ALGUNS TÓPICOS DE COMPILAÇÃO E UMA IMPLEMENTAÇÃO DA LINGUAGEM LAPA PARA O COMPUTADOR PADE.

INÊS S. HOMEM DE MELO

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 W. SETZER

São Paulo, setembro de 1978

aos meus pais

e ao Paulo

AGRADECIMENTOS

Ao Prof. Dr. Valdemar W. Setzer, pela sua dedicação desde o início de minha formação acadêmica. Seu interesse e incentivo constante foram significativos na realização deste trabalho, tendo sido muito valiosa a sua contribuição como orientador e amigo;

Aos amigos do Departamento de Matemática Aplicada do IME-USP e do Setor de Matemática Aplicada do IFUSP pelo apoio e estímulo durante a realização deste trabalho, e em particular à Graça Bressan, pelas discussões proveitosas;

Ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e à IBM, pelo auxílio que me concederam;

À Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) pelo suporte concedido ao projeto PADE;

À Srta. Regina Helena pelo excelente trabalho de datilografia.

São Paulo, setembro de 1978
I.S.H.M.

ÍNDICE

		Pág.
INTRODUÇÃO		1
CAPÍTULO I	- PROCESSAMENTO DE BLOCOS E PROCEDIMENTOS	3
	1. Introdução	3
	2. Tratamento de ativação e desativação de bloco	5
	3. Outras propostas e características distintas	14
	4. Parâmetros	22
	Referências bibliográficas	32
CAPÍTULO II	- MANIPULAÇÃO DE RECORDS	35
	1. Introdução	35
	2. Uma proposta que se adapta ao ALGOL 60	36
	3. COBOL e PL/I	46
	4. PASCAL	62
	5. ALGOL W	66
	6. Outras propostas	71
	7. Comentários finais	71
	Referências bibliográficas	73
CAPÍTULO III	- MATRIZES	75
	1. Introdução	75
	2. Matrizes estáticas e matrizes dinâmicas	76
	3. Outras propostas	87
	Referências bibliográficas	91

Pág.

CAPÍTULO IV - IMPLEMENTAÇÃO DA LINGUAGEM "LAPA". 93

1. Introdução. 93
2. O compilador. 94
3. Procedimentos do compilador 96
4. Catálogo de compilação (Código objeto gerado) 114
5. Algumas justificativas sobre o código objeto gerado. 144
6. Exemplo. 151

Referências bibliográficas. 159

CAPÍTULO V - COMENTÁRIOS GERAIS E CONCLUSÕES. 161

1. Matrizes. 161
2. Constantes. 162
3. Comandos for, case, expressões case e index encaixados. 162
4. Considerações sobre a arquitetura do PADE.. 163
5. Interação dos grupos de "software" e "hardware". 166
6. Conclusões. 167

APÊNDICE 1 - DIAGRAMA SINTÁTICO DA LAPA 169

APÊNDICE 2 - DESCRIÇÃO DA LIMPA. 175

INTRODUÇÃO

O objetivo deste trabalho é apresentar a implementação da linguagem LAPA. Essa linguagem foi projetada no Departamento de Matemática Aplicada do IME-USP, para o computador PADE, projetado e desenvolvido no Instituto de Física da USP.

Para a implementação dessa linguagem fizemos um levantamento bibliográfico dos aspectos mais críticos de um compilador. Desse levantamento resultou o capítulo I que trata do estudo de processamento de blocos e procedimentos; o capítulo II que trata do estudo de manipulação de *records*; o capítulo III que trata do estudo de manipulação de matrizes.

No capítulo IV descrevemos o compilador LAPA, o código objeto gerado para as diversas construções existentes na linguagem e também algumas justificativas do código objeto gerado (quando este não é feito da maneira convencional). Damos também um exemplo de um programa em LAPA e o respectivo código gerado pelo compilador.

Como conclusão deste trabalho, apresentamos no capítulo V alguns aspectos que podem ser melhorados no compilador.

O apêndice 1 é de autoria de G. Bressan e apresenta a descrição da linguagem LAPA sequencial através de diagramas sintáticos.

O apêndice 2 é um resumo que fizemos da descrição da LIMPA, descrição esta de autoria de C. Mammana. Nesse resumo apresentamos apenas as instruções da LIMPA usadas pelo compilador LAPA.

Como notação, sublinhamos as palavras reservadas da LAPA. Os itens dentro de cada capítulo são numerados a partir de 1. Ao referirmos a um item,1.2.4 por exemplo, fica subentendido tratar-se do item 1.2.4 do capítulo em questão. No caso de uma referência a um item de um outro capítulo, mencionaremos explici

tamente o número do capítulo, seguido do número do item (por exemplo, I.2.1).

No capítulo IV, ao apresentarmos a sintaxe de cada construção, colocaremos os não terminais entre os símbolos "<" e ">".

CAPÍTULO I

PROCESSAMENTO DE BLOCOS E PROCEDIMENTOS

1. INTRODUÇÃO

A linguagem ALGOL 60 permite matrizes com limites variáveis e chamadas recursivas a procedimentos; isso impede qualquer tentativa de alocação estática de memória, durante a execução do programa objeto. É necessário um esquema de alocação dinâmica pois, por exemplo, a uma variável declarada em um procedimento recursivo, pode corresponder mais do que um valor, num mesmo instante durante a execução. Nesse caso, apenas o valor associado à última chamada do procedimento pode ser usado.

Denominaremos de *ativação* de um bloco ao início de seu processamento, e *desativação* ao correspondente término. Diremos também que um bloco ou procedimento *foi ativado*, quando já foi feita sua ativação, mas ainda não foi feita sua desativação.

Segundo Wichmann /22/ a *profundidade de um bloco ou procedimento* é definida como:

a) a profundidade do bloco mais externo do programa é zero

b) a profundidade de um bloco ou procedimento qualquer é n se ele está declarado num bloco ou procedimento de profundidade $n-1$.

As variáveis de um programa em ALGOL só podem ser declaradas em blocos ou procedimentos, sendo que neste último caso referimo-nos aos parâmetros formais. Na medida que o tratamento de blocos e procedimentos for o mesmo, faremos referência somente ao primeiro.

O processamento de blocos segue um esquema de pilha , ou seja, o último bloco que foi ativado é o primeiro a ser desa-

:ivado. Isso também se aplica se, devido à execução de um go to para um bloco de profundidade muito menor que o atual, deve-se retirar vários blocos da pilha. Esse caso pode ser encarado como a retirada inicial do último bloco da pilha, seguida da retirada do penúltimo e assim por diante até chegar-se ao nível desejado. Por essa razão usa-se uma pilha para fazer a reserva de áreas para as variáveis declaradas localmente e parâmetros.

O tratamento da pilha é feito por blocos, não havendo em geral manipulação da mesma para variáveis individuais, a menos do caso de computadores com "estrutura de pilha"; nesses computadores, as operações são feitas em operandos que são previamente colocados no topo da pilha onde, nesse caso, são também colocadas as variáveis temporárias.

Em qualquer ponto durante a execução do programa objeto a pilha deve conter, no mínimo, o espaço de memória para as variáveis de cada bloco que foi ativado.

Diremos que uma variável está *disponível* se o bloco onde ela foi declarada foi ativado, e além disso pudermos ter acesso à variável. Assim, se tivermos um procedimento recursivo, uma mesma variável pode ter sido ativada diversas vezes, mas só estará disponível o valor correspondente à última ativação. Diremos também que um bloco está disponível, se suas variáveis estão disponíveis.

Conforme Rohl /18/ denominaremos *registro de ativação* de um bloco ao espaço da pilha contendo todas as variáveis e informações relativas a um determinado bloco, que devem ser guardadas na pilha. A *base do registro de ativação*, ou simplesmente *base*, será o apontador para a primeira posição de um desses registros.

As variáveis são declaradas em blocos ou procedimentos e seu endereçamento é feito relativamente à base do registro de ativação correspondente. Assim, ao endereço de uma variável pode ser associado um par ordenado (n,p), onde *n* é a profundidade

do bloco ou procedimento e p é o deslocamento dentro do mesmo, isto é, o endereço dessa variável em relação à base.

2. TRATAMENTO DE ATIVAÇÃO E DESATIVAÇÃO DE BLOCOS

Veremos agora um esquema de tratamento de ativação e desativação de blocos em ALGOL 60. Esse esquema foi proposto por Randell e Russell /17/, baseado em propôsta de Dijkstra /1/.

Uma maneira de se construir a pilha com os registros de ativação, é colocar no início de cada registro, um conjunto de informações que denominaremos de *marca de ativação* ou simplesmente *marca*. Para cada bloco de profundidade i, a marca consiste dos seguintes campos:

a) *apontador estático* - aponta para o início do registro de ativação do bloco ativo de profundidade $i-1$, onde o bloco atual foi declarado. Se considerarmos cada apontador estático como o início de uma cadeia de ligações que pode ser percorrida a partir dele (*cadeia estática* daquele bloco), conseguiremos acesso a todas variáveis disponíveis nesse ponto do processamento.

b) *apontador dinâmico* - aponta para o início do registro de ativação do bloco imediatamente anterior na pilha, ou seja, o último bloco que foi ativado antes de ser ativado o atual. O conjunto dos apontadores dinâmicos forma a cadeia dinâmica. Através dessa cadeia conseguimos acesso a todos os blocos que foram sucessivamente ativados.

c) profundidade do nível corrente (i)

d) estado da computação, isto é, posição para armazenar o apontador para o topo da pilha nesse bloco, valores dos registradores, endereço de retorno (para procedimentos), indicadores de condição, etc..

Essas informações são colocadas na pilha. Em seguida a

elas, colocam-se os parâmetros ou as variáveis locais. Os parâmetros são variáveis locais a um procedimento e que constituem o bloco mais externo do procedimento.

Por razões de eficiência, os apontadores da parte disponível da cadeia estática, em um determinado instante, podem ser duplicados num conjunto de registradores que chamaremos de *pilha de bases* (denominado por Dijkstra /1/ de *display*). Em um determinado instante do processamento, cada elemento da pilha de bases contém a base do registro de ativação de cada bloco disponível nesse momento. Assim, quando se faz referência a uma variável *global* , ou seja, uma variável declarada num bloco de profundidade menor que a do bloco atual, se consultarmos o elemento apropriado da pilha de bases, teremos diretamente a base do bloco em questão. Isso evita percorrer a cadeia estática a cada referência a uma variável. A consulta referida é muitas vezes feita automaticamente por "hardware" a partir do par (n,p).

Usando a pilha de bases, a qual nos algoritmos que se seguem será associado o identificador PILBAS, o acesso a uma variável de endereço (n,p) é dado por PILBAS [N] + P.

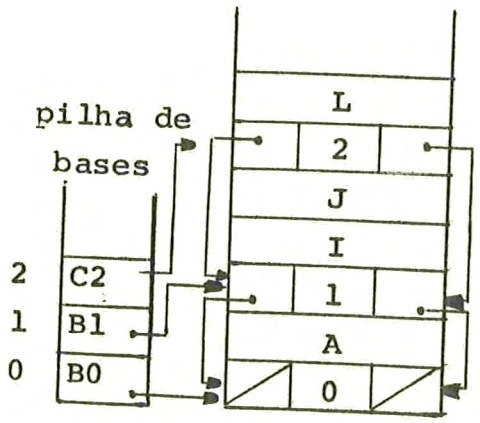
2.1 - Exemplo

O exemplo dado a seguir é baseado num semelhante , descrito por Randell e Russell /17/.

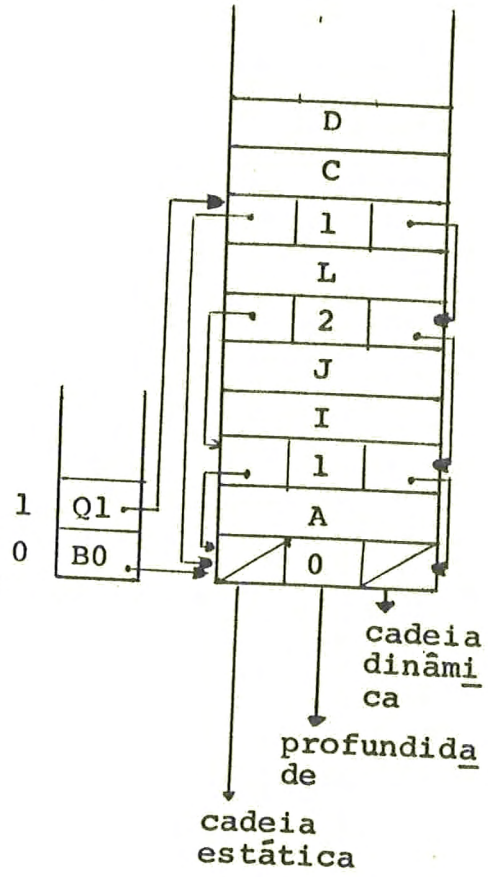
Na figura que se segue, esquematizamos o estado da pilha em momentos hipotéticos. Note-se que os procedimentos não tem parâmetros, de modo que suas variáveis locais são apenas as de seus blocos. A ligação vazia é representada por uma célula com um traço diagonal. Para simplificar, não indicamos o estado da computação na marca de cada bloco.

```
B0: begin real A;
      procedure P1,
        begin real B,C;
            :
        B2: begin real A;
              procedure P3;
                  begin real F,G;
                      :
                  L:...
                      :
                  end P3;
            B3: begin real H;
                      :
                  M:...
                      :
                    P3;
                end;
            end;
      end P1;
  procedure Q1;
    begin real C,D;
        :
    V:...
        :
    end Q1;
```

a) no rótulo T temos:

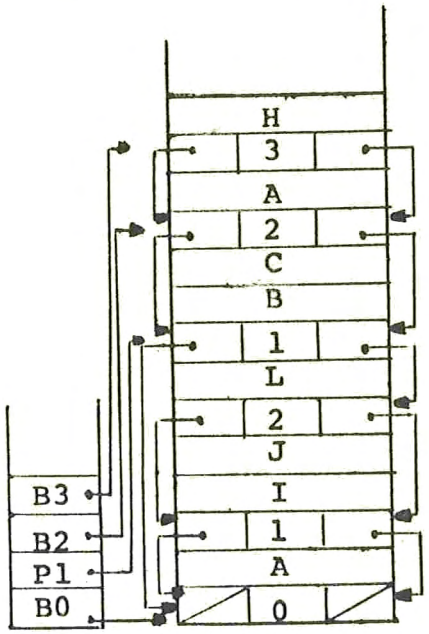


b) no rótulo V temos:

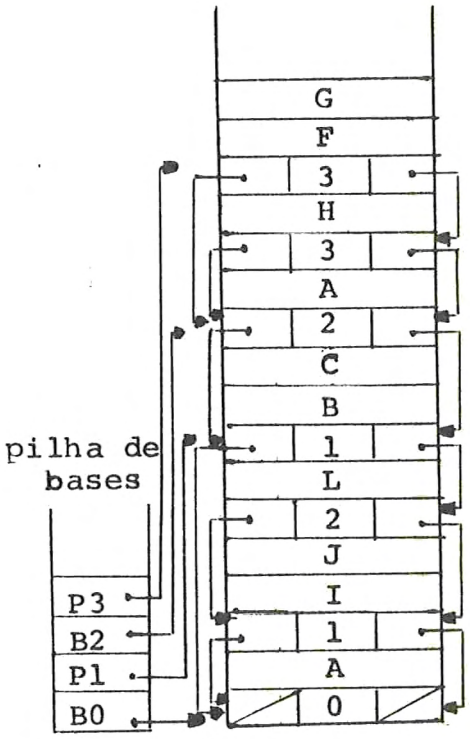


```
B1: begin real I,J;
      :
    C2: begin real L;
          :
        T: ...
           Q1;
          :
        N: ...
           P1;
          :
        end C2;
    end B1;
end B0;
```

c) no rótulo M temos;



d) no rótulo L temos:



.2.2 Algoritmos de manipulação da pilha

Nos algoritmos para tratamento de ativação e desativação de blocos, além dos apontadores e das duas pilhas (PILHA e PILBAS) mencionados no item anterior, vamos usar as seguintes variáveis:

a) AP - apontador para o topo da PILHA

b) TP - apontador para o topo da PILBAS (PILBAS [TP] é a base do registro de ativação do bloco que está sendo executado)

c) CI - apontador para a instrução corrente.

Devido à compactação empregada, usaremos a notação P.ESTAT, P.DINAM, P.PROF para referirmo-nos respectivamente aos apontadores da cadeia estática e da cadeia dinâmica e à profundidade do bloco armazenadas na variável P; também usaremos a notação Q.TOPO e Q.ENDV para referirmo-nos respectivamente ao apontador para o topo da pilha e endereço de volta, armazenados na variável Q.

Nos algoritmos que se seguem será reservada uma posição da pilha para cada variável simples e uma para cada matriz. A área para conter elementos de matrizes fica após a área de variáveis simples (ver III.3.2). Vamos supor também que, se o bloco é um procedimento, este não é parâmetro de outro procedimento. No item 4.2.7 comentaremos o tratamento de procedimentos como parâmetros.

2.2.1 - Ativação de um bloco de profundidade N, com L células para variáveis locais e parâmetros e com as instruções começando no endereço A da memória, supondo que o bloco não é parâmetro.

Este algoritmo é dividido em duas etapas. Na primei-

ra etapa é feito o cálculo dos parâmetros, o armazenamento do endereço de retorno e o desvio para o endereço A. A segunda etapa, que é gerada antes das instruções do bloco (evitando-se assim duplicação de código objeto), armazena os apontadores estático e dinâmico e a profundidade, atualiza o topo da pilha e atualiza a pilha de bases e seu topo.

Primeira etapa:

```
<<instruções para calcular os parâmetros>>
PILHA [AP+2] := (CI, "estado da computação");
    <<endereço de retorno e estado da computação ( compacta-
                                                    tos)>>

CI:= A; <<desvia para o bloco>>
```

Segunda etapa:

```
PILHA [AP+1]. ESTAT := PILBAS [N-1]; <<cadeia estática>>
PILHA [AP+1]. PROF := N;  <<profundidade>>
PILHA [AP+1]. DINAM := PILBAS [TP] ; <<cadeia dinâmica>>
PILBAS [N] := AP+1;  <<atualiza  PILBAS>>
TP := N; <<atualiza topo da PILBAS>>
AP := AP+L+2;  <<atualiza topo da PILHA; 2 células são para a
                                                    marca>>
```

Na realidade, Randell e Russell /17/ propõem dois esquemas, um para blocos e um para procedimentos; mas os dois esquemas são bastante análogos e podem ser compactados em apenas um.

2.2.2 - Desativação de bloco (normal, ou seja, pelo end)

```
CADEST:= PILHA [PILBAS [TP]]. DINAM;
APROF:= PILHA [PILBAS [TP]].PROF;
CADINAM:= PILHA [PILBAS [TP]].DINAM;
```

```
NOVPROF:= PILHA [CADINAM] . PROF; I:= NOVPROF;
if NOVPROF gtr APROF then
    <<atualização da PILBAS quando o bloco chamador tem profun
      didade maior que o bloco que está sendo deixado>>
   for I:= NOVPROF step -1 until APROF do
       begin
       PILBAS [I]:= CADEST;
       CADEST:= PILHA [CADEST] . ESTAT
       end;
<<atualização desde o nível que está sendo deixado até que a
  PILBAS e a cadeia estática coincidam>>
while  PILBAS [I] neq CADEST do
   begin
   PILBAS [I] := CADEST;
   CADEST:= PILHA [CADEST] . ESTAT;
   I:= I-1
   end;
AP:= PILBAS [TP] -1; <<atualiza topo da PILHA>>
TP:= NOVPROF; <<atualiza topo da PILBAS>>
CI:= PILHA [AP+2]. ENDV; <<endereço de volta>>
```

2.2.3 - go to para uma instrução de endereço ENDE, num bloco de profundidade N, supondo que ENDE não é parâmetro.

```
if TP = N <<TP indica a profundidade atual>>
   then  CI:= ENDE
   else begin
        TP:= N;
        AP:= PILHA [PILBAS [TP]+1].TOPO;<<atualiza topo
                                          da PILHA>>
        CI:= ENDE
        end;
```

2.3 - Justificativa dos algoritmos

2.3.1 - Ativação de um bloco

Reserva-se espaço na PILHA para os parâmetros e variáveis locais, uma posição para cada variável simples e uma ou mais para cada matriz (dependendo se na PILHA é colocado o *dope vector* ou apenas seu apontador) (ver III.2.3). Também guardam-se as informações que serão necessárias para a desativação do bloco. O elemento da PILBAS cujo índice é a profundidade do bloco que está sendo ativado, é a base do registro de ativação daquele bloco. O apontador para o topo da pilha indica a última posição reservada para o bloco. Se o bloco for um procedimento que é parâmetro de um outro procedimento o esquema deve ser alterado (ver 4.2.7).

2.3.2 - Desativação normal de um bloco

Na saída normal de um bloco, pela execução de seu end, devemos retornar ao ponto do programa onde estávamos antes da chamada e também restaurar o estado da PILBAS. No exemplo do item 2.1 temos (base (X) indica a base do bloco X).

a) em P1 a PILBAS contém:

PILBAS [0]= base (B0)
PILBAS [1]= base (P1)

b) quando P1 é desativado, volta-se ao bloco em que ele foi ativado (C2) e a PILBAS passa a conter:

PILBAS [0]= base (B0)
PILBAS [1]= base (B1)
PILBAS [2]= base (C2)

c) quando C2 é desativado, basta suprimir o último elemento da PILBAS. Nesse caso ela passa a conter:

PILBAS [0]= base (B0)
PILBAS [1]= base (B1)

Para atualizar a PILBAS, percorre-se a cadeia estática, que está nos vários registros de ativação da pilha, no sentido do topo para o fundo, usando cada elemento da cadeia estática para atualizar o elemento correspondente da PILBAS. Teoricamente poderíamos parar o processo quando este atingir a profundidade em que o elemento da cadeia estática e a PILBAS coincidem. Entretanto, é necessário um cuidado adicional quando um bloco é desativado e o retorno é feito para outro bloco com profundidade maior. Neste caso, os elementos da PILBAS para profundidades maiores contém valores obsoletos e podemos, por acaso, ter uma coincidência. (no exemplo de 2.1, ao sair de Q1 temos PILBAS [2]= C2 e já encontramos a igualdade, mas ainda temos que atualizar PILBAS [1] senão teremos um erro).

Chegamos à conclusão que é necessário verificar a PILBAS desde a profundidade que se vai ficar, até a profundidade do procedimento que está sendo deixado e daí em diante até que haja a primeira coincidência entre uma célula da PILBAS e uma célula da cadeia estática. Wichmann /22/ sugere que em alguns casos a atualização de todos os elementos da PILBAS pode ser mais rápida do que verificar, da maneira descrita acima, quais os elementos que devem ser atualizados.

Depois da atualização da PILBAS, colocam-se os novos valores dos apontadores para o topo da PILHA e topo da PILBAS e desvia-se para o endereço de volta.

2.3.3 - Desvio através do go to

Se o desvio é feito para o mesmo bloco, apenas é necessário atualizar o apontador para a instrução atual.

Se o desvio é feito para um bloco de profundidade N, menor que a atual, e global a este (desvio para um bloco de pro-

fundidade maior só pode ser feito através de rótulos como parâmetros e isto é tratado em 4.2.5), a base do bloco desejado é exatamente a contida em PILBAS [N]. Nesse caso é necessário atualizar os apontadores para o topo da PILBAS e topo da PILHA e o apontador para a instrução atual. Note que dessa maneira podemos desativar vários blocos de uma vez, o que ocorre por exemplo, no caso de desvio para fora de um procedimento que foi ativado recursivamente várias vezes. Se o desvio é feito para um rótulo que é parâmetro de procedimento o esquema deve ser alterado (ver 4.2.5).

2.4 - Características adicionais

Segundo Gries /4/, Randell e Russell /17/ e Wichmann /22/, o espaço acima do último registro de ativação pode ser usado para as variáveis temporárias. Nesse caso, é necessário mais um apontador (WP) que indica a última posição da pilha usada por uma variável temporária. Na ativação de cada bloco, WP recebe o valor de AP naquele bloco.

Randell e Russell /17/ também sugerem que para um procedimento tipo função pode-se reservar as duas primeiras posições do registro de ativação correspondente para colocar-se o resultado da função.

Estudando as características do compilador Dask ALGOL /12/ verifica-se também que a cadeia estática não é necessária (ver 3.4). Guardando na marca apenas o apontador dinâmico e a profundidade do bloco, conseguimos resolver todos os problemas de ativação e desativação. A cadeia estática apenas torna a execução do programa mais rápida.

3 - OUTRAS PROPOSTAS E CARACTERÍSTICAS DISTINTAS

O esquema descrito no item 2 é basicamente aquele usado pela maioria dos compiladores, como os descritos por

Dijkstra /2/ e Gries, Paul e Wiehle/3/, ou seja, usam exatamente o esquema proposto por Randell e Russell /17/. Veremos agora características de vários compiladores contendo adaptações desse esquema e também outras propostas de pequenas modificações.

3.1 - Proposta devida a Gries /4/ e Wichmann /22/

Uma primeira proposta é usar os elementos da pilha de bases apenas para procedimentos. Todos os blocos dentro de um procedimento são endereçados através do mesmo elemento da pilha de bases. Esses blocos constituem agora um só registro de ativação.Gries /4/ e Wichmann /22/ propõem sistemas em que a área de matrizes é alocada no fim do registro de ativação; assim, matrizes não introduzem problemas adicionais. No item III.3.2 voltaremos a abordar a implementação de matrizes. Além disso, a pilha de bases pode ser colocada no início do registro de ativação de cada procedimento; com perda de eficiência pode-se então dispensar o conjunto especial de registradores. Nesse caso usamos apenas dois registradores: um como base do bloco de profundidade zero e um como base do procedimento que está sendo executado no momento, ao qual é associado o identificador BASEATUAL. Assim, uma variável de endereço (n,p) é localizada através de: CONTEUDO (BASEATUAL +N) + P.

Quando a pilha de bases é guardada no início do registro de ativação, o algoritmo de ativação de um procedimento é mais complexo pois deve copiar todos os elementos da pilha de bases anteriores ao bloco atual no referido registro. Por outro lado, o algoritmo de desativação é mais simples, pois não precisa atualizar os elementos da pilha de bases, já que eles estão no registro de ativação anterior.

3.2 - Computador B-6700

Conforme Hauck e Dent /6/ e Organick /16/, o B-6700

tem trinta e dois elementos na pilha de bases. Dois desses elementos são usados pela própria máquina, portanto podemos ter até 30 níveis de blocos e procedimentos encaixados (tanto os blocos quanto os procedimentos usam um elemento da pilha de bases cada um). Todas as variáveis simples, apontadores para matrizes (ver III.2.4) e temporários são colocados na pilha. Na marca de ativação de cada bloco ou procedimento o B-6700 também guarda a profundidade do bloco ou procedimento chamador, para facilitar a operação de desativação daquele bloco.

3.3 - Proposta devida a Irons e Feurzeig /11/. Deteção de chamadas recursivas segundo Huxtable e Hawkins /9/

Uma das primeiras idéias que surgiu para o tratamento de procedimentos recursivos, devida a Irons e Feurzeig /11/ foi deixar de colocar as variáveis locais na pilha. Cada variável recebe uma posição fixa de memória. Apenas quando houver uma ativação recursiva de um procedimento é que as variáveis locais e parâmetros correspondentes à ativação anterior são colocados na pilha. Dessa maneira, só são colocados na pilha os valores que precisam ser salvos, juntamente com todas as marcas de ativação (ver item 2). Os novos valores disponíveis são colocados nas posições de memória correspondentes. Assim, as variáveis de procedimentos e blocos que não estão envolvidas em recursão, são alocadas estaticamente, em tempo de compilação.

Segundo Rutishauser /19/ dizemos que ocorre *recursividade direta* quando dentro do procedimento X temos uma ativação do próprio X. Se ocorrer um encadeamento de ativações de dois ou mais procedimentos, de tal maneira que a última ativação seja feita ao primeiro procedimento da cadeia, ocorre *recursividade indireta*.

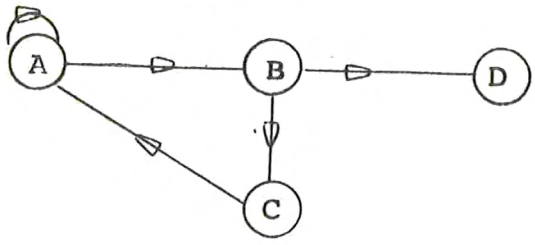
Quando a recursão é direta e algum *thunk* de parâmetro (ver 4.1.3)aponta para uma variável local ao próprio procedimento, o valor apontado deve ser o que está na pilha, correspondente à chamada anterior e não o valor disponível, associado à

profundidade corrente da recursão.

Para executar esse mecanismo precisamos saber quando a ativação de um procedimento é recursiva (direta ou indiretamente). Para isso Irons e Feurzeig /11/ propõem a construção, no programa objeto, de uma matriz C, contendo um elemento para cada procedimento existente no programa. Inicialmente todos os elementos da matriz são iguais a -1 (menos um). Cada vez que um procedimento PROC é ativado faz-se C[PROC]:=C[PROC]+1 e cada vez que é desativado faz-se C[PROC]:=C[PROC]-1.Toda vez que um procedimento P é ativado, verifica-se se C[P]> 0. Se isso acontecer a ativação é recursiva. Não é preciso ter um contador para cada bloco pois os blocos só são ativados recursivamente quando são declarados em procedimentos recursivos.

A recursão também pode ser detetada em tempo de compilaçãc, mas o processo é mais trabalhoso. Esse processo é descrito por Hawkins e Huxtable /9/ e consiste no seguinte. Constroí-se um grafo orientado onde cada nó corresponde a um procedimento do programa e existe uma aresta saindo do nó A e chegando ao nó B se e só se no procedimento A ocorre uma ativação direta do procedimento B. Baseado nesse grafo, constroí-se uma matriz booleana, onde as linhas e colunas correspondem aos procedimentos do programa. Se existe uma aresta saindo do nó A e chegando ao nó B do grafo, então o elemento da matriz cuja linha corresponde ao procedimento A e cuja coluna corresponde ao procedimento B, deve ter valor 1 (um). Os elementos da matriz para os quais não acontece essa condição devem ter valor zero.

Exemplo: se tivermos o grafo



a matriz correspondente será:

	A	B	C	D
A	1	1	0	0
B	0	0	1	1
C	1	0	0	0
D	0	0	0	0

Se construirmos uma matriz que é o fecho transitivo da matriz dada e examinarmos os elementos da diagonal dessa matriz, saberemos se um procedimento pode ser recursivo ou não . Os elementos que contêm 1 (um) na diagonal, correspondem a procedimentos que podem ser chamados recursivamente. É importante notar que em tempo de compilação só conseguimos descobrir os procedimentos que podem ser recursivos e não conseguimos saber se haverá ou não uma ativação recursiva.

O fecho transitivo pode ser construído através do algoritmo de Warshall /21/. Para a matriz dada no exemplo anterios o fecho transitivo é a seguinte matriz:

	A	B	C	D
A	1	1	1	1
B	1	1	1	1
C	1	1	1	1
D	0	0	0	0

Examinando a matriz acima verificamos que os procedimentos A, B e C podem ser recursivos.

Quando identificadores de procedimentos podem ser parâmetros, a deteção da recursão fica ainda mais difícil. Nesse caso precisamos relacionar todas as declarações de procedimentos com as respectivas ativações, verificando todos os valores que podem assumir os procedimentos, que são parâmetros formais de ou

tros procedimentos, para achar a correspondência entre parâmetros formais e atuais.

Na proposta de Irons e Feurzeig /11/ a pilha de bases não é utilizada. Por essa razão, o comando go to deve ser traduzido de maneira diversa em relação ao exposto em 2.2.3. Um go to pode levar a um bloco de profundidade muito menor que a do bloco atual. A tradução de um go to para um endereço ENDE num bloco de profundidade N é feita da seguinte maneira (supondo que PROFA indique a profundidade atual):

```
while  N ≠ PROFA  do
  "desempilha o último registro de ativação atribuindo
   a PROFA a profundidade do registro de ativação
   que passou a ficar no topo da pilha".
CI:= ENDE;
```

3.4 - Dask ALGOL de Jensen, Mondrup e Naur /12/

O Dask ALGOL /12/ não permite procedimentos recursivos e a razão é a economia de tempo de execução. Além disso, impõe que dois procedimentos declarados num mesmo bloco não podem estar disponíveis ao mesmo tempo. Com essa restrição, todas as variáveis simples podem ter endereços fixos. Dois blocos de mesma profundidade, mesmo pertencendo a procedimentos diferentes, compartilham o mesmo espaço, mas o endereço das variáveis simples é fixo. Assim, o registro de ativação de cada bloco se reduz ao espaço para as matrizes e para a marca daquele bloco.

