto* cujo 1º membro é u e 2º membro é v.

Exemplos de representações de S-expressões:

ATOMO

3.14159

(A . A)

(ALGOL . B6700)

(LISP . (ALGOL . B6700))

(A . (B . (C . NIL)))

((TRES . (PARES . COM)) . PONTO)

DEFINIÇÃO 1.1.2 - Vamos definir o *conjunto das S-expressões não atômicas* P como sendo:

$$P = S - A$$

DEFINIÇÃO 1.1.3 - Vamos definir o *conjunto das listas* L como sendo o conjunto dos elementos de S, tais que:

a) NIL $\in$ L

b) Se u $\in$ S e v $\in$ L então (u . v) $\in$ L.

NOTAÇÃO - Um elemento de L, com exceção de NIL, tem a forma

$$(u_1 \ . \ (u_2 \ . \ (\ldots \ . \ (u_n \ . \ \text{NIL}) \ldots))),$$

$n > 0$, $u_i \in S$, $i=1,2,\ldots,n$, e, por simplicidade, será representado pela notação:

$$(u_1 \quad u_2 \quad \ldots \quad u_n)$$

Por extensão desta notação, o elemento NIL, que chamaremos de *lista vazia*, será representado por ().

DEFINIÇÃO 1.1.4 - Vamos definir o *conjunto das listas não vazias* L' como sendo:

$$L' = L - \{NIL\}$$

Temos $L' \subset P$ pois qualquer elemento x de L' é da forma (u . v), com u $\in$ S e v $\in$ L $\subset$ S, logo x $\in$ P.

Exemplos de listas:

()

(A)

(A (B C) D)

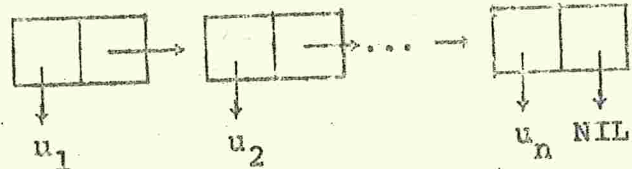
(OS 10 ELEM DESTA LISTA SAO 9 ATOMOS E
(UMA LISTA DE 5 ELEM))

1.2 - Representação gráfica de S-expressões

Uma lista formada por n elementos quaisquer

$$(u_1 \quad u_2 \quad \ldots \quad u_n)$$

é representada graficamente por:

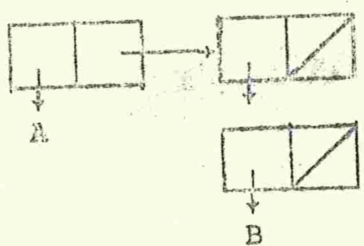


onde [ ] é também representado por [/]

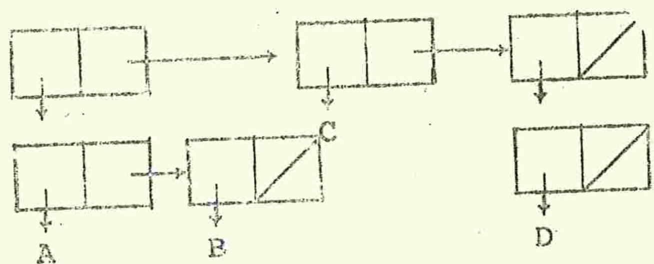
NIL

<u>Exemplos:</u>

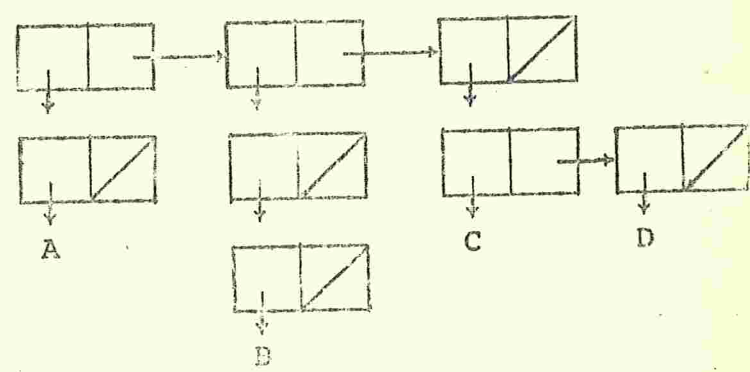
1) (A (B))



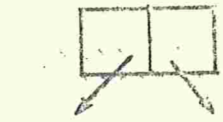
2) ((A B) C (D))



3) ((A) ((B)) (C D))



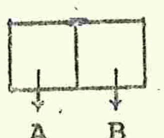
Uma S-expressão na forma de (1º membro. 2º membro) é representada graficamente por:



1º membro   2º membro

Exemplos:

1) (A . B)



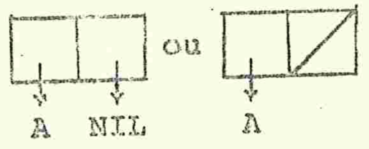
2) ((TRES . (PARES . COM)) . PONTO)

PONTO

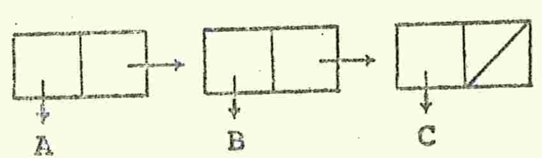
COM

TRES   PARES

3) (A . NIL)



4) (A . (B . (C . NIL)))



2. FUNÇÕES E FORMAS

2.1 - Meta-linguagem: m-expressões

A meta-linguagem LISP, formada por m-expressões, é

um sistema que nos permite escrever algoritmos para manipulação de S-expressões. A sintaxe das m-expressões é dada nesta seção e a sua semântica será vista logo a seguir nas seções seguintes.

<m-expressão>::=<função>|<forma>

<função>::=<identificador>|λ[[];<forma>]|
λ[[<seq. de variáveis>];<forma>]|
label[<identificador>;<função>]

<seq.de variáveis>::=<variável>|<seq.de variáveis>;<variável>

<forma>::=<constante>|<variável>|<função>[]|
<função>[<seq.de argumentos>]|
[<seq.de condicionais>]

<seq.de argumentos>::=<argumento>|
<seq.de argumentos>;<argumento>

<seq.de condicionais>::=<forma>→<forma>|
<seq.de condicionais>;<forma>→<forma>

<constante>::=<S-expressão>

<variável>::=<identificador>

<argumento>::=<forma>|<argumento funcional>

<argumento funcional>::=<função>

<identificador>::=<letra minúscula>|
<identificador><letra minúscula>|
<identificador><algarismo>

<letra minúscula>::=a|b|c|...|x|y|z

<algarismo>::=0|1|2|...|8|9

<S-expressão>::=<átomo>|<par com ponto>

<átomo>::=<átomo não numérico>|<átomo numérico>

<átomo não numérico>::=<letra>|<átomo não numérico><letra>|
<átomo não numérico><algarismo>

<letra>::=A|B|C|...|X|Y|Z

<par com ponto>::=(<S-expressão>.<S-expressão>)

Não será dada a sintaxe de <átomo numérico>.

2.2 - Funções e predicados elementares

Uma função LISP, que tem como domínio um subconjunto de S e valores num subconjunto de S, pode ser indicada por um identificador como car, cdr, cons, etc. cujos significados serão vistos a seguir. Um *predicado* é uma função que tem valores no conjunto {T, NIL}.

DEFINIÇÃO 2.2.1 - FUNÇÃO car

car : P ⟶ S

car[x] = u se x = (u . v)∈P

Observamos que car[x] = u_1 se x = (u_1 u_2 ... u_n)∈L'⊂P

Exemplos:

car[(A . B)] = A

car[(A B)] = A

car[((A B) C (D))] = (A B)

DEFINIÇÃO 2.2.2 - FUNÇÃO cdr

cdr : P ⟶ S

cdr[x] = v se x = (u . v)∈P

Observamos que cdr[x]= $(u_2 \ldots u_n)$ se

$x = (u_1\ u_2 \ldots u_n)$∈L'⊂P

Exemplos:

cdr[(A . B)] = B

cdr[(A   B)] = (B)

cdr[((A B) C (D))] = (C (D))

Notação

As funções compostas de car's e cdr's serão abreviadas com nomes que começam pela letra c, terminam pela letra r, tendo entre elas uma cadeia que indicaremos por $\alpha_1 \alpha_2 \ldots \alpha_k$, onde cada α_i, i=1,2,...,k, denota a letra a ou a letra d, isto é, $\alpha_i \in \{a,d\}$, com o seguinte significado:

$$c\ \alpha_1\ \alpha_2 \ldots \alpha_k\ r[x] = c\ \alpha_1 r[c\ \alpha_2 r[\ldots[c\ \alpha_k r[x]]\ldots]]$$

Exemplos:

cadr[(A B C)] = car[cdr[(A B C)]] = B

cddar[((A B C) D)] = cdr[cdr[car[((A B C) D)]]] = (C)

DEFINIÇÃO 2.2.3 - FUNÇÃO cons

cons : S x S ⟶ P

cons[x;y] = (x . y)

Exemplos:

cons[A ; B] = (A . B)

cons[A ; (B)] = (A . (B)) = (A B)

cons[(A B) ; (C (D))] = ((A B) . (C (D))) = ((A B) C (D))

DEFINIÇÃO 2.2.4 - PREDICADO atom

atom : S ⟶ {T, NIL}

$$\text{atom}[x] = \begin{cases} T & \text{se } x \in A \\ NIL & \text{se } x \notin A \end{cases}$$

Exemplos:

atom[A] = T

atom[(A)] = NIL

atom[()] = T

atom[1975] = T

DEFINIÇÃO 2.2.5 - PREDICADO numberp

numberp : S ⟶ {T, NIL}

$$\text{numberp}[x] = \begin{cases} T & \text{se } x \in N \\ NIL & \text{se } x \notin N \end{cases}$$

Exemplos:

numberp[1975] = T

numberp[(1975)] = NIL

numberp[(A B)] = NIL

DEFINIÇÃO 2.2.6 - PREDICADO null

$$null : S \longrightarrow \{T, NIL\}$$

$$null[x] = \begin{cases} T & \text{se } x = NIL \\ NIL & \text{se } x \neq NIL \end{cases}$$

Exemplos:

null[()] = T

null[NIL] = T

null[(A)] = NIL

null[(NIL)] = NIL

DEFINIÇÃO 2.2.7 - PREDICADO eq

$$eq : A \times A \longrightarrow \{T, NIL\}$$

$$eq[x;y] = \begin{cases} T & \text{se } x = y \\ NIL & \text{se } x \neq y \end{cases}$$

Exemplos:

eq[A ; A] = T

eq[A ; B] = NIL

2.3 - FUNÇÕES E PREDICADOS ARITMÉTICOS

As funções aritméticas são definidas para argumentos que são átomos numéricos. Operações aritméticas são efetuadas sobre os números associados a estes átomos numéricos (números esses dentro de certos limites de grandeza e com

um certo número fixo de algarismos). Seja n um número que é o resultado das operações aritméticas. Então o valor da função aritmética será um átomo numérico que é associado a este número n, ou a uma aproximação do mesmo sem alterar a grandeza. Caso tal átomo numérico não exista, o valor da função aritmética é indefinido.

a) plus $[x_1;x_2;...;x_n] = x_1+x_2+...+x_n$

b) difference $[x;y] = x-y$

c) minus $[x] = -x$

d) times $[x_1;x_2;...;x_n] = x_1\ x_2\ ...\ x_n$

e) add1 $[x] = x + 1$

f) sub1 $[x] = x - 1$

g) entier $[x] = \lfloor x \rfloor$ ou maior inteiro contido em x

h) quotient $[x;y] = x/y$

i) remainder $[x;y] = x - \lfloor x/y \rfloor y$

j) divide $[x;y] = \quad (\lfloor x/y \rfloor \quad x - \lfloor x/y \rfloor y)$

k) max$[x_1;x_2;...;x_n]$ = máximo de $\{x_1,x_2,...,x_n\}$

l) min$[x_1;x_2;...;x_n]$ = mínimo de $\{x_1,x_2,...,x_n\}$

m) expt $[x;y] = x^y$

n) sqrt $[x] = \sqrt{|x|}$

o) recip $[x] = 1/x$

p) absval $[x] = |x|$

q) log $[x] = \log_{10}x$

r) ln $[x] = \log_e x$

s) sin $[x]$ = seno de x

t) cos [x] = cosseno de x

u) tan [x] = tangente de x

v) arcsin [x] = arco seno de x

w) arccos [x] = arco cosseno de x

x) sinh [x] = seno hiperbólico de x

y) cosh [x] = cosseno hiperbólico de x

z) tanh [x] = tangente hiperbólica de x

Os argumentos dos predicados aritméticos são átomos numéricos. Caso contrário o valor do predicado não é definido.

a) zerop [x] = $\begin{cases} T & \text{se } x = 0 \\ NIL & \text{se } x \neq 0 \end{cases}$

b) onep [x] = $\begin{cases} T & \text{se } x = 1 \\ NIL & \text{se } x \neq 1 \end{cases}$

c) minusp [x] = $\begin{cases} T & \text{se } x < 0 \\ NIL & \text{se } x \geq 0 \end{cases}$

d) greaterp [x;y] = $\begin{cases} T & \text{se } x > y \\ NIL & \text{se } x \leq y \end{cases}$

e) lessp [x;y] = $\begin{cases} T & \text{se } x < y \\ NIL & \text{se } x \geq y \end{cases}$

2.4 - Ocorrências livre e ligada de um identificador Operador de substituição

Veremos algumas definições que serão usadas na seção seguinte para avaliação de funções e formas.

NOTAÇÃO 2.4.1

Se α é uma cadeia que satisfaz à sintaxe de <x> ou de <x> é o lado esquerdo de uma regra da gramática apresentada na seção 2.1 deste capítulo, diremos que "α é <x>".

NOTAÇÃO 2.4.2

A notação de uma sequência qualquer $\alpha_1, \alpha_2, \ldots, \alpha_n$ sem referência a n implica a possibilidade de $n = 0$, obtendo se a sequência vazia.

DEFINIÇÃO 2.4.1

Sejam ξ <identificador>,

ϕ <função>,

$\alpha_1, \ldots, \alpha_n$ <argumento>,

$\rho, \pi_1, \ldots, \pi_n, \varepsilon_1, \ldots, \varepsilon_n$ <forma>.

Uma ocorrência de ξ em uma <função> ou <forma> é *livre* se e somente se tal pode ser provado pelas regras seguintes:

i) A ocorrência de ξ no contexto "ξ" é livre.

ii) Qualquer ocorrência livre de ξ em ϕ, ou em algum dos $\alpha_1, \ldots, \alpha_n$, é livre em

$$\phi[\alpha_1; \ldots; \alpha_n]$$

iii) Qualquer ocorrência livre de ξ em algum π_i ou ε_i, i=1, ...,n, é livre em

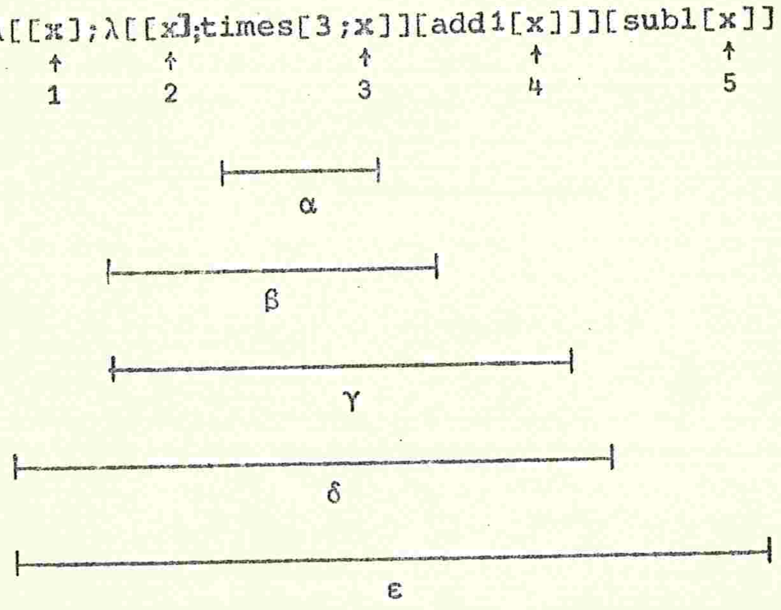
$$[\pi_1 \longrightarrow \varepsilon_1; \ ...; \ \pi_n \longrightarrow \varepsilon_n]$$

iv) Se cada ζ_i, i=1,...,k, é um <identificador> distinto de ξ, então qualquer ocorrência livre de ξ em ρ é livre em

$$\lambda[[\zeta_1; \ ...; \ \zeta_k]; \ \rho]$$

Uma ocorrência de ξ é *ligada* em uma <função> ou <forma> se e somente se não é livre.

Exemplo:



Suponhamos que $\alpha, \beta, \gamma, \delta$ e ε denotam as sub-expressões sublinhadas e numeremos as ocorrências de x de 1, 2, 3, 4 e 5. Temos:

a ocorrência 1 é ligada em δ

a ocorrência 2 é ligada em β

a ocorrência 3 é livre em α mas ligada em β

a ocorrência 4 é livre em γ mas ligada em δ

a ocorrência 5 é livre em ε.

NOMENCLATURA

Seja uma λ-expressão

λ[[<seq.de variáveis>];<forma>].

Cada <variável> da <seq.de variáveis> é chamada de *variável ligada* da λ-expressão (ou λ-variável), e a <forma> é chamada de *corpo* da λ-expressão.

Notemos que cada ocorrência livre de x no exemplo apresentado, exceto a ocorrência 5, passa do estado livre para ligado à medida que consideramos sub-expressões cada vez maiores. Mesmo a ocorrência 5 poderá vir a ficar ligada em alguma λ-expressão que engloba ε no seu corpo. Veremos na seção seguinte que cada ocorrência livre de x em ε será associada com a variável ligada da particular λ-expressão em que muda o seu estado.

DEFINIÇÃO 2.4.2 - Operador Σ de substituição

Sejam ρ<forma>,

$\xi_1, \ldots, \xi_n$ <identificador>,

$\psi_1, \ldots, \psi_n$ <função> ou <forma>.

O resultado da substituição de ψ_i, $i=1,\ldots,n$, para todas as ocorrências livres das correspondentes ξ_i em ρ será denotado por

$$\sum\nolimits_{\psi_1,\ldots,\psi_n}^{\xi_1,\ldots,\xi_n} \rho.$$

2.5 - Avaliação de funções e formas

Veremos nesta seção como podemos obter o valor de uma cadeia satisfazendo à sintaxe de <m-expressão>.

DEFINIÇÃO 2.5.1

Seja ψ <função> ou <forma>. Para obter o valor de ψ, vamos definir uma meta-função val. O valor de ψ, que indicaremos por val $\{\psi\}$, será ou <constante> ou <função>, ou ainda, indefinido.

A meta-função val será definida por:

a) Se ψ é <função>, então

val$\{\psi\} = \psi$.

Exemplos:

```
val{car} = car
val{eq} = eq
val{λ[[x;y];cons[y;x]]} = λ[[x;y];cons[y;x]]
val{label[ff;λ[[x];[atom[x]→x;T→ff[car[x]]]]]}
    = label[ff;λ[[x];[atom[x]→x;T→ff[car[x]]]]]
```

b) Se ψ é <forma>, então estaremos num dos seguintes casos:

b_1) ψ é <constante>, então

$$\underline{val}\{\psi\} = \psi.$$

Exemplos:

$\underline{val}\{25\} = 25$

$\underline{val}\{A\} = A$

$\underline{val}\{(A\ (B))\} = (A\ (B))$

b_2) ψ é a <forma>

$$\phi[\rho_1;\ \ldots;\ \rho_n]$$

onde ϕ é <identificador>, $\rho_1,\ldots,\rho_n$ são <forma> cujos valores são <constante>, isto é, $\underline{val}\{\rho_i\}$ é <constante>, i=1,...
..,n, então

$$\underline{val}\{\phi[\rho_1;\ldots;\rho_n]\} = f[\underline{val}\{\rho_1\};\ldots;\underline{val}\{\rho_n\}],$$

supondo que ϕ é a representação de uma função f e entende-se pela notação acima como a aplicação de f às S-expressões representadas por $\underline{val}\{\rho_1\},\ldots,\underline{val}\{\rho_n\}$, caso estas S-expressões estejam dentro do domínio de f; caso contrário, $\underline{val}\{\psi\}$ é indefinido.

Exemplos:

$\underline{val}\{car[(A\ .\ B)]\} = A$

$\underline{val}\{cons[car[(A\ .\ B)];\ C]\} = (A\ .\ C)$

$\underline{val}\{car[A]\}$ = indefinido

b_3) ψ é a <forma>

$$[\pi_1 \rightarrow \varepsilon_1; \ldots; \pi_n \rightarrow \varepsilon_n]$$

onde $n > 0$ e π_j, ε_j, $j=1,\ldots,n$, são <forma> em que existe i, $i \leq n$, tal que $\underline{val}\{\pi_i\} \neq NIL$, então

$$\underline{val}\{[\pi_1 \rightarrow \varepsilon_1; \ldots; \pi_n \rightarrow \varepsilon_n]\} = \underline{val}\{\varepsilon_i\}$$

para algum i tal que $1 \leq i \leq n$ e $\begin{cases} \underline{val}\{\pi_j\} = NIL, & j=1,\ldots,i-1 \\ \underline{val}\{\pi_i\} \neq NIL \end{cases}$

Exemplos:

val{[null[(A)]→A;T→cdr[(A)]]} = val{cdr[(A)] } = NIL

val{[null[(A)]→A;atom[car[(A)]]→B;T→cdr[(A)]]} = val{B}=B

b_4) ψ é a <forma>

$$\lambda[[\xi_1;\ldots;\xi_n];\rho][\alpha_1;\ldots;\alpha_n]$$

onde ρ é <forma>, $\xi_1,\ldots,\xi_n$ são <variável> e $\alpha_1,\ldots,\alpha_n$ <argumento>, então

$$\underline{val}\{\lambda[[\xi_1;\ldots;\xi_n];\rho][\alpha_1;\ldots;\alpha_n]\} = \underline{val}\{\sum\nolimits_{\underline{val}\{\alpha_1\},\ldots,\underline{val}\{\alpha_n\}}^{\xi_1,\ldots,\xi_n} \rho\}$$

Exemplos:

1) val{λ[[x];λ[[x];times[3;x]][add1[x]]][5]}

$$=\underline{val}\{\sum^{x}_{\underline{val}\{5\}} \lambda[[x];times[3;x]][add1[x]]\}$$

=val{λ[[x];times[3;x]][add1[5]]}

$$=\underline{val}\{\sum^{x}_{\underline{val}\{add1[5]\}} times[3;x]\}$$

=val{times[3;6]} = 18

2) val{λ[[x;y];y[x]][(A B);λ[[x];car[x]]]}

$$=\underline{val}\{\sum^{x,y}_{\underline{val}\{(A\ B)\},\underline{val}\{\lambda[[x];car[x]]\}} y[x]\}$$

=val{λ[[x];car[x]][(A B)]}

$$=\underline{val}\{\sum^{x}_{\underline{val}\{(A\ B)\}} car[x]\}$$

=val{car[(A B)]} = A

b_5) ψ é a <forma>

$$label[\xi;\phi][\alpha_1;\ldots;\alpha_n] \quad (*)$$

onde ξ é <identificador>, φ <função> e $\alpha_1,\ldots,\alpha_n$ <argumento>, então

$$\underline{val}\{label[\xi;\phi][\alpha_1;\ldots;\alpha_n]\} = \underline{val}\{\sum^{\xi}_{label[\xi;\phi]} \phi[\alpha_1;\ldots;\alpha_n]\}.$$

(*) ver observação sobre o uso da notação label no fim desta seção.