Uma outra característica do Dask ALGOL é que ele não utiliza, e portanto não guarda, a cadeia estática. Só são armazenados na pilha, fazendo parte da marca, o apontador dinâmico, a profundidade do bloco e o estado da computação. Assim, quando é feito um go to para um bloco de profundidade N, deve-se percorrer todos os registros de ativação através da cadeia dinâmica,

até que a profundidade de um determinado registro seja igual a N (ou seja, deseja-se desviar para o primeiro bloco de profundidade N que for encontrado na pilha). Toda a informação que está acima desse registro é retirada da pilha.

3.5 - Computador KDF9 /9/

O compilador ALGOL para o KDF9, descrito por Huxtable e Hawkins /9/ adota um esquema um pouco diferente para o acesso a variáveis não locais. O acesso a uma variável não local pode ser feito de duas maneiras:

a) mantendo a pilha de bases, com cada elemento apontando para a base de um bloco disponível ou,

b) procurando o último registro de ativação do bloco onde ela foi declarada, através das informações da cadeia estática.

Problemas decorrentes desses esquemas são,no primeiro, manter a pilha de bases atualizada e no segundo,ineficiência no tempo de execução.

Baseado nesses fatos,o esquema adotado foi uma combinação dos dois. O esquema a) é usado para os procedimentos que não são recursivos e o esquema b) para os procedimentos recursivos. É provável que os autores tenham feito essa proposta por considerarem que procedimentos recursivos tendem a manipular frequentemente a pilha de bases. O algoritmo proposto para verificar se um procedimento é recursivo ou não é aquele comentado em 3.3.

3.6 - Gier ALGOL /14/

No compilador Gier ALGOL, descrito por Naur /14/ , os elementos da pilha de bases, embora existam, não são usados - como registradores base. Usa-se um registrador para as variáveis do bloco mais externo, que tem sua posição sempre fixa em rela-

ção a ele, e mais outros dois registradores, P e S, que são usados como base da seguinte maneira:

a) variáveis do bloco local corrente são endereçados em relação ao conteúdo do registrador P. Portanto esse registrador sempre aponta para o bloco atual.

b) para as variáveis em blocos intermediários, usa-se o registrador S. O elemento apropriado da pilha de bases é atribuído a esse registrador e em seguida é feito o endereçamento.

A decisão de usar os registradores base dessa maneira é baseada no fato de que, em programas práticos verificou-se que a maioria das referências é feita a variáveis locais e a variáveis do bloco mais externo.

3.7 - Solução com restrição à linguagem

Na linguagem SPL do HP-3000 /7/, o problema de manutenção da pilha de bases é resolvido impondo-se restrições à linguagem. Assim, na linguagem SPL não são permitidas declarações de procedimentos encaixados, ou seja, um procedimento não pode conter uma declaração de um outro procedimento.Como nessa linguagem também é adotada a proposta de Gries /4/ e Wichmann /22/ (ver 3.1) ou seja, todos os blocos de um procedimento são endereçados através do mesmo registrador base, só necessitamos de dois desses registradores. Um deles aponta para a base do registro de ativação do bloco mais externo e o outro para a base do procedimento que está sendo executado no momento. Dessa maneira, a manutenção da pilha de bases se restringe à manutenção do registrador base do procedimento disponível.

Essa restrição deve-se à própria organização do computador HP-3000 /8/, que conta com apenas dois registradores base.

4. PARÂMETROS

Segundo Randell e Russell /17/ os parâmetros de um procedimento podem ser considerados como um canal de comunicação entre o corpo do procedimento e as condições externas, válidas na hora da chamada. Em diferentes chamadas de um mesmo procedimento pode-se ter parâmetros atuais diferentes e portanto deve-se calculá-los a cada chamada, como especificamos abaixo. No texto que se segue denominaremos de *parâmetro atual* a expressão que ocorre como parâmetro na sequência que ativa o procedimento. *Parâmetro formal* é a variável que é declarada no cabeçalho do procedimento.

4.1 - Técnicas de passagem de parâmetros e sua implementação

Existem diversas técnicas de passagem de parâmetros; várias delas são descritas por Gries /4/. Concentrar-nos-emos nas que consideramos principais: chamada por nome, chamada por valor e chamada por referência (conservaremos essas denominações tradicionais, que não consideramos adequadas pois a decisão do tipo de parâmetro não depende da chamada e sim da declaração do mesmo no procedimento). Para seu estudo basear-nos-emos no que foi descrito por Gries /4/, Randell e Russell /17/, Rohl /18/ e Wichmann /22/.

4.1.1. - Parâmetros chamados por valor

Essa é a técnica mais simples de passagem de parâmetros. Neste caso o parâmetro atual é avaliado antes de ser feito o desvio para o procedimento e seu valor é colocado numa posição local ao mesmo. O parâmetro comporta-se como uma variável local ao procedimento. A única diferença é que o parâmetro é inicializado com o valor do parâmetro atual, na ativação do procedimento; dessa maneira, este não pode mudar o valor do parâmetro atual.

Como a posição do parâmetro é fixa em relação à base do registro de ativação do procedimento, o parâmetro pode ser avaliado e colocado exatamente na posição correta da pilha.

Exemplo:

- comando de ativação do procedimento P cujos parâmetros são chamados por valor:

```
P(X, Y+Z, U-V)
```

- supondo que as variáveis X, Y, Z, U e V são locais ao bloco de profundidade D e que ocupam posições consecutivas na pilha, a partir de PILHA[PILBAS[D]+L],o código objeto para calcular os parâmetros é o seguinte:

```
PILHA[AP+3]:= PILHA[PILBAS[D]+L];
PILHA[AP+4]:= PILHA[PILBAS[D]+L+1]+PILHA[PILBAS[D]+L+2];
PILHA[AP+5]:= PILHA[PILBAS[D]+L+3]-PILHA[PILBAS[D]+L+4];
```

A técnica de chamada por valor é a mais eficiente das três técnicas mencionadas e deve ser usada sempre que um parâmetro atual não for alterado pelo procedimento.

4.1.2 - Parâmetros chamados por referência

Um parâmetro chamado por referência é passado como o endereço de uma variável. Esse tipo de parâmetro pode ser usado para mudar o valor do parâmetro atual, o que não acontece com os parâmetros chamados por valor. Segundo Nicholls /15/ essa técnica combina propriedades da chamada por nome e da chamada por valor; a diferença entre elas será esclarecida mais adiante. Em ALGOL 60 não existe esse tipo de parâmetro, mas tudo se passa como se ele fosse empregado, como por exemplo no caso de matrizes chamadas através de seu identificador. No FORTRAN essa é a modalidade de passagem de parâmetros utilizada /4/.

Se o parâmetro chamado por referência fôr uma variável simples ou constante, o próprio endereço é passado ao procedimento. Se fôr uma expressão, ela é avaliada, o resultado é colocado num temporário e o endereço deste é passado ao procedimento.

O acesso a parâmetros chamados por referência é mais lento que o acesso a parâmetros chamados por valor, pois as variáveis são endereçadas indiretamente.

Exemplo:

- comando de ativação do procedimento P cujos parâmetros são chamados por referência:

```
P(X, Y+Z)
```

- supondo que as variáveis X, Y e Z são locais ao bloco de profundidade D, que ocupam posições consecutivas na pilha, a partir de PILHA[PILBAS[D]+L], e que a posição PILHA[K] é usada como temporário, o código objeto para calcular os parâmetros é o seguinte:

```
PILHA[AP+3]:= PILBAS[D]+L;
PILHA[K]:= PILHA[PILBAS[D]+L+1]+PILHA[PILBAS[D]+L+2];
PILHA[AP+4]:= K;
```

O acesso aos parâmetros é indireto ou seja, na posição da pilha correspondente ao parâmetro temos o endereço do parâmetro atual. Por exemplo, se quisermos acessar o segundo parâmetro do procedimento acima, supondo que seu endereço está na posição PILHA[PILBAS[TD]+M], fazemos acesso à seguinte posição:

```
PILHA[PILHA[PILBAS[TD]+M]]
```

4.1.3 - Parâmetros chamados por nome

Um parâmetro chamado por nome deve ser avaliado toda vez que for referenciado dentro do procedimento. O procedimento se comporta como se o parâmetro atual fosse substituído textualmente em todas as ocorrências do parâmetro formal.

A técnica usual para implementar este tipo de parâmetro é construir, para cada parâmetro atual, um procedimento que o avalia, gerado pelo compilador. Esse procedimento foi denominado por Ingermam /10/ de *thunk*. Toda vez que o *thunk* é ativado, ele avalia o parâmetro atual e devolve como resultado o endereço associado ao mesmo. Cada vez que for preciso acessar um parâmetro chamado por nome, faz-se uma ativação do *thunk* e usa-se o endereço retornado por ele para fazer referência ao parâmetro atual.

É importante notar que, para calcular um parâmetro atual, é necessário ter capacidade de sair temporariamente do corpo do procedimento e retornar às condições válidas na sua chamada. Isso porque as variáveis envolvidas no cálculo do parâmetro atual podem não estar disponíveis ao procedimento que está sendo executado. Para isso, uma das informações usadas pelo *thunk* é um apontador para o registro de ativação do bloco onde se deu a chamada do procedimento. Assim, quando um procedimento tem um parâmetro formal chamado por nome, o que ele necessita para calculá-lo é o endereço do *thunk* e o apontador para a base do registro de ativação do bloco onde se deu a chamada do procedimento.

Este tipo de parâmetro é menos eficiente que um parâmetro chamado por referência, pois ele é avaliado toda vez que é acessado, além do tempo gasto na ativação e desativação do *thunk* propriamente dito.

Exemplo:

. comando de ativação do procedimento P cujos parâmetros são chamados por nome:

P(X, Y+Z)

. vamos supor que nas posições PILHA[K] e PILHA[K+1] temos respectivamente o endereço do *thunk* do primeiro parâmetro e o apontador para a base do registro de ativação do bloco onde se deu a chamada do procedimento. Nas posições PILHA [K+2] e PILHA [K+3] temos as mesmas informações relativas ao segundo - parâmetro. Além disso, vamos supor que as variáveis X, Y e Z possuem respectivamente os endereços (N1, P1), (N2, P2) e (N3, P3) (em termos de profundidade do bloco e deslocamento em relação à base do registro de ativação do mesmo) e que a profundidade do bloco onde se deu a chamada de P é N. Nesse caso, o código do *thunk* é representado pelo seguinte algoritmo (usamos a notação descrita em 2.2):

```
go to A;
   begin
   procedure P1;<<thunk do primeiro parâmetro>>
      begin
      PRO:=N;<<profundidade do bloco onde se deu a chamada>>
      ESTA:=PILHA[PILHA[K+1]]. ESTAT;
      PILBAS[PRO]:=PILHA[K+1];<<topo da pilha de bases>>
      <<reconstroi a pilha de bases de maneira a retornar às
        condições válidas na chamada>>
      if PRO > PILHA[AP-2]
         then <<atualização da PILBAS quando a profundidade do
                bloco chamador é maior que a do bloco chamado>>
              for I:= PRO-1 step -1 until  PILHA[AP-2] do
                 begin
                 PILBAS[I]:=ESTA;
                 ESTA:=PILHA[ESTA].ESTAT
                 end
         else I:=PRO-1;
      <<atualização da PILBAS até coincidir com a cadeia está-
        tica>>
      while PILBAS[I] ≠ ESTA do
```

```
        begin
        PILBAS[I]:=ESTA;
        I:=I-1;
        ESTA:=PILHA[ESTA].ESTAT;
        end;
      <<cálculo do endereço do parâmetro>>
      ENDEREÇO:=PILBAS[N1]+P1;
      TP:=PILHA[AP-2];
      PILBAS[TP]:=PILHA[AP-1];
      ESTA:=PILHA[PILBAS[TP]].ESTAT;
      <<retorna a pilha de bases da maneira que ela estava an-
        tes do início do cálculo dos parâmetros>>
      if  TP > PRO
        then <<atualização da PILBAS quando a profundidade  do
               bloco chamado é maior que a do bloco chamador>>
             for I:=TP-1 step -1 until PRO do
                 begin
                 PILBAS[I]:=ESTA;
                 ESTA:=PILHA[ESTA].ESTAT
                 end
        else I:= TP-1;
      <<atualização da PILBAS até coincidir com a cadeia estáti
        ca>>
      while PILBAS[I] ≠ ESTA do
        begin
        PILBAS[I]:= ESTA;
        I:= I-1;
        ESTA:=PILHA[ESTA]. ESTAT
        end;
      CI:=PILHA[AP];<<desvia para o endereço de volta>>
      end;
procedure P2;<<thunk do segundo parâmetro>>
      begin
      <<o corpo deste procedimento é igual ao corpo do procedi-
        mento P1, trocando K por K+2 e trocando o comando  que
```

calcula o endereço do parâmetro pelos seguintes comandos:

```
PILHA[L]:=PILHA[PILBAS[N2]+P2]+PILHA[PILBAS[N3]+P3];
ENDEREÇO:=L;
Estamos supondo que L é a posição onde ficará o temporário com o resultado da expressão Y+Z>>
end;
A:...
```

. as informações sobre os parâmetros, colocadas na pilha são:

```
PILHA[AP+3]:= end (P1);<<endereço do procedimento P1>>
PILHA[AP+4]:= PILBAS[TP];<<base do registro de ativação>>
PILHA[AP+5]:= end (P2);<<endereço do procedimento P2>>
PILHA[AP+6]:= PILBAS[TP];<<base do registro de ativação>>
```

Nesse caso, no algoritmo de ativação de um procedimento (ver 2.2.1), ao calcular o valor de L (número de células da PILHA para variáveis locais e parâmetros) devemos lembrar que cada parâmetro chamado por nome ocupa duas células.

. o acesso a um parâmetro cujo endereço do *thunk* está em PILHA[PILBAS[TP]+J] é feito da seguinte maneira:

```
PILHA[AP+1]:=TP;<<profundidade atual>>
PILHA[AP+2]:=PILBAS[TP];<<base do registro de ativação atual>>
PILHA[AP+3]:=CI;<<endereço de retorno>>
AP:=AP+3;<<atualiza topo da PILHA>>
CI:=PILHA[PILBAS[TP]+J];<<desvia para o thunk>>
AP:=AP-3;<<atualiza topo da PILHA>>
```

A atualização da pilha de bases é feita da mesma maneira que foi descrita em 2.2.2 (para sua justificativa ver o item 2.3.2).

4.2 - Parâmetros atuais

Vamos descrever os tipos mais importantes de parâmetros atuais existentes, baseando-nos no que foi descrito por Gries /4/, Randell e Russell /17/ e Wichmann /22/.

4.2.1 - Variável simples

O procedimento recebe, eventualmente do *thunk*, o endereço da variável. Se for chamada por valor é feita uma cópia de seu conteúdo numa posição temporária, local ao procedimento.

4.2.2 - Constantes

O procedimento tanto pode receber o valor da constante como seu endereço, dependendo da convenção adotada.A primeira solução é mais adequada pois impede a alteração da constante, empregando-se um temporário ao qual se atribui o valor da constante no momento da chamada e cujo endereço é então passado.

4.2.3 - Matrizes

Se a matriz é chamada por nome, o procedimento pode receber o endereço do *dope vector* (ver III.2.3), no caso do computador possuir um tipo de endereçamento indireto que permita manipulá-lo. Se isso não for possível, o procedimento recebe o próprio *dope vector*. Em qualquer caso ele recebe informações adequadas para permitir o acesso aos diversos elementos da matriz, já que os mesmos não são passados.

Matrizes chamadas por valor são mais trabalhosas.É preciso fazer uma cópia da matriz na área local ao procedimento. Por essa razão elas são bem menos eficientes que matrizes chamadas por nome.

4.2.4 - Expressões

Quando um parâmetro é uma expressão, o compilador gera um procedimento (*thunk*) para avaliá-la; isto vale também para variáveis indexadas. Se o parâmetro for chamado por nome passa-se o endereço do *thunk* e um apontador para a base do registro de ativação do bloco onde houve a chamada do procedimento. Esse apontador permite retornar-se temporariamente às condições da chamada do procedimento, para conseguir-se acesso às variáveis disponíveis ao *thunk*. Toda vez que o parâmetro formal é referenciado faz-se uma chamada ao *thunk*.

Se o parâmetro é chamado por valor, o *thunk* só é chamado uma vez, ao entrar-se no procedimento, e o valor resultante é colocado num temporário, local ao procedimento.

4.2.5 - Rótulos

Um rótulo é conhecido pela profundidade do bloco onde ele foi definido e pelo seu endereço dentro do bloco.

Se o rótulo for parâmetro deve-se ter também um apontador para a base do registro de ativação do bloco onde foi feita a chamada do procedimento, que é o mesmo apontador usado pelo *thunk*.

Quando faz-se desvio para um rótulo que é parâmetro, volta-se às condições da chamada, através do apontador descrito no parágrafo anterior, e em seguida faz-se o desvio para o rótulo da maneira usual, descrita anteriormente (ver 2.2.3).

4.2.6 - Parâmetro formal

Um parâmetro atual pode ser um parâmetro formal de um procedimento onde ocorreu a chamada do procedimento em questão. Nesse caso precisamos conseguir acesso ao parâmetro atual o

riginal. Assim, o procedimento recebe o endereço de onde estão as informações que permitem calcular o parâmetro atual original, que eventualmente pode ser o endereço do *thunk* associado ao parâmetro formal. Dessa maneira o cálculo do parâmetro é feito em duas etapas. Na primeira, fazemos um desvio para o trecho de código onde estão as informações que permitem o cálculo ao parâmetro atual original. Na segunda, calculamos esse parâmetro, correspondente ao parâmetro formal.

4.2.7 - Identificador de procedimento

Quando um parâmetro atual é o identificador de um procedimento, deve ser passado o endereço da primeira instrução do procedimento e um apontador para a base do registro de ativação do bloco onde ele foi declarado. Assim, a um parâmetro atual que é um identificador de um procedimento, correspondem duas informações. O endereço da primeira instrução do procedimento permite que se faça um desvio (com endereço de retorno) para ele. O apontador para o registro de ativação do bloco onde ele foi declarado permite o acesso às suas variáveis globais. Se esse procedimento tem parâmetros, eles são tratados da mesma maneira que em um procedimento comum.

REFERÊNCIAS BIBLIOGRÁFICAS

/1/ Dijkstra, E.W. - Recursive Programming - *Numerische Matematik* 2,5, (1960), pp 312-318.

/2/ Dijkstra, E.W. - An ALGOL 60 Translator for the XI - *Annual Review in Automatic Programming*, 3, (1963), Pergamon Press, London, pp 329-345.

/3/ Gries, D., Paul, M., Wiehle, H.R. - Some Techniques Used in the ALCOR ILLINOIS 7090 - *Comm ACM* 8,8 (Aug. 1965), pp 496-500.

/4/ Gries, D. - *Compiler Construction for Digital Computers* - (1971), Wiley, New York.

/5/ Griffits, M. - Run Time Storage Management - In *Compiler Construction an Advanced Course* , Bauer, F.L., Eickel, J. (Eds) - Lecture Notes in Computer Science, (1974), v. 21.

/6/ Hauck, E.A., Dent, B.A. - Burroughs B6500/B7500 Stack Mechanisms-*Spring Joint Comp. Conference, AFIPS Press*, v.32, (1968), pp 245-252.

/7/ HP-3000 - SPL - *System Programming Language* - 03000-9000, 2A, (Nov. 1972).

/8/ HP-3000 - *Reference Manual* - 03000-90019, (Sept 1973)

/9/ Huxtable, D.H.R., Hawkins, E.N. - A Multipass Translation Scheme for ALGOL 60 - *Annual Review in Automatic Programming*, 3, (1963), Pergamon Press, Oxford, pp 163-205.

/10/ Ingerman, P.Z. - Thunks - A Way of Compiling Procedure Statements With Some Comments on Procedure Declarations -

Comm ACM 4,1 (Jan. 1961), pp 55-58.

/11/ Irons, E.T., Feurzeig, W. - Comments on the Implementation of Recursive Procedures and Blocks in ALGOL 60 - *Comm ACM* 4,1 (Jan. 1961), pp 65-69.

/12/ Jensen, J., Mondrup P., Naur P. - A Storage Allocation Scheme for ALGOL 60 - *Comm ACM* 4,10 (Oct. 1961), pp 441-445

/13/ Jensen, J., Naur, P. - An Implementation of ALGOL 60 Procedures - *BIT*, 1, (1961), pp 38-47.

/14/ Naur, P. - The Design of the Gier ALGOL Compiler - *Annual Review in Automatic Programming*, 4, (1965), Pergamon Press, London, pp 49-85.

/15/ Nicholls, J.E. - *The Structure and Design of Programming Languages* - (1975), Addison-Wesley, New York.

/16/ Organick, E.I. - *Computer System Organization - The B5700/ B6700 Series* - (1973), Academic Press, London.

/17/ Randell, B., Russell, D.J. - *ALGOL 60 Implementation* - (1964), Academic Press, London.

/18/ Rohl, J.S. - *An Introduction to Compiler Writing* - (1975), Macdonald and Jane's, New York.

/19/ Rutishauser, H. - The Use of Recursive Procedures in ALGOL 60-*Annual Review in Automatic Programming*, 3, (1963), Pergamon Press, London, pp 43-51.

/20/ Samet, P.A. -The Efficient Administration of Blocks in ALGOL 60 - *Computer Journal* 8,1, (1966),pp 21-23.

/21/ Warshall, S. - A Theorem on Boolean Matrices - *Journal ACM*, 9,2, (1962), pp 11-12.

/22/ Wichmann, B.A. - *ALGOL 60 Compilation and Assessment* - (1973), Academic Press, London

CAPÍTULO II

MANIPULAÇÃO DE RECORDS

1. INTRODUÇÃO

Em muitas áreas de aplicação da computação aparecem problemas envolvendo vários tipos de estruturas de dados. Uma delas são os "records". Assim, um aspecto importante de uma linguagem de programação é a facilidade de associar identificadores a estruturas de informação ou seja, a "records".

Como descrito por Wegner /13/, no ALGOL 60 não há necessidade de distinção entre a definição de uma estrutura de dados e a sua criação. Isto porque em cada declaração está implícita a especificação completa da estrutura do identificador declarado. Entretanto, quando é introduzida uma maior flexibilidade para a definição e criação de dados estruturados, é conveniente separar a definição da estrutura, da sua criação.

Uma das linguagens pioneiras no projeto e estudo de *records* foi a AED-O (Automatic Engineering Design ou ALGOL Extended for Design), que deu origem à linguagem AED-I, ambas devidas a Ross /11,12/. Ela se baseou na existência de problemas nos quais é necessário manipulação simbólica e numérica. Por essa razão, os problemas passaram a ser vistos como compostos de estruturas de dados, algumas das quais tinham *n* componentes de tipos gerais e interligados.Esses *n* componentes interligados são os *records*.

Foram desenvolvidas outras linguagens, que definem e manipulam estruturas de informação, cada uma delas adequada a determinadas aplicações.

Analisaremos os conceitos envolvidos em manipulação de estruturas de informação e algumas linguagens ou propostas de linguagens, que oferecem esse tipo de recurso.

2. UMA PROPOSTA QUE SE ADAPTA AO ALGOL 60

A primeira proposta a ser vista, é a de uma linguagem com recursos para manipulação de estruturas de informação, que se adapta ao ALGOL 60, e que foi introduzida por Hoare /7/. Este item é baseado nesse trabalho, onde usaremos sua notação. Nesse trabalho, as estruturas de informação são chamadas de *records* ou variáveis estruturadas.

Os princípios observados neste projeto de manuseio de *records* foram:

a) sintaxe simplificada, ou seja, fazer com que o programador compreenda muito bem os princípios fundamentais da linguagem, para usá-los de maneira adequada

b) segurança, ou seja, garantia de que certos erros que possam produzir grandes perdas, sejam detetados em tempo de compilação, para não ocorrerem em tempo de execução

c) eficiência, ou seja, fazer com que o código objeto gerado seja rápido e compacto.

2.1 - Conceitos básicos

Comumente, quando desejamos resolver um problema num computador, precisamos construir um modelo para o qual a solução do problema será aplicada. Nesse modelo, cada objeto de interesse deve ser representado por uma estrutura quantificável. Uma dessas estruturas poderá ser um *record*.

Cada objeto representado por um *record* possui diversos atributos. Em geral podemos representar cada atributo por um valor de algum domínio apropriado, domínio este representado por um tipo. Para representar os diversos atributos que compõem um dado objeto, o *record* que o representa terá diversos identificadores, chamados *campos* ("fields") do *record*.

Os objetos podem ser classificados em *classes*. Em ge

ral todos os membros de uma dada classe tem os mesmos atributos. Também é comum os *records* não aparecerem isolados, mas agrupados com outros, formando classes de *records*.Todos os *records* de uma dada classe tem o mesmo número de campos e campos correspondentes tem o mesmo tipo e mesmo identificador.

Num programa devem ser declaradas todas as classes de *records* desejadas e cada declaração introduz o identificador da classe e dá a lista de todos os campos que fazem parte de cada *record* da classe.

```
Exemplo : record class CONTABANCARIA  (integer NUMERODACONTA,
                                              CAPITAL;
                                       real   TAXADEJUROS );
```

Para se referir a um campo de um particular *record* deve-se colocar o identificador do campo e o identificador da classe à qual o *record* pertence.

Exemplo: TAXADEJUROS (CONTABANCARIA)

(Neste caso, CONTABANCARIA é o último *record* criado, da classe CONTABANCARIA).

Num programa, um identificador de campo pode aparecer em qualquer lugar onde é permitida uma variável.

Um *record* é implementado como um grupo de uma ou mais palavras consecutivas na memória do computador. Cada campo recebe um número de palavras suficiente para acomodar qualquer valor do tipo declarado. Em termos de armazenamento, o número de palavras de memória que precedem um determinado campo de um *record* é chamado de *offset* do campo.

2.2 - Referências

Os campos de um *record* podem ser usados para representar relações funcionais, ou seja, relações muitos-para-um ou rela-

ções um-para-um. Em particular, essas relações são fáceis de serem representadas em um computador, alocando-se, em um *record* , um campo que contém o endereço de máquina da primeira palavra do *record* com o qual o primeiro se relaciona.

Para cada relação funcional deve-se declarar um campo tipo *referência*. Para cada declaração de campo tipo referência , deve-se dar a classe de *records* com a qual ele se relacionará.

```
Exemplo: record class  PESSOA  (integer NASCIMENTO;boolean SEXO;
                                reference (PESSOA) PAI,
                                  IRMÃOMAISVELHO,FILHOCAÇULA );
```

A especificação explícita da classe para a qual o campo tipo referência pode apontar, permite que a validade de seu uso seja verificada em tempo de compilação e também permite recuperação de dados inativos ("garbage collection") em tempo de execução.

Quando a relação de um campo tipo referência não está definida, ele assume o valor nil.

Se desejamos trabalhar com vários *records* ao mesmo tempo, precisamos ter várias variáveis apontando para os diferentes *records* com os quais se deseja trabalhar. Para isso podemos declarar variáveis tipo referência.

```
Exemplo: reference (PESSOA)  S;
```

Essas variáveis recebem atribuições da maneira usual.

```
Exemplo: S:= IRMÃOMAISVELHO (S);
```

Uma variável tipo referência é implementada como o endereço da primeira palavra do *record* a que ela se refere. Assim, reserva-se espaço suficiente para armazenar qualquer endereço. O valor nil pode ser representado como qualquer endereço de memória fora dos endereços reservados para *records*. Quando um bloco é ativado, todas as variáveis tipo referência nele declaradas são

inicializadas com o valor nil.

O acesso a um identificador de campo da forma

NASCIMENTO (PAI(S))

pode ser traduzido como uma sequência de três instruções.A primeira carrega num registrador o endereço contido na variável S. Em seguida verifica-se qual é o *offset* do campo PAI e carrega-se no mesmo registrador o conteúdo do campo PAI do *record* previamente apontado. A terceira instrução executa as mesmas operações com o campo NASCIMENTO.

Para cada variável tipo referência deve-se declarar a classe de *records* para a qual ela aponta. Assim,mesmo que duas classes de *records* tenham campos com mesmo identificador,nunca haverá dúvida sobre qual é o campo desejado.

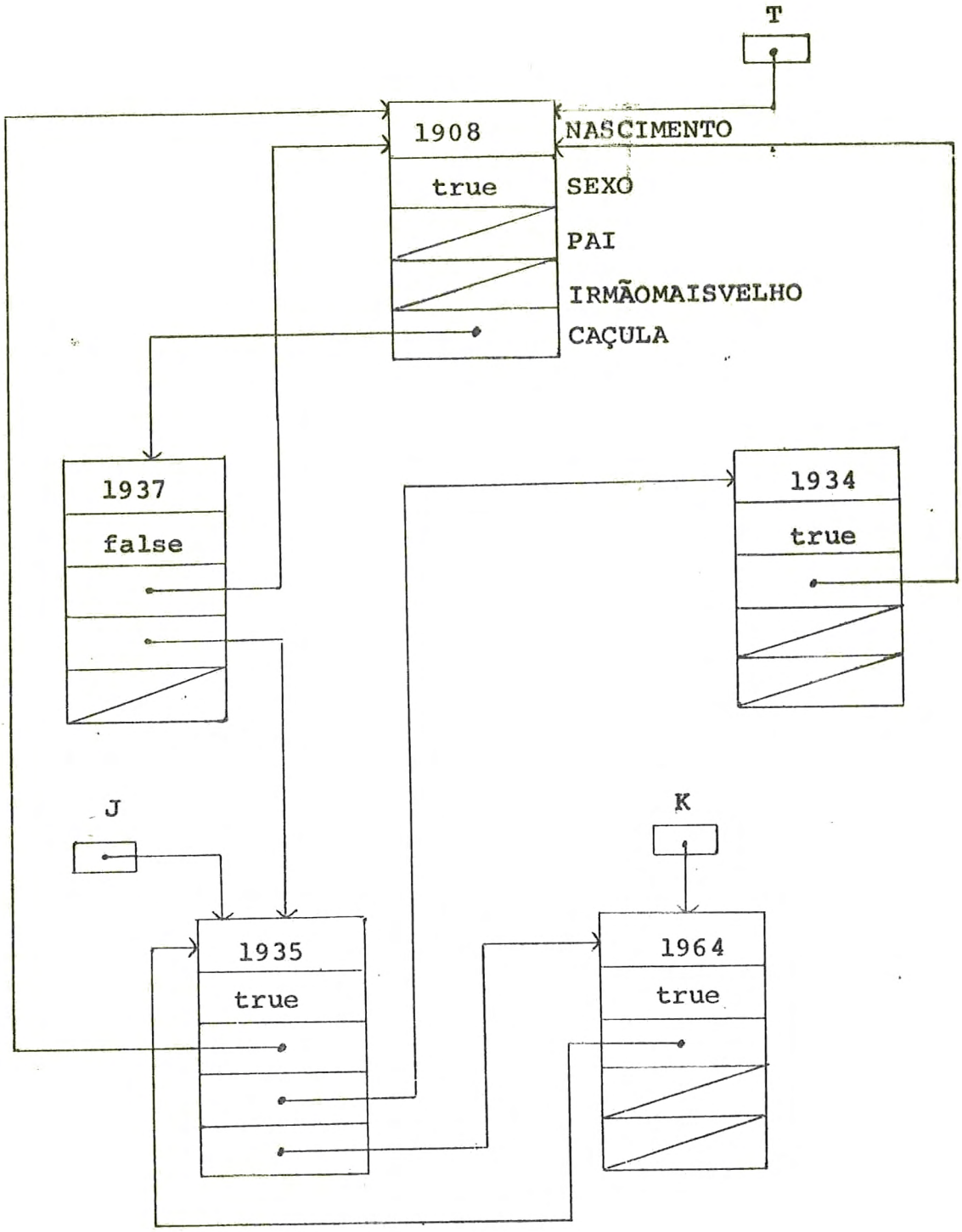
2.3 - Geração de *records*

Para cada classe de *records* existe uma função geradora, que cria um novo *record* daquela classe e devolve como seu valor uma referência a esse *record* criado. Essa função tem uma lista de parâmetros, cada um correspondendo a um campo dos *records* da classe. Cada parâmetro atual é uma expressão que define o valor inicial do campo correspondente. É conveniente usar o próprio identificador da classe de *records* como identificador da função que gera *records* daquela classe.

Exemplo: reference (PESSOA) T, J, K;

```
T:= PESSOA (1908, true, nil, nil, nil);
CAÇULA (T):= PESSOA (1934, true, T, nil, nil);
CAÇULA (T):= J:= PESSOA (1935, true,T,CAÇULA (T),nil);
CAÇULA (T):= PESSOA (1937, false, T, J, nil);
K:= CAÇULA (J):= PESSOA (1964, true, J, nil, nil);
```

Uma célula com um traço diagonal representa a ligação vazia.



Quando se ativa um bloco no qual existe uma declaração de classe de *records*, uma certa área de memória é reservada para ela. Essa área é chamada de *segmento*. Um apontador chamado *first-free* aponta para a primeira palavra do segmento. Toda vez que a função geradora da classe é chamada, o valor do *first-free* fica sendo a referência ao *record* e o *first-free* é incrementado do comprimento do *record* em questão. Quando o segmento se esgota, aloca-se uma nova extensão e o *first-free* passa a apontar para a primeira palavra dessa extensão. Quando o bloco em que a classe foi declarada é desativado, libera-se o segmento inicial e todas as suas extensões, podendo essas áreas serem usadas para outras finalidades.

Se um *record* é gerado num bloco interno àquele em que a classe foi declarada, e se essa geração necessitou alocar um segmento de extensão, este não deve ser alocado como parte da pilha dinâmica de variáveis declaradas. Isto porque esse *record* terá uma validez maior que a das variáveis locais ao bloco em que ele foi gerado, pois um *record* é válido enquanto estiver ativo o bloco onde foi declarada a classe à qual ele pertence. Dessa maneira, quando o bloco onde o *record* foi gerado for desativado, a área dedicada a este não deve ser liberada. Esse fato pode levar a um aumento significante na complexidade da administração da pilha e eventualmente a uma irrecuperável fragmentação de memória.

A solução para esse problema é reservar os segmentos de extensão em área externa à pilha de variáveis declaradas. Pela mesma razão descrita, a administração dessa área não será uma simples administração de pilha dinâmica. Precisamos de um mecanismo de lista livre entre os segmentos de extensão. Toda vez que um bloco onde há declaração de classe de *records* é desativado, todos os seus segmentos devem ser ligados à lista livre, e toda vez que se deseja um novo segmento, ele deve ser retirado da lista livre.

Pode acontecer uma sobreposição entre a área usada para segmentos e a área usada para a pilha dinâmica, e isto pode a

contecer mesmo quando ainda houverem segmentos livres no meio da área de extensão. Nesse caso, só podemos continuar se existir um mecanismo que permita usar segmentos disjuntos ou um mecanismo de compactação.

Se durante a execução do programa necessitarmos de um segmento de extensão e o único disponível estiver muito perto da pilha dinâmica, é aconselhável usar um procedimento de recuperação de dados inativos.

2.4 - Subclasses de *records*

Hoare define também a possibilidade de um *record* ter subclasses, cada uma delas com campos próprios. Assim, na declaração da classe de *records* enumera-se as subclasses às quais um determinado *record* da classe pode pertencer. Os campos de cada subclasse devem ser declarados em seguida ao identificador da subclasse. Dessa maneira, uma declaração de subclasse tem a mesma forma de uma declaração de classe de *records* . Podemos também , dentro de uma subclasse, declarar outra subclasse, e assim por diante.

```
Exemplo: record class  POLIGONO (real AREA;
                 subclasses  TRIANGULO (real LADO,ANGULO1,ANGULO2),
                             RETANGULO (real LADO1, LADO2),
                             CIRCULO (real DIAMETRO));
```

As variáveis tipo referência podem apontar para classes de *records* ou para subclasses.

A especificação da subclasse à qual um *record* pertence é feita quando o *record* é gerado, usando o identificador da subclasse como função geradora.

Para *records* que contêm subclasses, é alocado um campo especial. Quando um *record* da classe é criado esse campo especial vai conter um valor inteiro que indica à qual particular

subclasse o *record* pertence. Esse campo especial deve ser seguido por todos os campos comuns a todas subclasses, para que o *offset* desses campos seja sempre o mesmo, em todos os *records* da classe. Campos privativos a uma subclasse aparecem no fim do *record*.

Records que contêm subclasses, em geral, tem tamanhos diferentes. Nesse caso, Hoare recomenda a criação de uma lista livre para cada tamanho possível do *record*, o que nos parece muito difícil de ser implementado de forma eficiente.

2.5 - Discriminação da classe de *records*

Quando uma variável tipo referência aponta para uma determinada classe de *records* que contêm subclasses, podemos querer saber à qual subclasse pertence o *record* apontado e executar comandos que dependam da subclasse. Isso pode ser feito por uma construção chamada "discrimadador da classe de *records*". Dentro do discriminador devemos dar um nome temporário ao *record*.Um exemplo da forma do discriminador é o seguinte:

```
consider  T = D   when  TRIANGULO  then ...
                  when  RETANGULO  then ...
                  when  CIRCULO    then ...
```

Na execução desse comando é verificada a que subclasse o *record* D pertence e, dependendo da subclasse, apenas o comando seguinte ao respectivo then é executado. Se a referência pode assumir o valor nil, ou se não se deseja discriminar todas as subclasses existentes, pode-se terminar o comando acima com um else. O nome temporário que é dado ao *record* é o que deve ser usado em todos identificadores de campo dentro do discriminador. Dessa forma, em cada comando que se segue ao then, a variável temporária T funciona como se sua validez fosse restrita à particular subclasse introduzida pelo comando selecionado. Assim, pode-se fazer testes de validade em tempo de compilação. Note que a variá

vel T não deve ser usada no comando que segue ao else, pois nesse caso não se tem certeza da subclasse à qual o *record* pertence.

Dentro do comando consider, as subclasses devem aparecer na ordem em que foram declaradas.

A implementação do discriminador de classe de *records* é feita da seguinte maneira:

a) Primeiramente avalia-se a expressão do consider e atribui-se esse valor ao temporário associado.

b) Se existir else, verifica-se se o valor é nil e nesse caso o controle é passado ao comando que segue o else.

c) Faz-se acesso ao campo especial do *record* e adiciona-se seu conteúdo ao contador de instruções. O efeito é pular para a n-ésima instrução seguinte à atual, onde n é o valor do campo especial.

d) Essa instrução de salto é seguida de m instruções de salto onde m é o número de subclasses da classe dada. O n-ésimo salto tem como destino a sequência de código de máquina que deve ser selecionada se o *record* pertence à n-ésima subclasse possível.

e) Cada seção do consider é traduzida como se o temporário associado ao discriminador fosse associado à subclasse relevante. O código de cada seção termina com um desvio para o final do discriminador da classe do *record*.

2.6 - Extensões diversas

Hoare descreve extensões, que não são essenciais, mas que ampliam a quantidade de aplicações para as quais os *records* podem ser usados de maneira eficiente.

a) Matrizes

Às vezes um objeto fica bem representado por uma matriz com elementos do mesmo tipo. Assim, um campo de um *record* pode ser uma matriz.

Exemplo: record class POLIGONO (real AREA; integer COR;
real array [1:4] ANGULO,LADOS);

Para se referir a um elemento da matriz, devemos colocar seu identificador de campo, seguido do índice desejado.

Exemplo: X:= LADO (P) [I] ;
ANGULO (Q) [J] := 3.7 * arctan (X/2) + 7;

Para gerar um *record* que contém campos que são matrizes, a função geradora de *records* deve receber informações relativas à estrutura, tamanho e valores iniciais da matriz em questão.

Se forem permitidas matrizes com mais dimensões, precisamos especificar o número de dimensões, para que o compilador possa reservar, na área relativa a cada *record*, espaço o *dope vector* da matriz.

Dentro de um *record* um campo que é matriz é representado por três quantidades, que especificam os limites inferior e superior e o endereço do primeiro elemento da matriz. Os elementos da matriz ficam no fim do *record*.

No manuseio de *records*, é essencial verificar se um índice de uma matriz está dentro dos limites permitidos, senão corre-se o risco de estragar todo o mecanismo de lista livre (ver III.2.2).

b) Arquivos

Podemos considerar um arquivo como um caso especial de

uma classe de *records*, onde cada registro é um *record* e onde a posição relativa de cada *record* é importante. Num arquivo, a contiguidade física dos *records* reflete a sequência em que eles foram criados. Podemos considerar um arquivo como um conjunto ordenado de *records*, enquanto que uma classe de *records* é um conjunto desordenado.

Em arquivos, introduz-se o operador next, que é a maneira de passar de um *record* para o seu sucessor na sequência.

Deve ser feita uma restrição com relação aos campos tipo referência. Só é permitido a esses campos apontarem para *records* do próprio arquivo ou de outros arquivos que coexistam com o primeiro. A razão é que, quando o programa termina, só os arquivos podem continuar existindo, e portanto, qualquer outra referência deixa de ser válida.

c) Novos tipos

A declaração de classe de *records* fornece uma boa técnica para se definir novos tipos de valores em uma linguagem. Por exemplo, um número complexo pode ser definido como um *record* com dois campos:

Exemplo: record class COMPLEXO (real PARTEREAL,PARTEIMAGINARIA);

Nesse caso, também é interessante definir-se operações aritméticas para manipular os novos tipos.

3. COBOL e PL/I

Para estudar os *records* do COBOL e PL/I, basear-nos-emos no que foi descrito por Berztiss /2/, Gates e Poplawski /5/,Gries /6/, Hoare /7/, Knuth /9/ e Nicholls /10/.

3.1 - Características gerais

As duas linguagens de programação mais usadas hoje em dia, e que fornecem recursos para manipulação de variáveis estruturadas são COBOL e PL/I. Normalmente, vários atributos de dados como tipo e comprimento, podem ser especificados para cada campo da estrutura. Entretanto, para o nosso propósito, essa informação é supérflua, e não será levada em consideração.

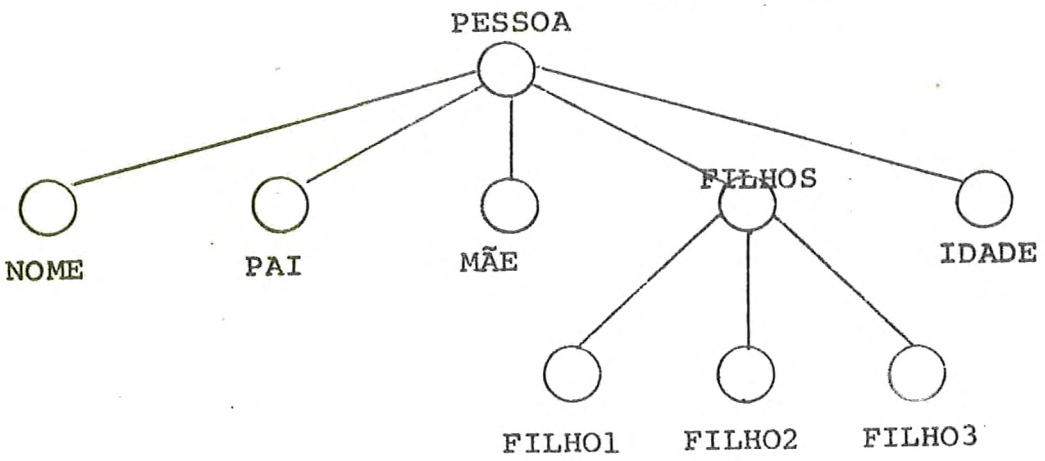
Conforme Nicholls /10/ os princípios usados para especificação de *records* em COBOL e PL/I são essencialmente os mesmos, embora difiram em alguns detalhes.

Segundo Gries /6/ e Nicholls /10/, os *records* do COBOL e PL/I são mais complicados que os *records* de Hoare (ver 2),pois cada campo do *record* pode ter subcampos e, quando for necessário, podemos tratar o *record* como um todo e outras vezes podemos acessar partes dele. Dessa maneira, verificamos que os campos de um *record* podem ser agrupados formando novos campos, que são estruturas maiores, que por sua vez podem ser agrupadas para formar estruturas maiores ainda. Em COBOL, os campos agrupados são chamados de itens de grupo. Em PL/I, o *record* é chamado de estrutura maior e cada conjunto de campos agrupados é chamado de estrutura menor.

Conforme descrito por Berztiss /2/, Gries /6/ e Nicholls /10/, um *record* , em geral, é representado por uma árvore . Cada nó está associado a um campo e os nós finais tem os valores dos dados. Os nós finais formam os campos que constituem o *record* físico na memória.

Exemplo: (em PL/I) declare 1 PESSOA

```
                    2 NOME  char (20),
                    2 PAI   pointer,
                    2 MÃE   pointer,
                    2 FILHOS
                      3 FILHO1  pointer,
                      3 FILHO2  pointer,
                      3 FILHO3  pointer,
                    2 IDADE  fixed;
```



na memória, a representação é:

NOME	PAI	MÃE	FILHO1	FILHO2	FILHO3	IDADE

A raiz, PESSOA, tem nível 1. Os nós que podem ser alcançados por um caminho de comprimento I-1 da raiz tem nível I. O identificador da raiz é precedido pelo inteiro 1 e os identificadores dos campos por inteiros maiores que 1. Esses inteiros que precedem o identificador do campo são chamados de número de nível.

Podemos fazer acesso a campos (ex.: MÃE), ítens de grupo (ex.: FILHOS) ou ao *record* todo (ex.: PESSOA). O acesso a um campo de um *record* é feito através de endereçamento relativo ao iní-

cio do *record* .

Em COBOL o acesso a um campo é feito através de:

FILHO2 of FILHOS of PESSOA

e em PL/I, através de:

PESSOA . FILHOS . FILHO2

Essa forma de acesso é chamada de qualificação do nome.

Em PL/I os campos tipo referência são declarados como pointer e podem apontar para qualquer estrutura. O valor nil é fornecido por um procedimento padrão chamado NULL.

No exemplo dado, usamos identificadores distintos para cada campo e a lista completa dos identificadores para acessar um determinado campo. Entretanto, conforme descrevem Gates e Poplawski /5/ e Nicholls /10/, podemos usar nomes parcialmente qualificados, e ainda, identificadores duplicados. A única restrição é que qualquer nome qualificado se refira a um único elemento. Assim, só são necessários os qualificadores que tornam a referência única, embora outros sejam permitidos.

```
Exemplo:  1  A
             2  B,
             2  C
                3  X,
                3  Y,
             2  D
                3  X,
                3  Y;
```

Neste exemplo, proposto por Nicholls /10/, X e Y estão repetidos, mas não há ambiguidade pois cada nome totalmente qualificado é único.

Segundo Hoare /7/, uma declaração de classe de *records* corresponde, em PL/I, a declaração de um *record* controlled ou de um *record* based

Conforme Nicholls /10/, os dois tipos mencionados acima tem as seguintes diferenças. Quando um *record* controlled é alocado, se já havia sido alocado um outro *record* da mesma classe, este *record* anterior não é perdido. Os dois *records* são mantidos, mas de tal maneira que, em um determinado instante, apenas o alocado por último está disponível. O que se passa efetivamente é que o *record* anterior é colocado em uma pilha. Da mesma forma , quando um *record* é liberado, o último *record* que foi colocado na pilha torna-se disponível. Dessa maneira, existe uma pilha para cada *record* controlled declarado, isto é, para cada classe.

No caso dos *records* based, em um determinado instante podemos ter disponíveis vários *records* de uma mesma classe. Isto porque o acesso a um *record* based é sempre feito através de variáveis tipo referência. Assim, para conseguir acesso a mais de um *record* de uma mesma classe precisamos ter várias variáveis tipo referência.Cada *record* ao qual se deseja ter acesso é criado tendo uma dessas variáveis apontando para ele.

No PL/I, um *record* é sempre uma variável, ocupando portanto certo espaço durante a execução do programa. Já no PASCAL, como veremos adiante (ver 4), um *record* é encarado como um tipo , não funcionando como uma variável.

```
Exemplo: declare 1 PESSOA based (X)
                  2 DATADONASCIMENTO  fixed,
                  2 SEXO  bit (1),
                  (2 PAI, 2 IRMÃOMAISVELHO, 2 CAÇULA) pointer;
```

Nessa declaração, o identificador X é implícitamente declarado como uma variável tipo referência, que normalmente será usada para apontar para um *record* da classe de *records* PESSOA.

Para gerar um *record* correspondente a uma estrutura based ou controlled, deve ser executado um comando allocate. Por exemplo, para construir um *record* PESSOA escrevemos

allocate PESSOA set (X); ou

allocate PESSOA; se X foi associado ao *record* PESSOA

Como não existe recuperação automática de dados inativos, se o programador deseja liberar a área reservada para *records* based ou controlled, ele é o responsável por essa tarefa. A liberação é feita através do comando free.

Exemplo: free PESSOA;

3.2 - Acesso a um campo

Um problema que aparece no COBOL e PL/I é fazer o reconhecimento de um nome qualificado. Existem várias propostas para a solução deste problema. Aqui veremos duas dessas propostas, uma devida a Gates e Poplawski /5/ e a outra devida a Knuth /9/.

3.2.1 Proposta devida a Gates e Poplawski /5/

Conforme descrito por Gates e Poplawski /5/, uma necessidade básica de um compilador é manter uma tabela de símbolos. A técnica de usar-se uma tabela de símbolos ordenada sequencialmente como em FORTRAN ou ALGOL não é boa para o COBOL e PL/I , pois podemos ter um mesmo identificador aparecendo mais de uma vez, com atributos diferentes.

Exemplo: 1 A

2 B

3 C,

```
        3 D,
    2   C
        3 D,
1   B
    2   A;
```

A solução desse problema possui duas etapas. Primeiro aplica-se uma função de "hash" ao qualificador inicial em um nome qualificado. O endereço resultante é uma entrada em uma tabela, cuja estrutura é uma réplica da variável estruturada original, com algumas ligações adicionadas. Em seguida percorre-se essa tabela, procurando uma ou mais ocorrências do nome qualificado. Para cada campo existente na estrutura, existe uma entrada na referida tabela, e cada entrada é constituída de quatro partes:

NOME - contém o identificador desse campo.

SUC - contém uma ligação para o próximo campo com mesmo identificador, formando uma lista ligada circular.

FILHO - contém uma ligação para o primeiro subcampo desse campo.

IRMÃO - contém uma ligação para o próximo subcampo no item de grupo que contém esse campo.

A função de "hash" fornece o endereço da primeira ocorrência do identificador.

Para o último exemplo dado, a tabela é a seguinte:

HASH

A	1
B	2
C	3
D	4

TABELA

	NOME	SUC	FILHO	IRMÃO	
1	A	8	2	7	infor
2	B	7	3	5	mações
3	C	5		4	
4	D	6			adicio
5	C	3	6		nais
6	D	4			
7	B	2	8		
8	A	1			

Uma célula com um traço diagonal representa uma ligação vazia. No algoritmo usamos a notação "λ".

Suponhamos que o nome qualificado seja $A_0.A_1...A_n$, que HASH (A) devolve como valor a primeira ocorrência de A na tabela e que temos funções que manipulam uma fila, ou seja, que F<= (P) significa inserir o elemento P na fila F e que (P)<= F significa retirar o primeiro elemento da fila F e colocar em P (P pode ser um par ordenado). Dessa maneira, podemos elaborar um procedimento que busca um nome qualificado na tabela. O procedimento terá uma fila auxiliar, chamada de FILA e uma fila de candidatos finais chamada de FF. Na fila auxiliar, os elementos são da forma (P,Q) onde P é um apontador e Q é um inteiro. Na fila de candidatos finais os elementos são da forma (P), onde P é um apontador.

Damos a seguir um algoritmo em ALGOL; note-se que a descrição de Gates e Poplawski /5/ é informal.

```
begin
<<inicializações>>
Po← HASH (A_O);
P ← Po;
K ← 0;
FILA<= (P,K);
P ← SUC (P);
<<lista os caminhos iniciais>>
while  P ≠ Po  do
  begin
    FILA <=(P,K);
    P ← SUC (P);
  end;
<<examina cada candidato>>
while  (P,K)<= FILA ≠  λ do
  begin <<compara os níveis>>
    if  A_K = NOME (P) and  K=N
        then  FF <= (P)
        else  begin
              if  A_K = NOME (P) then K ← K+1;
              Q ← FILHO (P);
              if  Q ≠ λ
                  then  begin <<acrescenta primeiro filho>>
                        FILA <=(Q,K);
                        P ← IRMÃO (Q);
                        <<acrescenta os irmãos>>
                        while  P ≠ λ do
                          begin
                           FILA<= (P,K);
                           P ← IRMÃO (P);
                           end;
                        end;
              end;
  end;
end;
```

Quando a execução do algoritmo termina precisamos verificar a lista de candidatos finais. Se não há candidato final, o nome qualificado não é definido. Se há apenas um candidato final o nome qualificado é definido unicamente e é referenciado. Se há mais do que um candidato final o nome qualificado é ambíguo e portanto, não é permitido.

Uma desvantagem desse método é a necessidade de enumerar todos os candidatos possíveis.A estrutura interna da tabela de símbolos é fácil de ser construída pois reflete diretamente a variável estruturada original.

Uma solução para o método de reconhecimento se baseia no fato de que a estrutura de dados original é apenas uma outra maneira de descrever uma expressão regular, e os nomes qualificados válidos são casos dessa expressão regular. Assim, se construírmos um autômato de estados finitos que aceite a expressão regular, e se esse autômato for construído de maneira adequada, ele produzirá todos os possíveis candidatos finais em apenas uma passada através da lista de qualificadores.

Se definirmos "λ" como a cadeia vazia e $\tilde{X}$ como a expressão regular (λUX), que significa zero ou uma ocorrência do símbolo X, então podemos aplicar a linguagem de expressões regulares ao exemplo do início do item 3.2.1.

expressão regular: $A(B(C \cup D) \cup \tilde{C}\tilde{D}) \cup B\tilde{(C \cup D)} \cup \tilde{CD} \cup C \cup D \cup \tilde{BA} \cup A$

Como a estrutura de dados pode ser representada por uma expressão regular, existe um autômato de estados finitos não determinístico que aceita essa expressão. Mas, devido à natureza não determinística da solução, é necessário enumerar todas as soluções. Entretanto, se existe uma máquina não determinística do tipo mencionado, existe uma máquina determinística equivalente , o que elimina a necessidade de enumeração.

A partir do estado inicial do autômato, aplicam-se os

qualificadores da esquerda para a direita. Um nome qualificado é ilegal se necessitamos de alguma transição que não existe no autômato.Se, ao terminar a lista de qualificadores o autômato determinístico está num estado que representa um único estado do autômato não determinístico, o nome qualificado é definido univocamente e é referenciado. Se o estado final do autômato determinístico representa dois ou mais estados do autômato não determinístico, o nome qualificado é ambíguo.

No caso do COBOL, como a referência a um campo é feita do elemento mais específico para o menos específico, ou seja, de um nó final da árvore para a raiz, o algoritmo não pode ser aplicado. Uma solução poderia ser colocar a lista de qualificadores numa pilha e depois retirá-los em ordem inversa e aplicar o algoritmo.

O artigo de Gates e Poplawski /5/ também trata da construção da tabela de símbolos e do autômato.

3.2.2 Proposta devida a Knuth /9/

Esta segunda proposta, devida a Knuth /9/, descreve uma solução para a manipulação de *records* por um compilador COBOL.Ela consta de dois algoritmos. O primeiro processa a descrição dos identificadores e números de nível, colocando as informações relevantes em tabelas que serão usadas pelo compilador. O segundo verifica se um nome qualificado é válido, e se for, localiza o endereço correspondente ao campo referenciado.

Vamos supor que dispomos de um procedimento "tabela de símbolos" que converte um identificador num apontador para a entrada desse identificador na tabela. Além da tabela de símbolos, existe uma outra, chamada tabela de dados e que contém uma entrada para cada campo existente no programa que está sendo compilado. A diferença entre ambas ficará clara no texto que segue.

Para reconhecer o nome qualificado:

A_o of A_1 of ... of A_n $n \geq 0$

precisamos, primeiramente procurar A_o na tabela de símbolos.Deve haver uma série de ligações de cada entrada da tabela de símbolos para todas as entradas da tabela de dados correspondentes ao identificador em questão. Cada entrada na tabela de dados tem três campos de ligação:

PREV - uma ligação para a entrada anterior com mesmo identificador (se existir)

PAI - uma ligação para o ítem de grupo de nível estrutural imediatamente superior, que contém esse campo (se existir)

NOME - uma ligação para a entrada da tabela de símbolos desse campo

Como um exemplo, consideremos as seguintes estruturas em COBOL:

```
1  A
   3  B
      7  C,
      7  D,
   3  E,
   3  F
      4  G;
```

```
1  H
   5  F
      8  G,
   5  B,
   5  C
      9  E,
      9  D,
      9  G;
```

	LIGAÇÃO	
informação adicional	A1	A
	B5	B
	C5	C
	D9	D
	E9	E
	F5	F
	G9	G
	H1	H

Tabela de símbolos

	PREV	PAI	NOME	
A1			A	informação adicional
B3		A1	B	
C7		B3	C	
D7		B3	D	
E3		A1	E	
F3		A1	F	
G4		F3	G	
H1			H	
F5	F3	H1	F	
G8	G4	F5	G	
B5	B3	H1	B	
C5	C7	H1	C	
E9	E3	C5	E	
D9	D7	C5	D	
G9	G4	C5	G	

Tabela de dados

O primeiro algoritmo constrói a tabela de dados. Ele recebe uma sequência de pares (L,P), onde L é um inteiro positivo indicando o número do nível do campo e P aponta para a entrada correspondente na tabela de símbolos. Quando P aponta para uma entrada na tabela de símbolos que não foi referenciada antes, LIGAÇÃO (P) é vazia. Esse algoritmo usa uma pilha auxiliar, que é chamada de PILHA, cujas células contém pares (L,P), onde L é um inteiro e P é um apontador. O algoritmo faz referência a um procedimento REMOVE, que remove o topo da pilha e supõe que a operação "PILHA ⟵(L,P)" insere o par (L,P) no topo da pilha, e que a operação "(L,P)← PILHA" copia (sem remover) o topo da pilha no par (L,P). Conforme o algoritmo é executado, a pilha contém o número de nível e os apontadores para as últimas entradas de dados de níveis maiores que o nível corrente. Por exemplo, imediatamente antes de encontrar o par (3, F) do exemplo dado, a pi

lha deve conter "(0,λ), (1,A), (3,E)".

No algoritmo supomos ainda que a função ENTRADA devolve como resultado o próximo par (L,P) do *record* para o qual estamos construindo a tabela de dados, e que quando essa função devolve o par (λ,λ), a tabela está toda construída e portanto o algoritmo deve terminar. ("λ" significa a ligação vazia).

Damos abaixo uma versão estruturada do algoritmo de Knuth /9/.

```
begin
<<inicialização>>
PILHA <=(0,λ); Q←0; <<Q aponta para o último elemento u
                      sado na tabela de dados>>
<<próximo par>>
  while (L,P)<= ENTRADA ≠ (λ,λ)  do
      begin
      <<faça as primeiras ligações>>
      Q ← Q+1;
      PREV (Q) ← LIG (P);
      LIG (P) ← Q;
      NOME (Q)← P;
      (L1, P1) ← PILHA;
      <<compare os níveis>>
      if  P1 ≠ λ
          then if  L1 ≥ L
                   then begin
                        <<remova o nível no topo  da pi
                                      lha>>
                        while  L1> L  do
                          begin
                          REMOVE;
                          (L1,P1)← PILHA;
                          end;
```

```
                    if L1 < L then ERRO <<números
                                        misturados>>
                              else begin
                                   REMOVE;
                                   (L1,P1) ← PILHA;
                                   end;
                    end;
<<faça a ligação PAI>>
if P1 ≠ λ
   then  PAI(Q) ← LIG(P1)
   else  PAI(Q) ← λ ;
PILHA ← (L,P);
end;
```

O segundo algoritmo localiza a entrada na tabela de dados correspondente ao nome qualificado:

A_o of A_1 of ... of A_n $\quad n \geq 0$

O algoritmo verifica se a referência tem um erro, ou se é ambígua. Se for correta, coloca na variável Q o endereço da tabela de dados correspondente ao campo referenciado. O procedimento "tabela de símbolos" é usado para achar os apontadores P_o, P_1,...P_n das entradas A_o, A_1,...A_n.

Nesse algoritmo, S é uma variável tipo apontador, que percorrerá a árvore a partir de P_o, de maneira ascendente, através das ligações PAI; K será uma variável inteira que assumirá valores de 0 (zero) a N. Sua versão estruturada é:

```
begin
<<inicialização>>
Q ← λ;
P ← LIG(P_o) ;
while  P ≠ λ  do
  begin
  S ← P;
  K ← 0;
  IND ← true;
  while  IND  do
     begin  <<correspondencia completa?>>
     if K = N  then if Q ≠ λ  then ERRO
                              else begin
                                   Q ← P;
                                   P ← PREV (P);
                                   IND ← false
                                   end
               else begin <<aumenta K>>
                    K ← K+1; <<suba na árvore>>
                    S ← PAI(S);
                    while  S ≠ λ and NOME(S) ≠ P_K do
                        S ← PAI(S);
                    if  S = λ  then begin
                                    P ← PREV(P);
                                    IND ← false;
                                    end
     end;
  end;
if Q = λ then ERRO;
end;
```

Knuth /9/ também propõe um algoritmo para tratar o comando move corresponding do COBOL e nesse caso precisamos acrescentar à tabela de dados mais dois campos de ligação (FILHO e IRMÃO). Esse algoritmo não será visto aqui pois o consideramos mui-

to específico ao COBOL.

Existe uma outra proposta para fazer o reconhecimento de um nome qualificado, proposta esta devida a Berztiss /2/. Entretanto, ela não será descrita aqui pois é mais restrita que as duas propostas apresentadas (Berztiss supõe que dois campos distintos de um *record* não podem ter identificadores iguais).

4. PASCAL

Para estudar os *records* do PASCAL, basear-nos-emos no que foi descrito por Jensen e Wirth /8/ e Wirth /15,16/.

4.1 - Características gerais

A linguagem PASCAL foi baseada no ALGOL 60, seguindo seus princípios de estruturação e formação de expressões. Entretanto, conforme Jensen e Wirth /8,15/, o ALGOL 60 não pode ser considerado um subconjunto do PASCAL, pois as duas linguagens diferem em alguns princípios. Uma das principais extensões em relação ao ALGOL 60 é a facilidade de estruturação de dados.

Toda variável que ocorre num comando deve ser introduzida por uma declaração que associa um identificador e um tipo a essa variável. O tipo do dado define o conjunto de valores que podem ser assumidos pela variável. Em PASCAL, um tipo de dado pode ser descrito diretamente na declaração da variável ou pode ser referenciado por um identificador de tipo; nesse caso, o identificador deve constar explícitamente em uma declaração de tipo.

```
Exemplo: type COMPLEXO = record RE, IM : real end;
         var X : COMPLEXO;
```

Uma declaração de tipo definido por um *record* é semelhante à declaração de classe de *records* vista em 2.1.

Os dados estruturados são definidos descrevendo-se o tipo de seus componentes e indicando o método de estruturação.

Um dos métodos de estruturação é o *record*, onde os componentes (campos) não são necessariamente do mesmo tipo. Um tipo *record* pode ter muitas *variantes*, que são análogas às subclasses de 2.4. Se um *record* possui variantes, ele deve ter também um campo especial, chamado *tag*, cujo valor indica qual a variante escolhida. Cada estrutura variante é identificada por uma especificação case. Todos os identificadores de campo devem ser distintos, mesmo se ocorrem em diferentes variantes.

```
Exemplo: 1) record  DIA : 1..31; MES: 1..12; ANO: 0..2000 end;

         2) record  X,Y: real; AREA: real;
                    case S: FORMA of
                       TRIANGULO: (LADO: real;INCLINAÇÃO,ANG1,
                                   ANG2:ANGULO);
                       RETANGULO: (LADO1, LADO 2:real;ANG3:ANGU-
                                   LO);
                       CIRCULO: (DIAMETRO:real)
            end;
```

Se para um certo rótulo a lista de campos é vazia escreve-se R:( ); onde R é o rótulo especificado.

A parte variante de um *record* deve vir sempre após a parte fixa, embora possa existir uma parte variante dentro de outra.

Um campo de um *record* é denotado pelo identificador do *record*, seguido do identificador do campo.

```
Exemplo:  X.RE := 5; X . IM := 3;
```

Em PASCAL existe também o tipo *classe* que é uma estrutura constituída por uma classe de componentes, todos do mesmo tipo. O número de componentes é variável, e no início da execu-

ção de um programa, é zero. Os componentes são variáveis dinâmicas, criadas durante a execução do programa, através do procedimento alloc. O número máximo de componentes é especificado na declaração do tipo classe.

A cada variável tipo classe é associada uma variável tipo referência, que aponta para os componentes daquela classe.

Exemplo: FAMILIA: class 100 of PESSOA;

P1,P2: ↑FAMILIA; (apontadores para a classe FAMILIA)

Dois procedimentos fazem a alocação de componentes de uma classe:

- alloc (P) - reserva um novo componente da classe à qual P pertence, e faz P apontar para esse componente
- alloc (P,T) - análogo ao anterior, com a diferença que o componente tem um campo *tag*, que recebe o valor T.