Exemplo:

Seja a função label[ff;λ[[x];[atom[x]→x;T→ff[car[x]]]]] que aplicada a uma S-expressão qualquer produz o primeiro símbolo atômico da S-expressão. Por motivo de simplicidade, vamos denotar por ϕ a λ-expressão λ[[x];[atom[x]→x;T→ff[car[x]]]].

val{label[ff;λ[[x];[atom[x]→x;T→ff[car[x]]]]][(A B)]}

=val{$\sum^{ff}_{label[ff;\phi]}$ λ[[x];[atom[x]→x;T→ff[car[x]]]][(A B)]}

=val{λ[[x];[atom[x]→x;T→label[ff;ϕ][car[x]]]][(A B)]}

=val{$\sum^{x}_{val\{(A\ B)\}}$ [atom[x]→x;T→label[ff;ϕ][car[x]]]}

=val{[atom[(A B)]→(A B);T→label[ff;ϕ][car[(A B)]]]}

=val{label[ff;λ[[x];[atom[x]→x;T→ff[car[x]]]]][car[(A B)]]}

=val{$\sum^{ff}_{label[ff;\phi]}$ λ[[x];[atom[x]→x;T→ff[car[x]]]][car[(A B)]]}

=val{λ[[x];[atom[x]→x;T→label[ff;ϕ][car[x]]]][car[(A B)]]}

=val{$\sum^{x}_{val\{car[(A\ B)]\}}$ [atom[x]→x;T→label[ff;ϕ][car[x]]]}

=val{[atom[A]→A;T→label[ff;ϕ][car[A]]]}

=val{A}

= A

b_6) ψ é uma <forma> não enquadrada em (b_1) a (b_5), então

val{ψ} é indefinido.

Exemplos:

val{x} = indefinido

val{car[car]} = indefinido

val{[null[(A)]→NIL;atom[(A)]→A]} = indefinido

Observação sobre o uso da notação label:

O uso principal da notação label é para escrever funções definidas recursivamente. Seja o exemplo clássico da função fatorial de um número inteiro $n \geq 0$ definida recursivamente por:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n(n-1)! & \text{se } n \neq 0 \end{cases}$$

Para isso, queremos escrever a função fatorial como sendo

λ[[n];[zerop[n]→1;T→times[n;fatorial[sub1[n]]]]].

A fim de que o <identificador> fatorial se refira a esta λ-expressão, vamos utilizar a notação label

label[ξ;φ]

em que o <identificador> ξ é associado à <função> φ, de modo que referências a este <identificador> podem ser feitas

em ϕ. Assim, a função fatorial é escrita como:

label[fatorial;λ[[n];[zerop[n]→1;

T→times[n;fatorial[sub1[n]]]]]].

2.6 - Algumas funções definidas recursivamente

NOTAÇÃO

Sejam ϕ <identificador>,

$\xi_1,\ldots,\xi_k$ <variável>,

$\pi_1,\ldots,\pi_n$, $\varepsilon_1,\ldots,\varepsilon_n$ <forma>.

Por motivo de simplicidade, em lugar de

$$\phi = \text{label}[\phi;\lambda[[\xi_1;\ldots;\xi_k];[\pi_1\to\varepsilon_1;\ldots;\pi_n\to\varepsilon_n]]],$$

vamos escrever

$$\phi[\xi_1;\ldots;\xi_k] = [\pi_1\to\varepsilon_1;\ldots;\pi_n\to\varepsilon_n].$$

Exemplo:

Em lugar de

fatorial = label[fatorial;λ[[n];[zerop[n]→1;

T→times[n;fatorial[sub1[n]]]]]]

vamos escrever

fatorial[n] = [zerop[n]→1;T→times[n;fatorial[sub1[n]]]].

DEFINIÇÃO 2.6.1 - Função length

length : L ⟶ N

length [x] dá o comprimento, ou seja, o número de elementos, da lista x e é definida por:

length[x] = [null[x]→0;
T→add1[length[cdr[x]]]].

Exemplos:

length[()] = 0

length[((A B))] = 1

length[((A B) C (D))] = 3

DEFINIÇÃO 2.6.2 - Função last

last : L' ⟶ S

last[x] dá o último elemento da lista x e é definida por:

last[x] = [null[cdr[x]]→car[x];
T→last[cdr[x]]]

Exemplos:

last[(A B C D)] = D

last[(A ((B)))] = ((B))

DEFINIÇÃO 2.6.3 - Função equal

equal : S x S ⟶ {T, NIL}

equal[x;y] verifica a condição de igualdade entre duas S-expressões x e y e é definida por:

```
equal[x;y] = [and[atom[x];atom[y]]→eq[x;y];
             or[atom[x];atom[y]]→NIL;
             equal[car[x];car[y]]→equal[cdr[x];cdr[y]];
             T→NIL].
```

OBSERVAÇÃO: as funções and e or serão definidas na seção seguinte.

Exemplos:

```
equal[(A B) ; (A C)] = NIL
equal[(A (B)) ; (A (B))] = T
```

DEFINIÇÃO 2.6.4 - Função member

member : S x L ⟶ {T, NIL}

member[x;y] verifica se x é um elemento da lista y e é definida por:

```
member[x;y] = [null[y]→NIL;
               equal[x;car[y]]→T;
               T→member[x;cdr[y]]].
```

Exemplos:

```
member[B ; ( A (B) C D )] = NIL
member[(A (B) ) ; ( C D (A (B) ) E )] = T
```

DEFINIÇÃO 2.6.5 - Função append

append : L x L ⟶ L

append[x;y] faz a concatenação de duas listas, e é definida por

```
append[x;y] = [null[x]→y;
               T→cons[car[x];append[cdr[x];y]]].
```

Exemplos:

```
append[(A (B) C) ; (D E)] = (A (B) C D E)
append[(A B C) ; NIL] = (A B C)
```

DEFINIÇÃO 2.6.6 - Função pair

pair: L x L ⟶ L

Se $x = (u_1\ u_2\ \ldots\ u_n)$ e $y = (v_1\ v_2\ \ldots\ v_n)$, $u_i, v_i \in S$, $i=1,2,\ldots,n$, então pair[x;y] = $((u_1 . v_1) \ldots (u_n . v_n))$.

A função pair é definida recursivamente por:

```
pair[x;y] = [null[x]→NIL;
             T→cons[cons[car[x]; car[y]];
                    pair[cdr[x];cdr[y]]]].
```

Exemplos:

```
pair[(X Y Z) ; (A B C)] = ( (X . A) (Y . B) (Z . C))
pair[(ONE TWO) ; (1 2)] = ( (ONE . 1) (TWO . 2) )
pair [(X Y) ; (A)] = ((X. A)(Y))
```

2.7 - Funções lógicas

Sejam $\rho_1, \rho_2, \ldots, \rho_n$ <forma>.

a) Função not

not[x] = null[x], $x \in S$

b) Função and

$$\text{and}[\rho_1;\rho_2;\ldots;\rho_n] = [\text{null}[\rho_1]\to\text{NIL};$$
$$\text{null}[\rho_2]\to\text{NIL};$$
$$\vdots$$
$$\text{null}[\rho_n]\to\text{NIL};$$
$$T\to T].$$

c) Função or

$$\text{or}[\rho_1;\rho_2;\ldots;\rho_n] = [\rho_1 \to T;$$
$$\rho_2 \to T;$$
$$\vdots$$
$$\rho_n \to T;$$
$$T \to \text{NIL}].$$

3. TRANSFORMAÇÃO DE m-EXPRESSÕES PARA S-EXPRESSÕES

A linguagem externa LISP está na forma de representações de S-expressões. Vamos definir uma meta-função <u>tran</u> que transforma uma <m-expressão> ψ numa <S-expressão> que indicaremos por <u>tran</u>$\{\psi\}$.

DEFINIÇÃO 3.1

<u>tran</u> $\{\psi\}$ =

- ψ -se ψ é T, NIL ou <átomo numérico>
- (QUOTE ψ) -se ψ é <átomo não numérico> diferente de T e NIL
- <u>maiúscula</u> $\{\psi\}$ -se ψ é <identificador>
- (<u>tran</u> $\{\phi\}$ <u>tranl</u> $\{\alpha_1\}$ ... <u>tranl</u> $\{\alpha_n\}$) -se ψ é a <forma> $\phi[\alpha_1;\ldots;\alpha_n]$, onde ϕ é <função> e $\alpha_1,\ldots,\alpha_n$ são <argumento>
- (COND (<u>tran</u> $\{\pi_1\}$ <u>tran</u> $\{\varepsilon_1\}$)...(<u>tran</u> $\{\pi_n\}$ <u>tran</u> $\{\varepsilon_n\}$)) -se ψ é a <forma>$[\pi_1\to\varepsilon_1;\ldots;\pi_n\to\varepsilon_n]$, onde $\pi_1,\ldots,\pi_n,\varepsilon_1,\ldots,\varepsilon_n$ são <forma>
- (LAMBDA (<u>tran</u> $\{\xi_1\}$ ... <u>tran</u> $\{\xi_n\}$) <u>tran</u>$\{\rho\}$) -se ψ é a <função> $\lambda[[\xi_1;\ldots;\xi_n];\rho]$, onde $\xi_1,\ldots,\xi_n$ são <variável> e ρ é <forma>
- (LABEL <u>tran</u> $\{\xi\}$ <u>tran</u> $\{\phi\}$) -se ψ é a <função> label$[\xi;\phi]$, onde ξ é <identificador> e ϕ é <função>

DEFINIÇÃO 3.2

A meta-função <u>maiúscula</u> associa a cada <identificador> uma <S-expressão>. A cadeia de letras minúsculas e algarismos do <identificador> é transformada em uma outra cadeia, semelhante

ã original, onde as letras minúsculas são substituídas por letras maiúsculas.

DEFINIÇÃO 3.3

Supondo que α é um <argumento>, a meta-função tranl é definida por:

$$\text{tranl}\ \{\alpha\} = \begin{cases} \text{(FUNCTION } \underline{\text{tran}}\ \{\alpha\}) & \text{se } \alpha \text{ é <função>} \\ \underline{\text{tran}}\ \{\alpha\} & \text{em caso contrário.} \end{cases}$$

Exemplos:

<m-expressão> ψ	<S-expressão> tran {ψ}
1) NIL	NIL
2) 3417	3417
3) A	(QUOTE A)
4) (A . B)	(QUOTE (A . B))
5) car	CAR
6) lisplponto5	LISP1PONTO5
7) car[x]	(CAR X)
8) eq[car[x];A]	(EQ (CAR X) (QUOTE A))
9) [null[x]→x;T→car[x]]	(COND ((NULL X) X)(T (CAR X)))
10) λ[[x;y];cons[y;x]]	(LAMBDA (X Y)(CONS Y X))
11) label[ff;λ[[x]; [atom[x]→x;T→ff[car[x]]]]]	(LABEL FF (LAMBDA (X) (COND ((ATOM X) X) (T (FF (CAR X))))))
12) λ[[x;y];y[x]][(A B);car]	((LAMBDA (X Y) (Y X)) (QUOTE (A B)) (FUNCTION CAR))

OBSERVAÇÃO: Ao encerrar o Capítulo I, queremos lembrar que no sistema LISP existem pseudo-funções (como CSET, DEFINE, etc.) que serão abordadas no Capítulo II.

- □ -

CAPÍTULO II

DESCRIÇÃO DO SISTEMA LISP

1. SISTEMA LISP

Chamaremos de *sistema LISP* um conjunto de algoritmos e estruturas de informações, usados para calcular valores de funções LISP. Os algoritmos que fazem a avaliação de funções constituem o *interpretador*.

Um sistema LISP pode ser implementado num computador, recebendo como entradas uma função LISP e seus argumentos, convenientemente representados, e dando como saída o valor da função. Neste capítulo faremos uma descrição de um sistema LISP sem nos restringirmos a nenhuma implementação em particular.

2. ESPAÇO LIVRE E SUA ORGANIZAÇÃO

2.1 - Célula

DEFINIÇÃO

Vamos definir *célula* como um elemento, identificável univocamente por uma referência, que é ou vazio (célula

vazia), ou um par ordenado de itens, que denominaremos de 1º membro e 2º membro do par ordenado, cada um dos quais contendo ou uma quantidade fixa de informação (numérica ou alfanumérica), ou uma referência a uma célula (eventualmente à mesma ou à célula vazia).

A célula é a constituinte básica para a construção de estruturas de lista, correspondentes a representações internas de S-expressões dentro do sistema LISP.

Representação

A célula vazia será representada por nil. Uma célula não vazia será representada por um retângulo dividido em dois compartimentos [|]. O compartimento da esquerda corresponde ao 1º membro do par ordenado e o da direita corresponde ao 2º membro. Se um membro contém uma referência a uma célula, então o compartimento correspondente contém uma seta apontando para a representação da célula. Uma seta que aponta para nil pode ser substituída por uma linha diagonal ligando dois cantos do compartimento.

Exemplo:

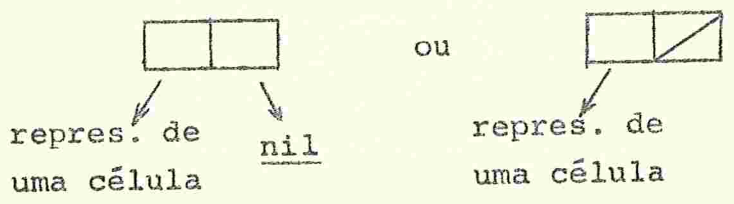


Fig. 2.1

2.2 - Espaço livre

O sistema LISP dispõe de um certo número de células, as quais serão usadas para construção de estruturas de lista. No início do funcionamento do sistema, todas as células são disponíveis e são organizadas num espaço chamado *espaço livre*. Células são liberadas ou cedidas pelo espaço livre, à medida que elas sejam requeridas durante o funcionamento do sistema LISP. Extrações sucessivas de células do espaço livre podem eventualmente esgotar o mesmo. Neste caso é executado um algoritmo de "garbage collection". Faremos neste trabalho uma tradução tentativa dos termos "garbage collector" e "garbage collection" para *recuperador de células inativas* e *recuperação de células inativas*, respectivamente. O problema de recuperação de células inativas será abordado na seção 7 deste capítulo.

2.3 - Organização do espaço livre

Veremos nesta seção dois esquemas que podem ser usados para organizar o espaço livre.

1º esquema:

Podemos organizar todas as células do espaço livre numa *lista linear ligada* [Knu68] que é definida como uma sequência de n células c_1, c_2, ..., c_n, tais que:

a) c_1 é o início da lista linear ligada,

b) para $1 \leq i < n$, o 2º membro de c_i contém a referência de c_{i+1},

c) o 2º membro de c_n contém a referência à célula vazia.

Fig. 2.2

2º esquema:

Suponhamos que as referências de todas as células do sistema LISP, ou transformações das mesmas, constituem uma enumeração, de modo que cada célula pode ser identificada por um número de ordem. Então podemos manter uma variável delimitadora cujo conteúdo é um particular número de ordem n, tal que uma célula pertence ao espaço livre se e somente se o seu número de ordem é superior a n.

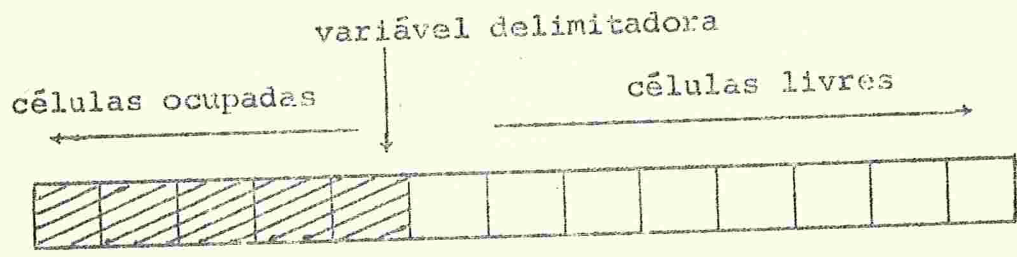


Fig. 2.3

3. REPRESENTAÇÕES INTERNAS

3.1 - Representação interna de átomos não numéricos

Cada átomo não numérico é representado internamente dentro do sistema LISP por meio de uma lista chamada lista de propriedade [Mcc62, Rib69], ou p-lista, que dá as propriedades do átomo. Para indicar quais as propriedades que um símbolo atômico possui, usam-se dentro da p-lista os seguintes indicadores de propriedade: PNAME, SUBR,APVAL,EXPR, FSUBR e FEXPR. A cada indicador de propriedade é associada uma indicação da propriedade. Daremos a seguir uma maneira de representar p-listas. Esta representação funciona para o interpretador descrito neste capítulo. Outros tipos de interpretadores podem adotar outras representações de p-listas.

Representação de uma p-lista

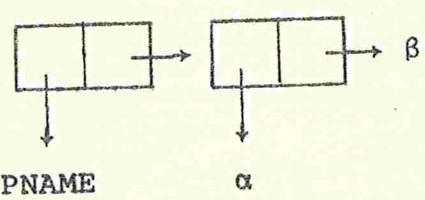


Fig. 2.4

α é uma lista linear ligada de uma ou mais células cujos primeiros membros apontam para células contendo infor

mações alfanuméricas. Estas informações, quando concatenadas, constituem o nome de impressão do símbolo atômico.

Se o átomo representado não possui outro indicador de propriedade a não ser PNAME, então β é nil.

Exemplos:

Supondo que uma célula pode conter no máximo 5 caracteres, a p-lista do símbolo atômico ITU é:

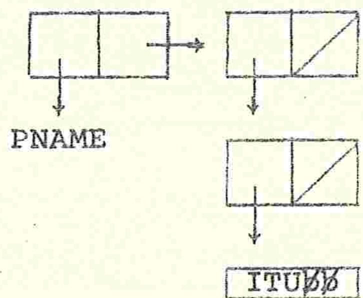


Fig. 2.5

A p-lista do símbolo atômico CAMANDUCAIA é:

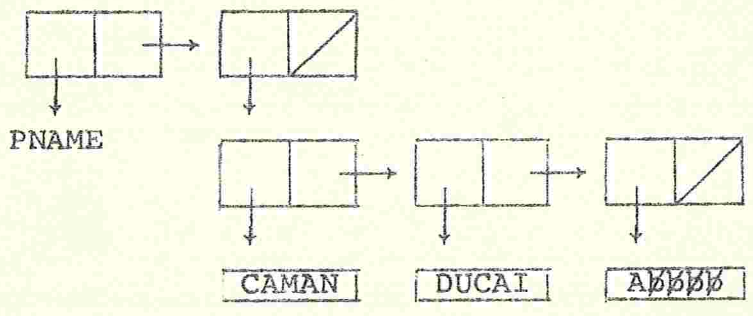


Fig. 2.6

Se o átomo representado possui outra propriedade, então β pode ser um dos seguintes tipos:

a)

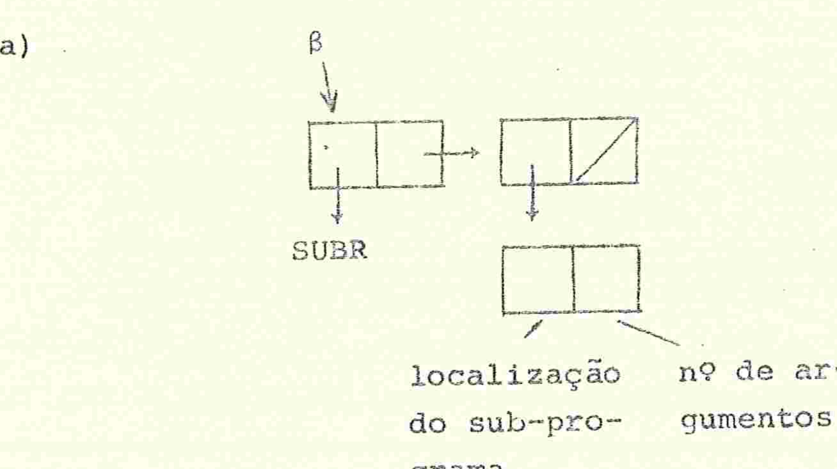


Fig. 2.7

O indicador SUBR indica que o símbolo atômico representado é uma função cuja definição se exprime sob a forma de um sub-programa. A indicação da propriedade associada a SUBR fornece a localização do sub-programa e o número de argumentos da função. CAR, ATOM, EQ, DIFFERENCE são exemplos de átomos do tipo SUBR.

Exemplo:

A p-lista de CAR é:

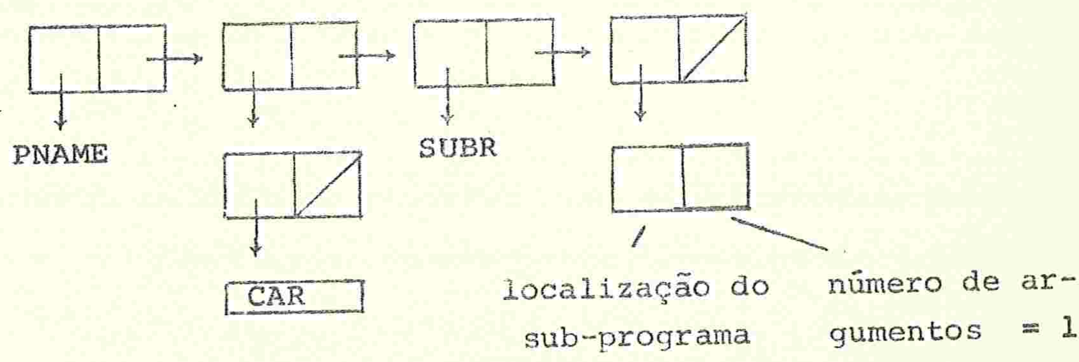


Fig. 2.8

b)

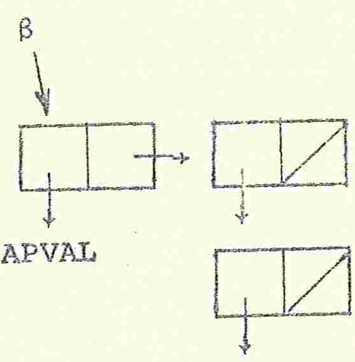


Fig. 2.9 .

O indicador APVAL indica que o símbolo atômico representado possui um valor permanente. A indicação da propriedade associada a APVAL é a representação de uma S-expressão que é esse valor permanente.