4.2 - Aspectos da implementação

Conforme descrito por Jensen e Wirth /8, 16/, a unidade básica de um programa em PASCAL é o procedimento. Cada procedimento é representado por um segmento de programa, que contém o código executável e as constantes, mais um segmento de dados,que contém as variáveis locais ao procedimento.

Todas as variáveis de um segmento de dados devem ser alocáveis pelo compilador e é por essa razão que foram eliminadas as matrizes dinâmicas do ALGOL 60.

Em relação ao tipo *record*, desde que cada campo é conhecido pelo compilador, este pode determinar seu deslocamento (*off set*). Dessa maneira, os designadores de campo tem um endereço fixo, e não necessitam de indexação em tempo de execução. Em geral, cada campo ocupa um número inteiro de palavras.

A parte variante de um *record* deve sempre seguir a parte fixa, para que os *offsets* dos campos desta independam da variante. Se um *record* tem uma parte variante, o tamanho reservado para ele é o maior tamanho passível de ocorrência (considera-se a maior variante). Cada parte variante de um *record* tem um campo *tag* associado a ela, e o conteúdo desse campo indica qual a variante escolhida.

O valor de uma variável tipo referência é o endereço absoluto de um componente da classe à qual ela pode apontar. Para as variáveis tipo classe é reservada uma área da memória, cujo tamanho é calculado pelo compilador a partir do tamanho de um componente da classe, e do número máximo de componentes, indicado na definição do tipo classe. Além disso, a cada classe é associada uma variável tipo referência, que é inicializada com a origem da área reservada para aquela classe. Cada vez que um novo componente é requisitado, através do procedimento alloc, a variável tipo referência é aumentada do tamanho do componente. Se o componente tiver estrutura de *record* variante, o procedimento alloc recebe um parâmetro adicional indicando qual a variante desejada. Assim, podemos ter componentes de tamanhos diferentes em uma mesma classe.

No PASCAL não existe recuperação automática de dados inativos.

Sempre que possível, as computações com valores conhecidos são feitas em tempo de compilação. Por exemplo, se tivermos a variável:

```
X: array [10..25] of record  A: integer;
                             B: array [-13..0] of
                                  record U:char;V:real
                                  end;
                             C: boolean;
                        end;
```

é possível, em tempo de compilação, calcular os endereços de:

X , X[13] , X [15] . A , X[19]. B [-7], X[25]. B[-5]. V

Wirth /16/ descreve alguns aspectos de um particular compilador feito para o PASCAL, no qual a tabela de símbolos é estruturada como uma classe de *records*. Todos os identificadores - que compõem a tabela de símbolos são colocados junto com a descrição do objeto que eles representam. Como vários objetos são caracterizados por diferentes conjuntos de atributos, o *record* tem uma parte variante.

5. ALGOL W

Para estudar os *records* do ALGOL W, basear-nos-emos no que foi descrito por Bauer, Becker e Graham /1/, Gries /6/, Nicholls /10/ e Wirth e Hoare /14/.

5.1 - Características gerais

O ALGOL W surgiu de uma proposta feita por Wirth e Hoare /14/ para mudanças no ALGOL 60. Uma das mudanças efetuadas foi a introdução de tipos especiais, como os *records*.

Nesta proposta, os *records* são sempre criados dinâmicamente por comandos do programa. A cada *record* é associado um único valor, do tipo referência, que aponta para o *record*. Essa referência pode ser o valor de um campo de um outro *record*, que mantêm relação com o *record* dado.

Como os *records* são criados dinâmicamente e existem enquanto houver alguma variável do tipo referência apontando para ele, o mecanismo de reserva de memória deve ser algo mais complicado do que uma técnica de pilha; também deve existir um procedimento que faz recuperação de dados inativos, ou seja, um procedimento que libera o espaço de um *record* que está inacessível.

Conforme descrito por Gries /6/ e Nicholls /10/, a cada

variável tipo referência, deve ser associada a classe para a qual a variável vai apontar.

No ALGOL W não existem subclasses (ver 2.4). Elas podem ser simuladas se um campo de um *record* for uma variável tipo referência, que aponta para um outro *record*.

Gries /6/,Nicholls /10/ e Wirth e Hoare /14/ descrevem a maneira de construir um novo *record* de uma dada classe. Isso é feito,usando uma função cujo nome é o identificador da classe de *records*.

Exemplo:- declaração de uma classe de *records* e de uma variável tipo referência que aponta para ela

```
record PESSOA (string NOME; integer IDADE;logical SEXO;
     reference (PESSOA) PAI, MÃE, PRIMOGENITO, IRMÃO);
reference (PESSOA) P;
```

- criação de um novo *record* da classe, apontado por P

```
P:= PESSOA ("carol", 0, false, JACK, JILL, null,PRIMOGE
          NITO (JACK));
```

Para criar um novo *record*, chama-se um procedimento que reserva a quantidade de memória necessária, armazena os valores especificados para o *record* nessa área, e coloca o endereço dessa área na variável tipo referência.

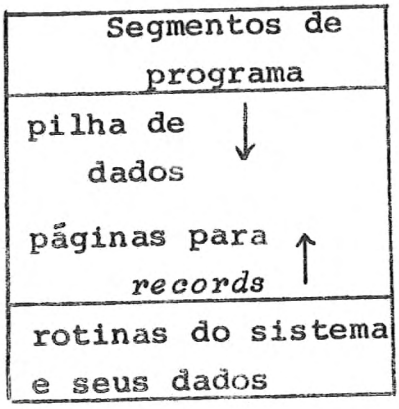
Quando não existe mais espaço livre, chama-se um procedimento de recuperação de dados inativos, pois não existe nenhum comando explícito para liberação de áreas. Esse procedimento deve verificar todas as variáveis tipo referência do programa, inclusive as que são campos de *records*.

O acesso aos campos de um *record* é feito através de : NOME (P), IDADE (P), etc... (considerando o exemplo acima).

5.2 - Detalhes de um compilador

Bauer, Becker e Graham /1/ descrevem um compilador que foi desenvolvido para o ALGOL W, para o sistema IBM/360. Veremos agora alguns detalhes desse compilador, no que diz respeito aos *records*.

A reserva de memória é feita da seguinte maneira. Os segmentos de programa, que são estáticos, são colocados numa extremidade da memória e são seguidos da pilha de dados, que é reservada dinâmicamente. As rotinas do sistema e seus dados, que são alocados estáticamente, são colocados na outra extremidade da memória, seguidos das páginas para *records*, que são reservadas dinâmicamente. Se a pilha de dados e as páginas para *records* colidem, a execução termina.



Os blocos de um procedimento necessitam de um segmento de dados, e quando um bloco ocorre dentro do corpo de um procedimento, seu segmento de dados faz parte do segmento de dados do procedimento. Na base de cada segmento de dados existem cinco palavras que contém as informações necessárias para a sua administração. Uma dessas cinco palavras, contém, em seus dois primeiros *bytes*, o número de variáveis tipo referência, locais ao bloco em questão (parâmetros "value/result" tipo referência são tratados como variáveis locais). Todas as variáveis e parâmetros tipo referência são colocados juntos para facilitar o processamen-

to do recuperador de dados inativos. Os dois últimos *bytes* dessa palavra, apontam para a primeira variável ou parâmetro tipo referência, relativo à base desse segmento de dados. Uma outra palavra, dentre as cinco palavras mencionadas, contém o mesmo tipo de informação para matrizes tipo referência. Assim, os dois primeiros *bytes* indicam quantas matrizes existem naquele bloco e os dois últimos *bytes* apontam para o descritor da primeira matriz tipo referência daquele segmento de dados. Todos os descritores de matrizes tipo referência estão contíguos, e o primeiro *byte* de cada descritor indica a dimensão da matriz. Dessa maneira, o recuperador de dados inativos é capaz de localizar todos os descritores de matrizes tipo referência.

O espaço para os *records* é alocado por páginas, sendo que cada página é dedicada a uma classe de *records* . O tamanho das páginas é um parâmetro das rotinas de execução. As páginas são formatadas de tal maneira que cada *record* dentro da página é apontado pelo *record* anterior.

Existe uma tabela chamada RCT ("*record* class table")que contém uma entrada para cada classe de *records* do programa e é indexada pelo número da classe. As classes são numeradas de 1 (um) em diante. A posição RCT(0) é usada como apontador para o início da cadeia de páginas livres de *records*.

Cada entrada na RCT contém as seguintes informações sobre cada classe de *records*.

- FRC - ("free *record* chain") - cadeia de *records* livres
- PC - ("page chain") - cadeia de páginas
- RL - ("*record* length") - tamanho do *record* em *bytes*
- NR - ("number of reference fields") - número de campos tipo referência.

Além disso, o número da entrada na RCT indica o número da classe do *record*.

A informação denominada de FRC contém a origem da cadeia de *records* livres para uma dada classe de *records* .

A posição RCT(0), que também é chamada de FRPC ("free *record* page chain") contém a origem da cadeia de páginas livres de *records* . Essas páginas livres são liberadas para a memória livre, na medida da necessidade de expansão desta última. Dessa maneira, a memória livre pode ser usada tanto pelos segmentos de dados, quanto pelas páginas de *records*.

Uma nova variável tipo referência apontando para uma dada classe de *records* é sempre obtida através do FRC correspondente à classe. Se o primeiro *byte* do FRC é zero, isto significa que não temos mais *records* livres naquela página; portanto, o recuperador de dados inativos é chamado.

A rotina de recuperação de dados inativos consiste de três fases:

a) marcar os *records* de todas as classes que estão sendo usados;

b) coletar os *records* que não estão sendo usados, ou seja, colocar os *records* que não estão sendo referenciados na cadeia de *records* livres da sua classe;

c) devolver as páginas não usadas, o que significa que se alguma página só tem *records* não referenciados, ela é retornada à cadeia de páginas livres de *records*. Além disso, se a página está contígua ao espaço livre para segmentos de dados, ela é devolvida a ele.

Depois que toda devolução de memória é feita, o recuperador de dados inativos deve fornecer um *record* da classe desejada. Se não existe nenhum *record* livre da classe desejada, é reservada uma nova página para essa classe de *records*, que é ligada à cadeia de páginas dessa classe. Se não existir espaço para uma nova página, a execução do programa termina.

6. OUTRAS PROPOSTAS

Christopher /3/ Dahl e Nygaard /4/, e Hoare /7/ descrevem ainda outras propostas relativas a *records*.

Dahl e Nygaard /4/ descrevem a linguagem SIMULA, que é uma linguagem projetada para facilitar a geração de programas que simulam eventos discretos. Nessa linguagem, declaram-se processos, que analogamente aos procedimentos em ALGOL tem uma seção de declaração de dados e um corpo ("de atividades"). Se esse corpo for vazio, o processo é bastante similar a um *record*.

Hoare /7/ propõe também uma outra maneira de implementar *records*, que seria reproduzi-los no ALGOL 60 através do manuseio de matrizes. Uma classe de *records* poderia ser declarada como um grupo de matrizes, uma para cada campo da classe.

Exemplo: o tipo *record*

```
type COMPLEXO = record RE, IM: real end;
```

poderia ser declarado como:

```
real array RE [1:N] , IM [1:N];
```

onde N é o número máximo de *records* que serão usados.

Essa proposta tem algumas desvantagens pois precisamos saber o número limite de *records* em uma classe quando as matrizes são declaradas. Além disso, o mecanismo para acessar variáveis indexadas,em geral, é menos eficiente que o para acessar campos de um *record*.

7. COMENTÁRIOS FINAIS

Consideramos a proposta devida a Hoare /7/, bastante completa, contendo todos os pontos importantes a serem considerados na manipulação de *records*, como definições de classe, variá-

veis tipo referência, subclasses, etc.. Outro ponto importante dessa proposta, consiste nas referências à implementação dos diversos aspectos considerados.

No COBOL e PL/I, o que aparece de novo são os números de nível, que trazem como consequência o fato dos identificadores de campo poderem ser iguais dentro de um mesmo *record*. Assim,nessas linguagens, a grande diferença é a existência de algoritmos não triviais para reconhecer um nome qualificado. Nas outras linguagens apresentadas, o reconhecimento de um nome qualificado é imediato.

No PASCAL, os *records* podem ser encarados como a definição de um novo tipo de dados. As classes do PASCAL, devem ter um número fixo de componentes, pois a filosofia da linguagem é a de que o compilador faça o máximo de tarefas possíveis, para que o programa objeto seja bem eficiente.

A diferença do PASCAL para o ALGOL W, é que no último os *records* são sempre criados dinâmicamente. Seu espaço só é reservado durante a execução do programa objeto.

REFERÊNCIAS BIBLIOGRÁFICAS

/1/ Bauer, H., Becker, S. and Graham, S. - *ALGOL W Implementation* - CS98, (1968), Computer Science Department, Stanford.

/2/ Berztiss, A.T. - *Data Structures Theory and Practice* - (1975), Academic Press, New York.

/3/ Cristopher, T.W. - *A Proposed Data Type Resembling the SIMULA 67 Class* - (1974), Institute of Computer Research, University of Chicago.

/4/ Dahl, J. and Nygaard, K. - SIMULA an ALGOL Based Simulation Language - *Comm ACM* 9,9 (Sept 1966), pp 671-678.

/5/ Gates, G.W. and Poplawski, D.A. - A Simple Technique for Structured Variable Lookup - *Comm ACM* 16,9 (Sept 1973), pp 561-565.

/6/ Gries, D. - *Compiler Construction for Digital Computers* - (1971), Wiley, New York.

/7/ Hoare, C.A.R. - Record Handling In *Programming Languages* (1968), Academic Press, London, pp 291-347.

/8/ Jensen, K. and Wirth, N. - *PASCAL - User Manual and Report* Lecture Notes in Computer Science, 18, (1974), New York.

/9/ Knuth, D.E. - *The Art of Computer Programming* - Vol. 1, (1968), Addison-Wesley, New York.

/10/ Nicholls, J.E. - *The Structure and Design of Programming Languages* - (1975), Addison-Wesley, New York.

/11/ Ross, D.T. - A Generalised Technique for Symbol Manipulation and Numeral Calculation - *Comm ACM* 4,3 (Mar. 1961),

pp 147-150

/12/ Ross, D.T. - The AED Free Storage Package - *Comm ACM* 10,8 (Aug. 1967), pp 481-492

/13/ Wegner, P. - *Programming Languages, Information, Structures and Machine Organization* - (1968), McGraw-Hill, New York

/14/ Wirth, N. and Hoare, C.A.R. - A Contribution to the Development of ALGOL - *Comm ACM* 9,6 (June 1966), pp 413-432

/15/ Wirth, N. - The Programming Language PASCAL - *Acta Informatica* 1, (1971), pp 35-63

/16/ Wirth, N. - The Design of a PASCAL Compiler - *Software Practice and Experience* 1, (1971), pp 309-333.

CAPÍTULO III

PROCESSAMENTO DE MATRIZES

1. INTRODUÇÃO

Um tipo de estrutura de dados existente no ALGOL 60 é a matriz. Esse tipo é bastante conhecido pois também existe em outras linguagens, como por exemplo no FORTRAN. Além disso, a matriz fornece uma representação natural para muitos dados encontrados na prática, sendo indispensável para certos tipos de problemas.

Uma matriz consiste de um conjunto ordenado de elementos, todos do mesmo tipo. O acesso a um elemento é feito através de seu índice e portanto é um acesso direto. Conforme Wirth /12/, todos os elementos podem ser selecionados diretamente, e são igualmente acessíveis. O índice de um elemento da matriz deve estar dentro do intervalo que foi especificado na sua declaração. Há sistemas que verificam essa condição e outros que não a verificam, sendo portanto sujeito a erros (ver 2.2).

Exemplo: declaração de uma matriz: real array A[1:10]
acesso ao elemento de índice 2: A[2].

Uma matriz pode ter mais do que uma dimensão e nesse caso, para selecionar um elemento da matriz, colocamos um índice para cada dimensão existente. Na declaração da matriz devem ser especificados os limites de variação dos índices para cada dimensão.

Exemplo: declaração: real array A[1:10, 2:15, 0:5]
acesso: A[2, 10, 0]

Segundo Nicholls /7/, as matrizes além de representa

rem dados que ocorrem naturalmente, também podem ser implementadas eficientemente. Algumas razões para isso são:

a) como todos os elementos de uma matriz são do mesmo tipo, só precisamos ter uma descrição de tipo para toda a matriz.

b) a estrutura de uma matriz está bastante relacionada com a estrutura de endereçamento de muitas máquinas. Mesmo no caso de matrizes de mais de uma dimensão, a correspondência entre os índices de um elemento e seu endereço é simples e pode ser calculada eficientemente.

2. MATRIZES ESTÁTICAS E MATRIZES DINÂMICAS

Se o tamanho de uma matriz é conhecido em tempo de compilação, ou seja, os limites de cada dimensão são constantes durante toda a execução do programa onde ela é declarada,dizemos que a matriz é estática. Nesse caso, seu espaço pode ser reservado em tempo de compilação, pois sabemos exatamente qual é o seu tamanho.

A partir deste ponto vamos supor que cada elemento de uma matriz ocupa sempre uma palavra de memória. A generalização dos algoritmos quando um elemento ocupar mais do que uma palavra é imediata.

Exemplo: real array A[1:10, 0:5]

A matriz A ocupará 60 palavras de memória.

No ALGOL 60 foi introduzida a possibilidade dos limites de cada dimensão de uma matriz serem variáveis. A cada ativação de um bloco eles assumem um valor fixo, mas em diferentes ativações de um mesmo bloco podem ter valores diferentes. Nesse caso dizemos que a matriz é dinâmica.

Uma matriz dinâmica só poderá ter seu espaço reser

vado durante a execução do programa objeto pois somente nesse momento é que saberemos quanto espaço será necessário para ela.

2.1 - Matrizes estáticas

Para fazer a compilação da declaração e do acesso a uma matriz estática, basear-nos-emos no que foi descrito por Aho e Ullman /1/ e Rohl /9/.

Para cada matriz estática que for declarada, precisamos guardar as seguintes informações:

a) descrição do tipo.

b) endereço base (definido adiante).

c) limites inferior e superior de cada dimensão (o que implica em guardar-se o número de dimensões).

Suponhamos que as matrizes sejam armazenadas por colunas, ou seja, a matriz declarada como real array A[1:10, 1:20] é armazenada na seguinte ordem: A[1,1], A[2,1],..,A[10,1], A[1,2],... A[10,2],... A[1,20],... A[10,20]. Posteriormente teceremos considerações sobre esse tipo de armazenamento, comparado com o correspondente por linhas.

Vamos supor inicialmente que estamos trabalhando com matrizes de uma dimensão (vetores): $A[L_1:U_1]$.

O deslocamento do endereço de um elemento $A[I]$ qualquer em relação ao endereço do elemento inicial $A[L_1]$ é $(I-L_1)$. Assim, para calcular seu endereço precisamos somar ao endereço de $A[L_1]$ o índice I e em seguida subtrair L_1. Nesse caso, é preferível fazer apenas uma operação, calculando-se inicialmente o endereço de $A[L_1]$ menos L_1, ou seja, o endereço do elemento $A[0]$. Em seguida, fazemos todos os endereçamentos em relação ao endereço de A[0], que é chamado de *endereço base*. Note que o elemento $A[0]$ pode não existir, mas o endereço base sempre existe.

Assim, o acesso a um elemento A[I] é feito da seguinte maneira (onde end (X) indica o endereço de X):

$$\text{end}(A[I]) = \text{end}(A[0]) + I$$

ou seja, o endereço de A[I] é o endereço base de A modificado pelo valor de I.

A área reservada para a matriz A pode ser calculada quando é compilada a sua declaração, sendo colocada em seguida à última área de dados reservada pelo compilador. Seu tamanho é (U_1-L_1+1) palavras. O endereço base de A é o endereço do início da área reservada para A menos L_1.

Vamos agora considerar o caso de uma matriz de k dimensões: $A[L_1:U_1, L_2:U_2, \ldots, L_k:U_k]$. Para calcular o endereço base de A, é necessário saber quanto espaço cada dimensão ocupará. Para isso vamos introduzir as constantes:

$$C_1 = 1$$
$$C_2 = (U_1-L_1+1) * C_1$$
$$C_3 = (U_2-L_2+1) * C_2$$
$$\vdots$$
$$C_k = (U_{k-1}-L_{k-1}+1) * C_{k-1}$$
$$C_{k+1} = (U_k-L_k+1) * C_k$$

Cada (U_i-L_i+1) é denominado por Randell e Russell /8/ de *largura* do i-ésimo índice. Portanto, cada C_i dá a quantidade pela qual o endereço de um elemento muda, correspondendo à mudança de uma unidade no i-ésimo índice.

A constante C_{k+1} indica o tamanho da matriz, ou seja, quantas posições de memória ela ocupa. O cálculo do endereço de um elemento $A[I_1, I_2, \ldots, I_k]$ qualquer, é feito da seguinte forma:

$$\text{end}\,(A[I_1,I_2,\ldots,I_k]) = \text{end}\,(A[L_1,L_2,\ldots L_k]) + \sum_{j=1}^{k}(I_j-L_j)*C_j = \quad (1)$$

$$= \text{end}\,(A[L_1,L_2,\ldots L_k]) - \sum_{j=1}^{k} L_j*C_j + \sum_{j=1}^{k} I_j*C_j$$

Os dois primeiros termos da última expressão,que são fixos para cada matriz, dão o endereço base da mesma (endereço de A[0,0,... 0]).

Assim, a compilação da declaração de uma matriz $A[U_1:L_1,\ U_2:L_2,\ldots\ U_k:L_k]$ é feita pelas seguintes etapas:

a) cálculo das constantes $C_i \quad 1 \le i \le k+1$

b) reserva de C_{k+1} palavras para a matriz A

c) cálculo do endereço base de A: endereço base (A) =

$$\text{end}\,(A[L_1,L_2,..,L_k]) - \sum_{j=1}^{k} L_j*C_j$$

O acesso a um elemento $A[I_1,I_2,\ldots,I_k]$ pode ser indicado através dos seguintes comandos do ALGOL 60:

```
R:= 0,
for  J:= 0 step 1 until K do
     R:= R + I[J] * C[J];
```

e o endereço do elemento desejado é:

endereço base (A) + R

Se as matrizes fossem armazenadas por linhas, ao invés de colunas, o que se modificaria no esquema acima seria somente o cálculo dos C_i, $1 \le i \le k+1$

Nesse caso teríamos:

$$C_k = 1$$
$$C_{k-1} = (U_k - L_k + 1) * C_k$$
$$\vdots$$
$$C_1 = (U_2 - L_2 + 1) * C_2$$
$$C_0 = (U_1 - L_1 + 1) * C_1$$

O elemento C_0 indicaria quantas posições de memória seriam necessárias para a matriz. Os demais cálculos seriam iguais ao caso de matrizes armazenadas por colunas.

A vantagem do armazenamento por colunas reside no fato do compilador ser mais simples, pois o cálculo dos C_i $1 \leq i \leq k+1$ pode ser feito à medida que é feito o reconhecimento sintático de cada dimensão da matriz. No caso de armazenamento por linhas, esse cálculo só pode ser feito após o reconhecimento sintático de todas as dimensões da matriz. Note-se que em tempo objeto nada se ganha com uma ou outra forma.

2.2 - Verificação do acesso a matrizes estáticas

Para fazer a verificação do acesso a matrizes estáticas, basear-nos-emos no que foi descrito por Randell e Russell /8/, Rohl /9/ e Wichmann /11/.

Um erro comum em programas é o que resulta do uso de índices que estão fora dos limites declarados para uma matriz.

Em geral, a verificação dos índices a cada acesso a um elemento de uma matriz aumenta bastante o tamanho do código objeto gerado (a não ser no caso da máquina possuir "hardware" para essa finalidade). Assim, um bom compilador deve permitir que o programador especifique se ele deseja ou não a verificação de índices.

Essa verificação pode ser feita de duas maneiras diferentes:

a) verificar cada índice separadamente

b) verificar se o endereço final está na área reservada para a matriz.

Se for usada a técnica de verificar cada índice separadamente, para cada dimensão da matriz precisamos guardar três informações:

a) o C_i correspondente àquela dimensão (ver 2.1)

b) o valor máximo dos índices daquela dimensão (U_i)

c) o valor mínimo dos índices daquela dimensão (L_i)

Nesse caso, a verificação do acesso é feita juntamente com o cálculo do endereço (ver expressão (1) em 2.1). À medida que é feita a somatória de $(C_j * I_j)$, faz-se também a verificação da validade do índice I_j, ou seja, verifica-se se $L_j \leq I_j \leq U_j$. Se a relação não for satisfeita é emitida uma mensagem de erro.

Se for usada a técnica de apenas verificar-se o endereço final, é necessário uma quantidade menor de informações e de operações. Nesse caso, para cada matriz, não necessitamos guardar os limites inferior e superior de cada dimensão, mas precisamos de outras duas informações, que podem ser guardadas juntamente com o endereço base. Essas informações são denominadas por Rohl /9/ de limite inferior equivalente (L') e de limite superior equivalente (U'), e são dadas por:

$$L' = \sum_{j=1}^{k} L_j * C_j \qquad U' = \sum_{j=1}^{k} U_j * C_j$$

A verificação do acesso é feita após o cálculo do endereço por meio da relação

$$L' \leq \sum_{j=1}^{k} I_j * C_j \leq U'$$

Se a relação não for satisfeita deve ser emitida uma

mensagem de erro. Esta solução é mais simples e barata que a anterior, pois para cada acesso a um elemento de uma matriz é feita apenas uma verificação, enquanto que na outra solução é feita uma verificação para cada índice. Também nesta solução necessitamos de menos informações (U' e L') do que na outra (U_j e L_j, $1 \leq j \leq k$). A desvantagem desse método é que um determinado índice pode estar fora dos limites declarados para o mesmo, sem que seja detetado o erro, se o endereço final estiver dentro da área reservada para a matriz.

2.3 - Matrizes dinâmicas

Para fazer a compilação da declaração e do acesso a uma matriz dinâmica, basear-nos-emos no que foi descrito por Gries /2/, Huxtable e Hawkins /4/, Randell e Russell /8/, Rohl /9/ e Wichmann /11/.

Se os limites de cada dimensão de uma matriz só forem conhecidos durante a execução do programa objeto, não é possível reservar espaço para a matriz durante a compilação.

Nesse caso, vamos distinguir as variáveis cujos endereços são conhecidos em tempo de compilação, que serão chamadas de variáveis estáticas, daquelas cujo endereço é conhecido durante a execução, que serão chamadas de variáveis dinâmicas.

O cálculo do endereço de um elemento de uma matriz dinâmica pode ser feito da mesma maneira que o de uma matriz estática. A diferença é que as informações que auxiliam o cálculo só são conhecidas em tempo de execução. O endereço base da matriz, assim como as constantes C_i (ver 2.1), passam a não ser mais conhecidos em tempo de compilação. Eles devem ser calculados e armazenados em tempo de execução. Em tempo de compilação, sabemos apenas quantas posições de memória serão necessárias para armazenar essas informações: uma posição para o endereço base e k+1 posições para os C_i, onde k é a dimensão da matriz em questão. O conjunto dessas informações é chamado de *dope vector* da

matriz. O endereço do *dope vector* é conhecido em tempo de compilação, pois seu tamanho é fixo, e ele pode ser armazenado como um conjunto ordenado de variáveis estáticas. Durante a execução do programa objeto, calcula-se seu conteúdo e reserva-se a área real da matriz (onde ficarão seus elementos).

Quando são permitidas matrizes dinâmicas, o esquema de alocação de memória para as mesmas passa a ser diferente do método usado para matrizes estáticas. O registro de ativação de cada bloco (ver I.1) será dividido em duas partes e terá sempre um apontador indicando seu término. Esse apontador será chamado de AP (apontador para o topo da pilha).

Na primeira parte do registro de ativação de um bloco ficarão as variáveis estáticas, e os *dope vectors* das matrizes declaradas naquele bloco; portanto, o tamanho desta parte é conhecido em tempo de compilação. Durante a execução do programa objeto, quando um bloco é ativado, o apontador AP deve apontar para o fim da área de variáveis estáticas daquele bloco. Na segunda parte do registro de ativação de um bloco ficarão os elementos das matrizes declaradas naquele bloco; portanto, o tamanho desta parte só é conhecido em tempo de execução.

Para cada declaração de matriz do tipo:

real array A[L_1:U_1, L_2:U_2,...L_k:U_k]

existente em um bloco, devem ser gerados os seguintes comandos (C_i $1 \leq i \leq k+1$ e o identificador ENDEREÇOBASE já tem seus endereços definidos), cuja finalidade é calcular o *dope vector* da matriz:

```
C[1] :=1;
S:=0;
for I:=2  step 1 until K+1 do
  begin <<calcula os C_i>>
    C[I]:= (U[I-1]- L[I-1] + 1)* C[I-1];
    S:= S+ L[I-1]* C[I-1]
```

```
    end;
ENDEREÇOBASE:= TP-S; <<calcula o endereço base>>
AP:= AP+C[K+1]; <<atualiza AP>>
```

O apontador AP apontará sempre para a primeira palavra disponível da área de matrizes.

O acesso a um elemento de uma matriz dinâmica é feito da mesma maneira que o de uma matriz estática (ver 2.1).

Se desejarmos uma verificação do acesso a matrizes dinâmicas, podemos proceder da mesma maneira que foi descrita para matrizes estáticas (ver 2.2). A única diferença é que o *dope vector* também deve ter posições para todos os limites inferiores e superiores e devemos acrescentar comandos que, durante a execução, preencham essas posições.

2.4 - Vectores de Iliffe

Veremos agora uma outra maneira de tratar as matrizes, que conforme Rohl /9/, foi sugerida inicialmente por Iliffe /5/. Nossa exposição está baseada no que foi descrito por Gries /2/, Rohl /9/ e Wichmann /11/.

Segundo Griffiths /3/, a idéia deste método é baseada na seguinte definição indutiva de uma matriz:

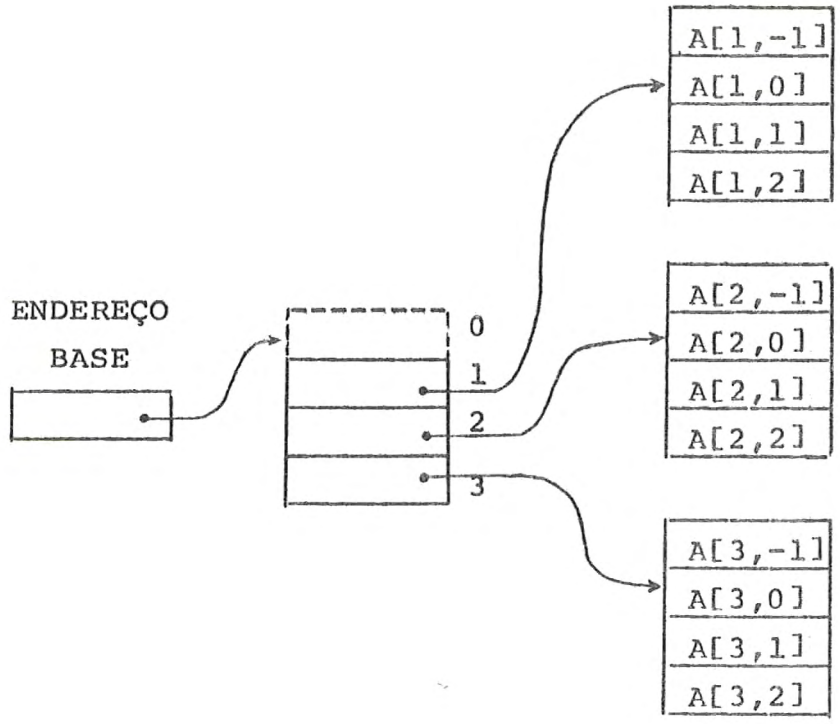
a) Uma matriz de uma dimensão é um vetor de posições de memória, isto é, um conjunto ordenado de posições consecutivas de memória, tendo um apontador para o elemento de índice zero desse vetor.

b) Uma matriz de k dimensões é um vetor de matrizes de k-1 dimensões e tem um apontador para o elemento de índice zero desse vetor.

Por exemplo, se tivermos uma matriz de duas dimensões, reservamos as posições correspondentes a cada linha e in-

troduzimos um vetor, onde cada elemento do mesmo, aponta para o elemento de índice zero de uma das linhas. O endereço base contém o endereço do elemento de índice zero do vetor de apontadores. Todo apontador aponta para o elemento de índice zero do próximo vetor para evitar a subtração do limite inferior daquela dimensão.

Exemplo: real array A[1:3, -1:2];



O acesso ao elemento A[I,J] é feito por:

end (A[I,J]) = conteúdo (endereço base (A)+I)+J

O cálculo do endereço não envolve nenhuma multiplicação, o que é uma vantagem em relação ao método do *dope vector* . Por outro lado gasta-se mais memória, pois além da área da matriz, precisamos das áreas para os apontadores. Pode-se otimizar o uso da memória, declarando-se a matriz de tal maneira que os índices mais à esquerda tenham menor limite de variação que os

mais à direita.