Exemplos:

DOLLAR e NIL são exemplos de átomos do tipo APVAL. A p-lista de DOLLAR é

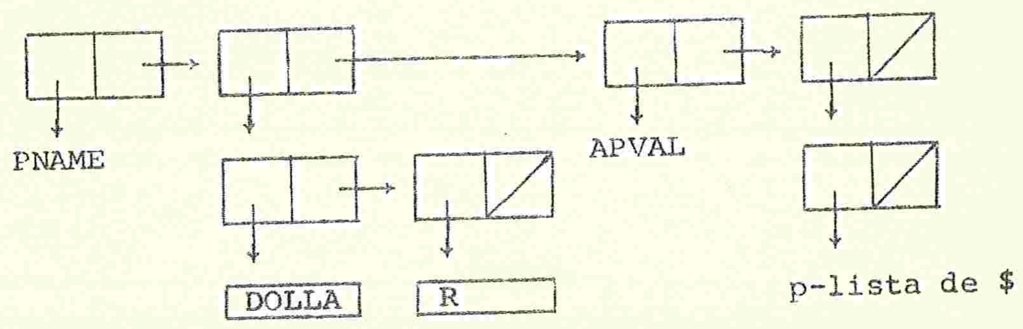


Fig. 2.10

A p-lista de NIL é:

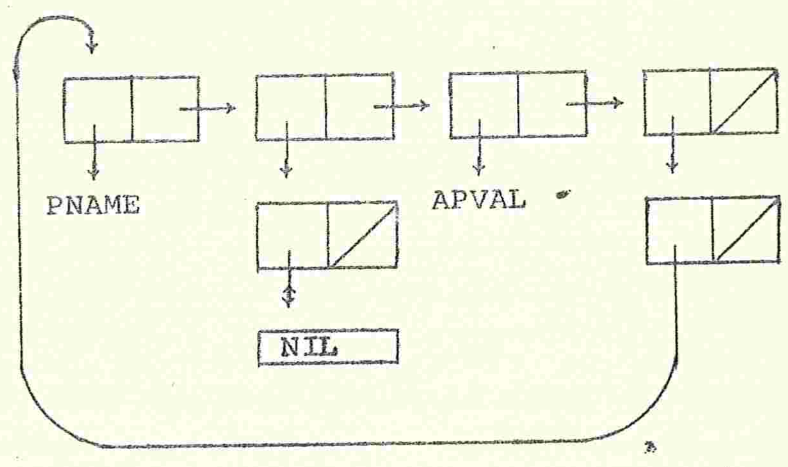


Fig. 2.11

Listamos abaixo os átomos usuais do tipo APVAL:

ÁTOMO	VALOR
BLANK	ƀ
COMMA	,
DASH	-
DOLLAR	$
EQSIGN	=
F	NIL
LPAR	(
NIL	NIL
PERIOD	.
PLUSS	+
RPAR	)
SLASH	/
STAR	*
T	T

c)

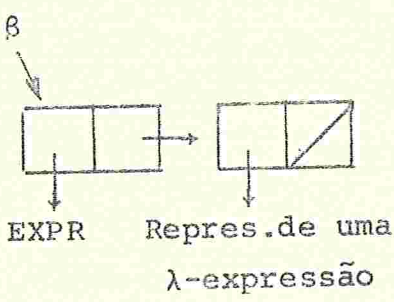


Fig. 2.12

O indicador EXPR indica que o símbolo atômico representado é uma função expressa sob a forma de uma λ-expressão.

Exemplo:

O símbolo atômico FATORIAL, após uma definição conveniente (ver seção 4.5 deste capítulo), possui a p-lista seguinte:

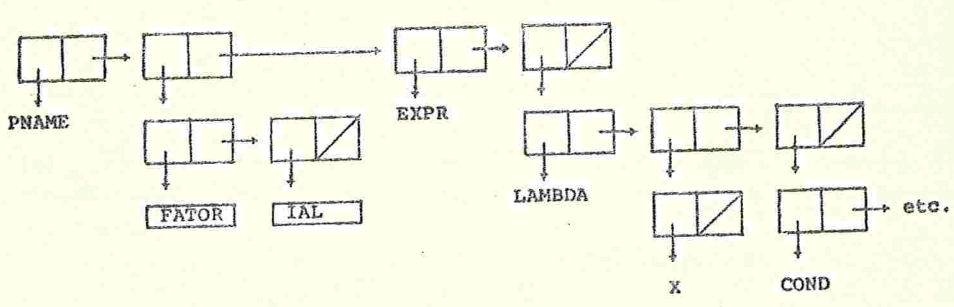


Fig. 2.13

d)

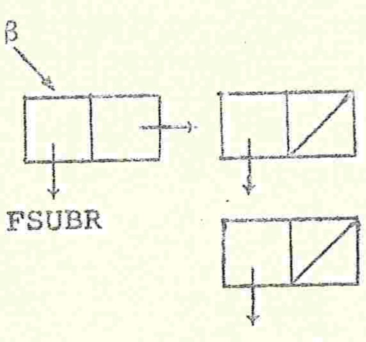


Fig. 2.14

O indicador FSUBR indica que o símbolo atômico representado é uma função cuja definição se exprime sob a forma de um sub-programa. A distinção entre os tipos FSUBR e SUBR é que nas funções do tipo FSUBR o número de argumentos pode ser qualquer (como PLUS, MAX, AND, OR, etc.), ou o argumento não deve ser avaliado (como QUOTE, CSETQ, etc.).

e)

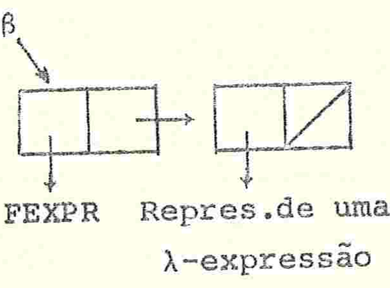


Fig. 2.15

O indicador FEXPR indica que o símbolo atômico representado é uma função e a indicação correspondente é a λ-expressão que define esta função. Uma função do tipo FEXPR

difere de uma do tipo EXPR no número indeterminado de argumentos ou na não avaliação de argumentos. A λ-expressão que define a função do tipo FEXPR possui exatamente duas λ-variáveis: a 1ª é uma lista dos argumentos da função e a 2ª é a lista de associação (a ser vista na seção 5.3 deste capítulo).

OBSERVAÇÕES:

a) Pelo exposto acima, verifica-se que a indicação da propriedade se encontra sempre no 1º membro da célula seguinte àquela que contém o indicador de propriedade:

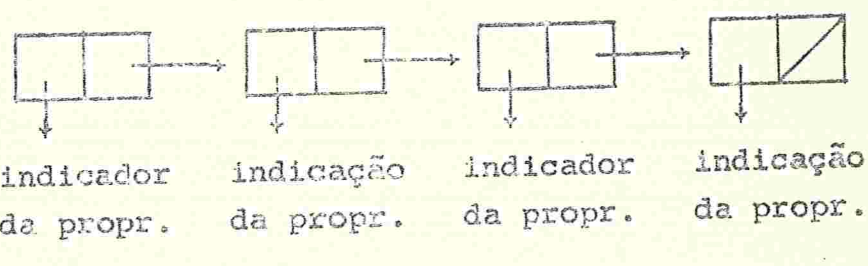


Fig. 2.16

b) A 1ª célula de uma p-lista deve ter uma marca qualquer que a distingue de uma lista comum.

c) No Apêndice I encontra-se uma lista com os nomes das funções LISP e seus tipos, assim como o número de argumentos.

3.2 - Lista dos símbolos atômicos

A lista dos símbolos atômicos é uma lista linear

ligada contendo as p-listas dos átomos existentes no sistema LISP. Na leitura de uma S-expressão, cada átomo lido é comparado com os símbolos atômicos já existentes na lista dos símbolos atômicos. Se o símbolo atômico lido não é encontrado no meio desses, será criada uma lista de propriedade para esse átomo, e a lista dos símbolos atômicos é devidamente atualizada, incorporando o novo átomo.

3.3 - Função para busca de propriedades:função get

A função get possui dois argumentos: uma lista x e um símbolo atômico y. Sua definição é:

```
get[x;y] = [null[x]→NIL;
            eq[car[x];y]→cadr[x];
            T→get[cdr[x];y]]
```

Esquematicamente, se x é a lista:

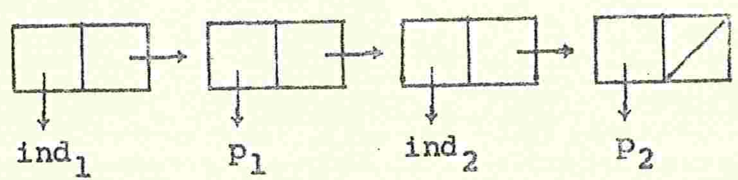


Fig. 2.17

então

$get[x;ind_1] = p_1$

$get[x;ind_2] = p_2$

$get[x;ind_3] = NIL$

A função get é utilizada para obter a indicação de propriedade que é associada a um indicador y sobre a p-lista de um símbolo atômico x. Se o indicador procurado está ausente, o valor da função get é NIL.

3.4 - Representação interna de S-expressões

a) Quando a S-expressão é um átomo não numérico, a representação interna é a lista de propriedade do mesmo.

b) Quando a S-expressão é um átomo numérico, a representação pode depender de cada implementação em particular, por exemplo:

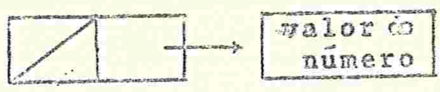


Fig. 2.18

c) Quando a S-expressão é uma lista

$$(u_1\ u_2\ \dots\ u_n)$$

a representação interna é:

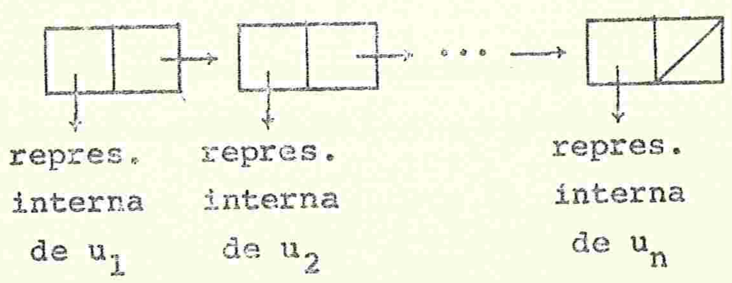


Fig. 2.19

Exemplos:

1) ALEXANDRE

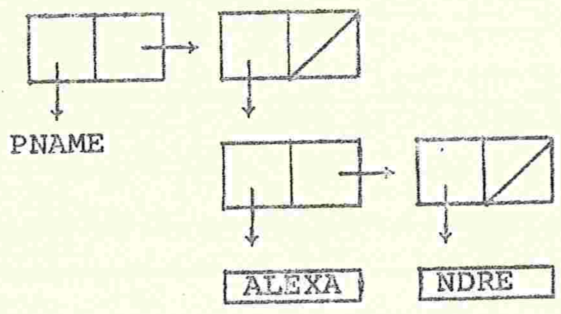


Fig. 2.20

2) (CONS X Y)

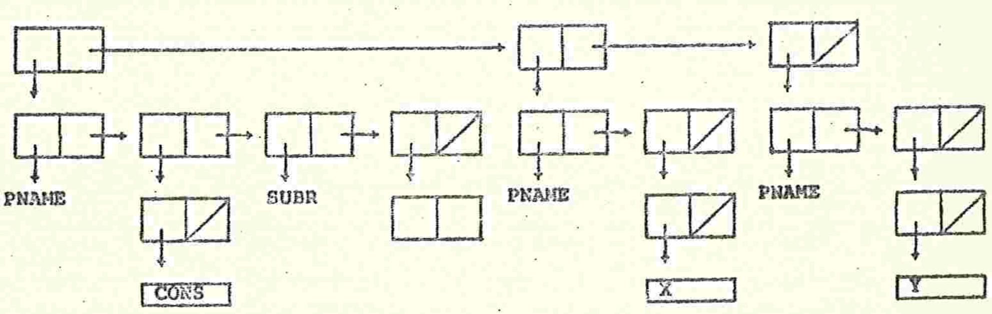


Fig. 2.21

3) (LAMBDA (X) (CAR X))

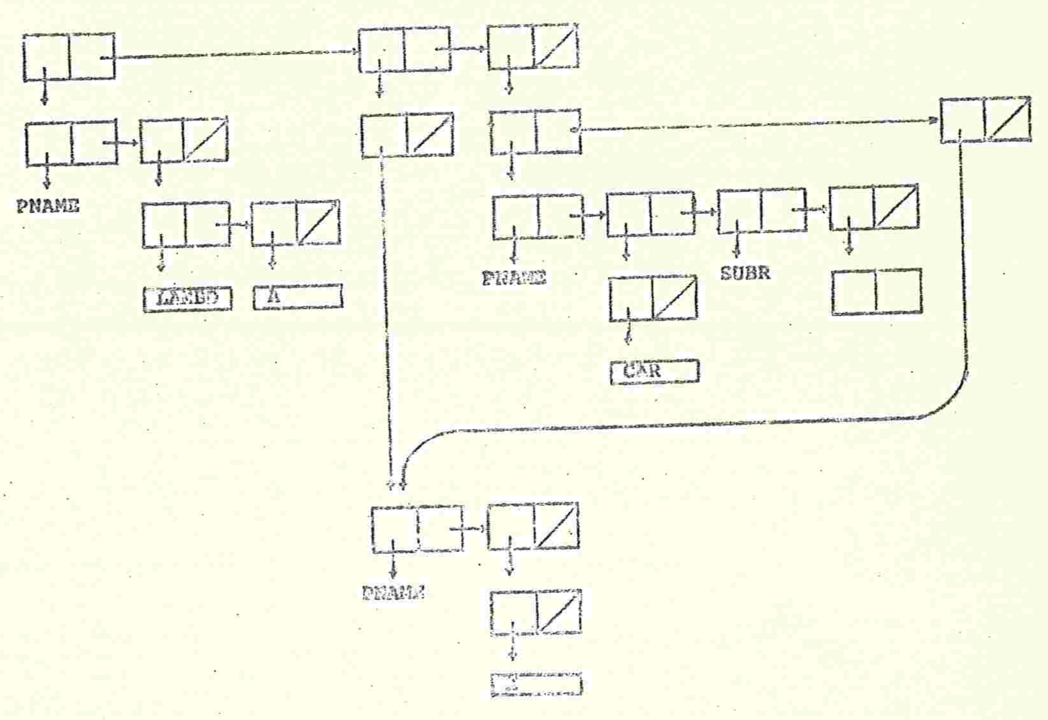


Fig. 2.22

4 - PSEUDO-FUNÇÕES

Uma pseudo-função é uma função LISP que além de possuir um valor, também apresenta outros efeitos colaterais como por exemplo a modificação da estrutura interna dos seus argumentos. Na maioria das pseudo-funções, como DEFINE, CSET, etc., interessa-nos mais o efeito que propriamente o valor da pseudo-função.

4.1 - Pseudo-funções RPLACA e RPLACD

A pseudo-função RPLACA possui dois argumentos: uma S-expressão não atômica x e uma S-expressão y. O seu valor é o argumento x cujo car é trocado por y. A pseudo-função RPLACD é análoga à RPLACA, com a diferença de que, em vez do car, o cdr de x é trocado por y.

4.2 - Pseudo-função NCONC

Possui dois argumentos que são listas. Tem por valor a primeira lista ao fim da qual se junta a segunda lista. A primeira lista original é portanto perdida.

4.3 - Pseudo-funções CSET e CSETQ

A pseudo-função CSET possui dois argumentos: o primeiro argumento tem valor atômico e o segundo uma S-expressão. CSET atribui ao símbolo atômico a S-expressão como seu valor permanente. O valor de CSET é a S-expressão.

A pseudo-função CSETQ possui dois argumentos: um símbolo atômico e uma S-expressão. O seu funcionamento é análogo ao da pseudo-função CSET, com a exceção de que o primeiro argumento não é avaliado.

Tanto na pseudo-função CSET como na CSETQ, a p-lista do símbolo atômico que é o primeiro argumento é modifica

da, acrescentando-se nela um indicador de propriedade APVAL. A indicação da propriedade correspondente é a S-expressão que é o valor do segundo argumento.

Exemplo:

A avaliação da forma

(CSET (QUOTE X) (QUOTE A))

ou da forma

(CSETQ X (QUOTE A))

tem o seguinte efeito na p-lista do átomo X:

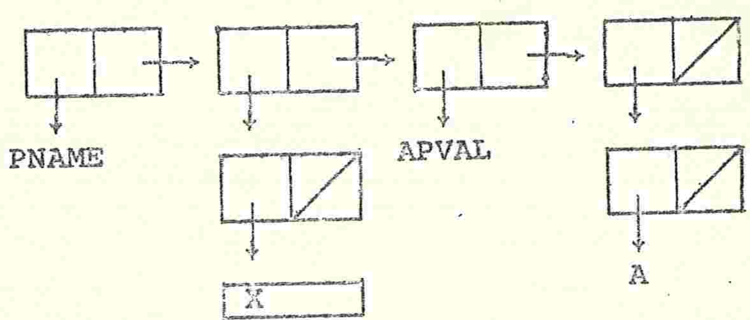


Fig. 2.23

4.4 - Pseudo-funções READ e PRINT

READ é uma pseudo-função sem argumentos com um efeito colateral de causar a leitura de uma S-expressão através de um dispositivo de entrada. O valor de READ é a própria S-expressão lida.

PRINT é uma pseudo-função de um argumento que é u-ma S-expressão. O valor de PRINT é a própria S-expressão e o efeito colateral de imprimir esta S-expressão em um dispositivo de saída.

4.5 - Pseudo-função DEFINE

DEFINE é uma pseudo-função de um argumento que é uma lista da forma:

$$((u_1\ v_1)\ (u_2\ v_2)\ \ldots\ (u_n\ v_n)),$$

onde cada um dos u_i é o nome de uma função cuja definição é v_i. O valor de DEFINE é uma lista contendo os nomes u_i:

$$(u_1\ u_2\ \ldots\ u_n).$$

O efeito desta pseudo-função é a definição dos nomes $u_1, u_2, \ldots, u_n$ como sendo as funções $v_1, v_2, \ldots, v_n$, respectivamente, de modo que estes nomes podem ser usados como se eles fizessem parte da linguagem externa.

A maneira de produzir esse efeito é a introdução, para cada par $(u_i\ v_i)$, do indicador de propriedade EXPR sobre a p-lista de u_i, e a indicação de propriedade é v_i.

Exemplo:

```
DEFINE (( (FF (LAMBDA (X) (COND
                  ((ATOM X) X)
                  (T (FF (CAR X)) )  ))) ))
```

tem o seguinte efeito na p-lista de FF:

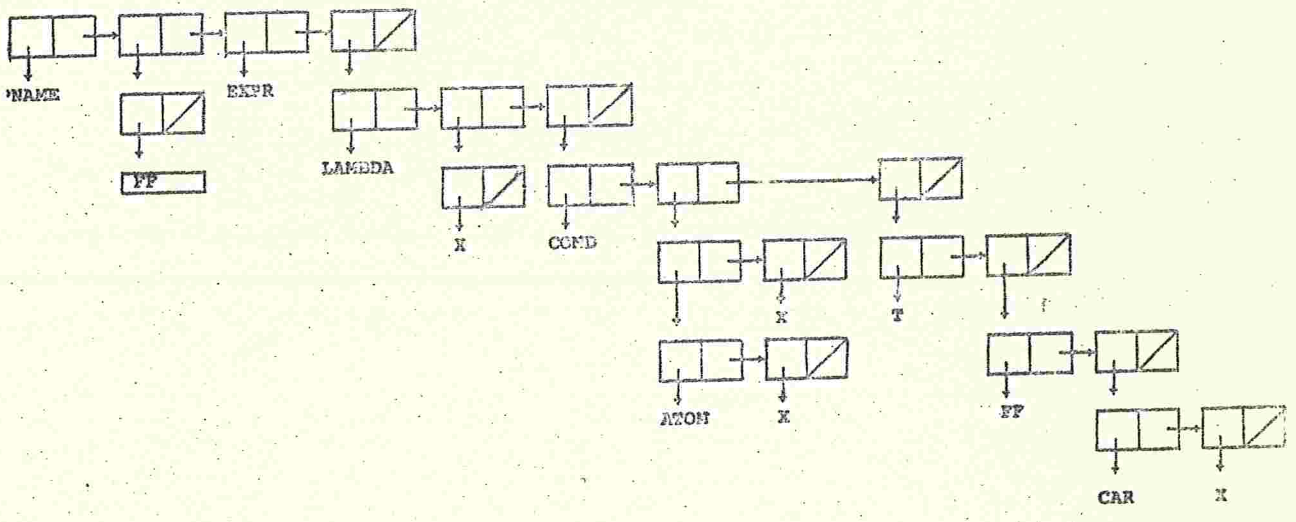


Fig. 2.24

5 - INTERPRETADOR

5.1 - Função universal evalquote

O interpretador é constituído essencialmente de três funções: evalquote, apply e eval, que são auxiliadas por algumas outras funções, como evlis, evcon, etc. A função evalquote é a única dentre estas funções que não faz parte da linguagem externa.

Sejam ϕ <função>,

$\varepsilon, \pi_1,\ldots,\pi_j, \varepsilon_1,\ldots, \varepsilon_j$ <forma>,

$\rho_1,\ldots,\rho_k$ <forma>,

$\xi_1,\ldots,\xi_n$ <constante>,

$\alpha_1,\ldots,\alpha_m, \beta_1,\ldots,\beta_m$ <constante>.

Vamos denotar

$\underline{tran}\ \{\phi\}$ por f,

$\underline{tran}\ \{\varepsilon\}$ por e,

$(\ \xi_1\ \ldots\ \xi_n\ )$ por x,

$(\underline{tran}\ \{\rho_1\}\ \ldots\ \underline{tran}\ \{\rho_k\})$ por r,

$((\underline{tran}\{\pi_1\}\underline{tran}\{\varepsilon_1\})\ldots(\underline{tran}\{\pi_j\}\underline{tran}\{\varepsilon_j\}))$ por s,

$(\ (\ \alpha_1\ .\ \beta_1\ )\ \ldots\ (\ \alpha_m\ .\ \beta_m)\ )$ por a,

onde tran é a meta-função definida na seção 3 do Capítulo I.

A função evalquote é uma função dita função universal tal que:

evalquote[f;x]

e $\underline{val}\{\phi[\xi_1;\ldots;\xi_n]\}$

são iguais quando definidos.

Exemplos:

val{car[(A B)]} = evalquote[CAR;((A B))] = A

val{λ[[x;y];cons[y;x]][NIL;A]}

= evalquote[(LAMBDA (X Y)(CONS Y X));(NIL A)] = (A)

OBSERVAÇÃO

evalquote[f;x] é definido também nos casos em que f contêm funções que produzem efeitos colaterais (pseudo-

funções) ou átomos do tipo APVAL, casos estes para os quais <u>val</u> não é definido.

<u>Exemplos</u>:

```
evalquote[(LAMBDA () DOLLAR) ; ()] = $
evalquote[(LAMBDA () (CSETQ PI 3.14159) ) ; ()] = 3.14159
```

efeito colateral: atribuição do átomo numérico 3.14159 ao átomo PI como seu valor permanente (isto é, modificação da p-lista de PI).

<u>DEFINIÇÃO</u>:

A função evalquote é definida por:

```
evalquote[f;x] =
    [or[get[f;FSUBR];get[f;FEXPR]]→
            eval[cons[f;evalq[x]];NIL];
                T→apply[f;x;NIL]]
```

onde

```
evalq[x] = [null[x]→NIL;
           T→cons[list[QUOTE;car[x]];evalq[cdr[x]]]]

apply[f;x;a] =
 [null[f]→NIL;
  atom[f]→[get[f;EXPR]→apply[get[f;EXPR];x;a];
           get[f;SUBR]→chamada do sub-programa de endereço
                   car[get[f;SUBR]],levando-se os  ele-
                   mentos da lista x como dados.
           T→apply[cdr[sassoc[f;a;(LAMBDA () (QUOTE
                   (UNDEFINED FUNCTION)))]];x;a]];
```

```
eq[car[f];LAMBDA]→eval[caddr[f];
          nconc[pair[cadr[f];x];a]];
eq[car[f];LABEL]→apply[caddr[f];x;
          cons[cons[cadr[f];caddr[f]];a]];
eq[car[f];FUNARG]→apply[cadr[f];x;caddr[f]];
T→apply[eval[f;a];x;a]]
eval[e;a] =
  [null[e]→NIL;
   numberp[e]→e;
   atom[e]→[get[e;APVAL]→car[get[e;APVAL]];
       T→cdr[sassoc[e;a;(LAMBDA ()
                              (QUOTE (UNDEFINED VAR)))]]];
  eq[car[e];QUOTE]→cadr[e];
  eq[car[e];COND]→evcon[cdr[e];a];
  eq[car[e];FUNCTION]→list[FUNARG;cadr[e];a];
  atom[car[e]]→[get[car[e];EXPR]→
                         apply[car[e];evlis[cdr[e];a];a];
                get[car[e];SUBR]→
                         apply[car[e];evlis[cdr[e];a];a];
                get[car[e];FEXPR]→
                         apply[get[car[e];FEXPR];
                               list[cdr[e];a];a];
                get[car[e];FSUBR]→chamada do sub-programa  de
                               endereço
                               car[get[car[e];FSUBR]], le-
                               vando-se como dados  cdr[e]
                               e a.
                T→eval[cons[cdr[sassoc[car[e];a;(LAMBDA ()
                 (QUOTE (UNDEFINED  FUNCTION)))]];cdr[e]];a]];
  T→apply[car[e];evlis[cdr[e];a];a]]
```

```
evlis[r;a] =
     [null[r]→NIL;
      T→cons[eval[car[r];a];evlis[cdr[r];a]]]

evcon[s;a] =
     [null[s]→error;
      eval[car[s];a]→eval[cadar[s];a];
      T→evcon[cdr[s];a]]

sassoc[c;a;h] =
     [null[a]→apply[h;NIL;NIL];
      eq[caar[a];c]→car[a];
      T→sassoc[c;cdr[a];h]]
```

onde c é uma S-expressão e h uma função constante sem argumentos.

Nas definições acima usamos a função list cuja finalidade é abreviar composições de cons:

$$list[x_1;x_2;\ldots;x_n]$$

$$= cons[x_1;cons[x_2;cons[\ldots;cons[x_n;NIL]\ldots]]]$$

5.2 - Exemplos de interpretação

EXEMPLO 1:

```
 evalquote[(LAMBDA(X)((LAMBDA(X)(TIMES 3 X))(ADD1 X)));(5)]
=apply[(LAMBDA(X)((LAMBDA(X)(TIMES 3 X))(ADD1 X)));(5);NIL]
=eval[((LAMBDA(X)(TIMES 3 X))(ADD1 X));((X.5))]
=apply[(LAMBDA(X)(TIMES 3 X));
                      evlis[((ADD1 X));((X.5))];((X.5))]
=apply[(LAMBDA(X)(TIMES 3 X));
                      cons[eval[(ADD1 X);((X.5))];
                           evlis[NIL;((X.5))]];((X.5))]
=apply[(LAMBDA(X)(TIMES 3 X));
                      cons[apply[ADD1;evlis[(X);((X.5))];((X.5))];
                           NIL];((X.5))]
=apply[(LAMBDA(X)(TIMES 3 X));
              cons[apply[ADD1;(5);((X.5))];NIL];((X.5))]
=apply[(LAMBDA(X)(TIMES 3 X));(6);((X.5))]
=eval[(TIMES 3 X);((X.6)(X.5))]
=apply[TIMES;evlis[(3 X);((X.6)(X.5))];((X.6)(X.5))]
=apply[TIMES;(3 6);((X.6)(X.5))]
= 18
```

EXEMPLO 2:

Por motivo de simplificação, denotemos por g a expressão:

```
(LABEL FF (LAMBDA(X)(COND ((ATOM X) X)(T (FF (CAR X))) )))
```

e por h a expressão:

```
(LAMBDA(X)(COND ((ATOM X) X)(T (FF (CAR X))) )).
```

```
evalquote[g;((A))]
=apply[g;((A));NIL]
=apply[h;((A));((FF.h))]
=eval[(COND ((ATOM X) X)(T (FF (CAR X))) );((X.(A))(FF.h))]
=evcon[( ((ATOM X) X)(T (FF (CAR X))) );((X.(A))(FF.h))]
=evcon[( (T (FF (CAR X))) );((X.(A))(FF.h))]
=eval[(FF (CAR X));((X.(A))(FF.h))]
=eval[( h (CAR X));((X.(A))(FF.h))]
=apply[ h;evlis[((CAR X));((X.(A))(FF.h))];((X.(A))(FF.h))]
=apply[ h; (A) ;((X.(A))(FF.h))]
=eval[(COND ((ATOM X) X)(T (FF (CAR X))) );((X.(A))(FF.h))]
=evcon[(((ATOM X) X)(T (FF(CAR X))) );((X.(A))(FF.h))]
=eval[X;((X.(A))(FF.h))]
= A
```

<u>EXEMPLO 3</u>:

Por motivo de simplificação, denotemos por g a expressão

```
(LAMBDA NIL (ATOM X)).
```

```
evalquote[(LAMBDA(X)((LAMBDA(X)(X))(FUNCTION g)));(A)]
=apply[(LAMBDA(X)((LAMBDA(X)(X))(FUNCTION g)));(A);NIL]
=eval[((LAMBDA(X)(X))(FUNCTION g));((X.A))]
=apply[(LAMBDA(X)(X));evlis[((FUNCTION g));((X.A))];((X.A))]
=apply[(LAMBDA(X)(X));((FUNARG g((X.A))) );((X.A))]
=eval[(X);( (X.(FUNARG g ((X.A)))) (X.A) )]
=eval[((FUNARG g ((X.A))));( (X.(FUNARG g ((X.A)))) (X.A) )]
=apply[(FUNARG g((X.A)));NIL;( (X.(FUNARG g((X.A)))) (X.A) )]
=apply[g;NIL;((X.A))]
=apply[(LAMBDA NIL (ATOM X));NIL;((X.A))]
=eval[(ATOM X);((X.A))]
=apply[ATOM;(A);((X.A))]
= T
```

5.3 - Lista de Associação

A lista de pares com ponto, que foi denominada de *a* na seção 5.1 deste capítulo, é chamada de *lista de associação*. Esta lista tem a finalidade de ligar ou associar valores às variáveis ligadas de λ-expressões, e ligar definições de funções a seus nomes.

Vemos pelas definições apresentadas na seção 5.1 deste capítulo que a lista de associação é inicializada pela função evalquote como uma lista vazia. No decorrer da interpretação, novos pares com ponto são inseridos no início, ou à esquerda, da lista de associação de duas maneiras:

a) Toda vez que a função apply encontra uma λ-expressão como seu primeiro argumento, pares com ponto do tipo

$$(\alpha_i . \beta_i),$$

onde β_i é o valor da variável ligada α_i da λ-expressão, são acrescentados à esquerda da lista de associação. Na avaliação do corpo da λ-expressão, as variáveis que nele ocorrem buscam seus valores na lista de associação, percorrendo-a da esquerda para a direita e terminando a busca na primeira ocorrência de um par onde a referida variável comparece como 1º membro. Esta busca é feita pela função sassoc. O exemplo 1 da seção anterior ilustra um caso de interpretação de λ-expressões.

b) Toda vez que a função apply encontra uma função em notação label como seu primeiro argumento, digamos (LABEL α_j β_j), então um par $(\alpha_j . \beta_j)$ é acrescentado à esquerda da lista de associação. Uma posterior ocorrência de α_j causará uma busca semelhante ao caso (a) acima, pela função sassoc. O exemplo 2 da seção anterior ilustra um caso de interpretação de uma função em notação label.

<u>Considerações sobre o uso de variáveis livres</u>

Chamamos de *variáveis livres* aquelas que ocorrem no corpo de uma λ-expressão e que não fazem parte da lista de variáveis ligadas da mesma λ-expressão. O valor de uma variável livre é definido fora desta λ-expressão e deve ser

disponível quando a mesma é usada. O uso de variáveis livres pode reduzir o tamanho da lista de associação [Wai73]. Consideremos, por exemplo, a função predicado FIND, de argumentos A, B e C, que verifica se há a presença de dois elementos consecutivos iguais a B e C na lista A:

```
DEFINE (( (FIND (LAMBDA (A B C) (FIND2 A) ))
          (FIND2 (LAMBDA (A) (COND
             ( (LESSP (LENGTH A) 2) NIL)
             ( (EQUAL (CAR A) B) (OR (EQUAL (CADR A) C)
                                     (FIND2 (CDDR A))) )
             (T  (FIND2 (CDR A)) )  )))   ))
```

As variáveis ligadas na função FIND são A, B e C. FIND não é recursiva mas chama a função recursiva FIND2 para fazer a busca na lista A. Notemos que B e C não são ligadas em FIND2 mas aparecem *livres*. Isto significa que os valores de B e C, que não se alteram durante a interpretação, são colocados sobre a lista de associação somente uma vez. Se essas variáveis fossem ligadas por FIND2, os seus valores teriam que ser colocados sobre a lista de associação em cada recursão de FIND2. Assim, usando as variáveis B e C como variáveis livres, economiza-se espaço na lista de associação. Por outro lado, enquanto o nível de recursão aumenta, os valores de B e C se distanciam cada vez mais do início da lista de associação, significando que um maior tempo será necessário para achá-los.

5.4 - Argumentos funcionais

Na sintaxe da meta-linguagem apresentada na seção 2.1 do Capítulo I, <argumento> pode ser <forma> ou <função>. Pela definição da meta-função tran, apresentada na seção 3 do Capítulo I, vimos que um <argumento> α_j do tipo <função> é transformado para

(FUNCTION $\underline{tran}\{\alpha_j\}$).

Suponhamos que um dos argumentos $x_1,\ldots,x_n$ de uma função f é uma função g (também chamado argumento funcional), e que durante uma interpretação, temos que avaliar

eval[(f x_1 ... (FUNCTION g) ... x_n) ; a].

A função f deverá ser aplicada aos valores de seus argumentos. Em particular, o valor de (FUNCTION g),pelas definições da seção 5.1 deste capítulo, é

(FUNARG g a).

O átomo FUNARG indica que g é um argumento funcional e que quando esta função for aplicada a seus argumentos, a lista de associação a ser considerada deverá ser a.

A aplicação de f aos valores de seus argumentos,pela função apply, pode alterar a lista de associação de a para a'. Mas no momento da aplicação da função g, realizada pela função apply, esta obterá a sua lista de associação como o terceiro elemento de (FUNARG g a), portanto a.

O exemplo 3 da seção anterior ilustra um caso de interpretação de argumentos funcionais.

Veremos a seguir a título de exemplo, duas funções que usam argumentos funcionais.

a) Função maplist

A função maplist possui dois argumentos: uma lista x e uma função f. O valor da função maplist é uma lista cujos elementos são f[x], f[cdr[x]], f[cddr[x]], etc. A função maplist é definida por:

```
maplist[x;f] = [null[x]→NIL;
                         T→cons[f[x];maplist[cdr[x];f]]]
```

Exemplos:

```
maplist[(A B C) ; length] = (3 2 1)
maplist[(A B A C A X I);λ[[j];member[A;j]]]
        = (T T T T T NIL NIL)
```

b) Função mapcar

A função mapcar possui dois argumentos: uma lista x e uma função f. O valor da função mapcar é uma lista cujos elementos são os resultados da aplicação da função f a cada um dos elementos da lista x. A definição de mapcar é:

```
mapcar[x;f] = [null[x]→NIL;
                         T→cons[f[car[x]];mapcar[cdr[x];f]]]
```

Exemplos:

mapcar[(1 2 3 4) ; add1] = (2 3 4 5)

mapcar[(A B C) ; λ[[j];list[QUOTE;j]]]

= ((QUOTE A) (QUOTE B) (QUOTE C))

6 - AVALIAÇÃO DA FORMA ESPECIAL PROG

Na linguagem externa LISP, a forma que constitui o corpo de uma λ-expressão pode ser uma forma especial chamada forma PROG que tem a seguinte estrutura:

(PROG lista de prog-variáveis sequência de comandos e rótulos)

As prog-variáveis e os rótulos são símbolos atômicos. Os comandos não são símbolos atômicos e podem ser de alguns tipos que passaremos a descrever. As operações que definem o valor da forma PROG são:

a) Seja $(v_1 \ldots v_n)$ a lista de prog-variáveis.

Com cada uma das v_i, $i=1,\ldots,n$, forma-se um par com ponto onde o 1º membro é v_i e o 2º membro é NIL. Os pares assim formados são acrescentados à esquerda da lista de associação, formando a lista de associação atualizada. Seja ela:

$$a = ((v_n \, . \, \mathrm{NIL}) \ldots (v_1 \, . \, \mathrm{NIL}) \underbrace{a_1 \, a_2 \ldots a_r}_{\text{elementos da lista de associação original}})$$

b) A sequência de comandos e rótulos é percorrida para detetar rótulos (símbolos atômicos). Para cada rótulo encontrado, constrói-se um par com ponto onde o 1º membro é este rótulo e o 2º membro é a lista contendo todos os comandos e rótulos que seguem este rótulo, e acrescenta-se o par assim formado à esquerda de uma lista que inicialmente é vazia. Esta lista é denominada de go-lista.

c) Sejam $r, s, p_1, \ldots, p_n$ formas,

$e_1, \ldots, e_n$ formas ou comandos,

v variável.

Os comandos são executados sequencialmente, com os rótulos ignorados. A execução termina quando é executado um comando (RETURN r), ou na ausência deste, quando é executado o último comando. A execução depende do tipo de comando, da seguinte maneira:

c_1) Comando (RETURN r):

Este comando define o valor da forma PROG como sendo eval[r;a].

c_2) Comando (SETQ v r):

Por meio de sassoc, varre-se a lista de associação a para obter o primeiro par com ponto com v como seu 1º membro. Substitui-se o 2º membro do par encontrado por

eval[r;a].

c_3) Comando (SET s r):

A execução é semelhante a (c_2), salvo no fato de que v é obtido como resultado de eval[s;a] que deve ser atômico.

c_4) Comando (GO v):

Por meio de sassoc, varre-se a go-lista para obter o primeiro par com ponto onde o 1º membro é v e transfere-se a execução para os comandos que constam no 2º membro do par encontrado.

c_5) Comando (COND (p_1 e_1) ... (p_n e_n)):

A execução deste comando é idêntica à da forma condicional, com a diferença de que se todas as formas p_i tiverem valor NIL, a execução simplesmente continua para o comando seguinte, em vez de produzir um valor indefinido.

c_6) Se o comando é outra forma qualquer r não enquadrada acima, a execução consiste em efetuar eval[r;a].

d) A forma especial PROG obtém seu valor com o comando (RETURN r) ou, na ausência deste, quando é executado o último comando. Neste caso, o valor de PROG será NIL.

É preciso dizer ainda que na execução dos comandos, condições de erro podem surgir quando, por exemplo, o comando (GO v) refere-se a um rótulo v inexistente, ou (SETQ v r) usa um símbolo v que não consta na lista de associação, etc.

Exemplo do uso da forma PROG

Vamos definir a função fatorial de N pelo diagrama:

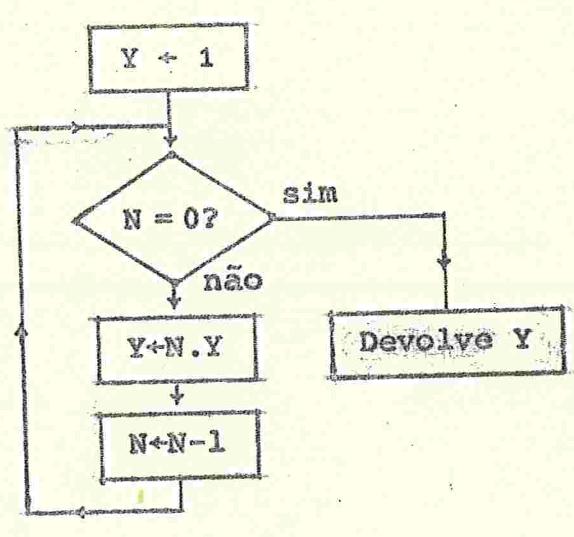


Fig. 2.25

```
DEFINE (( (FATORIAL (LAMBDA (N) (PROG (Y)
                (SETQ Y 1)
        COMP    (COND ((ZEROP N)(RETURN Y)) )
                (SETQ Y (TIMES N Y) )
                (SETQ N (SUB1 N) )
                (GO COMP)  )))  ))
```

7 - RECUPERADOR DE CÉLULAS INATIVAS

Quando o espaço livre se esgota, é ativado o recuperador de células inativas para devolver ao espaço livre as células de estruturas de lista inúteis ou inativas. A quantidade dessas estruturas pode ser significantemente gran

de, pelas seguintes considerações:

a) Na interpretação de cada λ-expressão, são adicionadas à lista de associação (portanto retiradas do espaço livre) 2n células, sendo n o número de λ-variáveis. Estas células são acrescentadas à lista de associação como ilustra a figura abaixo:

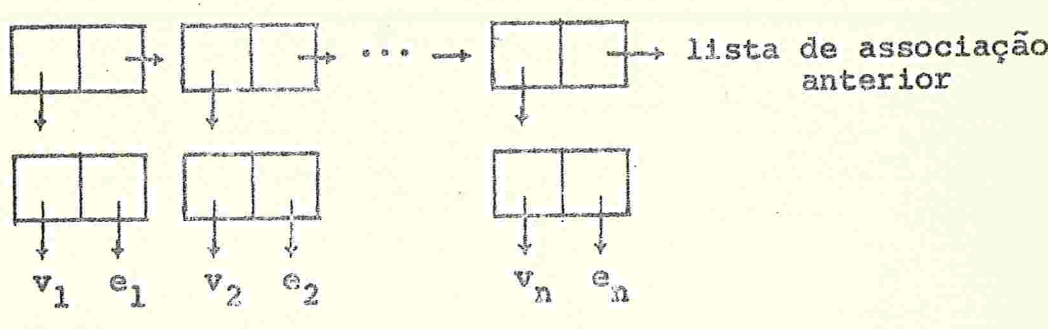


Fig. 2.26

onde v_i são as λ-variáveis e e_i seus respectivos valores. Quando a interpretação da λ-expressão terminar, essas células são simplesmente desprezadas, pois não haverá nenhum apontador que permita ao sistema atingí-las. Processo semelhante ocorre com as prog-variáveis da forma PROG.

b) Sempre que o valor de uma variável for alterado (por CSETQ, CSET, SET, SETQ, etc.), a estrutura interna que corresponde ao valor antigo pode tornar-se inútil, se não houver nenhuma outra variável à qual o mesmo esteja ligado. Isto é particularmente grave em relação aos átomos numéricos, que na maior parte das implementações não são únicos

(isto é, dois átomos numéricos iguais não têm necessariamente o mesmo endereço),pois a execução de cada função aritmética retira novas células do espaço livre.

Todas as células úteis ou ativas podem ser alcançadas por uma cadeia de car's e cdr's a partir de algumas células cujas posições dentro da memória devem ser conhecidas "a priori". Essas células são, por exemplo, a célula inicial da lista de associação, das representações internas do programa que está sendo interpretado, etc. Células que não podem ser alcançadas a partir desses endereços básicos devem ser consideradas inativas e podem ser devolvidas ao espaço livre.

Um recuperador de células inativas é constituído em geral de duas partes essenciais:

a) Uma rotina de marcação, que percorre todas as estruturas que se ramificam desses endereços básicos, deixando em todas as células percorridas um indicador de célula útil ou ativa (por exemplo, ligando um bit determinado da célula).

b) Uma rotina coletora, que percorre sequencialmente a área do espaço livre, coletando as células da seguinte maneira: Se é encontrada uma célula marcada (isto é, célula ativa), então prossegue em frente apagando antes a marca; se é encontrada uma célula não marcada (célula inativa), então devolve a mesma ao espaço livre.

Um problema de especial interesse é o da recuperação de células inativas em sistemas de computação com memória virtual. Segundo alguns estudos feitos [Bae72, Fen69] , num sistema com memória virtual, o problema é mais a *compactação* de dados ativos que a descoberta de espaços livres. Estritamente falando, recuperação de células inativas é desnecessária. À medida que se prosseguem as operações, porém, a eficiência do sistema pode cair. Isso ocorre pois o armazenamento das estruturas ativas pode ficar disseminado numa região cada vez maior de armazenamento virtual, podendo-se chegar talvez a um ponto em que cada referência a esta memória virtual requererá uma referência ao armazenamento secundário. Um recuperador de células inativas deve ser chamado, portanto, para fazer a compactação dos dados ativos.

Não há uma condição simples sob a qual uma recuperação deve ser feita para fazer a compactação. O critério primário para iniciar uma recuperação deve ser uma baixa razão observada entre ciclos do processador e o tempo decorrido. Qualquer estratégia adotada deve ser orientada para o comportamento estatístico do meio combinado apresentado por usuários e pelo sistema.

CAPÍTULO III

IMPLEMENTAÇÃO DO SISTEMA LISP

1. ORGANIZAÇÃO ESQUEMÁTICA DO SISTEMA LISP

Uma implementação de um sistema LISP foi feita por nós no computador B-6700 usando a linguagem ALGOL [Bur72-1, Bur72-2, Bur74]. O sistema tem a seguinte organização:

- p-listas básicas
- Espaço livre
- Recuperador de células inativas
- Entrada e Saída
- Funções pré-definidas do sistema LISP
- Interpretador
- Supervisor

2. ORGANIZAÇÃO DE DADOS

2.1 - Célula

Uma célula é formada por uma palavra do sistema B-6700, de 48 bits, que são numerados de bit 47, bit 46, ..

..,bit 1 e bit 0. Dependendo do seu uso, a célula pode conter os seguintes campos:

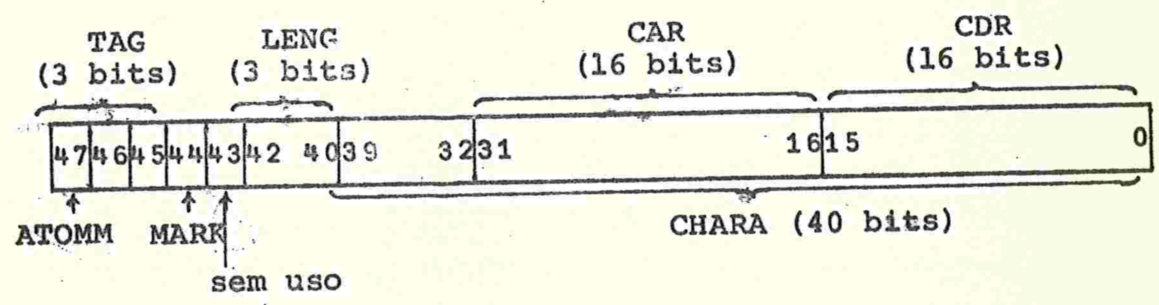


Fig. 3.1

Os significados destes campos são:

a) ATOMM (bit 47):

Normalmente os campos CAR e CDR de uma célula contêm apontadores para outras células. Porém, a célula pode também conter informações alfanuméricas e numéricas.

ATOMM = 1: A célula contém informação alfanumérica ou numérica, CAR e CDR não contêm apontadores.