O endereço de um elemento qualquer $A[I_1,I_2,..I_k]$ é dado por:

$$end\ (A[I_1, I_2,..I_k]) = conteúdo\ (...\ conteúdo\ (conteúdo\ (endereço\ base\ (A) + I_1)+I_2...)+I_k$$

Para fazer a verificação de índices usando esse método, precisamos guardar os limites inferior e superior de cada dimensão. Cada vez que fazemos acesso a um apontador já é feita a verificação dos limites daquela dimensão.

O compilador ALGOL do B-6700 usa essa técnica para armazenar matrizes. Além disso, a verificação dos limites é feita ao mesmo tempo que é feito o acesso a um elemento. Junto com cada apontador guarda-se o comprimento do vetor para o qual ele aponta, ou seja, a amplitude da dimensão correspondente. Em tempo de compilação os limites L_j e U_j de cada dimensão de uma matriz, são transformados nos limites 0 (zero) e (U_j-L_j) e consequentemente, o acesso ao índice I_j é transformado no acesso ao índice (I_j-L_j). Dessa maneira, apenas conhecendo a amplitude de cada dimensão podemos, em tempo de execução, verificar a validade de seus índices correspondentes, pois o valor de cada índice deve estar entre 0 (zero) e a amplitude daquela dimensão. O B-6700 possue um "hardware" especial que analisa os limites ao mesmo tempo que faz o acesso ao próximo elemento dos vetores de apontadores.

Para cada matriz declarada em um programa em ALGOL para o computador B-6700, apenas uma posição da pilha de dados é ocupada, contendo o endereço base da matriz. As demais posições de memória associadas à matriz ficam fora da pilha de dados. Além disso, o computador B-6700 faz segmentação dessas posições, sendo a única segmentação de dados executada automaticamente.

No caso do compilador FORTRAN do computador B-6700

essa técnica não é usada, já que o FORTRAN impõe que as matrizes devem ser guardadas por colunas. Nesse caso, a técnica usada é a de linearização, que foi descrita em 2.1. A matriz é tratada como um vetor unidimensional, havendo portanto uma verificação do fato do endereço do elemento à ser usado estar dentro da área ocupada pela matriz.

3. OUTRAS PROPOSTAS

Veremos agora outras propostas existentes sobre tratamento de matrizes.

3.1 - Matrizes tipo own

Para fazer o tratamento de matrizes tipo own, veremos duas propostas, uma devida a Randell e Russell /8/ e a outra devida a Sattley /10/.

3.1.1 - Proposta devida a Sattley /10/

Um identificador tipo own é aquele cujo valor não muda entre uma desativação do bloco onde ele foi declarado e uma posterior ativação do mesmo bloco. A maneira de implementar esse tipo de declaração é colocar variáveis e matrizes tipo own em uma área de memória que nunca é liberada, nem mesmo após a desativação do bloco onde elas foram declaradas. Tudo se passa como se as variáveis tipo own fossem globais a todo o programa.

Sattley sugere que a área normal de variáveis e matrizes pode ficar em uma extremidade da memória, e a área para variáveis e matrizes tipo own pode ficar na outra extremidade. Quando é feita a declaração de uma matriz tipo own, precisamos verificar se é ou não a primeira vez que a matriz é declarada. Se for a primeira vez, reserva-se área de memória para a matriz no local onde ficarão as matrizes tipo own. Se não for a primeira

vez, o mecanismo é mais complexo. Verificam-se as expressões que são limites de cada dimensão. Se elas possuem o mesmo valor que possuiam na última ativação do bloco onde a matriz é declarada, não há nada a fazer. Entretanto, se alguma expressão limite possuir um valor diferente que o da última ativação, deve-se rearranjar os elementos da matriz para que satisfaçam a nova declaração. É preciso reservar um novo espaço de memória para a matriz, copiando nesse novo espaço o valor de todos os elementos cujos índices são comuns a ambas as declarações. Após ser feita a cópia, a área antiga da matriz pode ser liberada.

Depois de fazer a cópia de uma matriz tipo own, é interessante usar-se algum mecanismo de compactação de memória para que a área anteriormente usada por ela, não fique inutilizada.

3.1.2 - Proposta devida a Randell e Russell /8/

Randell e Russell propõem que os limites de cada dimensão de uma matriz tipo own sejam constantes inteiras. Dessa maneira, não é preciso rearranjar elementos a cada ativação do bloco onde a matriz foi declarada (ver 3.1.1) e também, ao final da compilação, sabemos exatamente qual é a quantidade de memória necessária para a área de variáveis e matrizes tipo own. Essa área é reservada imediatamente antes da área normal de dados do programa e tem seu tamanho fixo, conhecido em tempo de compilação.

Quando é feita a declaração de uma matriz tipo own, precisamos apenas saber se se trata ou não da primeira vez que o bloco onde ela está declarada está sendo ativado. Se for a primeira vez, reserva-se o espaço para a matriz; se não for, usa-se o espaço que foi reservado na primeira ativação do bloco.

A desvantagem desse sistema é que o espaço para cada matriz tipo own é reservado antes de sua primeira declaração, sendo portanto estático. Entretanto, ele permite o uso do meca-

nismo normal de pilha para os outros dados.

3.2 - Procedimentos como unidade de reserva de memória

Para fazer o estudo de procedimentos como unidade de reserva de memória, basear-nos-emos no que foi descrito por Gries /2/ e Huxtable e Hawkins /4/.

Podemos usar o procedimento, ao invés do bloco, como unidade de reserva de memória. A cada ativação de um procedimento, reserva-se a área de memória necessária para todas as variáveis simples declaradas em todos os blocos internos ao procedimento.

Dessa maneira, ao invés do registro de ativação de cada bloco ser dividido em duas partes (ver 2.3), é o registro de ativação de cada procedimento que passa a ser dividido em duas partes. Na primeira parte ficam todas as variáveis estáticas de todos os blocos do procedimento. Assim, durante a execução do programa objeto, quando um procedimento é ativado, o apontador AP deve apontar para o final da área de variáveis estáticas daquele procedimento. Na segunda parte fica a área real de matrizes e ela é reservada e liberada à medida que os blocos vão sendo ativados e desativados, respectivamente.

Em alguns momentos teremos posições reservadas, mas não usadas (se o bloco ativo não foi o que requer mais variáveis estáticas, dentro do procedimento); mas como se trata apenas de variáveis simples e de *dope vectors* de matrizes, a perda não é muito grande. A vantagem do método é necessitar apenas um apontador base para cada procedimento (ver I.1), e não de um para cada bloco. Entretanto, devido à alocação dinâmica de matrizes, precisamos conhecer a posição do topo da pilha em cada bloco interno ao procedimento, pelo fato de existir a possibilidade de um bloco ser desativado antes de encontrarmos seu end. Para isso, podemos criar um vetor de topos da pilha (vetor de AP). A cada ativação de um bloco, o valor atual de AP é colocado na primeira posi

ção disponível do vetor de AP, sendo que essa posição passa a não ser mais disponível. Assim, quando esse bloco é desativado através do end ou de um desvio, podemos atualizar corretamente o valor de AP.

O tamanho do vetor de AP deve ser igual à profundidade máxima de blocos alcançada dentro de um procedimento.

Uma outra maneira de se conhecer o valor de AP em cada bloco é reservar, no início da sua área, uma posição de memória para guardar-se o topo da pilha naquele bloco. Assim, ao invés de reservar-se um vetor de AP, como foi sugerido na solução anterior, passamos a ter cada posição do vetor de AP junto com a área de matrizes do bloco correspondente. Esse processo é semelhante ao da manutenção da cadeia estática, descrito em I.2.2 e I.2.3.

REFERÊNCIAS BIBLIOGRÁFICAS

/1/ Aho, A.V., Ullman, J.D. - *Principles of Compiler Design* - (1977), Addison-Wesley, California.

/2/ Gries, D. - *Compiler Construction for Digital Computers* - (1971), Wiley, New York

/3/ Griffits, M. - Run Time Storage Management. In *Compiler Construction An Advanced Course*, Bauer, F.L., Erckel, J. (Eds.), Lecture Notes in Computer Science, v. 21, (1974)

/4/ Huxtable, D.H.R., Hawkins, E.N. - A Multipass Translation Scheme for ALGOL 60 - *Annual Review in Automatic Programming*, 3, (1963), Pergamon Press, Oxford, pp 163-205

/5/ Iliffe, J.K. - *Basic Machine Principles* - (1968), Macdonald, New York

/6/ Knuth, D.E. - *The Art of Computer Programming*, vol 1, (1968), Addison-Wesley, New York

/7/ Nicholls, J.E. - *The Structure and Design of Programming Languages* - (1975), Addison-Wesley, London

/8/ Randell, B., Russell, D.J. - *ALGOL 60 Implementation* - (1964), Academic Press, London

/9/ Rohl, J.S. - *An Introduction to Compiler Writing* - (1975) Macdonald and Jane's, New York

/10/ Sattley, K. - Allocation of Storage for Arrays in ALGOL 60 *Comm ACM* 4, 1 (Jan 1961), pp 60-65

/11/ Wichmann, B.A. - *ALGOL 60 Compilation and Assessment* - (1973), Academic Press, London

/12/ Wirth, N. - *Algorithms + Data Structures = Programs* - (1976), Prentice Hall Inc., New Jersey

CAPÍTULO IV

IMPLEMENTAÇÃO DA LINGUAGEM "LAPA"

1. INTRODUÇÃO

A LAPA - Linguagem Algorítmica do PADE - foi definida para ser a única linguagem de alto nível do computador PADE /4,5/. Esta linguagem tem por objetivo a programação científica e a programação de sistemas, destinando-se à produção de todo o "software" do computador.

Uma descrição completa da LAPA, versão sequencial , pode ser encontrada no trabalho de G.Bressan /1/.O diagrama sintático da LAPA encontra-se no apêndice 1 deste trabalho.

Neste capítulo será descrita a implementação da linguagem LAPA, implementação esta que foi executada por G. Bressan e I. S. Homem de Melo, com a colaboração de L. M. M. Lago.

Vamos apresentar um resumo das principais características do computador PADE, sendo que uma descrição sumária de seu "assembler" (LIMPA-Linguagem de Montagem do PADE) encontra-se no apêndice 2.

Características do PADE:

(a) - Palavra de 24 bits, podendo ser subdividida em grupos de bits de comprimento 1,2,3,4,6,8,12 e 24. Cada uma destas subdivisões denomina-se *ion*, e o número de bits em um *ion* é denominado valência.

(b) - Registradores

- 8 registradores que podem ser usados para endereçamento, operações aritméticas e como apontadores de pilha.
- 2 registradores especiais para base da área de dados e da área de código do programa, denominados β e β' respec

tivamente.

. 1 registrador base local, denominado γ, utilizado para endereçamento da área de dados de procedimentos, sendo relativo a β.

(c) - Instruções - além de instruções convencionais, o PADE contém: instruções especiais de pilha, que podem ser utilizadas para transmissão de parâmetros e ligação entre procedimentos; instruções denominadas *iônicas*, que efetuam operações entre *ions*; instruções de controle de sequência, que repetem um grupo de instruções o número de vezes que for especificado em um contador; instruções denominadas *polacas*, que executam operações em notação polonesa com até cinco símbolos, incluindo operandos, operadores unários e binários, compactados em uma só palavra.

2. O COMPILADOR

Está sendo desenvolvida uma versão inicial do compilador, para um subconjunto da linguagem LAPA. Nessa versão inicial não estão sendo tratados comandos de entrada e saída, funções, procedimentos external e forward, variáveis tipo pointer (referência), char, alpha, ion e o operador cat (concatenação). Posteriormente, será feita uma otimização desse compilador e será incluído o tratamento dos itens citados acima. Daqui para frente, essa versão inicial será denominada de *compilador LAPA* ou simplesmente - *compilador*.

O compilador LAPA é um "cross-compiler" de um único passo. Ele está escrito em PL/I, sendo executado no computador IBM/360-44 do Instituto de Física da USP e gerando código de montagem em LIMPA. Uma versão em ALGOL para o B-6700 do CCE-USP foi desenvolvida a partir da versão em PL/I e está sendo mantida em paralelo com esta.

Posteriormente, pretendemos produzir um compilador escrito na própria linguagem de máquina do PADE, através da técni

ca de "bootstrapping", para ser executado no processador PADE ; durante a programação foi tomado especial cuidado para que posteriormente essa conversão seja facilitada.

O compilador está dividido em dezenove procedimentos, sendo que devido a problemas de disponibilidade de memória no IBM/360, é feito o "overlay" de alguns desses procedimentos.

Veremos agora as diversas partes em que está dividida a compilação e os procedimentos do compilador encarregados de executar cada parte.

(a) - Análise sintática

Procedimentos: CAREGA e INTERP.

(b) - Inicializações

Procedimento: INICIA.

(c) - Análise léxica e criação da Tabela de Símbolos

Procedimento: ANALEX.

(d) - Impressões do compilador

Procedimentos: ESCREV e IMPRIM.

(e) - Geração de código objeto

Procedimentos: ALOCA, EXECUT, EXECEM, EXECFM, EXE200, EXE300, GERA e VEREG.

(f) - Tratamento de constantes

Procedimentos: CONTAD e GERCON.

(g) - Alocação de registradores

Procedimento: CATREG.

(h) - Controle de tabelas

Procedimento: CTLTAB.

(i) - Emissão de mensagens de erro

Procedimento: ERRO.

3. PROCEDIMENTOS DO COMPILADOR

Descreveremos agora a função de cada procedimento relacionado no item anterior, bem como as tabelas usadas por eles e o tipo de compilação feita.

3.1 - Análise sintática

O compilador faz a análise sintática do programa fonte usando a técnica de produções de Floyd-Evans com subrotinas recursivas. Uma descrição detalhada desse processo é feita por Haynes /3/. Para usar essa técnica o compilador emprega duas pilhas, uma sintática (SINTAT) e a outra "semântica" (SEMANT), que andam em paralelo. As unidades sintáticas são colocadas na pilha sintática e as informações "semânticas" correspondentes são colocadas na pilha semântica. Daqui para a frente denominaremos as informações "semânticas" simplesmente de "semântica".

Durante a fase de desenvolvimento do compilador, as produções estão sendo interpretadas; planejamos converte-las para código em etapa posterior a fim de aumentar sua eficiência.

3.1.1 - Procedimento CAREGA

A função deste procedimento é criar um arquivo com as produções de Floyd-Evans de maneira adequada ao seu uso pelo compilador. Atualmente temos aproximadamente 410 produções perfuradas em cartões. Cada produção consta de:

- rótulo
- cinco campos do lado esquerdo (antes da redução)
- indicador de redução
- dois campos do lado direito (após a redução)
- descritor da ação, que pode conter

. chamada a um procedimento de geração de código ou,
. chamada a um procedimento que emite mensagens de erro ou,
. código que indica o fim da análise sintática.

- indicador de chamada ao analisador léxico
- rótulo da subrotina (das produções) que deve ser chamada
- rótulo da próxima produção a ser analisada.

O procedimento CAREGA le as produções,numera-as de 1 (um) em diante, troca o rótulo da subrotina e o rótulo do desvio pelo número da produção correspondente e cria um arquivo em disco que é imagem dos cartões a menos das trocas dos rótulos.

3.1.2 - Procedimento INTERP

Este procedimento é o que controla toda a compilação. Ele faz a interpretação da análise sintática, ou seja, verifica com qual produção as cinco últimas posições da pilha sintática mantêm igualdade e executa todas as ações descrita nessa produção.

3.2 - Inicializações

Existe um procedimento para fazer todas as inicializações necessárias ao compilador. Esse procedimento só é chamado uma vez, no início da compilação de um programa.

3.2.1 - Procedimento INICIA

A primeira tarefa executada por esse procedimento é carregar na memória o arquivo de produções de Floyd-Evans criado pelo procedimento CAREGA . Em seguida ele cria a tabela de palavras reservadas e uma outra tabela, que é acessada através do mesmo índice da primeira, onde é colocada a "semântica" correspondente a cada palavra reservada. A última tarefa desse procedimento é inicializar as demais tabelas, todas as opções do compilador e todas as variáveis que auxiliam a compilação.

3.3 - Análise léxica e criação da Tabela de Símbolos

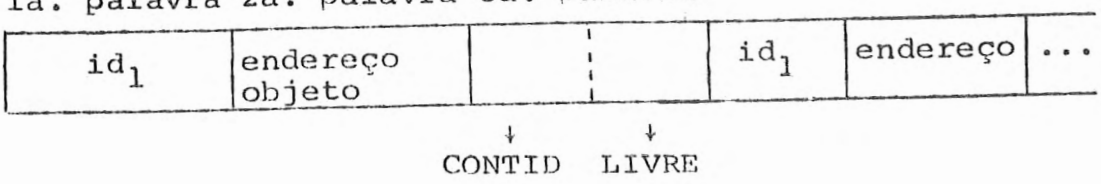
No compilador LAPA, o procedimento que obtém a próxima unidade léxica a ser analisada, também é encarregado de construir a tabela de símbolos (TABSIM) e a tabela de constantes - (TABCTE).

A estrutura da tabela de símbolos é a seguinte.

Um identificador é decomposto em n palavras de quatro caracteres, id_1, id_2,...id_n, das quais a primeira é colocada na TABSIM.

Cada identificador ocupa três palavras da TABSIM. A primeira contém id_1, a segunda contém seu endereço e a terceira consiste de dois apontadores. Os doze primeiros bits apontam para a tabela CONTID e os doze restantes para a tabela LIVRE.

TABSIM

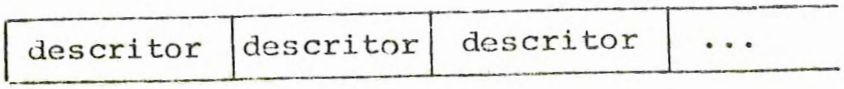


A tabela CONTID contém a segunda palavra, correspondente à continuação de um identificador. Por enquanto, as demais (n-2) palavras não são guardadas. Assim sendo, os oito primeiros caracteres é que distinguem o identificador.

Se na TABSIM o apontador para a CONTID assume o valor zero é porque o identificador consiste apenas de id_1.

A tabela LIVRE contém uma sequência de descritores. Cada descritor contém os atributos correspondentes a uma lista de identificadores com a mesma declaração.

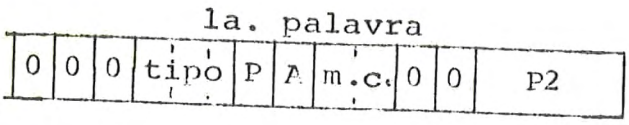
LIVRE



Cada descritor consiste de um grupo de palavras de 24 bits, com informações tais como: tipo, *dope vector*, tipo de chamada (quando o identificador for parâmetro), etc..

Um descritor tem uma das seguintes formas, de acordo com o tipo do identificador.

(a) Variável



P2: valência ou ponteiro para descritor

tipo: 0 0 0 - integer
0 0 1 - real
0 1 1 - ion
1 0 0 - unpackd ion
1 0 1 - pointer
1 1 0 - record

P: 0 - não é parâmetro formal
1 - é parâmetro formal

A: 0 - não é matriz
1 - é matriz

m.c. (modo de chamada)

0 0 - não foi especificado ou reference
0 1 - value
1 0 - result
1 1 - value result

As declarações boolean, char e alpha correspondem respectivamente a ion 24, ion 8 e ion 24.

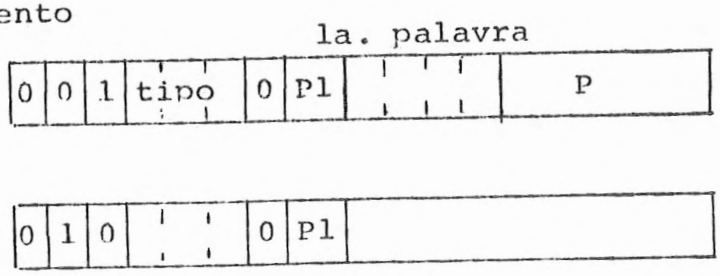
Se a variável é matriz, após a primeira palavra descrita acima virá o *dope vector*, que consiste da dimensão da matriz (k), de P1 e das constantes C_i, $2 \leq i \leq k$, onde

$$Pl = \sum_{j=1}^{k} L_j \, C_j$$

(ver III.2.1). Após o *dope vector* temos o número de elementos da matriz e o endereço do início da área da matriz (ver 3.5) . Se a matriz é um campo de um *record*, o endereço do início da área da matriz é substituído pelo endereço na área global em que se encontra o *dope vector* da mesma.

Quando a variável é um *record*, a segunda palavra é utilizada para dar o endereço do início da área do *record* (ver 3.5).

(b) Função ou procedimento

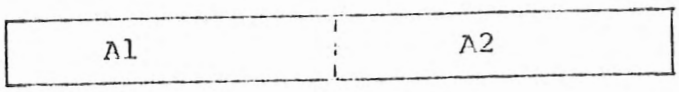


P : valência ou ponteiro para descritor

Pl : 0 - não tem parâmetros

1 - tem parâmetros

se Pl=1 então a segunda palavra do descritor será:

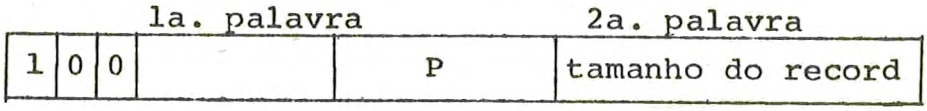


A1: apontador para o início da lista de parâmetros na TABSIM.

A2: apontador para o fim da lista de parâmetros na TABSIM.

Além disso, em seguida à palavra acima virão tantas palavras quantos parâmetros a função ou procedimento possuir, para descrever o tipo de seus parâmetros.

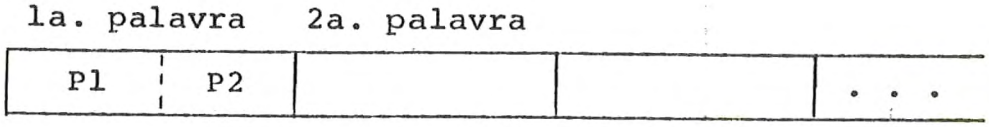
(c) *Record*



P: apontador para o primeiro componente do *record*

Se for matriz de *records*, ou uma variável cujo tipo é descrito por um *record*, o apontador P2 descrito em (a) aponta para o descritor do *record*, que é da forma descrita neste item.

O descritor de um grupo de componentes de um *record* é da forma:

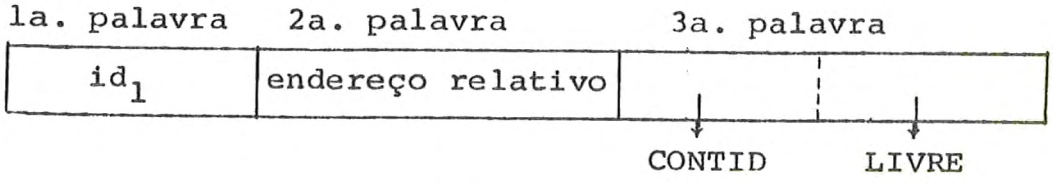


P1: ponteiro para o próximo componente após esse grupo de componentes

P2: ponteiro para o último componente deste grupo

A segunda palavra e as seguintes constituem o descritor propriamente dito desse grupo de componentes.

Cada componente de um *record* é colocado na LIVRE da seguinte forma:



Na segunda palavra o endereço é relativo ao início do *record* em que o componente se encontra.

(d) Rótulo

Além dessas tabelas descritas, existe umas outra , chamada LOCSIM, que é acessada através do mesmo índice da TABSIM, cuja finalidade é indicar a área de memória em que é alocado o correspondente identificador na TABSIM.

Se o identificador é uma variável, a LOCSIM pode assumir os seguintes valores com as respectivas interpretações:

0 - Global (variável alocada na área permanente)
1 - Dinâmica (variável alocada na área dinâmica)
2 - Local (variável relativa a Y, declarada em função ou procedimento)
3 - Registrador
4 - Matriz
8 - Parâmetro tipo reference
9 - Parâmetro tipo value
10 - Parâmetro tipo result
11 - Parâmetro tipo value result

Se o identificador é de um procedimento ou função, a LOCSIM pode assumir os seguintes valores, com as respectivas - interpretações:

0 - função ou procedimento declarado no programa
1 - função ou procedimento declarado como external
2 - função ou procedimento declarado como forward
3 - função ou procedimento pré-declarado (de biblioteca)

3.3.1 - Procedimento ANALEX

Este procedimento atualiza o valor da variável SINT com a próxima unidade léxica e da variável SEMA com a "semântica"

correspondente à unidade léxica. Essas duas variáveis são usadas pelo procedimento INTERP, que copia seus valores nos topos das pilhas sintática e semântica, respectivamente.

Se a unidade léxica é um identificador, e estão sendo compiladas declarações, esse identificador é colocado na TABSIM, e seu descritor, que foi construído à medida que foi sendo compilada o início da declaração, é colocado na LIVRE.

Se a unidade léxica é uma constante, ela é colocada na TABCTE, tomando-se o cuidado necessário para que essa tabela não tenha constantes duplicadas.

O procedimento ANALEX também compila os cartões de opção do compilador. Esses cartões podem ser colocados em qualquer ponto do programa fonte e têm o efeito de ligar ou desligar determinadas opções.

3.4 - Impressões do compilador

Todas as impressões produzidas pelo compilador são executadas por dois procedimentos: ESCREV e IMPRIM.

3.4.1 - Procedimento ESCREV

Este procedimento é responsável pelas seguintes impressões:

(a) - impressão de títulos, a cada nova página a ser impressa.

(b) - primeira impressão, onde é colocada a data da compilação.

(c) - impressão da TABSIM e TABCTE, após o primeiro comando de cada bloco (se estiver ligada a opção TABLE do compilador).

(d) - gravação do final do programa objeto, ao término da compilação (auxiliado pelo procedimento GERCON).

(e) - impressão do programa objeto (se estiverem ligadas as opções LIST e DECK do compilador).

3.4.2 - Procedimento IMPRIM

O procedimento IMPRIM é responsável pelas seguintes impressões:

(a) - impressão do programa fonte (se estiver ligada a opção SOURCE do compilador).

(b) - impressão do "trace" da análise sintática (se estiver ligada a opção TRACE do compilador).

(c) - impressão de títulos a cada nova página a ser impressa.

3.5 - Geração de código

A geração do código objeto está dividida em oito procedimentos.

Os procedimentos EXECUT, EXECEM, EXECFM, EXE200 e EXE300 poderiam constituir um só procedimento. Eles estão divididos devido a problemas com a memória disponível no IBM/360, para que possa ser feito seu "overlay". Esses procedimentos são chamados pelo procedimento INTERP. Os demais procedimentos de geração de código, ALOCA, GERA e VEREG, são chamados por um dos cinco procedimentos mencionados, para auxiliar a sua execução.

Esses procedimentos também são encarregados da alocação de memória. No computador PADE, o código e os dados de um programa constituem áreas distintas, endereçadas em relação aos registradores β' e β respectivamente.

A organização da área de dados é feita da seguinte maneira. As variáveis do programa, com exclusão dos dados de procedimento, são alocados em relação a β. Os dados locais a procedimentos são alocados em relação a γ, que constitui uma base local à área de dados. Matrizes e *records* situam-se em uma região distinta da de variáveis simples, devido ao alcance limitado do endereçamento direto. No caso de endereços relativos a β, o alcance é de 1024 palavras e relativos a γ é de 512 palavras.Dessa

forma, variáveis simples são endereçadas diretamente em relação a β ou γ, enquanto que o endereçamento de matrizes ou *records* é indireto, pós-indexado. Os apontadores (palavras de endereçamento indireto) de matrizes e *records* são mantidos nas primeiras 1024 palavras da área de dados, que podem ser endereçadas diretamente em relação a β.

Segundo a localização e forma de endereçamento, a área de dados compõe-se dos seguintes segmentos:

(a) - Segmento estático (SE), no qual são alocados dados globais e apontadores de matrizes e *records*. Situa-se nas primeiras posições da área de dados, sendo endereçadas em relação a β. No código objeto o rótulo GLB indica o início dessa área.

(b) - Segmento dinâmico de variáveis simples do programa (SDV), onde ficam as variáveis simples que não foram declaradas em procedimentos e que são endereçadas diretamente através de β. No código objeto o rótulo VAR indica o início dessa área.

(c) - Segmento dinâmico de matrizes e *records* do programa (SDM), onde ficam as matrizes e *records* que não foram declaradas em procedimentos e que são endereçadas através dos ponteiros do SE, com pós-indexação. No código objeto o rótulo MAT indica o início dessa área.

(d) - Segmento dinâmico de variáveis simples de procedimentos (SDVP), que são endereçadas diretamente através de γ. No código objeto, o rótulo GAMA indica o início dessa área.

(e) - Segmento dinâmico de matrizes e *records* de procedimentos (SDMP), que são endereçadas através dos ponteiros do SE, com pós-indexação. Para cada procedimento existe uma dessas áreas. No código objeto o rótulo MATGnp indica o início dessa área relativamente ao procedimento que recebeu o número np.

```
Exemplo: begin
           :
         procedure B: begin... end;
         procedure A: begin...; B;... end;
         procedure C: begin... end;
         A;
         C;
         end
```

Na figura 1 temos o esquema da área de dados durante a execução do procedimento B, chamado por A e na figura 2 temos o esquema da área de dados durante a execução do procedimento C.

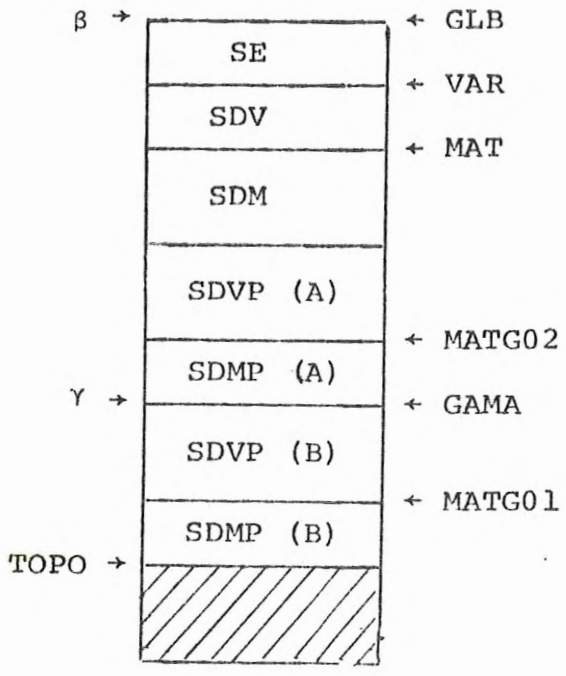


FIGURA 1

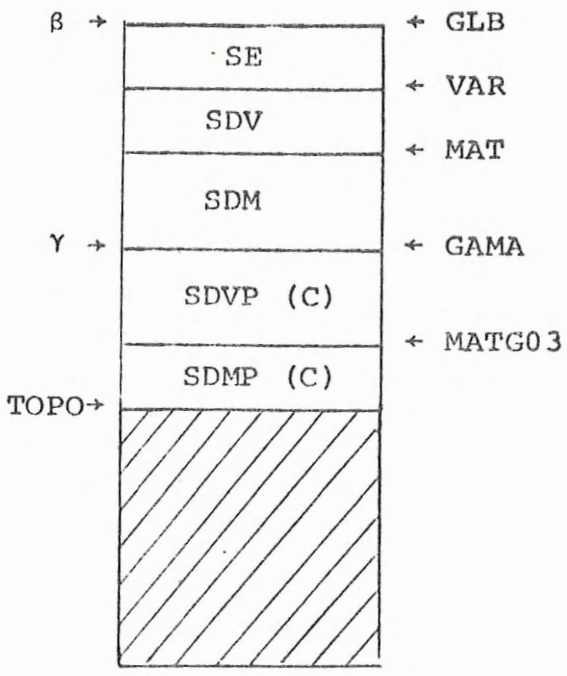


FIGURA 2

Para endereçar uma variável ou constante precisamos usar as informações da pilha semântica e através dela, as informações da TABSIM, LOCSIM e TABCTE. Na pilha semântica temos:

(a) - número do registrador (temporário) se a variável ou expressão está associada ao mesmo

(b) - endereço na TABCTE+1000 se é uma constante comum

(c) - endereço na TABCTE+2000 se é uma constante tipo string

(d) - endereço na TABSIM+4000 se as informações sobre a variável estão na TABSIM (nesse caso, para fazer o acesso precisamos consultar a LOCSIM)

(e) - número do temporário+5000 se é um temporário na memória.

No caso (a) o identificador associado à variável é "R" seguido do número do registrador.