ATOMM = 0: Caso contrário.

b) TAG (bits 47 a 45):

Indica o tipo de célula conforme a seguinte descrição:

bit 47	bit 46	bit 45	Valor decimal de TAG	SIGNIFICADO
0	0	0	0	célula de lista
0	0	1	1	1ª célula de uma p-lista
0	1	0	2	1ª célula de uma representação de número
0	1	1	3	-----
1	0	0	4	Célula contém as indicações de SUBR
1	0	1	5	Célula contém informações alfanuméricas
1	1	0	6	-----
1	1	1	7	Célula contém as indicações de APVAL

Fig. 3.2

c) MARK (bit 44):

É usado pelo recuperador de células inativas,a ser descrito na seção 7 deste capítulo.

MARK = 1: célula marcada

MARK = 0: célula não marcada

d) LENG (bits 42 a 40):

É usado em células que contêm informações alfanuméricas (TAG = 5). LENG indica o número de caracteres (no máximo 5) que compõem o dado alfanumérico.

e) CHARA (bits 39 a 0):

É usado em células que contêm informações alfanumé ricas (TAG = 5). Contém o dado alfanumérico (até 5 caracteres EBCDIC).

f) CAR (bits 31 a 16):

Contém um apontador para outra célula.

g) CDR (bits 15 a 0):

Contém um apontador para outra célula.

2.2 - Espaço livre

Organização do espaço livre

O espaço livre é mantido dentro de uma matriz unidimensional:

<u>array</u> cell[0:limit]

Cada elemento dessa matriz tem a forma de uma célula e todas as referências são índices de elementos dessa matriz. Neste capítulo, diremos que uma célula tem referência ou endereço i, ou é apontada por i, quando esta célula é cell[i].

No trecho inicial da matriz, de cell[0] até cell[untouchable], armazenam-se as p-listas básicas, a lista dos símbolos atômicos, as representações dos números inteiros de 0 a 100. Este trecho ocupa aproximadamente 1.800

células. O espaço livre ocupa o restante desta matriz e apresenta o seguinte esquema inicial, onde avail é uma variável que aponta para o início do espaço livre:

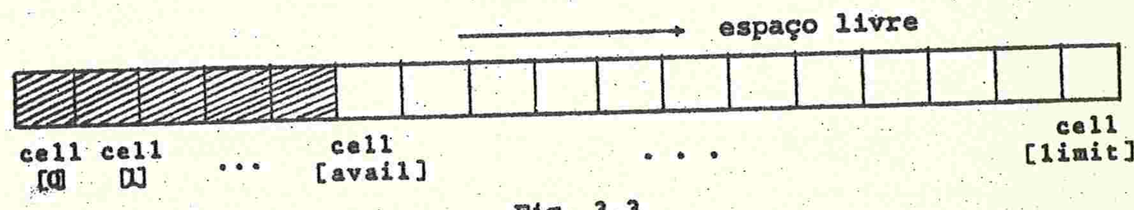


Fig. 3.3

Para facilitar a programação os campos de uma célula de referência i são definidos pelo comando define (do ALGOL B-6700) como segue:

```
define atomm(i) = cell[i].[47:1]#,
       tag(i)   = cell[i].[47:3]#,
       mark(i)  = cell[i].[44:1]#,
       leng(i)  = cell[i].[42:3]#,
       car(i)   = cell[i].[31:16]#,
       cdr(i)   = cell[i].[15:16]#,
       chara(i) = cell[i].[39:40]#
```

Em ALGOL B-6700, a notação x.[y:z] indica que se faz referência a z bits de x começando-se no bit de índice y $(0 \leq y \leq 47)$. A declaração define serve para definição de macro-instruções. Um identificador é associado a um texto colocado entre os sinais = e#. Identificadores colocados entre parênteses são considerados como parâmetros.

Manipulação do espaço livre

Para extrair uma célula do espaço livre, usamos

exfree que é definido por

```
define exfree = begin
                  if avail > limit
                     then garbage;
                  free: = avail;
                  avail: = avail+1
                  end#
```

A célula liberada fica apontada pela variável free. Caso o espaço livre se esgotar, será chamado o procedimento garbage, a ser visto na seção 7 deste capítulo.

2.3 - Lista dos símbolos atômicos

Todos os átomos que já têm p-listas são organizados numa lista chamada lista dos símbolos atômicos. Essa lista possui 101 elementos que são listas: sejam elas $lista_0$, $lista_1$, $lista_2$, ..., $lista_{100}$. O endereço de cada $lista_i$ fica num lugar fixo que é car de cell[i], como mostra a figura 3.4.

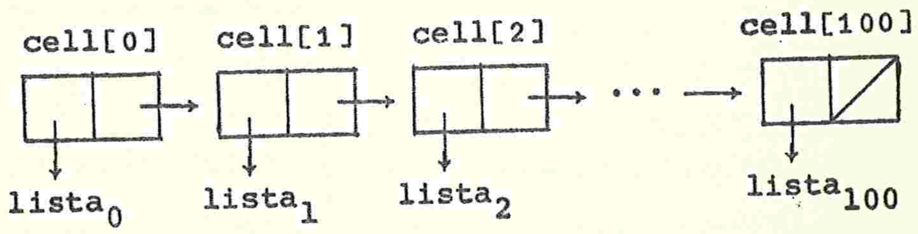


Fig. 3.4

Cada $lista_i$ contém os endereços das p-listas dos á

tomos cujos nomes de impressão gozam da seguinte proprieda-de: resto da divisão do número correspondente à representa-ção binária dos 5 primeiros caracteres (completados com bran-cos caso o nome tenha menos de 5 caracteres) por 101 é i-gual a i.

A razão dessa maneira de organizar a lista dos sím-bolos atômicos é para permitir uma busca sequencial indexa-da, onde o índice é obtido por "hashing".

OBLIST é um átomo do tipo APVAL cujo valor é a lis-ta dos símbolos atômicos.

O procedimento putinlas (i, carac) é usado para in-troduzir na lista dos símbolos atômicos uma nova p-lista,de endereço i e nome de impressão armazenado na matriz carac.

2.4 - Representação de números

Os números são representados por meio de duas célu-las, como mostra a figura abaixo:

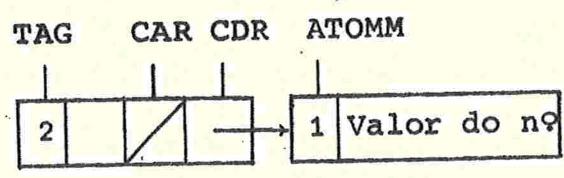


Fig. 3.5

Na maior parte das implementações, os números não são únicos, isto é, dois números iguais não necessariamente têm o mesmo endereço. Uma representação do número é criada

cada vez que ocorre o aparecimento de um número. Nesta implementação, os números inteiros de 0 a 100 são armazenados em células de posições determinadas e para estes números os endereços são únicos. Assim, se ZERO é o endereço do número 0, o número $n (0 \le n \le 100)$ terá endereço ZERO + 2n. Nenhuma tentativa é feita para unificar as representações de números maiores que 100.

Na segunda célula da representação acima, o valor do número na base binária é dado pelos bits 46 a 0, no mesmo formato de números do B-6700, isto é:

Bit	Uso
46	Sinal do número (0=positivo, 1=negativo)
45	Sinal do expoente (0=positivo, 1=negativo)
44-39	Expoente
38-0	Mantissa

3 - ENTRADA E SAÍDA

3.1 - Átomos especiais

Quando desejamos que certos caracteres especiais, tais como . ( ) * ƀ " etc., façam parte dos caracteres que compõem um átomo, usamos a seguinte construção:

$$d átomo especial d

Os dois caracteres $$ indicam a presença de um áto

mo especial e d é um delimitador do átomo especial e pode ser qualquer caráter desde que o mesmo não compareça no átomo especial.

Exemplos:

$$A(((A	átomo formado = (((
$$'LISP (1.5)'	átomo formado = LISP (1.5)
$$$16/04$	átomo formado = 16/04
$$X$$X	átomo formado = $$

3.2 - Criação de p-listas

O sistema LISP mantém um conjunto de p-listas de átomos considerados básicos tais como aqueles que são funções LISP pré-definidas do tipo SUBR ou FSUBR (CAR,CONS,EQ, PLUS, CSET, etc.), ou os do tipo APVAL (DOLLAR, COMMA, NIL, T, etc.), ou mesmo os átomos cuja única propriedade é PNAME mas que são frequentemente usados pelo sistema LISP (como os átomos PNAME, SUBR, LAMBDA, etc.), ou aqueles que são preferidos pelos usuários (os átomos com uma só letra no seu nome de impressão: A, B, C, ..., X, Y, Z). Também são considerados como básicos os átomos especiais $, +, -, *, (,),etc.

As p-listas básicas são criadas, antes mesmo de se iniciar a leitura dos programas LISP, pelo procedimento initialize. Os endereços destas p-listas são incorporados à lista dos símbolos atômicos. Na leitura de um programa LISP,es

ta lista é consultada para cada átomo lido, e novas p-listas podem ser criadas e incorporadas à lista dos símbolos atômicos.

A criação de p-listas é realizada por dois procedimentos:

a) Procedimento makepname (n, carac, i):

Este procedimento cria uma p-lista do tipo PNAME com n caracteres armazenados na matriz carac. O endereço da p-lista criada estará contido na variável i. Após o procedimento makepname, deve ser chamado o procedimento putinlas (i, carac) para introduzir a nova p-lista na lista dos símbolos atômicos.

Por exemplo, makepname pode criar a seguinte p-lista para o átomo MARGARIDA:

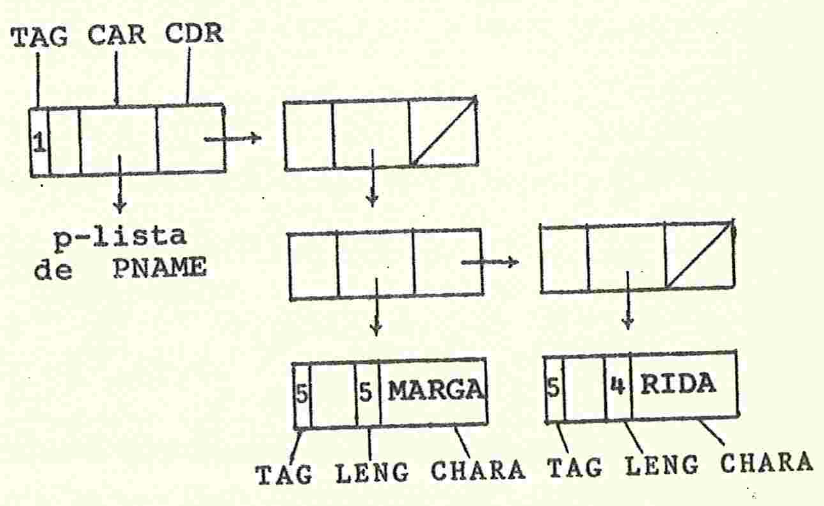


Fig. 3.6

b) Procedimento makeplist (n, carac, type, a, b, i):

Este procedimento cria uma p-lista para um átomo

com n caracteres no nome de impressão carac, o indicador de propriedade é type (que pode ser SUBR, EXPR, etc.), a e b são 1º e 2º membro da célula de indicação da propriedade, i conterá o endereço da p-lista. O procedimento, após a criação da p-lista, coloca a mesma na lista dos símbolos atômicos.

Por exemplo, makeplist pode criar a seguinte p-lista para o átomo EQ que é do tipo SUBR:

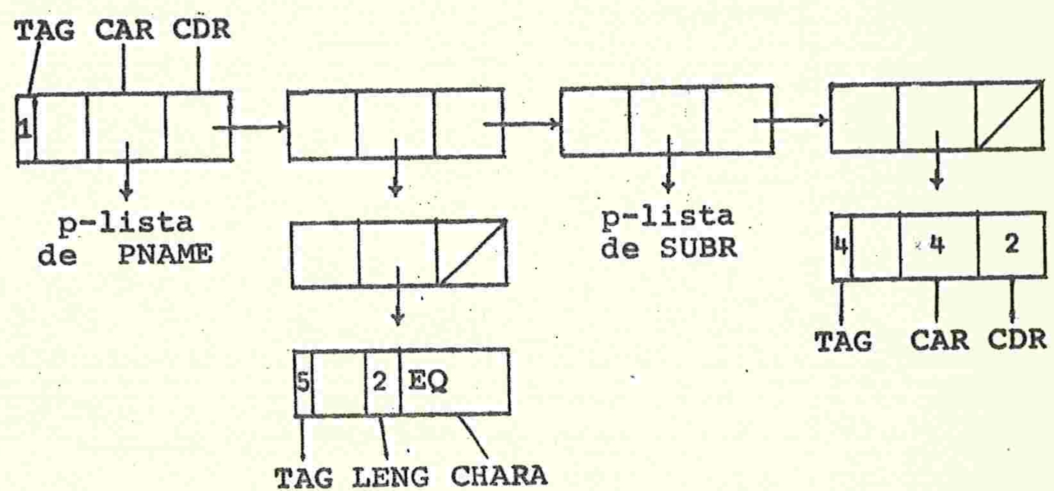


Fig. 3.7

Nesta implementação, todas as funções do tipo SUBR são identificadas por um número 0, 1, 2, ..., etc., e este número de identificação da função é colocado no 1º membro da célula de indicação da propriedade SUBR. Assim, no exemplo, a função EQ é identificada pelo número 4. O mesmo é feito para as funções do tipo FSUBR. No caso de função do tipo SUBR, o 2º membro da célula de indicação da propriedade contém o número de argumentos da função.

3.3 - Procedimentos de entrada

Por meio dos procedimentos de entrada, uma S-expressão é lida,a partir de leitora de cartões ou de disco, e é convertida em representação interna correspondente.As funções LISP que têm essa finalidade são READ (para ler uma S-expressão de leitora de cartões) e DREAD (para ler uma S-expressão de disco).

Os principais procedimentos de entrada são scanatom, plistnumber, readatom, read1 e readch.

a) Procedimento scanatom:

Cada vez que este procedimento é chamado, ele localiza, percorrendo um cartão ou um registro de disco (dependendo do valor de uma variável controladora swfin), um átomo (pode ser átomo especial), ou um número, ou ainda um dos seguintes símbolos: abre-parênteses, fecha-parênteses ou ponto. O elemento lido é então colocado numa matriz chamada atomin com o tamanho do elemento (ou seja, número de caracteres numa variável chamada lengthatom.Nesta implementação um átomo pode ter no máximo 120 caracteres. Uma variável booleana chamada isnumber será tornada true se o elemento lido é um número e false em caso contrário.

Por exemplo, se o cartão contém:

```
(   LISP   ( 1 . 5 )   ALGOL   B6700   )
```

então 10 chamadas sucessivas de scanatom terão os seguintes resultados:

atomin	lengthatom	isnumber
(	1	false
LISP	4	false
(	1	false
1	1	true
.	1	false
5	1	true
ALGOL	5	false
B6700	5	false
)	1	false

b) Procedimento plistnumber:

Este procedimento recebe um número em atomin, com o seu comprimento ou tamanho em lengthatom, e ele deve obter o endereço da representação interna do número. O procedimento verifica se o número é um inteiro compreendido entre 0 e 100. Em caso afirmativo, o número tem representação em lugar fixo e seu endereço vale ZERO + 2x número, onde ZERO é o endereço da representação do número 0. Em caso negativo, é criada uma representação para o número, extraindo-se duas células do espaço livre.

c) Procedimento readatom:

Este procedimento chama inicialmente o procedimen-

to scanatom, recebendo o elemento lido em atomin e o tamanho do mesmo em lengthatom. O procedimento deve obter o endereço do elemento lido. Se o elemento lido for um número (isnumber é true), então o endereço do mesmo será obtido chamando-se o procedimento plistnumber. Se o elemento não for um número, então é verificado se ele já está na lista dos símbolos atômicos. Caso não esteja, então deve ser criada uma p-lista para o elemento lido (por meio do procedimento makepname) e a nova p-lista deve ser incorporada à lista dos símbolos atômicos (por meio do procedimento putinlas).

d) Procedimento readl:

Este procedimento tem por finalidade a leitura de uma S-expressão e sua conversão em representação interna.

e) Procedimento readch:

Este procedimento lê um caráter a partir do dispositivo de leitora de cartões. O procedimento correspondente para disco é dreadch.

3.4 - Procedimentos de saída

Os procedimentos de saída têm por finalidade a impressão de uma S-expressão pelo dispositivo de saída, que pode ser impressora de linhas ou disco, a partir da representação interna correspondente. As funções LISP que têm essa finalidade são PRINT (dispositivo de saída = impressora) e

DPRINT (dispositivo de saída = disco).

Os principais procedimentos de saída são terpri, prinl e print (dprint para disco).

a) Procedimento terpri:

Este procedimento imprime uma linha na impressora ou um registro de disco (dependendo de uma variável controladora swfout). A função LISP correspondente é a função sem argumentos TERPRI.

b) Procedimento prinl(x):

O argumento x do procedimento prinl é o endereço de uma p-lista ou de uma representação de um átomo numérico. O procedimento usa uma área de saída que corresponde a uma linha de impressão. Caso essa área de saída não está totalmente cheia ainda, então o número a ser impresso ou os caracteres de impressão do símbolo atômico são colocados nessa área. Caso a área já está totalmente preenchida, então ela é impressa por meio de terpri e o número ou os caracteres são colocados no início da área.

c) Procedimento print(x):

O procedimento print tem um argumento que é o endereço da representação interna de uma S-expressão. A sua finalidade é a impressão dessa S-expressão pela impressora.

4 - FUNÇÕES COMPOSTAS DE CAR'S E CDR'S

A cada função composta de car's e cdr's até 4 níveis, isto é, CAAR, CADR, CDAR, CDDR, CAAAR, ..., CDDDDR, corresponde um procedimento distinto, com o mesmo nome da função composta. Isto dá um total de 4+8+16=28 procedimentos. O caso genérico de funções compostas de mais de 4 níveis, por exemplo CAAADADDAR, é solucionado da seguinte maneira:

Na entrada de uma S-expressão, o procedimento readatom, ao criar a p-lista para um símbolo atômico que começa pela letra C, termina pela letra R e tendo entre ambas uma cadeia com mais de 4 letras que são combinações das letras A e D, irá colocar sobre a p-lista deste símbolo atômico um indicador EXPR e uma λ-expressão que corresponde à função composta. Por exemplo, se é lido um símbolo CAAADAR e supondo que este símbolo ainda não figura na lista dos símbolos atômicos, então o procedimento readatom irá criar a seguinte p-lista:

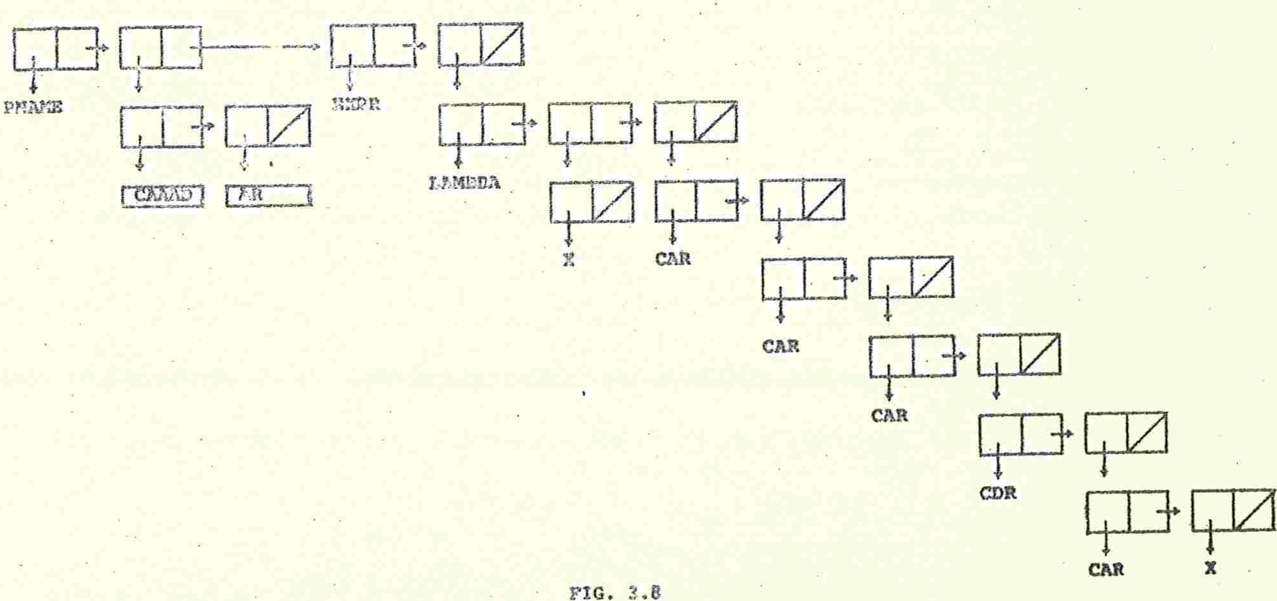


FIG. 3.8

5 - FUNÇÕES DO SISTEMA LISP

Cada função LISP do tipo SUBR (como car,cdr, null, sqrt, etc.) ou do tipo FSUBR (como times, max, and,or,etc.) corresponde a um procedimento cujo tipo é booleano, inteiro ou real.

a) Se a função LISP é um predicado, então o procedimento correspondente é do tipo booleano, por exemplo:

```
boolean procedure numberp(x);
     value x; integer x;
     numberp:=tag(x) = 2
```

No exemplo acima, x é o endereço da representação de uma S-expressão.

b) Se a função LISP é uma função aritmética, então o procedimento correspondente é do tipo real ou inteiro. O valor do procedimento é o valor numérico da função aritmética. Por exemplo:

```
real procedure expt(x,y);
     value x,y; integer x,y;
     if numberp(x) and numberp(y)
          then expt:=num(x) ** num(y)
          else error(8)
```

No exemplo acima x e y são endereços das representações de números, e num é definido por:

<u>define</u> num(i)=cell[cdr(i)]#.

c) Se a função LISP não é nem predicado nem função aritmética, então o procedimento correspondente é do tipo inteiro. Por exemplo:

```
integer procedure rplaca(x,y);
    value x, y; integer x, y;
    begin
        car(x):=y;
        rplaca:=x
    end
```

Podemos ver que há um conflito entre a função predicado LISP e o procedimento do tipo booleano correspondente, assim como entre a função aritmética LISP e o procedimento correspondente. No caso da função predicado, o seu valor é o símbolo atômico T ou NIL, ou seja a p-lista de T ou NIL, ao passo que o procedimento booleano correspondente dá como valor <u>true</u> ou <u>false</u>. No caso da função aritmética, o valor é o endereço da representação de um número. O seu procedimento correspondente dá simplesmente o valor numérico. Este problema é contornado por dois procedimentos cb(x) e ca(x) que fazem as conversões necessárias.

O procedimento cb converte o valor booleano <u>true</u> ou <u>false</u> em t ou nil:

```
integer procedure cb(x);
    boolean x;
    cb:=if x then t else nil
```

onde t e nil valem respectivamente os endereços das p-listas dos símbolos atômicos T e NIL.

O procedimento ca converte um número em sua representação interna.

6 - INTERPRETADOR

O interpretador consta principalmente de três procedimentos: apply, eval e evalquote, que correspondem às funções de mesmos nomes descritas no capítulo anterior. Convém fazer uma observação sobre a chamada de procedimentos que correspondem a funções LISP do tipo SUBR e FSUBR.

a) Procedimento callsubr(u,v):

Quando o procedimento apply encontra uma função do tipo SUBR, ele obtém a célula de indicação de SUBR (contendo o número de identificação da função e o número de argumentos da mesma) e chama o procedimento callsubr(u,v) onde os parâmetros u e v são o endereço da célula de indicação e a lista de argumentos da função.

O procedimento callsubr possui as variáveis locais w, x, y e z. À variável w atribui-se o número de identificação da função. Às variáveis x, y e z atribuem-se o 1º, 2º e

3º argumento da função, caso houver. A chamada do procedimento correspondente à função LISP é feita pelo comando ALGOL <u>case</u>:

```
callsubr:= case w of
          (      car(x),
                 cdr(x),
                   .
                   .
                   .
                 cb(atom(x)),
                 cb(eq(x,y)),
                   .
                   .
                   .
                 ca(addl(x)),
                 ca(subl(x)),
                 ca(difference(x,y)),
                   .
                   .
                   .       )
```

b) <u>Procedimento callfsubr(x,y,a):</u>

Quando o procedimento eval encontra uma função do tipo FSUBR, ele obtém a célula de indicação de FSUBR (o 1º membro da célula contendo o número de identificação da função) e chama o procedimento callfsubr(x,y,a) onde x é o endereço da célula de indicação, y é a lista dos argumentos da função e <u>a</u> é a a-lista.

O procedimento callfsubr chama o procedimento cor-

respondente à função LISP do tipo FSUBR através do comando case, do modo análogo ao callsubr.

7 - RECUPERADOR DE CÉLULAS INATIVAS

7.1 - Pilha de argumentos save

O sistema LISP mantém uma pilha de argumentos constituída por uma matriz uni-dimensional chamada save. A variável top aponta para o topo da pilha. No início de cada interpretação a pilha é tornada vazia.

A inserção e remoção de elementos da pilha save são feitas respectivamente por pushdown(x) e popup(n), definidos como sendo:

```
define pushdown(x)=top:=top+1;save[top]:=x#,
       popup(n)=top:=top-n#
```

loadval(u,x), definido como segue, é usado para carregar um valor x na pilha save, fazendo antes top igual a u.

```
define loadval(u,x)=val:=x;top:=u;save[top]:=val#
```

7.2 - Uso da pilha save

A pilha save é usada por todos os procedimentos correspondentes a funções LISP onde há extração de células do espaço livre, ou sejam, procedimentos que criam estrutu-

ras novas, chamando exfree ou cons direta ou indiretamente.

Seja proc um procedimento correspondente a uma função LISP (de n argumentos) enquadrada nas condições acima. Antes de chamar o procedimento proc, os n argumentos são colocados na pilha save, com o uso de pushdown:

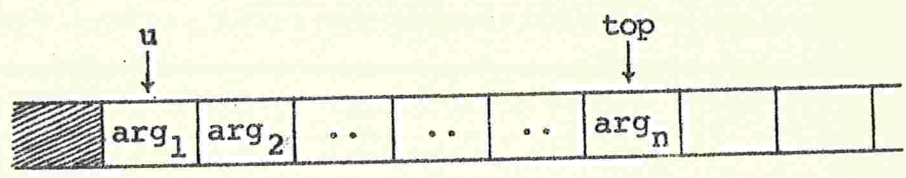


Fig. 3.9

Feito isso, é chamado o procedimento proc como seu único parâmetro de chamada igual a top-n+1, isto é, um apontador à pilha onde começam os n argumentos.

O procedimento proc(u) obtém os n argumentos da função LISP correspondente em save[u],save[u+1],...,save[u+n-1]. O valor de proc, seja ele v, é colocado na pilha por meio de loadval(u,v):

```
integer procedure proc(u);
    value u; integer u;
    begin
          .
          .
          .
            proc:=loadval(u,v)
          .
          .
          .
      end
```

À saída do procedimento proc, o topo da pilha save conterá o valor do procedimento:

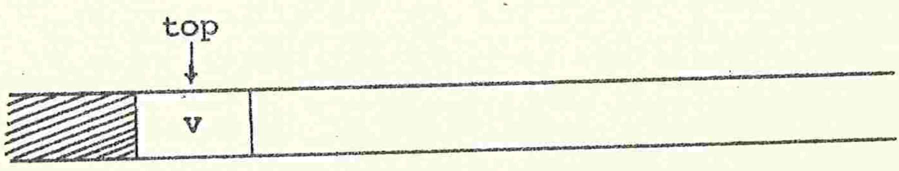


Fig. 3.10

7.3 - Alguns Exemplos:

O procedimento cons, correspondente à função LISP do mesmo nome, é definido como segue:

```
integer procedure cons(u);
    value u; integer u;
    begin
        exfree;
        car(free):=save[u];
        cdr(free):=save[u+1];
        cons:=loadval(u,free)
    end
```

O procedimento append, correspondente à função append cuja definição em m-linguagem é

```
append[x;y] = [null[x]→y;
                T→cons[car[x];append[cdr[x];y]]]
```

é declarado como segue:

```
integer procedure append(u);
    value u; integer u;
    begin
        define x=save[u]#,
               y=save[u+1]#;
        if null(x)
        then begin
            append:=loadval(u,y)
            end
        else if atom(x)
            then error(19)
            else begin
                pushdown(car(x));
                pushdown(cdr(x));
                pushdown(y);
                append(top-1);
                append:=loadval(u,cons(top-1))
                end
    end
```

Os demais procedimentos que usam a pilha save são o procedimento evalquote e aqueles correspondentes às seguintes funções LISP: and, apply, cset, csetq, define,eval,evcon, evlis,list,mapcar,maplist, max, min, or, pair, plus, prog, sassoc, times, etc.

7.4 - Finalidade da pilha save

A pilha save serve para salvar estruturas de lista construídas por procedimentos que chamam exfree ou cons direta ou indiretamente. Todas as estruturas de lista cujos apontadores ficam na pilha save são consideradas ativas ou úteis e não são destruídas pelo recuperador de células inativas, caso o mesmo seja ativado. Uma situação em que as estruturas de lista devem ser salvas é a seguinte:

$$f(u_1(x),\ u_2(y),\ \ldots,\ u_n(z))$$

em que u_1, u_2, ..., u_n são procedimentos que criam novas estruturas de lista e f um procedimento qualquer. Suponhamos que os procedimentos u_1, u_2, ..., u_i já criaram as i estruturas de lista e que na execução de u_{i+1} o espaço livre se esgote. Se as i estruturas não fossem salvas, elas seriam devolvidas ao espaço livre.

7.5 - Descrição do algoritmo

O algoritmo usado no recuperador de células inativas, para mover uma estrutura de lista para um outro lugar, compactando-a, deve-se a E.M.Reingold [Rei73], com ligeiras adaptações.

Para fixar idéia, suponhamos a seguinte lista apon

tada pela variável head:

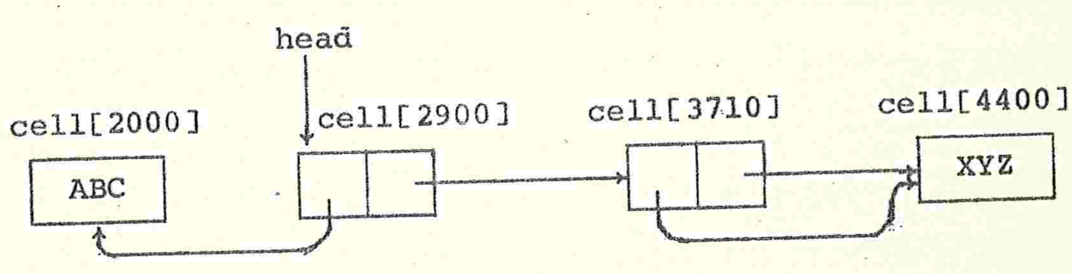


Fig. 3.11

Supondo que newcell é o nome da matriz para a qual é movida a lista acima, e supondo que newcell[0] é a 1ª célula disponível, teremos, após a execução do algoritmo:

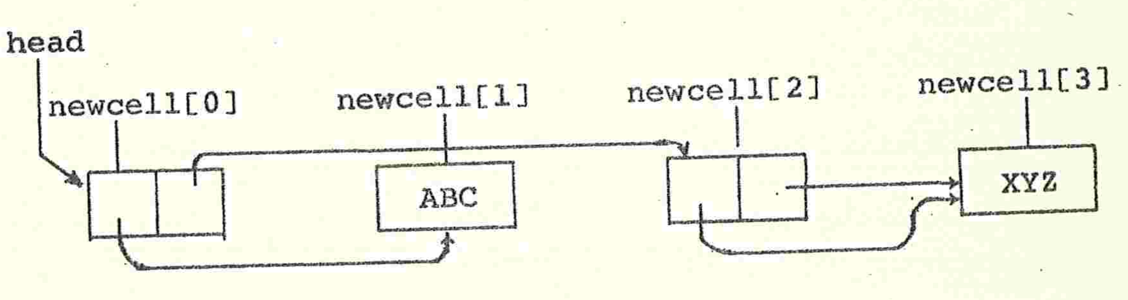


Fig. 3.12

O algoritmo de Reingold é baseado na combinação de duas idéias distintas. A primeira idéia é uma modificação de um algoritmo de recuperação devido a Schorr e Waite [Sch67] e independentemente a Deutsch [Knu68: pág.417-418]: o uso do cdr das células da lista original para armazenar apontadores que apontam de volta para células anteriores. A segunda idéia é devida independentemente a Cheney [Che70] e Edwards [Knu68: problema 9 do §2.3.5]: o uso do car das células da lista original para apontar para as células correspondentes na nova có-

pia da lista.

A estrutura de lista original apontada por head é movida consecutivamente para newcell [início], newcell [início+1], newcell[início+2], ..., etc., sendo início o índice da 1ª célula disponível de newcell. Se uma célula já foi movida, então o seu campo MARK (seção 2.1 deste capítulo) será feito igual a 1, caso contrário MARK contém 0. Após a cópia, head apontará para a estrutura da lista movida.

O algoritmo, dividido em vários passos, é apresentado a seguir. Neste algoritmo, j aponta para uma célula a ser copiada e jj aponta para a célula que acabou de ser copiada.

Passo 1 (Inicialização: j aponta para o início da lista a ser movida):

i:=início;

Se mark(head)=1 então head:=car(head) e termina o algoritmo;

j:=head;

jj:=NIL;

Passo 2 (Copia a célula apontada por j. Atualiza o valor de jj):

i:=i+1;

newcell[i]:= cell[j];

mark(j):=1;

car(j):=i;

cdr(j):=jj;

jj:=j;

Passo 3 (Verifica car da célula copiada):

Se atomm(car(j)) = 1 então vá para passo 6;

Se car(car(j)) = NIL então vá para passo 4;

Se mark(car(car(j))) = 0 então j:=car(car(j)) e volta ao passo 2;

Passo 4 (Verifica cdr da célula copiada):

Se cdr(car(j)) = NIL então vá para passo 5;

Se mark(cdr(car(j))) = 0 então j:=cdr(car(j)) e volta ao passo 2;

Passo 5 (Reposiciona apontadores na cópia):

Se atomm(car(j)) = 1 então vá para passo 6;

Se car(car(j))≠NIL então car(car(j)):=car(car(car(j)));

Se cdr(car(j))≠NIL então cdr(car(j)):=car(cdr(car(j)));

Passo 6 (Volta pelo cdr):

j:=cdr(j);

jj:=j;

Passo 7 (Faz head apontar para a cópia da lista, se já terminou):

Se j = NIL então head:=car(head) e termina o algoritmo senão volta ao passo 3.

Como ilustração, apresentamos a seguir as diversas

etapas de uma aplicação do algoritmo. Nas figuras apresentadas, uma célula já movida é representada por uma marca preta no canto esquerdo da célula:

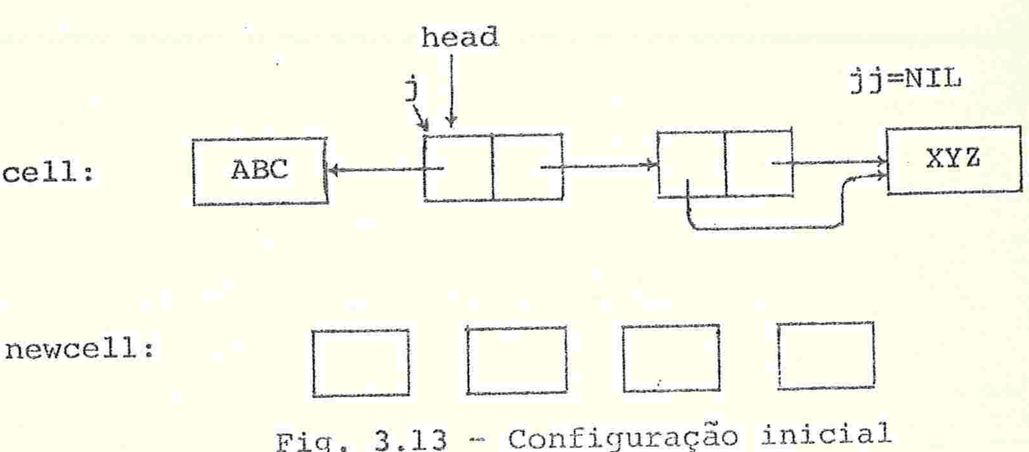


Fig. 3.13 - Configuração inicial

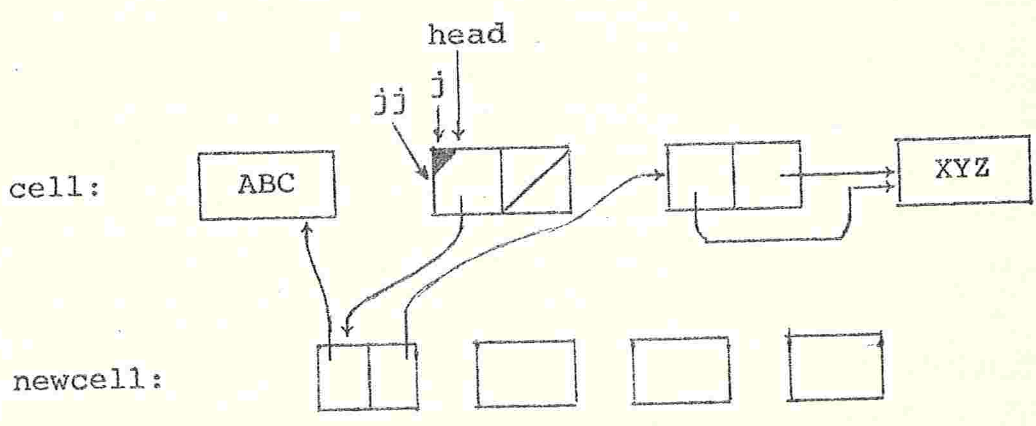


Fig. 3.14 - Cópia da 1ª célula

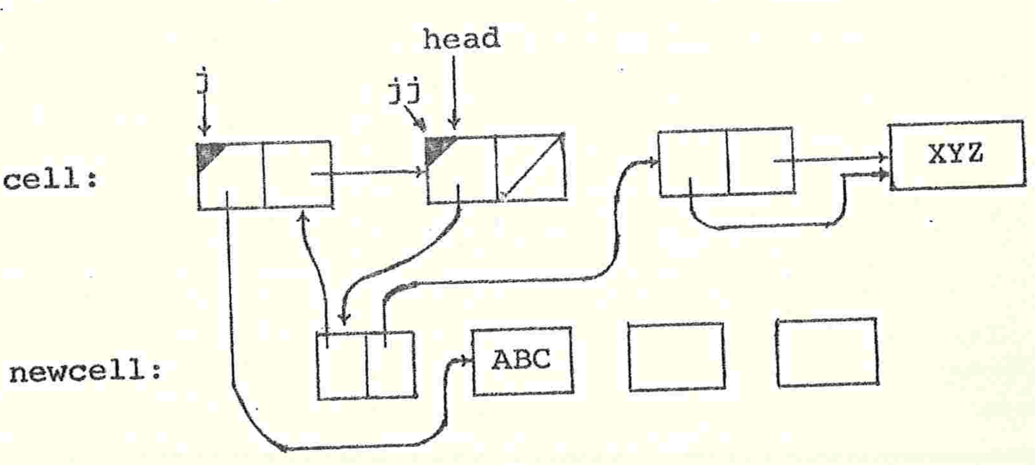


Fig. 3.15 - Cópia da 2ª célula

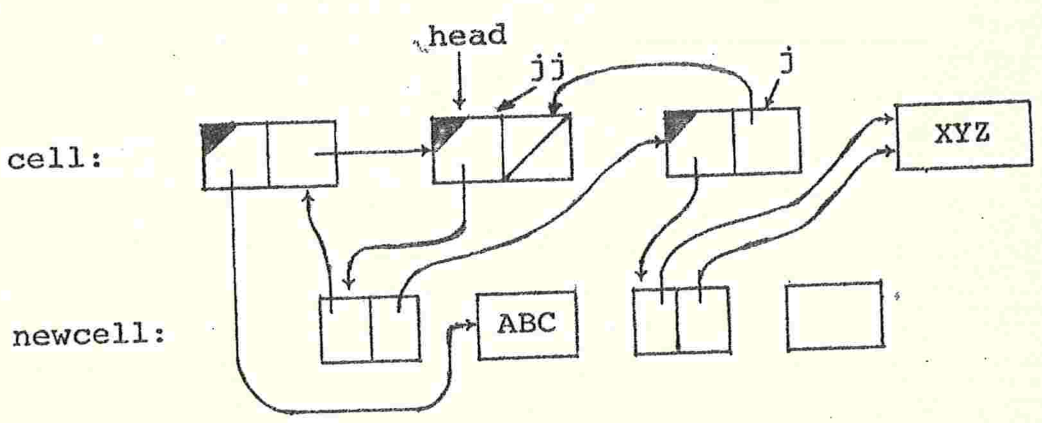


Fig. 3.16 - Cópia da 3ª célula

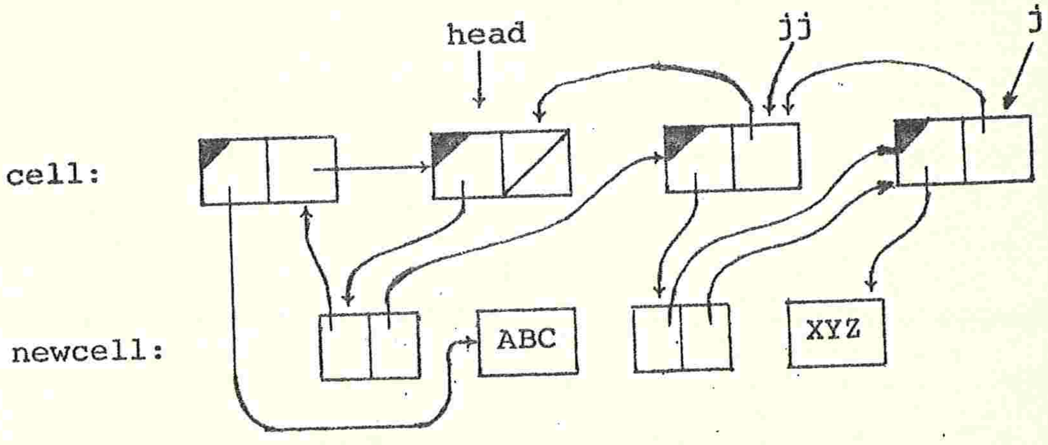


Fig.3.17 - Cópia da 4ª célula

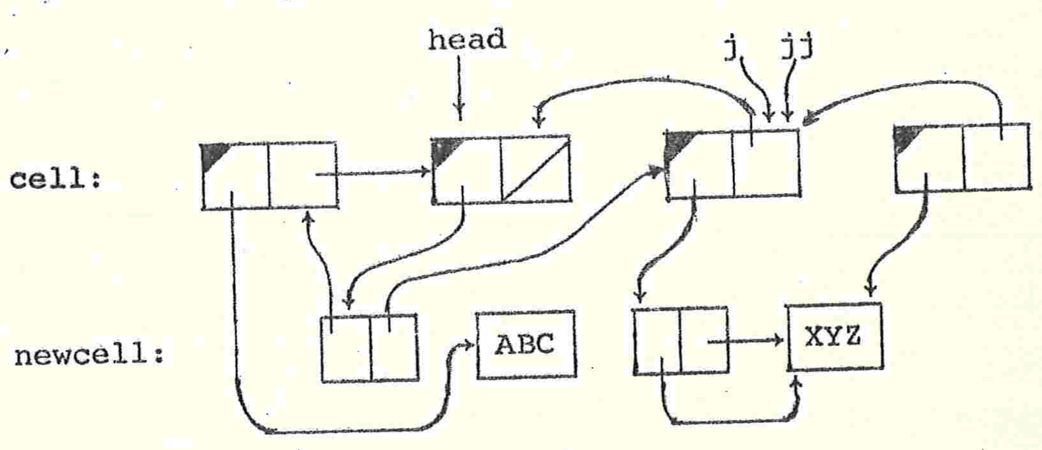


Fig. 3.18 - Posiciona apontadores de newcell[3]

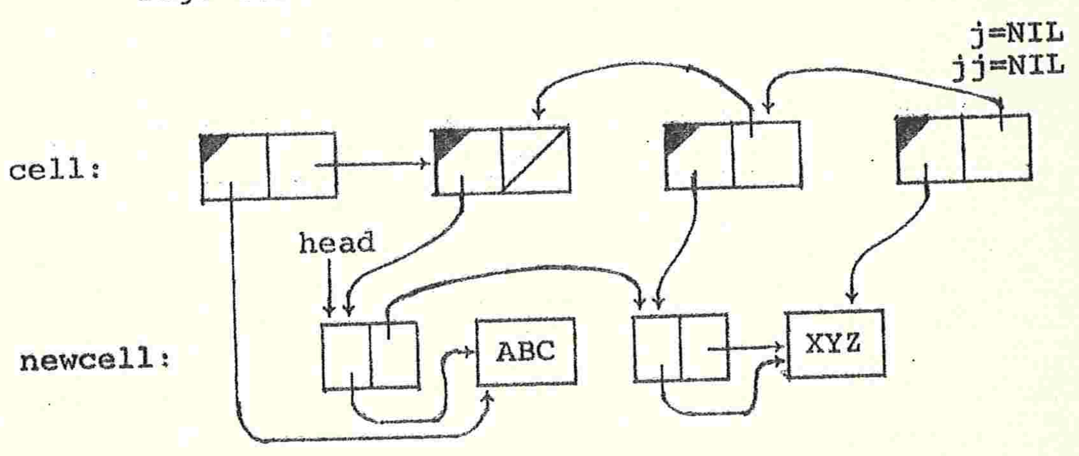


Fig. 3.19 - Posiciona apontadores de newcell[1]

7.6 - Comentários sobre o algoritmo

O sistema B-6700 é um sistema de computação com memória virtual. Pelo exposto na seção 7 do Capítulo II, convém utilizar um algoritmo de recuperação que faz a compactação dos dados ativos. O fato de o algoritmo adotado necessitar de uma outra área, que chamamos de newcell, não constitui nenhum problema, do ponto de vista de memória virtual, mas po-

de ser um fator de ineficiência se for necessário recopiar newcell para cell. No B-6700, existe um comando <u>swap</u> que troca entre si as referências de duas matrizes, de modo que não há necessidade de uma transferência física de dados.