No caso (b) a constante é associada ao identificador "Z" seguido do número que indica a célula da TABCTE onde se encontra a constante.

No caso (d) consultamos a célula adequada da LOCSIM, pois o identificador associado à variável é função desse valor.

valor da LOCSIM	identificador associado a variável
0 (variável global)	"GLB+" seguido do endereço que se encontra na segunda palavra da célula correspondente da TABSIM.
1 (variável dinâmica)	"VAR+" seguido do endereço que se encontra na segunda palavra da célula correspondente da TABSIM.
2 (variável local)	"GAMA+" seguido do endereço que se encontra na segunda palavra da célula correspondente da TABSIM , afetado do conteúdo de γ.
3 (registrador)	"R" seguido do número que se encontra na segunda palavra da célula correspondente da TABSIM.

No caso (e), o identificador associado ao temporário é "TEMP+" seguido do endereço que está na pilha semântica.

3.5.1 - Procedimento ALOCA

Este procedimento é encarregado da atribuição de um endereço a cada identificador declarado e da alocação de memória para cada variável declarada no programa fonte. Ele verifica o tipo da variável (variável simples, *record*, matriz, variável de procedimento, etc.), atribui um endereço a ela, sendo que esse endereço é colocado na posição devida da TABSIM.

Se a variável é um *record* ou matriz, o procedimento ALOCA gera a palavra para seu endereçamento indireto, e no caso de matriz, gera também seu *dope vector*. Essas informações são gerados na área global (SE).

Se a variável deve ser alocada num dos registradores gerais do PADE, o procedimento ALOCA chama o procedimento CATREG, que localiza um registrador disponível.

3.5.2 - Procedimento EXECUT

O procedimento EXECUT é chamado pelo procedimento INTERP e é encarregado da geração de código correspondente às declarações de procedimento. Além disso ele constrói os descritores de todas as variáveis, matrizes, *records* e procedimentos. Também chama o procedimento ALOCA para fazer a alocação de memória.

3.5.3 - Procedimento EXECEM

Este procedimento é chamado pelo INTERP e é encarregado da geração de código correspondente às expressões simples, expressões condicionais, atribuições e comandos seletivos (if , when). Ele chama os procedimentos CONTAD, GERA, GERCON e VEREG para que executem parte de suas tarefas. Quando os operan

dos de uma expressão são registradores, após o cálculo da expressão, o registrador do operando mais à direita é liberado.

3.5.4 - Procedimento EXECFM

Este procedimento é chamado pelo INTERP e é encarregado da geração de código correspondente aos comandos repetitivos for, repeat-until, repeat-forever, while-do, aos comandos seletivos index e case e às expressões seletivas. Ele chama os procedimentos CATREG, CONTAD, CTLTAB, ERRO, GERA e GERCON para que executem parte de suas tarefas.

3.5.5 - Procedimento EXE200

Este procedimento é chamado pelo INTERP e é encarregado da geração de código correspondente ao início e fim do programa, acesso a elementos de matrizes, chamada de procedimentos, rótulos e comando exit. Além disso ele também faz o controle do compilador sobre as entradas e saídas de blocos. Ele chama os procedimentos ALOCA, CATREG, CONTAD, CTLTAB, ERRO, ESCREV e GERA para auxiliá-lo nas suas tarefas.

3.5.6 - Procedimento EXE300

Este procedimento é chamado pelo INTERP e é encarregado da geração de código correspondente ao comando assemble.

Se entre as instruções pertencentes a um comando assemble houver operandos que fazem referência a um identificador de uma variável declarada no programa, ou a um rótulo de comando do programa (fora do assemble), este operando será substituído pelo endereço da variável ou rótulo. Ele chama os procedimentos ERRO e IMPRIM.

3.5.7 - Procedimento GERA

Este procedimento é chamado pelos procedimentos EXE-

EXECFM e EXE200.

Sua finalidade é gerar uma instrução executável, onde um operando é sempre um registrador e o outro operando está na pilha semântica (SEMANT), na célula indicada pelo parâmetro do procedimento (POS) e cujo mnemônico está em uma de três variáveis às quais o procedimento tem acesso. O procedimento verifica o tipo do operando que está na pilha semântica (constante, temporário, registrador, etc.) e, dependendo de seu tipo, gera uma instrução com um dos três mnemônicos relacionados. Ele chama os procedimentos CATREG, CONTAD e ERRO.

3.5.8 - Procedimento VEREG

Este procedimento é chamado pelo EXECEM. Sua finalidade é verificar se um determinado operando já está em algum registrador. Se não estiver, chama o procedimento GERA para gerar uma instrução que transfere o operando para um registrador disponível. Se o operando estiver em um registrador o número deste estará na pilha semântica.

3.6 - Tratamento de constantes

As constantes são alocadas na área de código, para que não possam ser modificadas. Dessa maneira, as instruções que manipulam as constantes devem acessar essa área, e portanto, no PADE, devemos utilizar instruções diferentes das que fazem acesso à área de dados.

Como o alcance das instruções que fazem referência à área de código é de 512 palavras, se o programa ocupar mais do que esse número de palavras, precisamos gerar as constantes antes do final do programa. Por essa razão, a qualquer momento durante a compilação precisamos saber o número de instruções geradas para verificar se esse número somado ao número de constantes na TABCTE é menor que 512. Se não for, todas as constantes existentes na TABCTE são geradas e essa tabela fica vazia. As novas

constantes encontradas são colocadas na TABCTE, mesmo se forem iguais a alguma constante já gerada (não existe endereçamento negativo no PADE).

3.6.1 - Procedimento CONTAD

Este procedimento recebe como parâmetro o número de instruções a serem geradas após sua chamada, e verifica se esse número somado ao número de palavras geradas até o momento (NPG) é menor que 512 palavras. Se for menor, ele incrementa NPG do valor recebido como parâmetro. Se não for menor ele chama o procedimento GERCON e inicializa NPG com o valor do parâmetro.

3.6.2 - Procedimento GERCON

Este procedimento é chamado pelos procedimentos CONTAD e ESCREV. Sua finalidade é gerar, na área de código, todas as constantes existentes na TABCTE, deixando esta tabela vazia.

Ele é chamado pelo procedimento ESCREV quando este último grava o final do programa objeto, pois nesse momento é necessário gerar todas as constantes que ainda estão na TABCTE.

3.7 - Alocação de registradores

O PADE possui oito registradores gerais denominados R0, R1,...R7 dos quais os sete últimos são usados pelo compilador como temporários, durante a compilação de uma expressão aritmética, ou de um comando for ou case. O compilador só aloca temporários na memória quando todos os registradores estão ocupados. O registrador R0 é usado como ponteiro de pilha, para que possa ser feito o salvamento dos demais registradores na ativação de um procedimento; o registrador R7 é também usado como indexador no endereçamento a matrizes e *records*.

Na LAPA existe a possibilidade de alocar variáveis em registradores, especificando qual o registrador desejado. No ca-

so de não ser especificado o registrador, o primeiro registrador disponível é alocado para a variável.

Para que as tarefas descritas acima possam ser feitas de maneira precisa e eficiente existe um procedimento para fazer todo o controle da alocação dos registradores.

3.7.1 - Procedimento CATREG

Este procedimento é chamado pelos procedimentos ALOCA, EXECUT, EXECEM, EXECFM e EXE200, toda vez que é necessário alocar um registrador.

Para controlar a alocação dos registradores ele usa duas tabelas auxiliares, INFO e APREG, com sete posições cada uma sendo que na posição I de cada uma delas temos informações sobre o registrador RI.

O procedimento recebe como parâmetro o número do registrador desejado ou um número maior que sete se puder ser alocado qualquer registrador.

A tabela INFO pode conter os seguintes valores, com as interpretações que os seguem:

(a) INFO (I) = 0 - significa que o registrador RI está disponível.

(b) INFO (I) = 1 - significa que o registrador RI está sendo usado pelo compilador, mas seu conteúdo pode ser transferido para um temporário na memória.

(c) INFO (I) = 2 - significa que o registrador foi alocado pelo programa em LAPA (uma declaração de variável especificava que ela deveria ser colocada em um registrador).

(d) INFO (I) = 3 - significa que o registrador I está sendo usado pelo compilador, e seu conteúdo não deve ser transferido para um temporário na memória.

Para os registradores usados como temporários, a tabela APREG indica a célula da pilha semântica que tem o registra-

dor como conteúdo.

Para os registradores alocados pelo programa em LAPA, a tabela APREG aponta para a célula da tabela de símbolos onde começam as declarações do bloco no qual o registrador foi alocado. Na desativação de um bloco usa-se essa informação para liberar ou não cada registrador.

Quando uma operação aritmética a ser compilada é uma multiplicação ou divisão, são necessários dois registradores consecutivos e nesse caso o procedimento recebe informação nesse sentido.

A alocação dos registradores é feita da seguinte maneira. A partir de R1 até R7, nessa ordem, procura-se um registrador livre. O primeiro registrador achado é alocado. Se não existir nenhum registrador livre, procura-se o registrador que está sendo usado como temporário (INFO (R) =1) e que é conteúdo da célula da pilha semântica mais próxima ao seu início, ou seja, o registrador usado há mais tempo. O conteúdo desse registrador é transferido para um temporário na memória e esse registrador é alocado. Se não existe nenhum registrador que está sendo usado como temporário, o procedimento ERRO é chamado. A liberação dos registradores é feita pelos procedimentos EXECEM, EXECFM, EXE200.

Se é gerada uma instrução para transferir o conteúdo de um registrador para um temporário na memória, chama-se também o procedimento CONTAD.

3.8 - Controle de tabelas

Existe um procedimento para controlar estouro ("over flow") de tabelas. As tabelas controladas são a TABSIM, TABCTE e LIVRE descritas em 3.3 e a tabela que auxilia a geração de código para o comando index (TINDEX).

3.8.1 - Procedimento CTLTAB

Este procedimento é chamado pelos procedimentos ANALEX, EXECFM e EXE200 para controlar o estouro das tabelas TABSIM, TABCTE, LIVRE e TINDEX. Cada uma dessas tabelas tem um apontador para a última célula utilizada. O procedimento recebe como parâmetro o número que deve ser acrescentado ao apontador. Após adicionar esse número ao conteúdo do apontador, ele verifica se o apontador não ultrapassou seu limite. Se ultrapassou, o procedimento ERRO é chamado para emitir uma mensagem de erro.

3.9 - Emissão de mensagens de erro

O compilador tem um procedimento encarregado de emitir todas as suas mensagens de erro.

3.9.1 - Procedimento ERRO

Este procedimento é chamado pelos procedimentos INTERP, EXECUT, EXECFM, EXE200, EXE300, GERA, ANALEX, CTLTAB e CATREG.

Ele emite uma mensagem com o número do erro e a posição do cartão do programa fonte onde ele ocorreu.

4 - CATÁLOGO DE COMPILAÇÃO (CÓDIGO OBJETO GERADO)

Veremos agora o código objeto gerado para as diferentes construções existentes na LAPA.

A notação usada será a seguinte. Para cada construção da LAPA será dada sua sintaxe e a geração de código da mesma. Se o código gerado para alguma parte da construção é visto em outro ítem, essa parte aparece entre os símbolos "{" e "}" (no código gerado). Dessa maneira, deve-se procurar o código objeto correspondente a elas e substituir no local apropriado.

Exemplo: comando when

Sintaxe: when <exp> then <com>

Código gerado: "código relativo ao when"
{<exp>}
"código relativo ao then"
{<com>}

Neste caso, as partes relativas a "<exp>" e "<com>" devem ser substituídas pela tradução da respectiva expressão e do respectivo comando. Para as outras partes é dado o código correspondente.

Em todos os comandos onde é necessário usar um registrador como temporário, usaremos a notação "Ri". Entende-se que "Ri" deve ser substituído pelo primeiro registrador disponível - (obtido pelo procedimento CATREG).

Os rótulos são representados pela letra I seguida de um número natural e os temporários de memória por TEMP+ℓ onde ℓ é um número natural.

Quando deve ser feito algum comentário a respeito de uma instrução gerada, colocaremos um número entre parênteses após a instrução e o comentário virá após o código gerado, precedido pelo mesmo número.

Se a sintaxe de alguma construção ficar longa, alguma parte da sintaxe será substituída por um símbolo não terminal , cuja sintaxe será dada em seguida.

4.1 - Comando if

Sintaxe: if <exp> then <com$_1$> else <com$_2$>

Código gerado:

```
{<exp>}
T     Ri,<exp>
DF    Ri,Ij
{<com1>}
```

(1)

```
        D      15, Ik
Ij      EQV    .
       {<com2>}
Ik      EQV    .
```

Comentários:

(1) - Omitida se a <exp> já estiver em um registrador.

4.2 - Comando when

Sintaxe: when <exp> then <com>

Código gerado:

```
        {<exp>}
        T      Ri, <exp>          (1)
        DF     Ri, Ij
        {<com>}
Ij      EQV    .
```

Comentários:

(1) - ver comentário (1) de 4.1.

4.3 - Comando while

Sintaxe: while <exp> do <com>

Código gerado:

```
Ij      EQV    .
        {<exp>}
        T      Ri, <exp>          (1)
        DF     Ri, Ik
        {<com>}
        D      15, Ij
Ik      EQV    .
```

Comentários:

(1) - ver comentário (1) de 4.1.

4.4 - Comando repeat-until

Sintaxe: repeat <com> until <exp>

Código gerado:

```
Ij      EQV     .
        {<com>}
        {<exp>}
        T       Ri, <exp>          (1)
        DF      Ri, Ij
```

Comentários:

(1) - ver comentário (1) de 4.1.

4.5 - Comando repeat-forever

Sintaxe: repeat <com> forever

Código gerado:

```
Ij      EQV     .
        {<com>}
        D       15,Ij
```

4.6 - Comando for

Sintaxe: [for <id cont>][from <exp_1>]to <exp_2>[by<exp_3>]do <com>

As partes entre colchetes podem ser omitidas.Se <exp_1> ou <exp_2> forem omitidas supomos seus valores iguais a 1 (um).

Código gerado:

```
        {<exp1>}
        T       Ri,<exp1>          (1)
        TM      Ri,<id cont>       (2)
        {<exp2>}
        {<exp3>}
        S       Ri, <exp2>
        T       Rj, <exp3>         (3)
```

```
            TM      Rj, TEMP+n              (4)
            FNP     Ri, Rj, /, +1,NP1       (5)
            DF      Ri, Ij
            D       15,Ik                   (2)
Im          T       Rj, TEMP+n              (6)
            AM      Rj, <id cont>           (2)
Ik          EQV     .
            {<com>}
            I       Ri, Im                  (7)
Ij          EQV     .
```

Comentários:

(1) - Gerada se existir a parte from. Se não existir trocar por

```
TI      Ri,1
```

(2) - Omitida se não existir a parte for.

(3) - Gerada se existir a parte by. Se não existir trocar por

```
AI      Ri,1
```

(4) - Omitida se não existir a parte for e a parte by.

(5) - Omitida se não existir a parte by.

(6) - Gerada se existir a parte for e a parte by. Se não existir a parte by mas existir a parte for trocar por

```
Im      TI      Rj,1
```

(7) - Gerada se existir a parte for. Se não existir trocar por

```
I       Ri, Ik
```

Observação - Se existirem vários for encaixados, usa-se o mesmo registrador para controlar todos eles. Isso é feito visando eco-

nomizar os registradores, já que eles são usados como temporários. Para isso, no começo de um for, que ocorre dentro do comando de um outro for, é gerada a instrução:

```
TM    Ri, TEMP+k
```

que salva o conteúdo do registrador do for externo, e no fim do for interno é gerada a instrução

```
T     Ri, TEMP+k
```

que restaura o conteúdo do registrador mencionado. Para implementar essa economia de registradores usamos uma solução que permite no máximo dez for's encaixados.

4.7 - Comando index

Sintaxe: index <exp> of <const$_1$>→<com$_1$>,<const$_2$>→<com$_2$>,..., <const$_n$>→<com$_n$>, out→<com$_{n+1}$>

Código gerado:

```
        {<exp>}
        T       Ri, <exp>                    (1)
        CP      Ri, Xn (@)
        D       11, Ik
        TP      Ri, Xn (@)
Ik      DP      15,Xs (@)'
Xs      END     Xj (Ri,@)'
IC1     EQV     .
        {<com1>}
        D       15,Ip
IC2     EQV     .
        {<com2>}
        D       15,Ip
        .
        .
        .
```

```
IC_n    EQV     .
        {<com_n>}
        D       15,Ip
Xn      C10     m                                   (2)
Iq      EQV     .
        {<com_n+1>}
        D       15,Ip
Xj      EQV     .
        END     Icte_1                              (3)
        END     Icte_2                              (4)
        :
        END     Icte_u                              (5)
        END     Iq
Ip      EQV     .
```

Comentários:

(1) - ver comentário (1) de 4.1.

(2) - m = max (<const$_1$>,<const$_2$>,... <const$_n$>)+1.

(3) - Icte$_1$ é o rótulo do comando precedido pela constante de valor numérico 1.

(4) - Icte$_2$ é o rótulo do comando precedido pela constante de valor 2.

(5) - se alguma constante de valor entre 1 e m-1 (ver (2)) não ocorre, a instrução "END Icte$_i$" correspondente será "END Iq" ou seja, o rótulo é o mesmo que o do comando que segue ao out.

Observação - Esse comando só deve ser usado quando as constantes assumem valores pequenos. Por uma questão de simplicidade do compilador, é utilizada uma tabela onde vão sendo armazenados os rótulos gerados nas constantes de endereço explicadas em (3),(4) e

(5); eles são armazenados na ordem que devem ser gerados. Como essa tabela tem 50 posições, a máxima constante admitida é 49. Além disso, quando temos index encaixados usa-se a mesma tabela para todos os index o que restringe ainda mais o valor máximo que a constante pode assumir.

4.8 - Comando case

Sintaxe: case <exp> of $<exp_1>\rightarrow<com_1>$, $<exp_2>\rightarrow<com_2>$, ..., $<exp_n>\rightarrow<com_n>$, out $\rightarrow <com_{n+1}>$

Código gerado:

```
                {<exp>}
                T     Ri, <exp>                    (1)
                {<exp1>}
                C     Ri,<exp1>
                D     6, Ik1
                {<com1>}
                D     15, Ik
Ik1             EQV   .
                {<exp2>}
                C     Ri, <exp2>
                D      6, Ik2
                {<com2>}
                D     15, Ik
                :
Ikn-1           EQV   .
                {<expn>}
                C     Ri, <expn>
                D      6, Ikn
                {<comn>}
                D     15, Ik
Ikn             EQV   .
                {<comn+1>}
Ik              EQV   .
```

Comentário:

(1) - ver comentário (1) de 4.1.

Observação - Se existirem vários case encaixados, usa-se o mesmo registrador para controlar todos eles. Isso é feito visando economizar os registradores, já que eles são usados como temporários. Para isso, no início de um case que ocorre dentro do comando de outro case é gerada a instrução

```
TM      Ri, TEMP+k
```

que salva o conteúdo do registrador do case externo e o comentário (1) passa a não existir pois a instrução que transfere a expressão para o registrador precisa existir já que queremos um registrador específico e certamente a expressão não está neste registrador. No fim do case interno é gerada a instrução

```
T       Ri, TEMP+k
```

que restaura o conteúdo do registrador mencionado. Para implementar essa economia de registradores usamos uma solução que permite no máximo dez case's encaixados.

4.9 - Comando de atribuição

Sintaxe: <var_1>:=<var_2>:= ... <var_n>:=<exp>

Código gerado:

```
{<exp>}
T       Ri,<exp >                    (1)
TM      Ri,<var_n>
:
TM      Ri,<var_2>
TM      Ri,<var_1>
```

Comentário:

(1) - ver comentário (1) de 4.1.

4.10 - Ativação de procedimento

Sintaxe: <nom do proc> (<exp>,<exp>,...,<exp>)

Se o procedimento não tiver parâmetros o trecho entre parentesis deve ser omitido.

Código gerado:

Inicialmente temos o cálculo dos parâmetros, que depende do tipo especificado para os mesmos e do fato da chamada estar sendo feita do programa principal ou de outro procedimento.

Para cada parâmetro que é matriz (tipo reference) é gerado o seguinte trecho de código:

```
T     Ri, GLB+n
TM    Ri, MAT+m+endpar                (1)
T     Ri, GLB+n+1                     (2)
TM    Ri, MAT+m+endpar+1              (3)
:                                     (4)
T     Ri, GLB+n+k
TM    Ri, MAT+m+endpar+k              (1)
                                      (5)
```

Para cada parâmetro cujo tipo é descrito por um *record* (parâmetro tipo reference) é gerado o seguinte trecho de código:

```
T     Ri, GLB+n
TM    Ri, MAT+m+endpar                (1)
                                      (2)
```

Para cada parâmetro tipo reference (que não seja matriz nem *record*), result ou value result é gerado o seguinte tre

cho de código:

```
{<exp>}
TE      Ri,<exp>
TM      Ri, MAT+m+endpar                    (2)
```

Para cada parâmetro tipo value é gerado o seguinte trecho de código:

```
{<exp>}
T       Ri,<exp>                            (6)
TM      Ri, MAT+m+endpar                    (2)
```

O trecho seguinte depende do fato da chamada estar sendo feita do programa principal ou de um outro procedimento.

Se a chamada é feita do programa principal temos:

```
TE      RO, MAT+m                           (7)
T       Ri, 0 (RO)
SI      Ri, 9
TM      Ri, ABP                             (8)
```

Se a chamada é feita de outro procedimento temos:

```
TE      RO, MATGnpro+m (@)                  (7)
```

O trecho final do código é o seguinte:

```
T       PO, 0 (@)
TEG     RO, 1 (RO)
DS      PO, Pnpro                           (9)
T       RO,0 (@)
SI      RO, 9
```

Comentários:

(1) - n é o endereço da palavra de endereçamento indireto da matriz ou do *record* em questão. Instruções para copiar o *dope vector* (no caso de matriz).

(2) - Se a chamada é feita de outro procedimento trocar por

```
TM    Ri, MATGnpro+m+endpar (@)
```

Se não existe matriz declarada no programa principal (ou no procedimento que chamou) até o ponto da chamada, m assume o valor nove; caso contrário, m assume o valor do tamanho da área de matrizes, cujo endereço inicial é MAT ou MATGnpro (programa principal ou procedimento), incrementado de dez; endpar assume o valor do endereço do parâmetro em relação ao início da área do procedimento que está sendo ativado; npro é o número associado ao procedimento chamador. Para o significado de MAT e MATGnpro ver o item 3.5.

(3) - Para o valor de n ver comentário (1). Omitida se não for matriz de *records* nem de *ions*.

(4) - Se a chamada é feita de outro procedimento trocar por

```
TM    Ri,MATGnpro+m+endpar+1 (@)
```

(ver comentário (2)). Omitida se não for matriz de *records* nem de *ions*.

(5) - Se a chamada é feita de outro procedimento trocar por

```
TM    Ri,MATGnpro+m+endpar+k (@)
```

onde k assume o valor da dimensão da matriz, pois se a matriz tem k dimensões precisamos copiar k+1 elementos na área do procedimento (ver comentário (2)).

(6) - Ver comentário (1) de 4.1.

(7) - Para os valores de m e npro ver o comentário (2).

(8) - Para o significado de ABP ver o item 5.4.

(9) - Pnpro indica o identificador associado ao procedimento.

Observação - A parte relativa a parâmetros não está implementada.

4.11 - Comando exit

Existem duas formas para a sintaxe do exit:

(a) - Sintaxe: exit <nom de proc>

Código gerado:

```
D     15, FPk                    (1)
```

Comentário:

(1) - FPk indica o rótulo associado ao fim do procedimento.

(b) - Sintaxe: exit <nom de com>

Código gerado:

```
T     RO, ABP                    (1)
TR    PO, B'1101111111'          (1)
D     15, Fk                     (2)
```

Comentário:

(1) - Omitida se o exit não é feito de um procedimento para o programa principal. Para o significado de ABP ver o item 5.4.

(2) - Fk indica o rótulo associado ao fim do comando.

4.12 - Comando assemble

Sintaxe: assemble (<inst>;...;<inst>)

Código gerado:

O código gerado é a própria instrução em LIMPA, formatada e com todos os identificadores e rótulos que constam da TABSIM, trocados pelo identificador usado no código objeto.

4.13 - Rótulo de comando

Sintaxe: <nom com>:<com>

Código gerado:

```
Ck    EQV    .                    (1)
      {<com>}
Fk    EQV    .                    (2)
```

Comentários:

(1) - Ck indica o rótulo associado ao início do comando.

(2) - Fk indica o rótulo associado ao fim do comando.

4.14 - Declaração de procedimento

Sintaxe: procedure (<$parm_1$>,...,<$parm_n$>)<nom proc>:<com>

Se o procedimento não tiver parâmetros, a parte entre parentesis deve ser omitida.

Código gerado:

O trecho inicial do código objeto é o seguinte:

```
      D      15, PPk
Pk    EQV    .                    (1)
      TRM    PO, B'0001111111'
```

Em seguida, para cada parâmetro tipo value result é

gerado o seguinte trecho de código:

```
        T     Ri, GAMA+endpar (@)'      (2)
        TM    Ri, GAMA+endparl (@)      (3)
```

Em seguida vem o trecho:

```
        {<com>}
FPk     EQV   .
```

O trecho seguinte é repetido para cada parâmetro tipo result e value result

```
        T     Ri, GAMA+endparl (@)      (3)
        TM    Ri, GAMA+endpar (@)'      (2)
```

O trecho final do código é o seguinte:

```
        TR    PO, B'1101111111'
PPk     EQV   .
        ORD   INICIO+m                  (4)
MATGk   MEM   0
        ORD   INICIO+n                  (5)
```

Comentários:

(1) - Pk é o rótulo que indica o início do procedimento, sendo que k é o número associado ao mesmo.

(2) - endpar é a posição onde fica o endereço do parâmetro atual (ver 5.6).

(3) - endparl é o endereço do parâmetro usado pelo procedimento (ver 5.6).

(4) - m assume o valor do tamanho da área de variáveis simples

de procedimento (SDVP).

(5) - n assume o valor do tamanho da área de dados globais e apontadores para matriz (SE).

4.15 - Declaração de matriz

Sintaxe: [<const$_1$>:<const$_2$>,...,<const$_{2n-1}$>:<const$_{2n}$>]
<tipo> <ident>

onde tipo é: <tipo simples> ou <id tipo rec> ou <decl rec>

Para cada matriz declarada é gerado no segmento estático (SE) a palavra para endereçamento indireto da matriz e em seguida seu *dope vector* que consta da constante P1 com sinal trocado (ver (a) de 3.3) e dos C_i $2 \leq i \leq k$ onde k é a dimensão da matriz.

Código gerado:

```
END     MAT-INICIO+m (R7)        (1)
C10     '-P1'                    (2)
C10     'C2'
 :
C10     'CK'
```

Comentários:

(1) - Se a matriz está declarada em um procedimento, trocar por

END MATGk-INICIO+m(R7,@)

onde k é o número associado ao procedimento e m assume o valor do tamanho atual da área de matrizes e *records* do procedimento (SDMP). Se a matriz está declarada no programa principal, m assume o valor do tamanho da área de matrizes e records (SDM). Se não for matriz de *records* nem matriz de *ions*, m assume o valor descrito acima decrementado de P1 (ver (a) de 3.3). Para o significado de MAT e MATGk

ver o item 3.5.

(2) - Primeira instrução correspondente à geração do *dope vector*.

4.16 - Declaração de uma variável cujo tipo é descrito por um *record*.

Sintaxe: <tipo> <ident>

onde <tipo> é da forma *record* (<dc>;<dc>;...<dc>) ou faz referência a um tipo definido por um *record* e <dc> é de uma das seguintes formas:

<tipo><lista id> ou (<dc>/<dc>/.../<dc>)

Código gerado: END MAT-INICIO+m (R7) (1)

Além dessa palavra para endereçamento indireto do *record*, se <tipo> for da forma "*record* (<dc>; <dc>;... <dc>)" , para cada componente que for matriz é gerado seu *dope vector* (ver 4.15).

Comentários:

(1) - ver comentário (1) de 4.15.

4.17 - Acesso a um elemento de uma matriz

Sintaxe: <ident> [$<exp_1>$,$<exp_2>$,... $<exp_n>$]

Código gerado :

O trecho inicial é o seguinte:

```
{<exp1>}
T       R7, <exp1>
TE      Ri, GLB+(m+2)                    (1)
{<exp2>}
T       Rj, <exp2>                       (2)
FNP     R7, Pi, Rj, *, +
:
{<expn>}
T       Rk, <expn>                       (2)
FNP     R7, Pi, Rk, *, +
A       R7, GLB+(m+1)                    (5)
T       Ri, p                            (6)
FNP     R7, Ri, *, NPl, NPl
A       R7, q                            (7)
```

Em seguida devemos acessar o elemento da matriz.

Se o elemento da matriz aparece do lado esquerdo de uma atribuição, cujo lado direito é $\langle exp_{n+1} \rangle$ o trecho final do código gerado é o seguinte:

```
{<expn+1>}
T       Rn, <expn+1>                     (2)
TM      Rn, GLB+m'                       (3)
```

Se o elemento da matriz aparece do lado direito de uma atribuição, o trecho final do código objeto gerado é:

```
T       Rn, GLB+m'                       (4)
```

Comentários:

(1) - Omitida se a matriz tiver uma única dimensão. Ri tem que ser um registrador par. m é o endereço da palavra de endereçamento indireto da matriz. Se a matriz tiver mais do

que uma dimensão e for parâmetro trocar por

```
TE      Ri, GAMA+s (@)
```

onde s assume o valor de m+2 se for matriz de *ions* ou matriz de *records* ou assume o valor de m+1 caso contrário.

(2) - Ver comentário (1) de 4.1.

(3) - m é o endereço da palavra de endereçamento indireto da matriz. Se a matriz for parâmetro trocar por

```
TM      Rn, GAMA+m (@)'
```

(4) - Para o valor de m ver comentário (3). Se a matriz for parâmetro trocar por:

```
T       Rn, GAMA+m (@)'
```

(5) - As instruções a partir desta devem ser omitidas se não for matriz de *records*. Para o valor de m ver o comentário (3). Se a matriz for parâmetro e for matriz de *records* trocar por

```
A       R7, GAMA+(m+1) (@)
```

(6) - p assume o valor do tamanho do *record*.

(7) - q assume o valor do *offset* do componente a ser acessado.

Observação - Quando um índice de uma matriz de mais de uma dimensão (que não seja o primeiro índice) é um elemento de outra matriz, ou quando elementos de matrizes aparecem de ambos os lados de uma atribuição, é necessário salvar o conteúdo do registrador R7, para fazer o acesso à outra matriz. Para isso, no início do cálculo do índice da segunda matriz é gerada a instrução

```
TM      R7, TEMP+k
```

que salva o conteúdo do registrador R7. Após o acesso ao elemen-

to da matriz introduz-se a instrução:

```
T       R7, TEMP+k
```

que restaura o conteúdo do registrador R7. Para implementar essa manipulação do registrador R7 foi necessário introduzir uma restrição. O número máximo que o nível de salvamentos pode atingir é dez ou seja, podemos salvar no máximo dez valores de R7 sem que nenhum deles seja restaurado.

Só estão implementadas as matrizes de uma dimensão e matrizes que não são parâmetros.

4.18 - Acesso a um componente de um *record*

Sintaxe: $\langle ident_1 \rangle \cdot \langle ident_2 \rangle \cdot \langle ident_n \rangle$ onde

$\langle ident_i \rangle$ pode ser da forma $\langle ident \rangle [\langle exp_1 \rangle, \ldots, \langle exp_n \rangle]$

Código gerado:

O código gerado vai depender do fato de o *record* possuir ou não componentes que são matrizes. Se nenhum componente do *record* for matriz, o código gerado é o seguinte:

```
T       R7, endcom                                    (1)
```

Se algum componente do *record* for matriz, precisamos calcular o endereço do elemento da matriz antes de fazer o endereçamento do componente. Neste caso o trecho inicial do código gerado consiste do seguinte.

Para cada componente que é matriz precisamos calcular o endereço de seu elemento e isso é feito da maneira que foi descrita em 4.17;nesse caso precisamos trocar (m+1) pelo endereço do primeiro elemento do *dope vector* e (m+2) pelo endereço do segundo elemento; também devemos omitir o trecho do código que faz acesso ao elemento da matriz.

No registrador R7 deve ficar a somatória dos valores encontrados para cada matriz, pois cada valor representa uma parcela do deslocamento.

O trecho seguinte do código é:

```
A       R7, endcom                                    (1)
```

Se o componente do *record* aparece do lado esquerdo de uma atribuição cujo lado direito é <exp>, o trecho final do código gerado é o seguinte:

```
{<exp>}
T       Ri, <exp>                                     (2)
TM      Ri, GLB+m'                                    (3)
```

Se o componente do *record* aparece do lado direito de uma atribuição, o trecho final do código objeto gerado é:

```
T       Ri, GLB+m'                                    (4)
```

Comentários

(1) - endcom assume o valor do *offset* do componente a ser acessado.

(2) - ver comentário (1) de 4.1.

(3) - m é o endereço da palavra de endereçamento indireto do *record*. Se o *record* for parâmetro trocar por

```
TM      Ri, GAMA+m (@)'
```

(4) - Para o valor de m ver comentário (3). Se o *record* for parâmetro trocar por

```
T       Ri, GAMA+m (@)'
```

Observação - O acesso a um componente de um *record* não está implementado.

4.19 - Expressões

Sintaxe: if <exp_1> then <exp_2> else <exp_3> ou
index <exp> of <$const_1$>→<exp_1>,...,<$const_n$> →
<exp_n>, out →<exp_{n+1}>

ou

case <exp> of <exp_1>→<exp_2>,...,<exp_{2n-1}>→
<exp_{2n}>,out →<exp_{2n+1}>

ou

<exp simples>

a) if <exp_1> then <exp_2> else <exp_3>

Código gerado :

```
        {<exp1>}
        T       Ri,<exp1>                (1)
        DF      Ri, Iℓ
        {<exp2>}
        T       Rj, exp2                 (1)
        D       15, Ik
Iℓ      EQV     .
        {<exp2>}
        T       Rj,<exp2>
Ik      EQV     .
```

Comentário:

(1) - Ver comentário (1) de 4.1.

b) index <exp> of <$const_1$>→<exp_1>,...,<$const_n$>→<exp_n>,out→<exp_{n+1}>

Código gerado:

```
        {<exp>}
        T       Ri,<exp>                 (1)
        CP      Ri, Xn (@)
        D       11, Ik
```

```
        TP      Ri, Xn (@)
Ik      DP      15, Xs (@)'
Xs      END     Xj (Ri,@)'
IC1     EQV     .
        {<exp1>}
        T       Ri, <exp1>
        D       15, Ip
        :
ICn     EQV     .
        {<expn>}
        T       Ri, <expn>
        D       15, Ip                                  (2)
Xn      C10     m
Iq      EQV     .
        {<expn+1>}
        T       Ri, <expn+1>
        D       15, Ip
Xj      EQV     .                                       (3)
        END     Icte1                                   (4)
        END     Icte2
        :                                               (5)
        END     Icteu
        END     Iq
Ip      EQV     .
```

Comentários:

Ver os correspondentes de 4.7.

Observação - Ver observação de 4.7. Além disso, se tivermos index encaixados, usa-se o mesmo registrador para conter o resultado - de todos eles. Para isso, no início de um index que ocorre dentro da expressão de outro index é gerada a instrução

```
TM      Ri, TEMP+k
```

que salva o conteúdo do registrador do index e o comentário (1) passa a não existir pois precisamos transferir a expressão para um registrador específico. No fim do index interno é gerada a instrução

```
T       Ri, TEMP+k
```

que restaura o conteúdo do registrador mencionado. Para implementar essa economia de registradores usamos uma solução que permite no máximo dez index's encaixados.

c) case <exp> of $\langle exp_1 \rangle \rightarrow \langle exp_2 \rangle, \ldots, \langle exp_{2n-1} \rangle \rightarrow \langle exp_{2n} \rangle$, out $\rightarrow \langle exp_{2n+1} \rangle$

Código gerado:

```
        {<exp>}
        T       Ri, <exp>                    (1)
        {<exp1>}
        C       Ri, <exp1>
        D       6, Ik1
        {<exp2>}
        T       Ri, <exp2>
        D       15, Ik
Ik1     EQV     .
        :
Ikn-1   EQV     .
        {<exp2n-1>}
        C       Ri, <exp2n-1>
        D       6, Ikn
        {<exp2n>}
        T       Ri, <exp2n>
        D       15, Ik
Ikn     EQV     .
        {<exp2n+1>}
        T       Ri, <exp2n+1>
Ik      EQV     .
```

Comentário:

(1) - ver comentário (1) de 4.1.

Observação - Ver observação de 4.8.

d) <exp simples>

Sintaxe: <termo$_1$> <op rel> <termo$_2$>

onde <op rel> pode ser >, >=, <, <=, = ou neq

Código gerado:

```
{<termo1>}
{<termo2>}
T     Ri, <termo1>          (1)
C     Ri, <termo2>
TI    Ri, -1
D     cod, .+2              (2)
TI    Ri, 0
```

Comentário:

(1) - ver comentário (1) de 4.1.

(2) - cod assume um dos seguintes valores dependendo de <op rel>

<op rel>	cod
=	1
<	2
<=	3
>	4
>=	5
neq	6

e) <termo>

Sintaxe: <ter sim$_1$> <op> <ter sim$_2$>

onde <op> pode ser +, -, fadd, fsub, or ou xor

Código gerado: {<ter sim_1>}
{<ter sim_2>}
T Ri, <ter sim_1> (1)
mne Ri, <ter sim_2> (2)

Comentários:

(1) - ver comentário (1) de 4.1.

(2) - mne assume um dos seguintes valores dependendo de <op>

<op>	mne
+	A
-	S
fadd	AR
fsub	SR
or	O
xor	X

Observação - Posteriormente será incluído o tratamento de "overflow" e mistura de tipos.

f) <ter sim>

Sintaxe: <$fator_1$> <op mul> <$fator_2$>

onde <op mul> pode ser *, /, fmul, fdiv, mul, mod, and, shl ou sha.

Código gerado: {<$fator_1$>}
{<$fator_2$>}

O trecho seguinte depende de <op mul>.

Se <op mul> for fmul, fdiv, * ou / o trecho seguinte fica:

```
T       Ri, <fator1>                    (1)
ZAD     Ri, 24                          (2)
mne     Ri, <fator2>                    (3)
```

Se <op mul> for mod ou mul o trecho seguinte fica:

```
T       Ri, <fator1>
T       Rj, <fator2>
FNP     Ri, Rj, cod, NP1, NP1           (4)
```

Se <op mul> for and, shl ou sha o trecho seguinte fica:

```
T       Ri, <fator1>                    (1)
mne     Ri, <fator2>                    (5)
```

Comentários:

(1) - ver comentário (1) de 4.1.

(2) - Se <op mul> for fmul ou * essa instrução deve ser omitida.

(3) - mne assume um dos seguintes valores, dependendo de <op mul>

<op mul>	mne
fmul	MR
fdiv	QR
*	M
/	Q

(4) - cod assume um dos seguintes valores, dependendo de <op mul>

<op mul>	cod
mod	MOD
mul	*

(5) - mne assume um dos seguintes valores dependendo de <op mul>

<op mul>	mne
and	E
shl	ZAS
sha	ZLS

Observação - Posteriormente será incluído o tratamento de "overflow" e mistura de tipos.

g) - <fator>

Sintaxe: <prim$_1$> ** <prim$_2$> ou
<op un> <prim> ou

onde <prim> pode ser: <const> ou
<ident> ou
<nom fun> (<exp$_1$>,...,<exp$_n$>) ou
(<exp>)

Para a sintaxe <prim$_1$>** <prim$_2$> temos:

Código gerado :

```
TM      R2, SALVA+9
TE      R2, SALVA+8
TRM     P2, B'0000011110'
AI      R2, 5
T       R6, <prim1>
T       R3, <prim2>
DS      P2, EXPO
T       R2, SALVA+9
T       Ri, PARM1
```

Para a sintaxe <op un> <prim> temos:

Código gerado:

```
T       Ri,<prim>                    (1)
FNP     Ri, cod,NP1, R1, NP2         (2)
```

Comentários:

(1) - ver comentário (1) de 4.1.

(2) - cod assume um dos seguintes valores,dependendo de <op un>

<op un>	cod
not	C1
fabs	ABR
incr	+1
decr	-1
halve	/2
dup	*2
neg	NBS
abs	ABS

<op un>	cod
-	TRS
fneg	NBR
clear	ZER
sign	TS
load false	C-1
minus	CR
norm	NRR

4.20 - Programa

Sintaxe: <com> ou
<decl proc ou fun>

Código gerado:

```
                ORD     $
INICIO          MEM     0
GAMA            MEM     0
GLB             MEM     0
                ORD     .
                {<com>}ou {<decl proc ou fun>}
                ORD     INICIO+m        (1)
VAR             MEM     n               (2)
TEMP            MEM     j               (3)
PARM1           MEM     1               (4)
SALVA           MEM     10              (4)
ABP             MEM     1
MAT             MEM     0
                ORD     .
```

```
                D       15, Iℓ
Zℓ1             C10     'k1'                    (5)
Zℓ2             C10     'k2'
                :
Zℓn             C10     'kn'
EXPO            TI      R5,1                    (6)
                ZLD     R3,1
                CI      R4,0
                D       5, .+5
                FNP     R5,R6,*,NP1,NP1
                D       7, .+2
* TRATA OVERFLOW
                TI      R3,0
                DF      R3,.+3
                FNP     R6,R6,*,NP1,NP1
                D       7,.-8
* TRATA OVERFLOW
                TM      R5, PARM1
                SI      R2, 4
                TR      P2, B'0100011110'
Iℓ              EQV     .
                FIM
```

Comentários:

(1) - m assume o valor do número de variáveis globais (variáveis no SE) do programa.

(2) - n assume o valor do número de variáveis (variáveis no SDV).

(3) - j assume o valor do número de temporários.

(4) - Omitida se não houve operação de exponenciação no programa.

(5) - Início da geração das constantes que ainda estão na TABCTE. $Z\ell_1$ é o identificador associado à constante e k_1 assume o

valor da constante.

(6) - Procedimento para exponenciação. Se não houve nenhuma operação de exponenciação no programa esse procedimento deve ser omitido.

Observação - Posteriormente será incluído o tratamento de "overflow" no procedimento de exponenciação.

5 - ALGUMAS JUSTIFICATIVAS SOBRE O CÓDIGO OBJETO GERADO

Para algumas construções da LAPA o código objeto gerado é feito da maneira convencional, não dependendo da arquitetura do PADE. Mas para outras construções o código objeto gerado é dependente da arquitetura. Veremos agora em quais construções ocorreu esse fato e a justificativa correspondente ou seja, que aspecto da arquitetura induziu a forma do código gerado.

Além disso, veremos também as justificativas das construções que foram baseadas nos estudos descritos nos três primeiros capítulos.

5.1 - Comando for

No comando for todas as expressões são calculadas uma única vez. Isso é feito para que, antes de iniciar a execução do comando correspondente, possamos saber quantas vezes o for deve ser executado. Dessa maneira, podemos usar a instrução itera (de mnemônico I) do PADE, que controla automaticamente o número de repetições.

5.2 - Comando index

No comando index, para fazer acesso ao trecho apropriado de código, usamos um desvio indireto e indexado sobre uma

tabela de endereços denominada Xj . Mas como o alcance de uma instrução desse tipo é de dezesseis palavras e, provavelmente, a tabela Xj estará num endereço que dista de mais de dezesseis palavras dessa instrução, o desvio é feito através de duas instruções. A primeira instrução faz um desvio indireto sobre uma palavra que contém um endereço (D 15, Xs (@)'). Essa palavra indica a tabela apropriada, o registrador usado para a indexação e a indicação de desvio indireto (Xs END Xj (Ri,@)' onde Xj é o rótulo associado ao início da tabela).

5.3 - Controle de case, index e for encaixados

No caso dos comandos case, index e for encaixados, tomamos providências no sentido de economizar registradores. Isso é feito pelo fato dos registradores também serem usados como temporários em expressões aritméticas, sendo que as operações de multiplicação e divisão usam dois registradores cada uma. Quanto mais registradores disponíveis tivermos, maior é a chance de gerarmos um código objeto eficiente, já que temporários na memória produzem um código objeto maior. Assim sendo, decidimos usar no máximo um registrador para cada sequência de comandos encaixados de um mesmo tipo. No caso de ocorrer um encaixamento, o conteúdo do registrador correspondente ao comando é armazenado em um temporário. O número deste último é então carregado em uma pilha , cujo tamanho máximo é de dez células, donde a restrição mencionada em 4.6, 4.8 e no item b) de 4.19.

5.4 - Ativação de procedimentos

No computador PADE só existem dois registradores base para dados, denominados β e γ, usados para o programa principal e o procedimento ativo respectivamente. Por essa razão, na linguagem LAPA não são permitidas declarações de procedimentos encaixados (ver I.3.7). Além disso, para que a alocação de memória

pudesse ser feita em tempo de compilação, as matrizes são estáticas (ver III.2.1). Baseados nesses fatos, adotamos na LAPA a proposta de Gries e Wichmann (ver I.3.1), ou seja, todos os blocos de um procedimento são endereçados através do mesmo registrador base. O compilador calcula a área máxima necessária em qualquer ponto do procedimento, determinando assim o valor de m de 4.14 . Para efeito do programa principal, o registrador β permanece estático. Assim sendo, a pilha de bases (ver I.2) reduz-se a apenas um elemento, o registrador γ.

Pelo que vimos acima, o algoritmo de ativação de um procedimento em LAPA é bem mais simples do que o que foi descrito em I.2.2.1. Precisamos apenas armazenar o estado da computação e o endereço de retorno, o apontador para a cadeia dinâmica, calcular os parâmetros e atribuir um novo valor a γ.

O salvamento dos registradores é feito através da instrução TR do PADE, colocando-se um operando adequado. Como essa instrução necessita de um registrador para ponteiro de pilha, escolhemos o registrador RO para essa finalidade.

Assim, a ativação de um procedimento é feita através dos seguintes passos:

a) calcula os parâmetros

b) atribui a RO o endereço da pilha onde será colocada a marca do procedimento

c) armazena na pilha o apontador para a cadeia dinâmica (salva γ)

d) atribui um novo valor a γ

e) desvia para o procedimento

As primeiras instruções do procedimento executam as seguintes tarefas:

a) salvamento dos registradores

b) cópia dos parâmetros tipo value result na área local ao procedimento (ver 5.6)

Além disso, quando a ativação do procedimento é feita no programa principal, colocamos na variável ABP o número de posições que foram reservadas na pilha para os dados do programa principal. Esse valor é usado pelo comando exit (ver 5.5).

5.5 - Desativação de procedimento

Na desativação normal (pelo end) de um procedimento, como é necessário atualizar a pilha de bases, só precisamos restaurar os valores dos registradores, de γ e desviar para o endereço de retorno. Todas essas tarefas são executadas pela instrução TRM do PADE, colocando-se um operando adequado.

A desativação também pode ser feita pelo comando exit. Se o exit é feito para o identificador do procedimento, o processo é igual ao descrito no parágrafo anterior.

Se o exit é feito para um rótulo declarado fora do procedimento, certamente o rótulo pertence ao programa principal - (na LAPA não são permitidas declarações de procedimentos encaixados). Nesse caso, todos os procedimentos que estão na pilha são desativados. Isso é feito com apenas uma instrução, retornando o primeiro conjunto de registradores que foi salvo e que está na pilha. A posição desse conjunto é indicada pela variável ABP.

5.6 - Cálculo dos parâmetros

Na LAPA existem quatro tipos de parâmetros: value, result, value result e reference. Para cada parâmetro tipo reference que não seja matriz nem *record*, é reservada uma posição de memória na área do procedimento, que contém o endereço do parâmetro atual. Para cada parâmetro tipo value é reservada uma posição de memória na área do procedimento, que contém o valor do parâmetro atual. Para cada parâmetro tipo value result ou result são reservadas duas posições de memória na área do procedimento. Uma posição contém o endereço do parâmetro atual e a outra con-

têm seu valor, sendo que essa última é a posição usada pelo procedimento.

Assim, o cálculo dos parâmetros consiste do seguinte. Se o parâmetro é tipo reference (a não ser matrizes e *records*), result ou value result, colocamos seu endereço na posição adequada. Se é tipo value, colocamos seu valor. No procedimento geramos instruções que copiam o valor de cada parâmetro tipo value result na posição adequada. No fim do procedimento, antes da instrução TRM descrita em 5.5, geramos instruções que retornam os valores dos parâmetros tipo result e value result.

Matrizes e *records* como parâmetros são tratados de maneira diferente. Neste caso sempre supomos que o tipo do parâmetro é reference.

Se o parâmetro for matriz, copiamos na área do procedimento a palavra de endereçamento indireto e o *dope vector* da matriz (ver 4.15).

Essa cópia poderia ter sido feita de maneira diferente em relação ao exposto em 4.10, constando das seguintes instruções:

```
TE      Ri, GLB+n                          (1)
TE      Rj, MAT+m+endpar                   (2)
R       15, k
FNP     Pi, Pj, :=, NPl, NPl
```

Comentários:

(1) - ver comentário (1) de 4.10.

(2) - ver comentário (2) de 4.10.

(3) - k assume o valor do número de dimensões da matriz incrementado de um, se for matriz de *records* ou matriz de *ions*; caso contrário k assume o valor do número de dimensões da matriz.

O esquema descrito em 4.10 usa sempre um registrador a menos que o esquema proposto neste item. Por outro lado,quando a matriz tiver uma dimensão, em geral o esquema descrito em 4.10 terá duas instruções a menos; para matrizes de duas dimensoẽs, em geral os dois esquemas terão o mesmo número de instruções (a não ser no caso de matriz de *records* ou *ions*). Para matrizes de mais dimensões, o esquema descrito em 4.10 terá sempre mais instruções. Quanto maior o número de dimensões da matriz, mais instruções terá o esquema descrito em 4.10 ao passo que o esquema descrito neste item terá sempre quatro instruções.

Seria interessante fazer um estudo estatístico,verificando o número de registradores e o número de dimensões usados por programas em geral, para verificar qual dos dois esquemas é melhor.

5.7 - Matrizes

A manipulação de matrizes é feita de maneira semelhante à descrita em III.2.1. As matrizes são estáticas e portanto reservamos sua área em tempo de compilação. Não testamos se um índice está dentro dos limites declarados pois isso aumenta o código objeto gerado. (ver III.2.2).

O endereçamento de matrizes é indireto pós-indexado (ver 3.5). Além disso, precisamos escolher um determinado registrador para indexação de matrizes, pois geramos a palavra para endereçamento indireto quando compilamos a declaração da matriz. Por essa razão, num mesmo instante, só podemos estar calculando o índice de um único elemento de uma matriz, já que qualquer índice tem que estar no registrador R7. Assim, precisamos elaborar um esquema de salvamento desse registrador e optamos por uma solução que limita o número de salvamentos (num mesmo instante) em dez.

Uma outra opção para o tratamento de matrizes,poderia ser deixar de gerar o *dope vector* na área global e apenas guardá

-lo na tabela de símbolos do compilador. Neste caso, a cada acesso a um elemento de uma matriz, introduzíríamos o elemento apropriado do *dope vector* através de uma instrução imediata. Entretanto, se esse elemento não coubesse numa instrução do tipo mencionado, teríamos que gerar uma constante com o valor do elemento a cada acesso (por possíveis problemas de alcance). Por essa razão e também por nos parecer mais "organizado" optamos por gerar o *dope vector* na área de dados do programa objeto e usar seus componentes a cada acesso a um elemento da matriz.

Pode-se estranhar que as matrizes locais a procedimentos estão com seu *dope vector* na área global (isto é, são endereçados em relação a β). Para colocá-los na área local (relativa a γ) teria sido necessário gerar um certo número de instruções adicionais para cada matriz, na ativação do procedimento. Isso implicaria em perda de espaço na área de código em geral não compensando o ganho obtido na área de dados. Seria interessante fazer um estudo com a finalidade de encontrar uma base para uma solução definitiva.

5.8 - *Records*

Na linguagem LAPA cada *record* pode ser encarado como uma definição do tipo (ver II.4). Na tabela de símbolos guardamos todas as informações necessárias para alocar uma variável daquele tipo (tamanho, descrição dos componentes, etc.)(ver 3.3). A alocação de *records* é estática, já que nas declarações de um programa em LAPA constam todas as variáveis cujo tipo é descrito por um *record*.

O acesso a um componente de um *record* é bem simples, pois junto com a descrição de cada componente, guardamos também o *offset* do mesmo. Assim, para fazer o acesso, colocamos o *offset* do componente no registrador R7 e em seguinda acessamos o mesmo através da palavra para endereçamento indireto da variável cujo tipo é descrito por um *record* (endereçamento indireto pós-indexado, como para matrizes (ver 3.5 e 5.7)).

Os componentes que ocupam a mesma posição de memória dentro de um *record* (definidos entre barras (/)) possuem o mesmo *offset* e o tamanho reservado é o do maior componente.

6 - EXEMPLO

Veremos agora um programa em LAPA, com as construções que se encontram implementadas e o respectivo código objeto gerado pelo compilador LAPA.

Na listagem do programa fonte, os comandos estão numerados de um em diante. Na listagem do programa objeto, antes da primeira instrução do código objeto relativo a um determinado comando, colocamos o número atribuído ao mesmo.

```
$ LIST DECK
          BEGIN
 1        (0:10) INTEGER M1;
 2        (2:5) INTEGER M2;
          INTEGER A,B,C,D,E,PEN,ULT;
          INTEGER F REGISTER;
 3        PROCEDURE ALFA;
             BEGIN
 4           (-1:4) INTEGER M3;
             INTEGER D REGISTER 6;
 5           M3(-1):=0;
 6           M3(2):=M1(0)+D;
 7           M3(M2(2)):=B:=A+2;
 8           D:=M1(3):=C+M1(2);
 9           END;
10        PROCEDURE BETA;
             BEGIN
11           FOR I FROM 1 TO 3 BY 1 DO
12               FOR K TO I DO
13                 A:=I+K;
14           ALFA;
15           END;
16        WHILE B<10 DO
17           IF B>8 THEN B:=B+2
18                  ELSE IF B<2 THEN B:=B+C
19                                 ELSE B:=M1(2*A+3);
20        INDEX F-C OF
21           3 => D:=C,
22           1 => BEGIN
23                INDEX C OF
24                   2 => D:=5,
25                   1 => E:=D:=(2*B)**2,
26                 OUT => WHEN B=5 THEN B:=-1
                  END,
27           4 => B:=C/D,
28         OUT => F:=0;
          ASSEMBLE(
29                 TI T0,0(R1);
30                  SP F,ROT(3);
31                  C  P2,A+5;
32                  DP X'F',ROT+1;
33            ROT:  MEM 0;
34                  T  F,B;
35                  TM F,A);
36        CASE A+B OF
37          C+D => BETA,
38          E*F => REPEAT BEGIN
39                        F:=PEN+ULT;
40                        ULT:= F;
41                        END
42                 UNTIL ULT > 100,
43          A/B => A:=3,
44          OUT => A:=(A+A+(A+A+(A+A+(A+A+(A+A+(A+A+(A+A+(A+A+(A+A+(A+A+(A+A+(A+A)))))))
                   )));
45        END.
```

LAPA

```
                ORD  $
        INICIO  MEM  0
        GAMA    MEM  0
        GLB     MEM  0
                ORD  .
                ORD  $
1               END  MAT-INICIO+00000(R7)
                C10  '00000'
2               END  MAT-INICIO+00009(R7)
                C10  '-00002'
3               D    15,PP0001
        P0001   EQV  .
                TRM  P0,8'0001111111'
4               END  MATG01-INICIO+00001(R7,')
                C10  '00001'
                ORD  .
5               TP   R2,Z1004(')
                FNP  R2,TRS,NP1,R1,NP2
                TI   R7,0(R2)
                TP   R2,Z1000(')
                TM   R2,GLB+0004'
6               TP   R7,Z1003(')
                TM   R7,TEMP+0000
                TP   R7,Z1000(')
                T    R2,GLB+0000'
                T    R7,TEMP+0000
                AI   R2,0(R6)
                TM   R2,GLB+0004'
7               TP   R7,Z1002(')
                T    R2,GLB+0002'
                TI   R7,0(R2)
                T    R2,VAR+0000
                AP   R2,Z1002(')
                TM   R2,VAR+0001
                TM   R2,GLB+0004'
8               TP   R7,Z1007(')
                TM   R7,TEMP+0000
                TP   R7,Z1002(')
                T    R2,GLB+0000'
                T    R7,TEMP+0000
                T    R3,VAR+0002
                AI   R3,0(R2)
                TM   R3,GLB+0000'
                TI   R6,0(R3)
9       FP0001  EQV  .
                TR   P0,8'1101111111'
        PP0001  EQV  .
                ORD  INICIO+0000
        MATG01  MEM  0
                ORD  INICIO+0006
10              D    15,PP0002
        P0002   EQV  .
                TRM  P0,8'0001111111'
                ORD  .
11              TP   R2,Z1004(')
```

```
            TM   R2,GAMA+0000(')
            SP   R2,Z1007(')
            TP   R3,Z1004(')
            TM   R3,TEMP+0000
            FNP  R2,R3,/,+1,NP1
            DF   R2,I102
            D    15,I101
      I103  T    R3,TEMP+0000
            AM   R3,GAMA+0000(')
      I101  EQV  .
12          TM   R2,TEMP+0001
            TI   R2,1
            TM   R2,GAMA+0001(')
            S    R2,GAMA+0000(')
            AI   R2,)
            DF   R2,I105
            D    15,I104
      I106  TI   R3,1
            AM   R3,GAMA+0001(')
      I104  EQV  .
13          T    R3,GAMA+0000(')
            A    R3,GAMA+0001(')
            TM   R3,VAR+0000
            I    R2,I106
      I105  EQV  .
            T    R2,TEMP+0001
            I    R2,I103
      I102  EQV  .
14          TE   R0,MATG02+9(')
            T    P0,0(')
            TEG  R0,.(R0)
            DS   P0,P0002
            T    R0,0(')
            SI   R0,9
15    FP0002 EQV  .
            TR   P0,B'1101111111'
      PP0002 EQV  .
            ORD  INICIO+0002
      MATG02 MEM  0
            ORD  INICIO+0006
            ORD  .
16    I107  EQV  .
            T    R2,VAR+0001
            CP   R2,Z1002(')
            TI   R2,-1
            D    2,.+2
            TI   R2,0
            DF   R2,I108
17          T    R2,VAR+0001
            CP   R2,Z1003(')
            TI   R2,-1
            D    4,.+2
            TI   R2,0
            DF   R2,I109
            T    R2,VAR+0001
```

```
                AP   R2,Z1002(')
                TM   R2,VAR+0001
                D    15,I110
          I109  EQV  *
18              T    R2,VAR+0001
                CP   R2,Z1002(')
                TI   R2,-1
                D    2,*+2
                TI   R2,0
                DF   R2,I111
                T    R2,VAR+0001
                A    R2,VAR+0002
                TM   R2,VAR+0001
                D    15,I112
          I111  EQV  *
19              TP   R2,Z1002(')
                M    R2,VAR+0000
                AP   R3,Z1007(')
                TI   R7,0(R3)
                T    R2,GLB+0000'
                TM   R2,VAR+0001
          I112  EQV  *
          I110  EQV  *
                D    15,I107
          I106  EQV  *
20              TI   R2,0(R1)
                S    R2,VAR+0002
                CP   R2,X101(')
                D    11,I113
                TP   R2,X101(')
          I113  DP   15,X102(')'
          X102  END  X100(R2,')'
21        I115  EQV  *
                T    R2,VAR+0002
                TM   R2,VAR+0003
                D    15,I114
22        I116  EQV  *
23              T    R2,VAR+0002
                CP   R2,X104(')
                D    11,I117
                TP   R2,X104(')
          I117  DP   15,X105(')'
          X105  END  X103(R2,')'
24        I119  EQV  *
                TP   R2,Z1003(')
                TM   R2,VAR+0003
                D    15,I118
25        I120  EQV  *
                TP   R2,Z1002(')
                M    R2,VAR+0001
                TM   R2,SALVA+9
                TE   R2,SALVA+8
                TRM  P2,B'0000011110'
                AI   R2,5
                TI   R6,0(R3)
```

LAPA

```
                TP   R3,Z1002(')
                DS   P2,EXPO
                T    R2,SALVA+9
                T    R2,PARM1
                TM   R2,VAR+0003
                TM   R2,VAR+0004
                D    15,I113
26      X104    C10  '0003'
        I123    EQV  *
                T    R2,VAR+0001
                CP   R2,Z1003(')
                TI   R2,-1
                D    1,*+2
                TI   R2,)
                DF   R2,I122
                TP   R2,Z1004(')
                FNP  R2,TRS,NP1,R1,NP2
                TM   R2,VAR+0001
        I122    EQV  *
                D    15,I118
        X103    EQV  *
                END  I120
                END  I119
                END  I12.
        I118    EQV  *
                D    15,I114
27      I123    EQV  *
                T    R2,VAR+0002
                ZAD  R2,34
                Q    R2,VAR+0003
                TM   R3,VAR+0001
                D    15,I114
28      X101    C10  '0005'
        I124    EQV  *
                TP   R1,Z1000(')
                D    15,I114
        X100    EQV  *
                END  I116
                END  I124
                END  I115
                END  I123
                END  I124
        I114    EQV  *
29              TI   TO,0(R1)
30              SP   R1,POT(')
31              C    P2,VAR+0000+5
32              DP   X'F',ROT+1
33      ROT     MEM  0
34              T    R1,VAR+0001
35              TM   R1,VAR+0000
36              T    R2,VAR+0000
                A    R2,VAR+0001
37              T    R3,VAR+0002
                A    R3,VAR+0003
                CI   R2,0(R3)
```

```
                D     6,I126
                TE    R0,MAT+0024
                TI    R3,0(R0)
                SI    R3,9
                TM    R3,ABP
                T     P0,0(*)'
                TEG   R0,1(R3)
                DS    P0,P0002
                T     R0,0(*)
                SI    R0,9
                D     15,I125
38      I126    EQV   .
                T     R3,VAR+0004
                MI    R3,0(R1)
                CI    R2,0(R4)
                D     6,I127
        I128    EQV   .
39              T     R3,VAR+0005
                A     R3,VAR+0006
                TI    R1,0(R3)
40              TM    R1,VAR+0006
42              T     R3,VAR+0006
                CP    R3,Z1009(*)
                TI    R3,-1
                D     4,.+2
                TI    R3,0
                DF    R3,I128
                D     15,I125
43      I127    EQV   .
                T     R3,VAR+0000
                ZAD   R3,24
                Q     R3,VAR+0001
                CI    R2,0(R4)
                D     6,I129
                TP    R3,Z1007(*)
                TM    R3,VAR+0000
                D     15,I125
44      I129    EQV   .
                T     R3,VAR+0000
                A     R3,VAR+0000
                T     R4,VAR+0000
                A     R4,VAR+0000
                T     R5,VAR+0000
                A     R5,VAR+0000
                T     R6,VAR+0000
                A     R6,VAR+0000
                T     R7,VAR+0000
                A     R7,VAR+0000
                TM    R3,TEMP+0000
                T     R3,VAR+0000
                A     R3,VAR+0000
                TM    R4,TEMP+0001
                T     R4,VAR+0000
                A     R4,VAR+0000
                TM    R5,TEMP+0002
```

```
         T    R5,VAR+0000
         A    R5,VAR+0000
         TM   R6,TEMP+0003
         T    R6,VAR+0000
         A    R6,VAR+0000
         AI   R5,0(R6)
         AI   R4,0(R5)
         AI   R3,0(R4)
         AI   R7,0(R3)
         T    R3,TEMP+0003
         AI   R3,0(R7)
         T    R4,TEMP+0002
         AI   R4,0(R3)
         T    R3,TEMP+0001
         AI   R3,0(R4)
         T    R4,TEMP+0000
         AI   R4,0(R3)
         TM   R4,VAR+0000
I125     EQV  *
         ORD  INICIO+0006
45
VAR      MEM  0007
TEMP     MEM  0004
PARM1    MEM  1
SALVA    MEM  10
ABP      MEM  1
MAT      MEM  0
         ORD  *
         D    15,I130
Z1000    C10  '0000000'
Z1001    C10  '0000001'
Z1002    C10  '0000002'
Z1003    C10  '0000015'
Z1004    C10  '0000001'
Z1005    C10  '0000004'
Z1006    C10  '0000016'
Z1007    C10  '0000013'
Z1008    C10  '0000008'
Z1009    C10  '0000100'
EXPO     TI   R5,1
         ZLD  R3,1
         CI   R4,0
         D    5,*+5
         FNP  R5,R6,*,NP1,NP1
         D    7,*+2
* IMPRIME OVERFLOW
         TI   R3,0
         DF   R3,*+3
         FNP  R6,R6,*,NP1,NP1
         D    7,*-8
* IMPRIME OVERFLOW
         TM   R5,PARM1
         SI   R2,4
         TR   P2,B'0100011110'
I130     EQV  *
         FIM
```

REFERÊNCIAS BIBLIOGRÁFICAS

/1/ - Bressan,G. - Linguagens de Implementação de Sistemas e a Linguagem LAPA - Dissertação de Mestrado apresentada no Instituto de Matemática e Estatística da USP, São Paulo, (dez. 1977).