Faremos um apanhado de diversos algoritmos que podem ser usados num recuperador de células inativas.

Algoritmos de marcação podem ser usados num recuperador [Knu68, Sch67]. As células ativas marcadas e salvas por esses algoritmos não ficam compactadas após a recuperação e suas referências mantêm-se inalteradas antes e após uma recuperação.

Os algoritmos de cópia de listas [Fen69, Lin74, Fis75] diferem-se dos de movimentação de listas [Che70, Rei73] em que a operação de cópia não pode destruir a lista original. Lindstrom apresenta em [Lin74] dois algoritmos de cópia. O primeiro não usa nenhum bit de indicação e a tarefa da cópia de uma lista de n células pode ser realizada em tempo n^2. O segundo algoritmo usa um indicador de um bit por célula. Nesse algoritmo, uma estrutura não cíclica pode ser copiada em tempo linear e uma estrutura cíclica em tempo menor que n(log n). O algoritmo de Fisher [Fis75] opera em tempo linear e não requer bits de indicação. A única restrição do algoritmo é que a cópia deve ser colocada numa seção contígua de memória.

O algoritmo de Reingold [Rei73] faz a movimentação

de uma lista, compactando-a, e esse algoritmo não requer nenhum outro espaço de armazenamento a não ser o espaço para onde a estrutura é movida. A estrutura original é destruída ao término do algoritmo. Esse algoritmo foi adotado pelo recuperador da nossa implementação pois ele realiza a compactação desejada de maneira eficiente (tempo proporcional ao número de células da estrutura), e o fato de ter a estrutura original destruída não constitui nenhum problema, uma vez que o sistema irá trabalhar no novo espaço.

7.7 - Procedimento garbage

O procedimento garbage é ativado por exfree, ou indiretamente pelo procedimento cons, quando o espaço livre se esgota. Todas as estruturas ativas na matriz cell são movidas para um outro espaço que é uma nova matriz chamada newcell dentro da qual as estruturas movidas apresentar-se-ão compactadas.

Para realizar essa movimentação, o procedimento garbage usa o algoritmo de Reingold que é codificado na forma de um procedimento chamado move(head), onde head aponta para uma estrutura a ser movida. Após a movimentação, head apontará para a estrutura movida. São consideradas como estruturas ativas a lista dos símbolos atômicos e todas as estruturas contidas na pilha save. Todos os endereços contidos na pilha sa

ve devem ser atualizados uma vez que a movimentação altera a posição das estruturas.

Após a movimentação de todas as estruturas ativas de cell para newcell, são trocadas entre si as referências das matrizes cell e newcell por meio do comando swap.

Vamos fazer uma ilustração através de um exemplo. Seja o espaço livre inicial com 15 células (para facilitar o desenho) e seja a variável freesize indicando este valor. Como veremos na seção seguinte, o valor de freesize pode ser especificado pelo usuário.

Espaço livre inicial

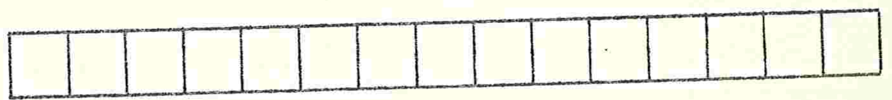


Fig. 3.20

As figuras 3.21 e 3.22 ilustram respectivamente as configurações antes e após a movimentação das estruturas ativas.

ANTES DA MOVIMENTAÇÃO

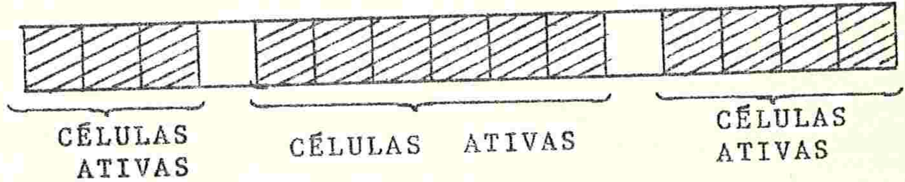


Fig. 3.21

APÓS A MOVIMENTAÇÃO

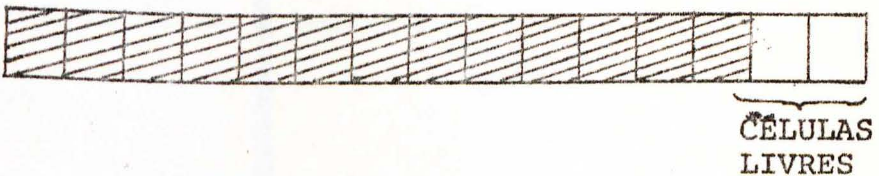


Fig. 3.22

Após a movimentação das estruturas, são impressas mensagens informando a quantidade de células livres recuperadas, o número de páginas ativas antes da movimentação, o número total de células ativas por página ativa (valor absoluto e porcentagem). O tamanho de cada página é de 256 células.

Em seguida, é verificado se a quantidade de células livres recuperadas é inferior a 1/3 do valor de freesize. Caso afirmativo, a matriz cell é redimensionada, de modo que o número de células livres seja igual a freesize/3. O valor de freesize é atualizado de acordo e uma mensagem é impressa informando este fato. No exemplo considerado, como a quantidade de células recuperadas (2) é inferior a freesize/3 (5), cell é então redimensionada para conter 5 células livres e o valor de freesize é incrementado de 3.

APÓS O REDIMENSIONAMENTO

freesize = 15 + 3 = 18

Fig. 3.23

Antes de terminar o procedimento garbage, o tempo gasto na sua execução é impresso.

Observações:

a) O usuário pode ativar uma recuperação por meio da função sem argumentos garbage, por exemplo,

(LAMBDA NIL (GARBAGE)) NIL

b) O fator 1/3, usado para decidir se deve ser feito o redimensionamento ou não, foi apenas adotado, sem maiores justificativas.

c) O valor padrão de freesize adotado pelo sistema LISP (também é o máximo permitido, ver seção 3.3 do Capítulo IV) é de 63700 células. Portanto o recuperador só é ativado quando todas essas células são usadas. O usuário que deseja ativar o recuperador antes pode especificar um valor de freesize menor, por meio de um cartão de controle, que veremos na seção seguinte.

8 - CARTÕES DE CONTROLE

Os cartões de controle para o sistema LISP têm o seguinte formato:

// <comentários> <opções> [<freesize>]

onde // devem ocupar as colunas 1 e 2 e o restante tem formato livre e pode ser colocado em qualquer ordem. As <opções>

existentes são:

a) Listagem ou não do programa:

NOLIST: não se deseja listagem do programa deste ponto em diante.

LIST: invalida a opção NOLIST anterior; o programa será listado daqui em diante.

A opção padrão é LIST.

b) Numeração ou não dos parênteses do programa:

NONUMBER: não se deseja numeração dos parênteses deste ponto em diante.

NUMBER: invalida a opção NONUMBER anterior; o programa terá seus parênteses numerados daqui em diante.

A opção padrão é NUMBER.

c) Verificação ou não da sequência dos cartões:

As colunas 73 - 80 dos cartões de programas LISP são reservadas para numeração (em ordem crescente) dos cartões, podendo também estarem totalmente em branco, ou conterem quaisquer comentários se a opção for NOCHECK.

NOCHECK: não se deseja verificação da sequência dos cartões deste ponto em diante.

CHECK: invalida a opção NOCHECK anterior; a verificação começa daqui em diante.

A opção padrão é CHECK.

O usuário que deseja especificar o número inicial de células do espaço livre pode colocar o número desejado <freesize> entre colchetes (sem nenhum espaço em branco entre o número e o par de colchetes). Este número não pode ultrapassar 63700. O número padrão adotado pelo sistema é 63700. Se o número especificado pelo usuário é menor que 3000 ou maior que 63700, o sistema adotará o número padrão.

Os cartões de controle podem aparecer em qualquer lugar do programa. Caso haja várias especificações de <freesize>, será tomado o último.

Quando o programa LISP tem dados (S-expressões) para serem lidos, estes dados devem ser colocados depois de um cartão:

```
// DATA
```

Exemplo:

```
// [20000]  PROGRAMA EXEMPLO      NOCHECK
// PROGRAMADOR: RIP VAN WINKLE 01/04/75
//     NONUMBER
             .
             .
             .
        Programa LISP
             .
             .
//  CHECK    NUMBER
             .
             .
             .
```

Programa LISP

.
.

// DATA

.
.
.

Dados

.
.
.

9 - SUPERVISOR

O supervisor controla o sistema LISP e tem as seguintes etapas:

a) Imprime um cabeçalho com a data corrente.

b) Lê cartões de programas, gravando-os no disco até que seja atingido o fim dos cartões ou até encontrar o cartão // DATA.

c) Dá as configurações iniciais do sistema, criando as p-listas básicas, reservando áreas de dados, etc.

d) Imprime o número inicial de células livres (freesize).

e) Lê um programa (um par de S-expressões) a partir do disco. Conforme as opções do usuário, imprime o programa lido, verifica a sequência dos cartões, numera os parênteses. Chama o procedimento evalquote para sua interpretação.

f) Imprime o resultado da interpretação. Imprime o tempo uti

lizado para interpretação.

g) Volta-se à etapa (e) se houver mais programas.

h) Imprime o número de células livres usadas (após a última chamada do garbage, se houver).

Observação:

No Apêndice II encontra-se uma lista das mensagens de erro que o sistema LISP pode acusar durante uma interpretação.

10 - RESUMO DAS PARTICULARIDADES DA IMPLEMENTAÇÃO

Resumimos abaixo as particularidades da implementação, algumas das quais já foram mencionadas.

a) Cada átomo pode conter no máximo 120 caracteres.

b) Um átomo pode conter caracteres especiais com exceção de $ ( ) . e ƀ. O primeiro caráter de um átomo não precisa necessariamente ser uma letra, podendo ser um caráter especial, com exceção de $ ( ) . ƀ + -.

c) Somente os números inteiros de 0 a 100 têm representação única.

d) Na entrada de uma S-expressão, os fecha-parênteses a mais são ignorados.

e) Na listagem do programa fonte LISP, os parênteses são nume

rados com os caracteres 1, 2, 3, ..., 9, A, B, C, ..., Z, de maneira cíclica.

f) Composições de funções de car's e cdr's podem ser escritas abreviadamente como cαr, onde α é qualquer cadeia contendo a's e d's com comprimento ilimitado.

g) A função zerop[x] tem valor verdadeiro T se $|x| < 10^{-10}$.

h) O número máximo de células livres é de 63700.

- □ -

CAPÍTULO IV

UMA OUTRA IMPLEMENTAÇÃO E COMENTÁRIOS

1 - IMPLEMENTAÇÃO MAGIDIN-SEGOVIA

Uma implementação do sistema LISP 1.6 foi feita no computador B-6700 da Universidad Autonoma Metropolitana, México, por M. Magidin e R. Segovia [Mag74-1, Mag74-2, Dia74]. O sistema LISP foi escrito em ALGOL B-6700. A seguir, faremos uma breve descrição de alguns aspectos da implementação.

1.1 - Organização de dados

O sistema LISP implementado usa uma matriz uni-dimensional chamada MEM, de 64K palavras (de 48 bits), divididas em módulos de 256 palavras cada. Os módulos são alocados dinamicamente para cada um dos diferentes tipos de dados que o sistema manipula: átomos, listas, números e matrizes. Dentro de cada módulo, as palavras são alocadas sequencialmente para cada tipo de dado. Quando se esgotam as palavras de um módulo, um novo módulo é alocado.

O sistema LISP mantém uma matriz uni-dimensional

chamada DATATYPE, de 256 palavras, uma para cada módulo, identificando o tipo de dado no módulo correspondente. Apontadores ligando módulos de igual tipo são mantidos. Também são guardados os números de palavras ainda sem uso de cada módulo. Cada palavra desta matriz tem o seguinte formato:

Nº DE PALAVRAS RESTANTES	TIPO DE DADO	APONTADOR PARA O PRÓXIMO MÓDULO DO MESMO TIPO

Fig. 4.1

Cada célula de lista corresponde a uma palavra e tem a seguinte divisão:

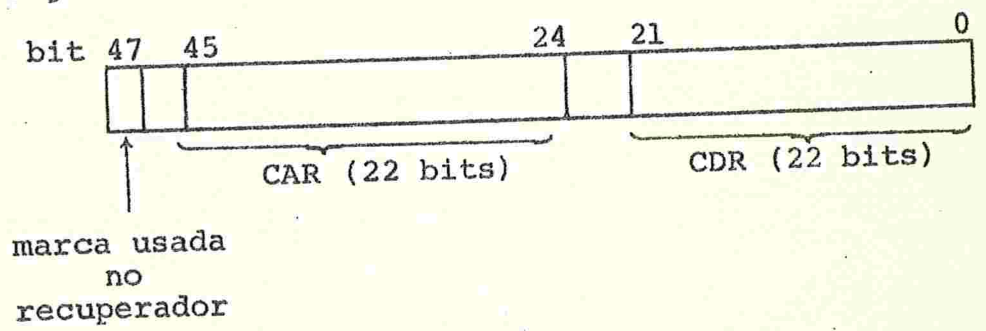


Fig. 4.2

Cada átomo pertence a uma das 128 listas dos símbolos atômicos existentes, segundo uma certa regra de "hashing". As cabeças destas 128 listas são organizadas numa matriz unidimensional de 128 palavras, chamada OBLIST. Novos átomos são inseridos no início de cada uma das 128 listas.

Um átomo usa no mínimo 3 palavras, como mostra a figura 4.3:

<table>
<tr><td>G
C</td><td colspan="2">VALOR</td><td>F
U
N
A
R
G</td><td colspan="2">PROPR</td></tr>
<tr><td>T
R
A
C
E</td><td>Comprimento do nome</td><td colspan="2">#RECUR</td><td>#FUN</td><td>OBL</td></tr>
<tr><td colspan="6">Nome de impressão</td></tr>
</table>

Fig. 4.3

O campo GC é usado pelo recuperador de células ina-tivas. O campo VALOR aponta ao valor do átomo. O campo #FUN le-va um número que univocamente identifica uma função do siste-ma LISP. Funções definidas por programas têm sua definição como sendo o primeiro elemento de PROPR. O campo TRACE é li-gado quando um "trace" deste átomo é desejado, e neste caso #RECUR contém o nível de recursão da função. O campo FUNARG indica um argumento funcional. Finalmente o campo OBL é usa-do para ligar átomos que pertencem à mesma lista de OBLIST.

Existe uma matriz uni-dimensional chamada ARGS que é usada como uma pilha, em que valores antigos associados ou ligados às variáveis em λ-expressões são salvos. Cada pala-vra de ARGS mantém dois apontadores: um para a ligação anti-ga, outro para o átomo.

A variável EJEC aponta para a S-expressão que está sendo interpretada.

Um procedimento chamado INICIALIZA é executado para dar ao sistema certas configurações iniciais. Átomos e funções pré-definidas são colocadas dentro da matriz MEM. São construídas as matrizes OBLIST e DATATYPE.

1.2 - Recuperador de células inativas

O procedimento GETSPACE é usado para alocar novos espaços e atualizar a matriz DATATYPE. Quando não há mais espaço disponível, o procedimento GARBAGE é chamado.

Quatro fases constituem o procedimento GARBAGE.

A primeira fase é a de marcação, onde todos os elementos ativos são marcados para separá-los dos inativos. As estruturas a serem salvas são: a estrutura apontada por EJEC (isto é, a S-expressão que está sendo interpretada), os elementos de pilhas usadas para avaliação (pilha de argumentos, pilha do ALGOL), e os átomos (incluindo seus valores e listas de propriedade). Isso é feito pelo procedimento MARCA que percorre uma estrutura de lista, marcando os seus elementos. Certas rotinas intrínsecas escritas em ESPOL [Bur72-3], que é uma linguagem para o sistema executivo, são usadas para varrer a pilha do ALGOL para procurar por apontadores de listas.

A segunda fase é a fase de movimentação de dados, onde todos os elementos marcados são movidos sequencialmente

para uma matriz chamada MEM1, modificando os elementos de MEM, que agora apontam para a posição correspondente do elemento movido para MEM1.

A terceira fase atualiza todos os apontadores em MEM1, fazendo uso dos endereços deixados em MEM.

Finalmente, na quarta fase, o conteúdo de MEM1 é copiado de volta para MEM. Numa versão futura, esta cópia não será mais feita, e a troca será realizada por alterações apropriadas de descritores de dados.

As figuras 4.4 a 4.6 ilustram as operações do recuperador de células inativas.

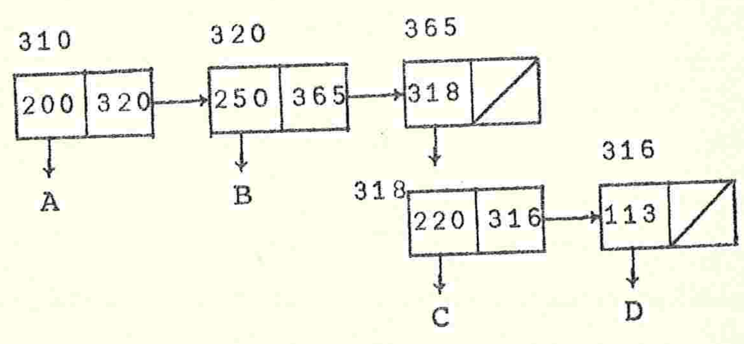


Fig. 4.4 - Estado inicial

(A, B, C e D são átomos)

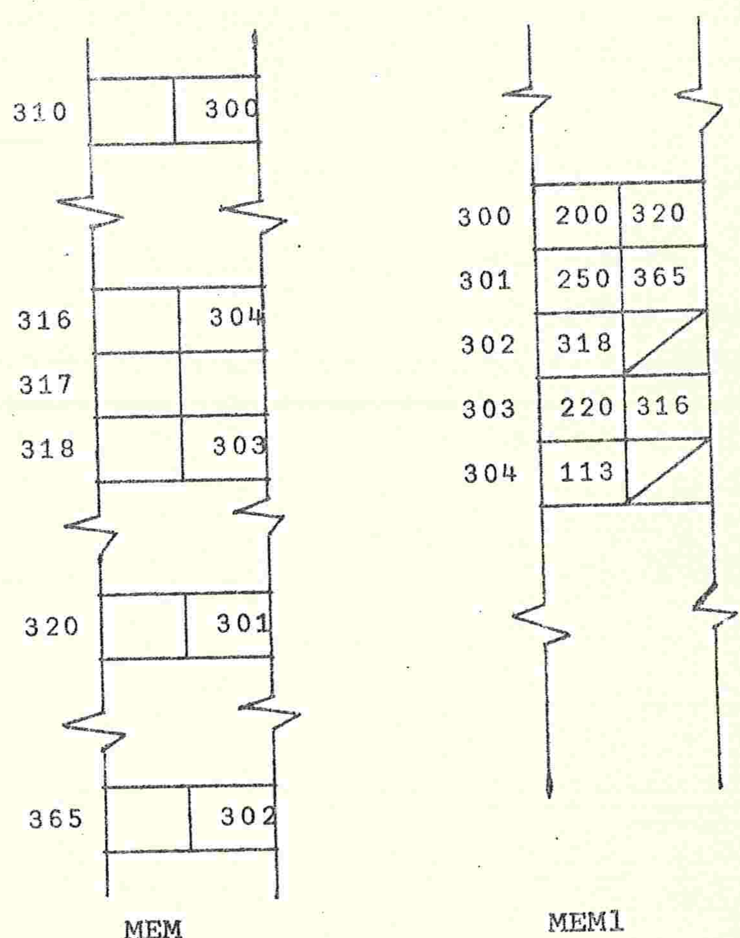


Fig. 4.5 - MEM e MEM1 após a fase de movimentação

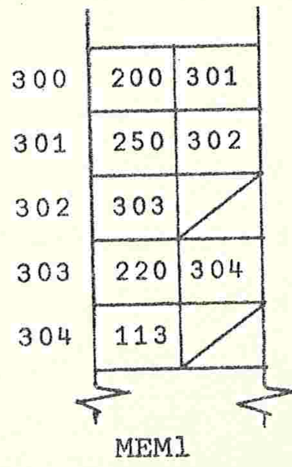


Fig.4.6 - MEM1 após a atualização dos apontadores (foi suposto que os átomos em 113, 200, 220 e 250 não foram movidos)

1.3 - Procedimentos de entrada e saída

Os procedimentos de entrada são controlados pelo procedimento LEE, que chama por sua vez CREALISTA (para um símbolo "("), CREANUMERO (para um número), FINDELISTA (para um símbolo ")"), CREATOMO (para um átomo) e CREASPECIAL (para um "$").

Os dois últimos procedimentos chamam INSERTATOMO para ou criar o átomo se ele não está presente no sistema, ou obter o apontador ao mesmo. Existe ainda o procedimento LEECH para ler um caráter.

É importante observar que, nesta implementação, enquanto uma S-expressão está sendo lida, a estrutura de lista está sendo construída como uma lista do tipo "threaded", com o último elemento apontando sempre ao nível anterior. Os "threads" são removidos quando a expressão está completa. A figura 4.7 mostra uma estrutura de lista construída após a leitura do átomo H da S-expressão

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

gações são feitas. Quando se sai da referida λ-expressão,os argumentos são re-ligados a seus valores prévios.

2 - ALGUMAS COMPARAÇÕES ENTRE AS DUAS IMPLEMENTAÇÕES

Serão feitas comparações sobre alguns aspectos das duas implementações que chamaremos de implementação Magidin e implementação IME.

2.1 - Representação de átomos

Um aspecto interessante é a quantidade de espaço gasto para representar um átomo na memória do computador,já que os átomos são os elementos constituintes de todas as S-expressões.

Na implementação Magidin, 3 palavras no mínimo são necessárias para um átomo e na implementação IME, 4 palavras. A fig.4.8 mostra um quadro contendo o número de palavras gastas para um átomo de n caracteres (os símbolos $\lfloor\ \rfloor$ representam "maior inteiro contido").

Nº DE CARACTERES DO ÁTOMO	n	1 a 5	6	7	8	9	10	11	12	13	14	15
Nº DE PALAVRAS IMPLEMENTAÇÃO MAGIDIN	$3+\left\lfloor\frac{n-1}{6}\right\rfloor$	3	3	4	4	4	4	4	4	5	5	5
Nº DE PALAVRAS IMPLEMENTAÇÃO IME	$4+2\left\lfloor\frac{n-1}{5}\right\rfloor$	4	6	6	6	6	6	8	8	8	8	8

Fig. 4.8

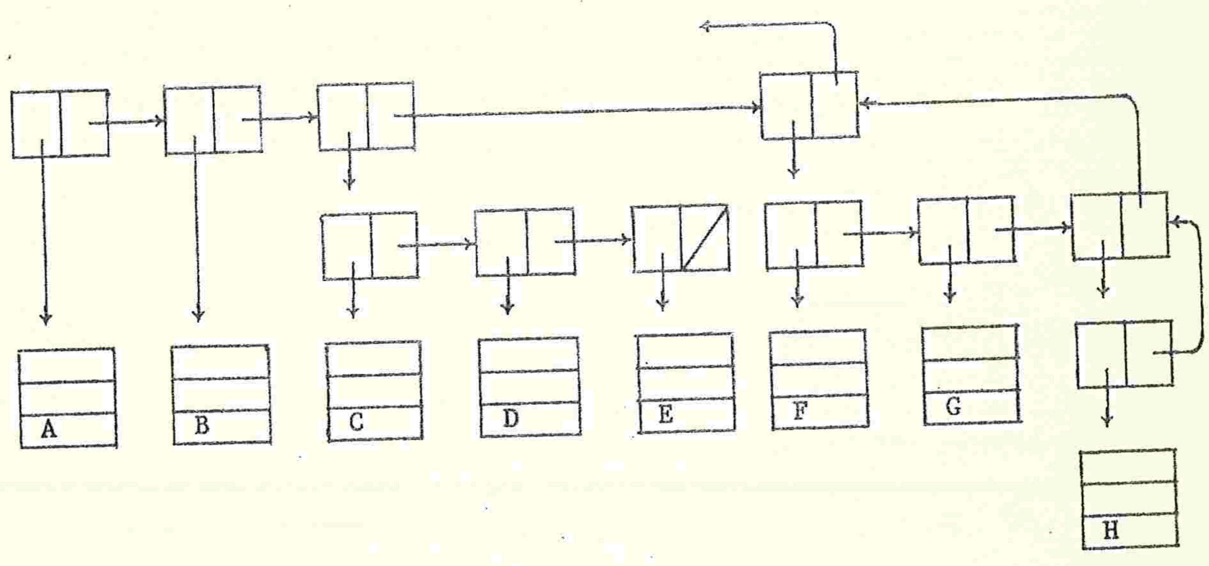


Fig. 4.7

Os procedimentos de saída são controlados por IMPR que chama PRINT para imprimir uma estrutura da lista, recursivamente. PRINT chama por sua vez PRINTNUM1 e PRINTNUM2 para números, PRINTATOM para átomos e PRINTARRAY para matrizes. Também fazem parte dos procedimentos de saída ESCRIBE para imprimir uma linha e IMPCH para imprimir um caráter.

1.4 - Procedimento de avaliação

A avaliação é feita basicamente pelo procedimento EVAL, que é auxiliado por diversos outros procedimentos.

O sistema armazena somente o último valor ligado a um argumento no campo VALOR do átomo [Mos70]. Quando uma λ-expressão é encontrada, os argumentos são avaliados, valores antigos ligados às λ-variáveis são salvos em ARGS e novas li

2.2 - Abreviatura de car's e cdr's

Na implementação Magidin, composições de car's e cdr's podem ser abrevidas até 4 níveis.Na implementação IME este número é ilimitado.

2.3 - Número de argumentos

Grande possibilidade é dada ao número de argumentos de uma função, na implementação Magidin, em que o usuário pode transmitir qualquer número de argumentos: argumentos faltantes são sempre supostos como NIL (mesmo em funções aritméticas), enquanto que argumentos extras são avaliados e ignorados. Na implementação IME o usuário deve fornecer o número exato de argumentos (para aquelas funções cujo número de argumentos é fixo): argumentos a mais ou a menos são considerados como erros e mensagens são acusadas.Do ponto de vista didático, essa solução nos parece mais adequada, permitindo localização sintática de erros devido a número incorreto de argumentos.

2.4 - Ligação de valores aos átomos

Na implementação Magidin, os valores dos átomos são ligados diretamente aos mesmos, tornando mais rápida a obtenção desses valores, enquanto que na implementação IME u-

ma lista de associação é usada para fazer a ligação dos valores aos átomos. Uma busca deve ser feita nesta lista toda vez que se deseja obter o valor de um átomo.

2.5 - Recuperação de células inativas

Nas duas implementações há a compactação das estruturas ativas. Os algoritmos usados, porém, são diferentes. Na implementação Magidin, todas as estruturas ativas são inicialmente marcadas numa primeira fase e após esta fase é que se inicia a movimentação de dados marcados para um outro espaço. O que se pode garantir após uma recuperação é que as estruturas ativas como um todo estarão compactadas, mas não cada estrutura ativa isolada. Como ilustração, sejam α, β e γ três estruturas ativas sem partes em comum. Temos:

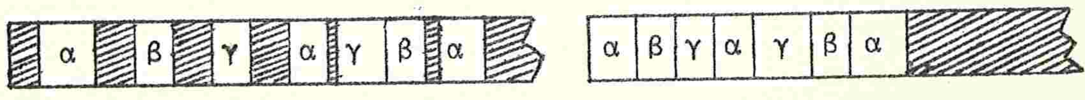


Fig. 4.9

Na implementação IME, não há a fase de marcação prévia. Cada estrutura ativa é movida, uma a uma, em um outro espaço de modo compactado. No exemplo acima, a situação após uma recuperação fica:

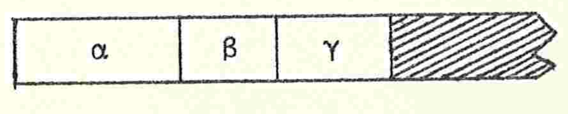


Fig. 4.10

3 - COMENTÁRIOS E CRÍTICAS

Serão feitos alguns comentários e críticas sobre o trabalho apresentado, assim como algumas propostas.

3.1 - Eliminação da notação do par com ponto

Para o ensino didático da linguagem LISP, é conveniente eliminar a notação do par com ponto. Proporemos a seguir algumas novas definições, como consequência desta eliminação.

a) Supõe-se a existência do conjunto A de átomos, do conjunto N, subconjunto de A, de átomos numéricos, e dos átomos T e NIL, pertencentes a A - N.

b) Define-se o conjunto S das S-expressões como sendo o menor conjunto tal que:

i) um átomo $\in S$

ii) se $u_1, u_2, \ldots, u_n \in S$ então $(u_1\ u_2\ \ldots\ u_n) \in S$

$(u_1\ u_2\ \ldots\ u_n)$ é denominada de lista com n elementos.

c) Define-se o conjunto L' das listas não vazias como sendo

$L' = S - A$, ou seja

$L' = \{x \in S \mid x = (u_1\ u_2\ \ldots\ u_n),\ n \geq 1, u_1, u_2, \ldots, u_n \in S\}$

d) Define-se o conjunto L das listas como sendo

$$L = L' \cup \{NIL\}$$

NIL será denominada de lista vazia e será represen tada também por ().

e) Função car

$car : L' \longrightarrow S$

$car[x] = u_1$ se $x = (u_1\ u_2\ \dots\ u_n) \in L'$

f) Função cdr

$cdr : L' \longrightarrow L$

$cdr[x] = (u_2\ \dots\ u_n)$ se $x = (u_1\ u_2\ \dots\ u_n) \in L'$

Observação: pela notação acima, entende-se que se $n = 1$,

$$cdr[(u_1)] = ()\ \text{ou NIL}$$

g) Função cons

$cons : S \times L \longrightarrow L'$

$$cons[x;y] = (u_1\ u_2\ \dots\ u_n)\ \text{se} \begin{cases} x = u_1 \in S \\ y = (u_2\ \dots\ u_n) \in L \end{cases}$$

Observação: pela notação acima, entende-se que se $n = 1$,

$$cons[u_1;()] = (u_1)$$

3.2 - Meta-função val

No Capítulo I, introduzimos uma meta-função val pa ra definir a semântica da linguagem LISP. Uma crítica que e ventualmente pode ser feita é a seguinte: já que existe a função universal evalquote (ou também chamada de outros nomes, como função universal apply, de J.McCarthy [Mcc60]) que

pode ser usada para definir a semântica da linguagem, para que introduzir a meta-função val?

Vamos justificar a introdução de val por meio dos seguintes argumentos: Uma pessoa, através da meta-função val, consegue entender com maior facilidade a semântica da linguagem do que pela função universal, que é mais adequada para uma máquina. J.McCarthy comenta que calcular os valores de funções usando apply é uma atividade mais apropriada a computadores eletrônicos que a pessoas humanas. Um outro ponto é que a meta-função val é definida num nível de abstração mais elevado do que a função universal que, além de depender da forma da representação da linguagem externa, também depende de certas particularidades do interpretador. Por exemplo, na meta-função val, a avaliação de λ-expressões é baseada apenas em conceitos de ocorrências livre e ligada de identificadores e do operador de substituição, ao passo que, para avaliar uma λ-expressão, a definição da função universal pode variar conforme a maneira como os valores são ligados às λ-variáveis: a função universal tem uma definição se é usada lista de associação e tem uma definição diferente se os valores são ligados diretamente aos átomos (como no interpretador usado por Magidin).

Neste trabalho, quando dissemos na seção 5.1 do Capítulo II que a função universal evalquote é uma função tal que:

$$\text{evalquote}[f;x]$$

e

$$\underline{\text{val}}\{\phi[\xi_1;\ldots;\xi_n]\}$$

são iguais quando definidos, não foi provada esta afirmação. Este é um ponto criticável e fica como proposta a demonstração desta afirmação.

3.3 - Uso do ALGOL na implementação

O uso do ALGOL para fazer a implementação faz com que a codificação de funções LISP em procedimentos ALGOL fique bastante simples e natural. Na maioria dos casos, quando não há problema de eficiência, a transformação de uma função LISP (expressa em meta-linguagem) para um procedimento ALGOL pode ser feita de maneira quase mecânica. Certos cuidados, entretanto, devem ser tomados devido à presença do recuperador de células inativas. Para evitar que estruturas ativas sejam destruídas por essa rotina, as mesmas devem ser salvas. Essa tarefa requer a possibilidade de se ter acesso à pilha do ALGOL. A implementação Magidin utilizou rotinas escritas em ESPOL que permitem tal acesso. A implementação IME, escrita só em ALGOL, foi obrigada a usar uma pilha própria para armazenamento de argumentos.

Uma restrição encontrada é a impossibilidade de se usar matrizes uni-dimensionais com mais de $2^{16}-1$ (≈65500) elementos. Esta limitação é inerente ao sistema B-6700. O ta

manho do espaço livre fica portanto limitado. Uma solução que pode ser proposta é a utilização de matrizes bi-dimensionais, já que a limitação de 65500 é para cada um dos índices. Isto requer alguns bits adicionais para representar o endereço de cada célula e rotinas um pouco mais complexas para ter acesso às mesmas. A sua adoção só se justifica se há real necessidade de um maior espaço de armazenamento livre.

3.4 - O interpretador

As definições usadas neste trabalho das funções evalquote, apply e eval foram obtidas do interpretador descrito por D. Ribbens [Rib69],no seu livro Programmation Non Numérique LISP 1.5. As definições de evalquote e eval, apresentadas na página 54 do referido livro, contêm dois erros que indicaremos abaixo:

1) A definição de evalquote, apresentada como:

```
evalquote[x;y] = [get[x;FEXPR]→eval[cons[x;y];NIL];
                  get[x;FSUBR]→eval[cons[x;y];NIL];
                  T→apply[x;y;NIL]]
```

deve ter a forma eval[cons[x;y];NIL] nas duas primeiras linhas acima alteradas para

```
eval[cons[x;evalq[y]];NIL]
```

onde:

```
evalq[y] = [null[y]→NIL;
                T→cons[list[QUOTE;car[y]];evalq[cdr[y]]]].
```

2) A definição de eval, cujas 4 primeiras linhas transcrevemos abaixo:

```
eval[x;a] = [null[x]→NIL;
              atom[x]→[get[x;APVAL]→car[get[x;APVAL]];
                       T→cdr[sassoc[x;a;λ[[];error[A8]]]]];
              numberp[x]→x;
                   .
                   .
                   .   ]
```

deve ter a 4ª linha, numberp[x]→x, colocada antes da 2ª linha, atom[x]→[...], uma vez que, se x é átomo numérico

atom[x]

é sempre verdadeiro.

- □ -

APÊNDICE I

LISTA DE FUNÇÕES

NOME DA FUNÇÃO	NÚMERO DE ARGUMENTOS	TIPO
ABSVAL	1	SUBR
ADD1	1	SUBR
AND	qualquer	FSUBR
APPEND	2	SUBR
APPLY	3	SUBR
ARCCOS	1	SUBR
ARCSIN	1	SUBR
ASSOC	2	SUBR
ATOM	1	SUBR
CAR	1	SUBR
CDR	1	SUBR
COND	qualquer	FSUBR
CONS	2	SUBR
COS	1	SUBR
COSH	1	SUBR
CSET	2	SUBR
CSETQ	2	FSUBR
DEFINE	1	SUBR
DIFFERENCE	2	SUBR
DIVIDE	2	SUBR
DPRINT	1	SUBR

NOME DA FUNÇÃO	NÚMERO DE ARGUMENTOS	TIPO
DREAD	0	SUBR
DREADCH	0	SUBR
ENTIER	1	SUBR
EQ	2	SUBR
EQUAL	2	SUBR
EVAL	2	SUBR
EVCON	2	SUBR
EVLIS	2	SUBR
EXP	1	SUBR
EXPT	2	SUBR
FUNCTION	1	FSUBR
GARBAGE	0	SUBR
GET	2	SUBR
GO	1	FSUBR
GREATERP	2	SUBR
LAST	1	SUBR
LENGTH	1	SUBR
LESSP	2	SUBR
LIST	qualquer	FSUBR
LN	1	SUBR
LOG	1	SUBR
MAPCAR	2	SUBR
MAPLIST	2	SUBR
MAX	qualquer	FSUBR
MEMBER	2	SUBR
MIN	qualquer	FSUBR
MINUS	1	SUBR
MINUSP	1	SUBR
NCONC	2	SUBR
NOT	1	SUBR

NOME DA FUNÇÃO	NÚMERO DE ARGUMENTOS	TIPO
NULL	1	SUBR
NUMBERP	1	SUBR
ONEP	1	SUBR
OR	qualquer	FSUBR
PAIR	2	SUBR
PLUS	qualquer	FSUBR
PRINT	1	SUBR
PROG	2	FSUBR
QUOTE	1	FSUBR
QUOTIENT	2	SUBR
READ	0	SUBR
READCH	0	SUBR
RECIP	1	SUBR
REMAINDER	2	SUBR
RETURN	1	SUBR
RPLACA	2	SUBR
RPLACD	2	SUBR
SASSOC	3	SUBR
SET	2	SUBR
SETQ	2	FSUBR
SIN	1	SUBR
SINH	1	SUBR
SQRT	1	SUBR
SUB1	1	SUBR
TAN	1	SUBR
TANH	1	SUBR
TERPRI	0	SUBR
TIMES	qualquer	FSUBR
ZEROP	1	SUBR

APÊNDICE II

MENSAGENS DE ERRO DO SISTEMA LISP

0 - FUNCTION 'LN' OR 'LOG' CANNOT TAKE ARGUMENT = OR < 0

1 - S-EXPRESSION CANNOT BEGIN WITH A PERIOD '.'

2 - INVALID FORMATION OF DOT NOTATION

3 - MORE THAN 120 CHARACTERS IN AN ATOM

4 - MISSING PROPRIETY PNAME (PRINT NAME)

5 - ARGUMENT OF FUNCTION 'LAST' CANNOT BE ATOMIC

6 - UNDEFINED VALUE OF FUNCTION 'GET'

7 - ARGUMENTS OF ARITHMETIC PREDICATE MUST BE NUMERIC

8 - ARGUMENTS OF ARITHMETIC FUNCTION MUST BE NUMERIC

9 - DIVISION BY ZERO

10 - NO VALUE FOR CONDITIONAL EXPRESSION 'COND'

11 - THE FOLLOWING FUNCTION IS UNDEFINED: <function name>

12 - CELLS OF TYPE APVAL OR SUBR CANNOT BE PRINTED

13 - WRONG NUMBER OF ARGUMENTS FOR THE FOLLOWING FUNCTION: <function name>

14 - 1ST ARGUMENT OF 'RPLACA' OR 'RPLACD' CANNOT BE ATOMIC

15 - INVALID FORMATION OF ARGUMENTS FOR 'DEFINE'

16 - MISSING ARGUMENTS OF 'SELECT'

17 - FIRST ARGUMENT OF 'CSET' MUST HAVE ATOMIC VALUE

18 - INVALID ARGUMENT OF 'CSETQ'

19 - ARGUMENTS OF 'APPEND' MUST BE LISTS

20 - FIRST ARGUMENT OF 'NCONC' MUST BE A LIST

21 - NO DATA FOR 'READ'

22 - SECOND ARGUMENT OF 'MEMBER' MUST BE A LIST

23 - INV ARGS FOR 'PAIR' OR WRONG # OF ARGS OF LAMBDA FUNC.

24 - 'CAR' CANNOT TAKE THE FOLLOWING ATOMIC ARGUMENT:
<variable name>

25 - 'CDR' CANNOT TAKE THE FOLLOWING ATOMIC ARGUMENT:
<variable name>

26 - INVALID ARGUMENTS FOR 'PROG'

27 - 1ST ARGUMENT OF 'SETQ' OR 'SET' MUST HAVE ATOMIC VALUE

28 - 'SETQ' OR 'SET' GIVEN ON FOLLOWING UNDEFINED VARIABLE:
<variable name>

29 - 'GO' MUST REFER TO AN ATOMIC SYMBOL

30 - 'GO' REFERS TO A NAME NOT LABELLED

31 - THE FOLLOWING VARIABLE IS UNDEFINED: <variable name>

32 - INVALID FORMATION OF CONDITIONAL EXPRESSION

33 - INVALID FORMATION OF LAMBDA EXPRESSION

34 - INVALID FORMATION OF LABEL EXPRESSION

35 - WRONG NUMBER OF ARGUMENTS OF 'SETQ' OR 'SET'

36 - SECOND ARGUMENT OF 'SASSOC' IS INVALID

37 - ARGUMENT OF 'LENGTH' MUST BE A LIST

38 - NO DATA FOR 'READCH' OR 'DREADCH'

REFERÊNCIAS BIBLIOGRÁFICAS

[Bae72] BAECKER, H.D.
Garbage collection for virtual memory computer systems
Comm. ACM 15, 11 (Nov. 1972)

[Bur72-1] Burroughs B6700/B7700
Extended ALGOL Compiler
Form 5000136 (June 1972)

[Bur72-2] Burroughs B6700/B7700
Extended ALGOL Language
Form 5000128 (june 1972)

[Bur72-3] Burroughs B6700/B7700
Executive System Programming Language (ESPOL)
Form 5000094 (June 1972)

[Bur74] Burroughs B6700/B7700
ALGOL Language Reference Manual
Form 5000649 (May 1974)

[Che70] CHENEY, C.J.
A nonrecursive list compacting algorithm
Comm. ACM 13, 11 (Nov. 1970)

[Dia74] DÍAZ, M.
Las funciones definidas en el sistema LISP B-6700
Public. Nº 5 (1974)
CIMAS, Universidad Autonoma Metropolitana, Mexico

[Fen69] FENICHEL, R.R., YOCHELSON, J.C.
A LISP garbage collector for virtual-memory computer systems
Comm. ACM 12, 11 (Nov. 1969)

[Fis75] FISHER, D.A.
Copying cyclic list structures in linear time using bounded space
Comm. ACM 18, 5 (May 1975)

[Knu68] KNUTH, D.E.
The Art of Computer Programming, Vol. I
Addison-Wesley, Reading, Mass. (1968)

[Lin74] LINDSTROM, G.
Copying list structures using bounded workspace
Comm. ACM 17, 4 (Apr. 1974)

[Mag74-1] MAGIDIN, M., SEGOVIA, R.
Implementation of LISP 1.6 on the B-6700 Computer
Comunicaciones Técnicas, Vol. V, Nº 70 (1974)
CIMAS, Universidad Autonoma Metropolitana, México

[Mag74-2] MAGIDIN, M., SEGOVIA, R., DÍAZ, M.
Manual del Sistema LISP B-6700
Public. Nº 4 (1974)
CIMAS, Universidad Autonoma Metropolitana, México

[Mcc60] McCARTHY, J.
Recursive functions of symbolic expressions and their computation by machine, Part I
Comm. ACM 3, 4 (Apr. 1960)

[Mcc62] McCARTHY, J. et al.
LISP 1.5 Programmers' Manual
The M.I.T. Press, Cambridge, Mass. (1962)

[Mos70] MOSES, J.
The function of FUNCTION in LISP
SICSAM Bulletin 15 (Jul. 1970)

[Rei73] REINGOLD, E.M.
A nonrecursive list moving algorithm
Comm. ACM 16, 5 (May 1973)

[Rib69] RIBBENS, D.
Programmation Non Numérique LISP 1.5
Dunod, Paris (1969)

[Sch67] SCHORR, H., WAITE, W.M.
An efficiente machine-independent procedure for garbage collection in various list structures
Comm. ACM 10, 8 (Aug. 1967)

[Wai73] WAITE, W.M.
Implementing Software for Non-Numeric Applications
Prentice-Hall Inc., Englewood Cliffs, N.J. (1973)

REFERÊNCIAS BIBLIOGRÁFICAS ADICIONAIS

[Bob64] BOBROW, D.G., RAPHAEL, B.
A comparison of list processing computer languages
Comm. ACM 7, 4 (Apr. 1964)

[Bob66] BOBROW, D.G. et al.
The BBN-LISP System
Bolt, Beranck and Newman, Inc.
Cambridge, Mass. (1966)

[Bob67] BOBROW, D.G., MURPHY, D.L.
Structure of a LISP system using two-level storage
Comm. ACM 10, 3 (Mar. 1967)

[Bob68] BOBROW, D.G., MURPHY, D.L.
A note on the efficiency of a LISP computation in a paged machine
Comm. ACM 11, 8 (Aug. 1968)

[Chu41] CHURCH, A.
The Calculi of Lambda Conversion
Annals of Mathematics Studies nº 6
Princeton University Press, Princeton, N.J. (1941)

[Fos67] FOSTER, J.M.
List Processing
MacDonald, London (1967)

[Jor73] JORDAN, C.R.
A note on LISP universal S-functions
The Computer Journal 16, 2 (May 1973)

[Lan64] LANDIN, P.J.
The mechanical evaluation of expressions
The Computer Journal 6, 4 (Jan. 1964)

[Lan65-1] LANDIN, P.J.
A correspondence between ALGOL 60 and Church's
Lambda-notation: Part I
Comm.ACM 8, 2 (Feb. 1965)

[Lan65-2] LANDIN, P.J.
A correspondence between ALGOL 60 and Church's
Lambda-notation: Part II
Comm. ACM 8, 3 (Mar. 1965)

[Lis69] LISP Bulletin
SIGPLAN Notices, Vol. 4, Nº 9 (Sep. 1969)

[Lis72] LISP/360 Reference Manual
Stanford Computation Center
Stanford University, Stanford, Calif.
Document number Scc 024, 4th Edition (Mar. 1972)

[Mau72] MAURER, W.D.
The Programmer's Introduction to LISP
American Elsevier Inc., New York, N.Y. (1972)

[Qua69] QUAM, L.H.
Stanford LISP 1.6 Manual
Stanford Artificial Intelligence Project
SAILON 28.3 (Sep. 1969)

[Roc71] ROCHFELD, A.
New LISP techniques for a paging environment
Comm. ACM 14, 12 (Dec. 1971)

[Ros67] ROSEN, S.
Programming Systems and Languages
McGraw-Hill, New York, N.Y. (1967)

[Weg68] WEGNER, P.
Programming Languages, Information Structures and Machine Organization
McGraw-Hill, New York, N.Y. (1968)

[Wei67] WEISSMAN, C.
LISP 1.5 Primer
Dickenson Publish. Co., Belmont, Calif. (1967)

[Woz69] WOZENCRAFT, J.M., EVANS, A.
Notes on Programming Linguistics
M.I.T., Cambridge, Mass. (1969)