/2/ - Bressan, G., Homem de Melo, I.S. - Uma Linguagem de Alto Nível Aderente a uma Arquitetura e sua Implementação - IV Seminário sobre Desenvolvimento Integrado de Software e Hardware, UFMG, Belo Horizonte (Julho 1977).

/3/ - Haynes, Herbert R. - An Optimizing Compiler for an Extended Version of the Floyd-Evans Production Language - TRM-12, Computation Center, The University of Texas at Austin, (Mar. 1969).

/4/ - Mammana, C.Z. et alli - Digital Processor for Data Acquisition (PADE) - Annual Report, Nuclear Physics Department, USP, (1975), pp 123-127.

/5/ - Mammana, C.Z. - Arquitetura de um Processador para Aquisição de Dados Estocásticos - IV Seminário sobre Desenvolvimento Integrado de Software, UFMG, Belo Horizonte,(Julho 1977).

CAPÍTULO V

COMENTÁRIOS GERAIS E CONCLUSÕES

O estudo sobre geração de código para matrizes, *records* e procedimentos permitiu uma boa implementação desses tipos de construções no compilador LAPA. Entretanto já pudemos constatar que tanto nessas construções, como nos demais comandos, implementações mais eficientes poderiam ter sido feitas.

Além disso, muitos aspectos da arquitetura do PADE não foram aproveitados pelo compilador. Na verdade trabalhamos apenas com uma subarquitetura do PADE.

Descreveremos agora os principais pontos que poderão ser melhorados após o término da versão inicial do compilador e também os aspectos da arquitetura que poderão ser utilizados.

1. MATRIZES

Quando geramos a palavra para endereçamento indireto de uma matriz (ver IV.4.15), pode acontecer da constante m assumir um valor negativo. Nesse caso podemos ter um endereço fora da partição de memória reservada para o programa, o que é um erro. Isso acontece se o valor negativo for maior que o tamanho da área de dados anterior à área de matrizes. No caso de matrizes declaradas no programa principal, o tamanho da área de dados anterior à área de matrizes é o tamanho de SE somado ao tamanho de SDV, e no caso de matrizes declaradas em um procedimento é o tamanho de SDVP (ver IV.3.5).

Esse problema pode ser resolvido da seguinte maneira. Ao gerarmos a palavra para endereçamento indireto da matriz sabemos qual o tamanho de SE somado ao tamanho de SDV (ou SDVP) até

aquele instante da compilação. Assim, podemos verificar se o endereço gerado está fora da partição. Se estiver, podemos gerar a constante m assumindo apenas o valor do endereço inicial da matriz. Em cada acesso a um elemento da mesma, acrescentamos uma instrução que decrementa P1 do endereço encontrado para o elemento. Se o endereço estiver dentro da partição, o código gerado é o mesmo que o descrito em IV.4.15 e IV.4.17.

Na tabela de símbolos, junto com a informação sobre a matriz, podemos ter um bit para indicar qual dos dois tipor de palavra para endereçamento indireto foi gerada.

Pretendemos também introduzir uma modificação no caso de acesso a um elemento de uma matriz cujos índices são constantes. Nesse caso, como guardamos na tabela de símbolos o endereço do início da área da matriz (ver (a) de IV.3.3) podemos calcular, em tempo de compilação, o deslocamento do elemento desejado em relação àquele endereço. Assim, a instrução que faz acesso ao elemento pode ter, diretamente, o endereço do mesmo.Dessa maneira evitamos o acesso através de endereçamento indireto , pós-indexado, que é mais lento que o endereçamento direto.

2 - CONSTANTES

No compilador LAPA as constantes são sempre geradas na área de código e são acessadas através das instruções que manipulam essa área. Entretanto, para todas as constantes cujo valor está entre -1023 e +1023, poderiam ser usadas instruções imediatas, gerando a constante na própria instrução. Com isso, o tamanho do código objeto diminuiria do número de palavras geradas para constantes com valores entre os limites mencionados.

3 - COMANDOS FOR, CASE, EXPRESSÕES CASE E INDEX ENCAIXADOS

No compilador LAPA, para os comandos for e case e

para expressões index e case usamos o mesmo registrador para todas as construções encaixadas do mesmo tipo. A solução adotada usa uma pilha do compilador para cada um dos diferentes comandos, e como cada pilha tem apenas dez células, o número de encaixamentos de cada construção mencionada ficou restrito a dez. Essa restrição poderia ser eliminada se usássemos a própria pilha semântica para colocar as informações necessárias. Nesse caso, para cada comando mencionado, precisaríamos de mais uma posição na pilha semântica.

Uma melhoria adicional seria a de atribuir um registrador diferente para cada novo comando ou expressão pertencente a qualquer dos tipos acima. Com isso pode-se aproveitar a eventual disponibilidade de registradores livres para guardar o número de repetições de um for, a expressão seletora de um case e o resultado de expressões case e index . Para isso basta utilizarmos o procedimento de controle de registradores e salvamento dos mesmos em temporários (com as alterações necessárias). (ver IV.3 7.1).

4. CONSIDERAÇÕES SOBRE A ARQUITETURA DO PADE

O compilador LAPA não usou todos os recursos disponíveis na arquitetura do PADE. Algumas construções não utilizadas foram as instruções para manipulação de *íons*, e instruções com resultado na memória. Além disso, a instrução Repete (mnemônico R) e as polacas (mnemônico FNP) não foram devidamente exploradas.

4.1 - Manipulação de *íons*

As instruções para manipulação de *íons* são difícieis de serem usadas pelo compilador. Ainda não conseguimos chegar a uma solução satisfatória. Pretendemos fazer mais estudos sobre esse assunto pois essas instruções deverão ser usadas para a manipulação de variáveis tipo ion e char da LAPA.

4.2 - Instruções com resultado na memória

Em alguns casos de comandos de atribuição da LAPA, poderíamos ter usado instruções com resultado na memória.Para isso, precisaríamos incluir no compilador, um algoritmo para verificar a possibilidade de uso desse tipo de instrução. Esse algoritmo constaria do seguinte.

Algoritmo:

Dado um comando de atribuição, cujo lado direito é uma expressão aritmética, gera-se o comando em notação polonesa.

Seja $x_1\ x_2\ \ldots\ x_n$ o comando em notação polonesa onde x_i tanto pode ser um operando como um operador.

Se forem válidas:

a) $x_1 = x_2$

b) x_{n-1} é uma operação comutativa

c) $x_3\ x_4\ \ldots\ x_{n-2}$ é uma expressão bem formada (uma expressão válida)

então podemos usar instruções de resultado na memória ao gerar o código objeto correspondente ao comando.

ex.: A:= A+B*C

O código objeto gerado pelo compilador é:

```
T       R1,A
M       R1,B
T       R3,A
AI      R2,0(R3)
TM      R2,A
```

Entretanto, o código gerado poderia ser:

```
T       R1,B
M       R1,C
AM      R2,A
```

No exemplo acima economizaríamos duas instruções.

Neste caso, de fato, as condições do algoritmo são satisfeitas, como mostramos abaixo.

expressão em notação polonesa:

A	A	B	C	*	+	:=
x_1	x_2	x_3	x_4	x_5	x_6	x_7

a) $x_1=x_2$ (x_1=A, x_2=A)

b) $x_{n-1}=x_6$= + operação comutativa

c) $x_3 \ldots x_{n-2}$= B C + expressão bem formada

No caso da operação x_{n-1} não ser comutativa não poderíamos adotar essa otimização. Para que ela pudesse ser usada seria preciso mudar a convenção adotada pelo PADE para instruções com resultado na memória. Por exemplo, o efeito da seguinte instrução

```
SM      R0,A
```

corresponde à execução do comando A:=R0-A. Para que pudéssemos aproveitar a otimização descrita, precisaríamos que a instrução acima correspondesse à execução de A:=A-R0.

Se fosse feita essa mudança no PADE, a condição b) passaria a ser:

b) x_{n-1} é uma operação qualquer

4.3 - Instrução Repete

A instrução Repete poderia ter sido usada no cálculo do endereço de um elemento de uma matriz. Nesse caso a primeira tarefa executada, seria a geração de código para o cálculo de todos os índices e o seu armazenamento em posições consecutivas da memória. Em seguida, usando uma instrução Repete faríamos

a multiplicação dos índices pelos elementos apropriados do *dope vector*. Essa solução só seria vantajosa para matrizes com um número grande de dimensões.

Além disso, precisaríamos também mudar a análise sintática para reconhecer todos os índices de um elemento da matriz antes de começar a gerar o código para fazer o seu acesso.

Seria interessante fazer um estudo para verificar, no caso geral, qual a melhor solução.

Na linguagem LAPA poderiam também ser incluídas construções para explorar o uso da instrução Repete. Um exemplo dessas construções são os comandos scan e move (para manipulação de caracteres) do ALGOL do B-6700.

4.4 - Instruções polacas

Muitas expressões poderiam ter seu código objeto gerado de maneira mais eficiente através do uso das instruções polacas. Entretanto para que isso pudesse ser feito, o compilador deveria conter um algoritmo que transformasse a expressão para notação polonesa e verificasse qual o operando da instrução polaca a ser gerada. Como isso aumentaria ainda mais o tamanho do compilador, não introduzimos essa otimização do código gerado.

5 - INTERAÇÃO DOS GRUPOS DE "SOFTWARE" E "HARDWARE"

Durante a fase de implementação do compilador foi possível uma interação com o grupo de desenvolvimento da arquitetura do computador PADE. Essa interação foi bastante proveitosa pois permitiu a introdução de pequenas modificações no "hardware", de maneira a tornar possível a geração de um código objeto mais eficiente. Por exemplo, o "hardware" para subrotinas (existência do registrador Y e ordem de salvamento dos registradores) foi influenciado pelos implementadores do "software".

Um outro ponto discutido foi a ordem de crescimento da pilha. A convenção adotada para o empilhamento (por exemplo , no caso da instrução TRM) é a de crescer a pilha em sentido contrário ao do crescimento da memória.

Essa convenção foi adotada pelo fato do deslocamento da indexação ser positivo o que justifica um crescimento contrário ao da memória. Se o deslocamento para indexação tiver o mesmo sinal do crescimento da pilha o acesso às células que ficam antes do topo (ante-topo e outras) fica prejudicado. Entretanto, no caso do compilador, essa convenção tornou um pouco ineficiente o código gerado no caso específico da ativação de procedimentos. Isso porque, no compilador, o próprio registrador Y é usado para a indexação dos parâmetros (além das variáveis locais). Dessa maneira, torna-se necessário ajustar o conteúdo do ponteiro PO (registrador RO) (ver IV.4.10).

Para esse caso, uma alteração no "hardware", do sentido de crescimento da pilha, permitiria um código objeto mais eficiente.

6 - CONCLUSÕES

A linguagem LAPA foi projetada para ser a única linguagem de alto-nível do computador PADE, tendo como objetivos a programação de sistemas e a programação científica.

Como linguagem de programação de sistema, seria conveniente que ela fosse aderente à arquitetura, dando ao programador acesso ao maior número possível de recursos elementares da máquina. Entretanto, esse objetivo inicial não foi atingido,pois muitos recursos da máquina não foram explorados (ver 4). Por essa razão, foi introduzido o comando assemble, para inclusão de código de montagem. Esse comando foge totalmente às regras da programação estruturada e também ao tipo de comandos em linguagens de alto-nível; mas, é só por meio dele que conseguimos aces

so a todos os recursos da máquina.

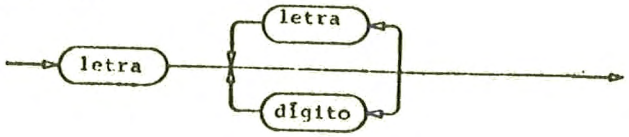
Para a programação científica, a linguagem parece ser bastante satisfatória, tendo os recursos encontrados nas mais sofisticadas linguagens do tipo ALGOL.

Os estudos descritos nos capítulos I, II e II contribuíram bastante para a obtenção de um bom conhecimento na área de compilação e contribuíram também para um bom desenvolvimento do compilador LAPA.

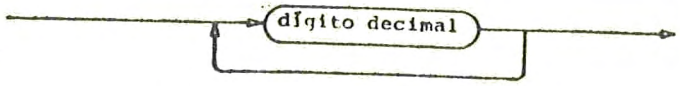
Como uma primeira versão, o compilador LAPA nos parece razoável. Futuramente, quando forem incluídas todas as modificações já relacionadas, bem como o tratamento de todas as construções não incluídas nesta versão inicial (ver IV.2) provavelmente obteremos um compilador bastante elaborado e eficiente.

Uma fase importante da compilação não foi abordada, qual seja, a da geração de programa objeto em linguagem máquina, eliminando-se a fase de montagem do "assembler". Entretanto, a inclusão do estudo correspondente ultrapassa os limites deste trabalho.

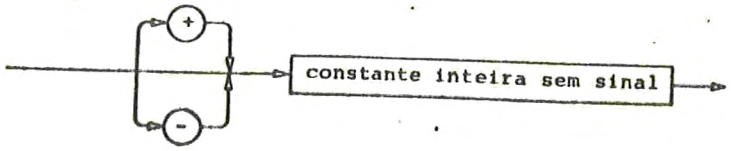
identificador



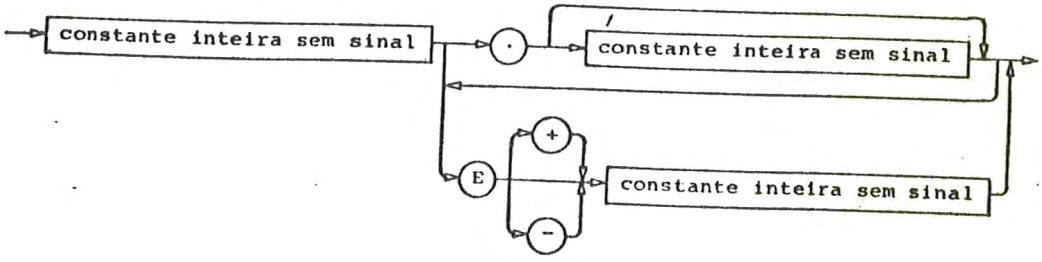
constante inteira sem sinal



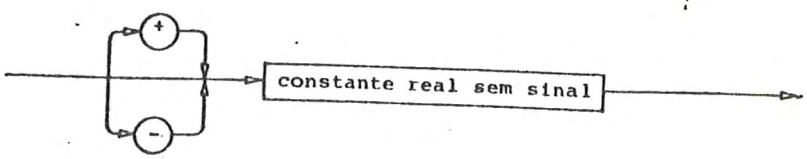
constante inteira



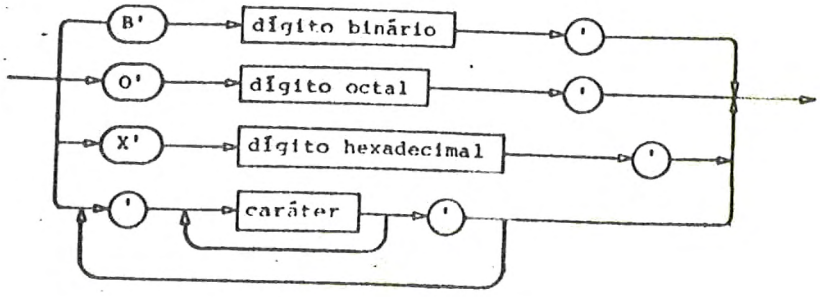
constante real sem sinal



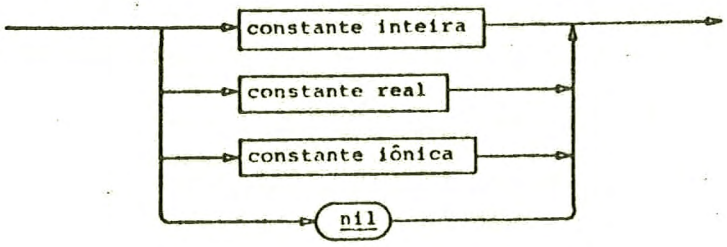
constante real



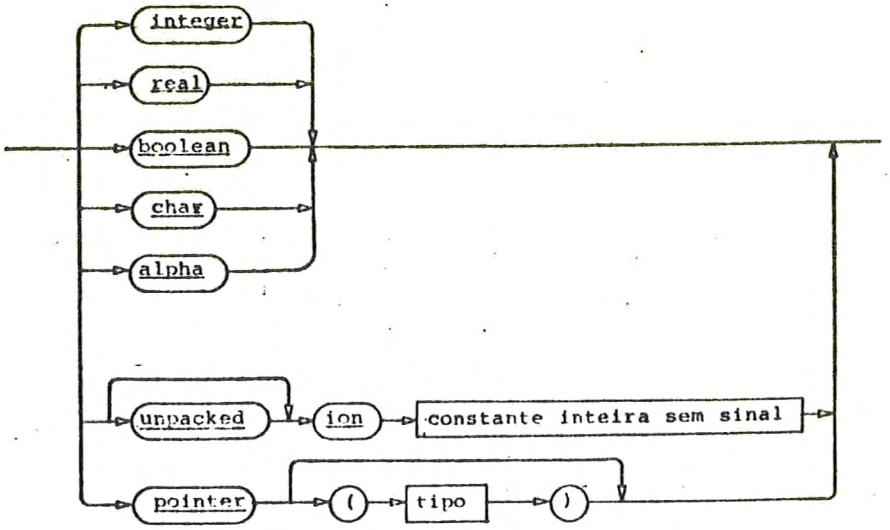
constante iônica



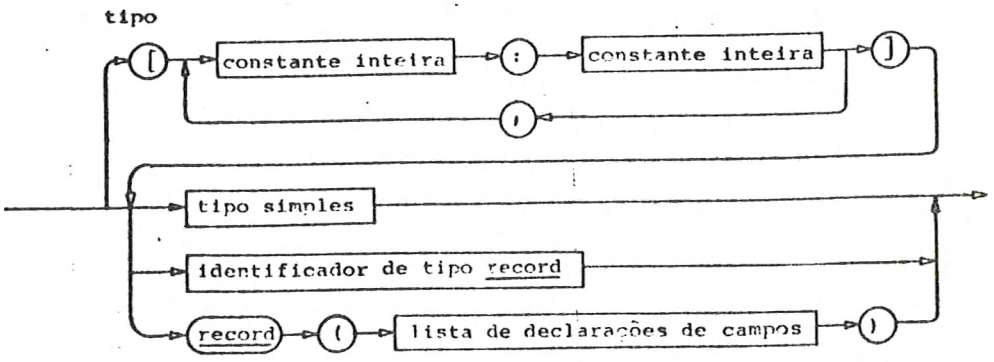
constante



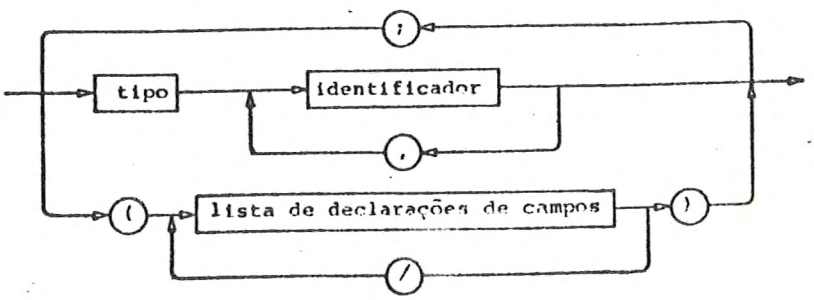
tipo simples



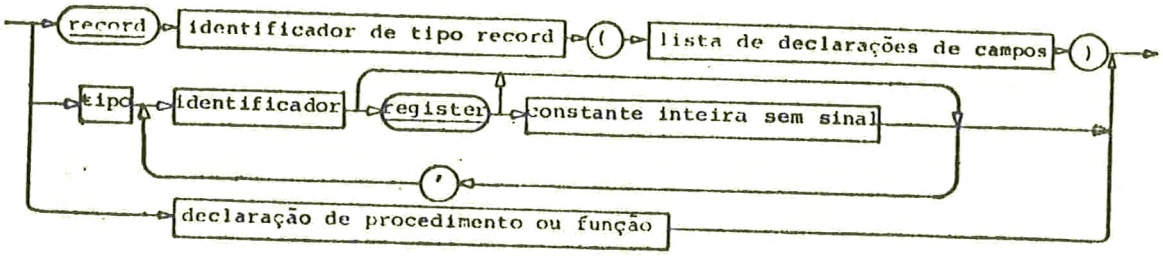
tipo



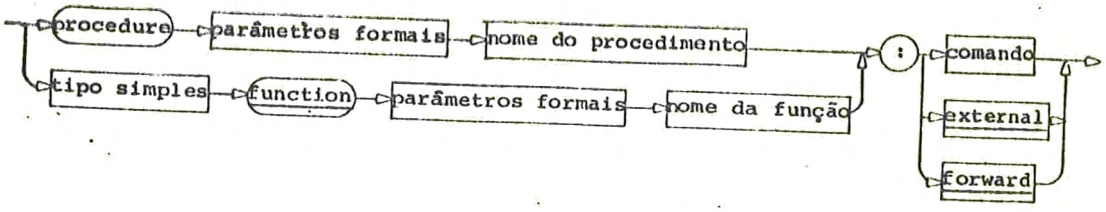
lista de declarações de campos



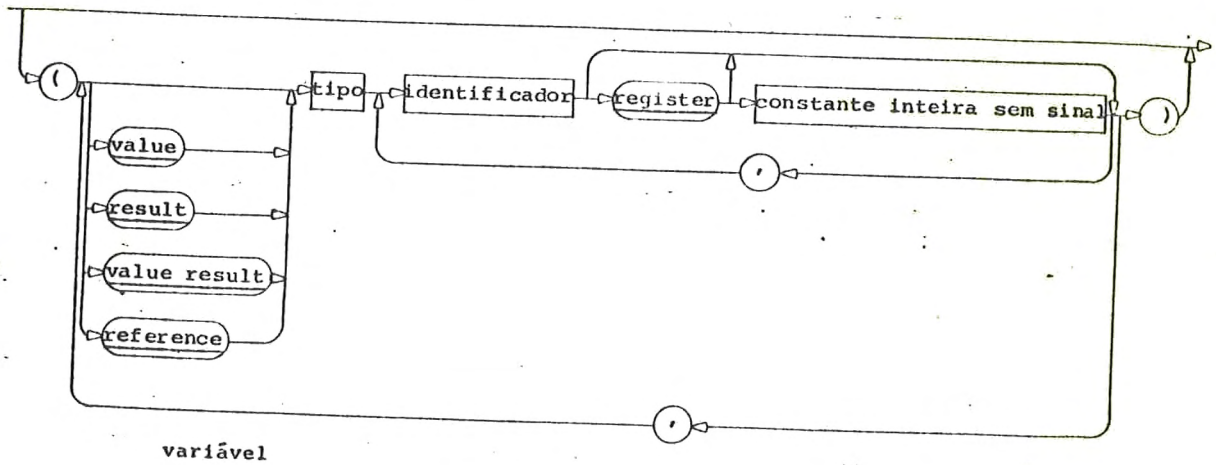
declaração



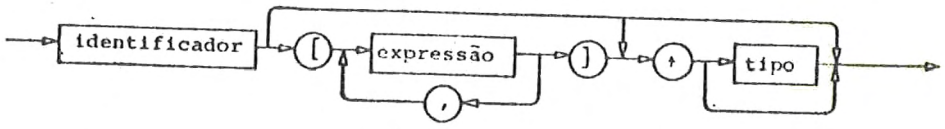
declaração de procedimento ou função



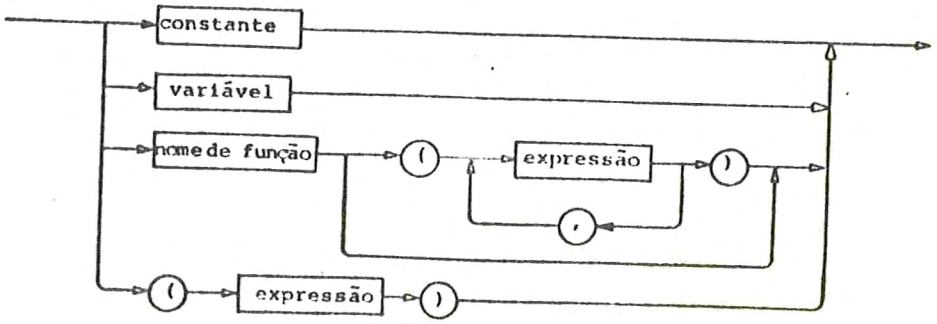
parâmetros formais



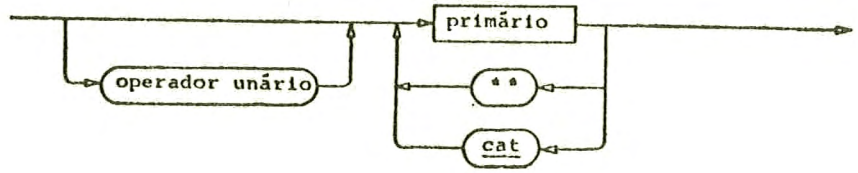
variável



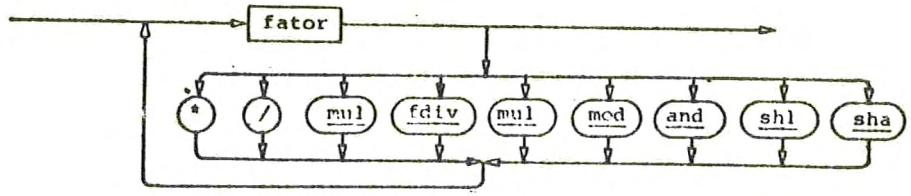
primário



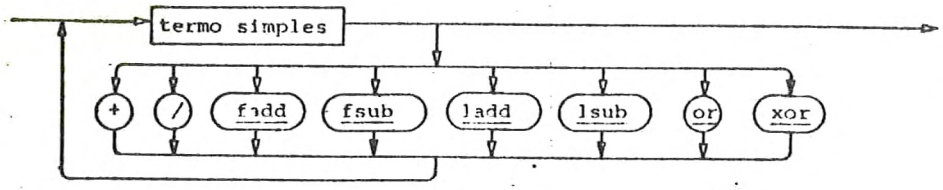
fator



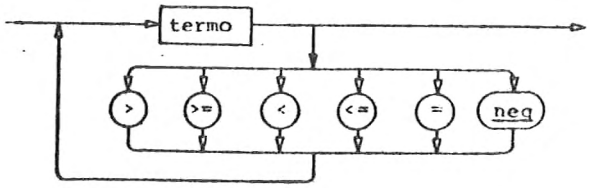
termo simples



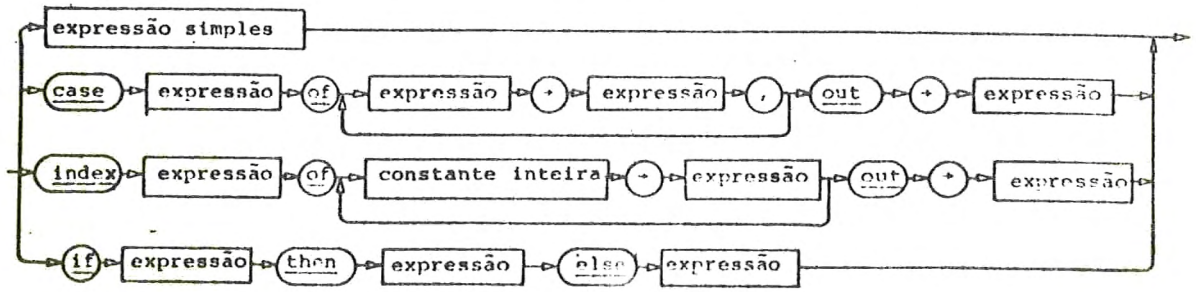
termo



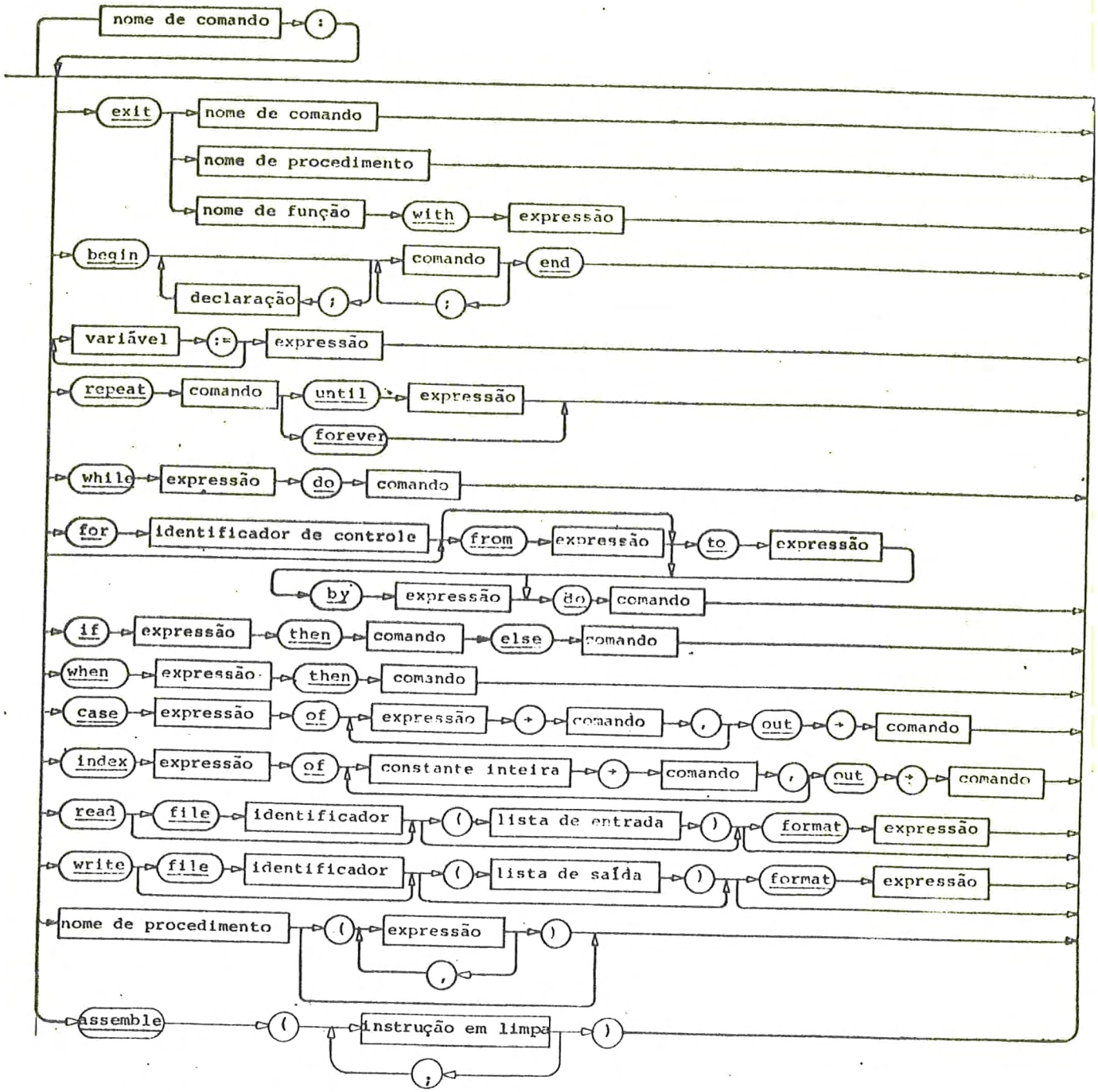
expressão simples



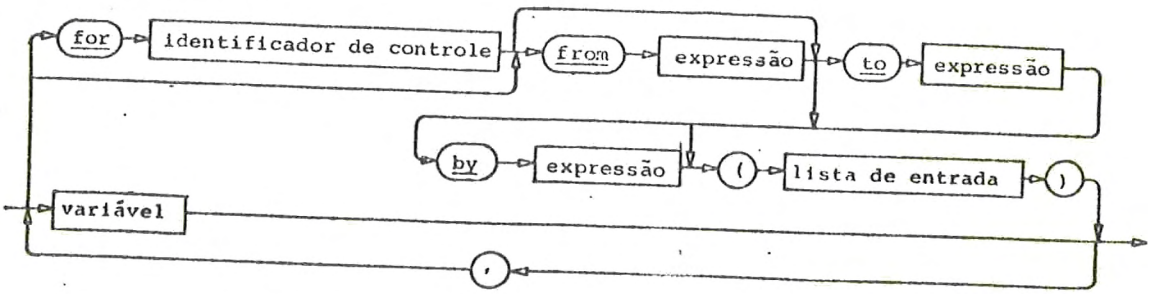
expressão



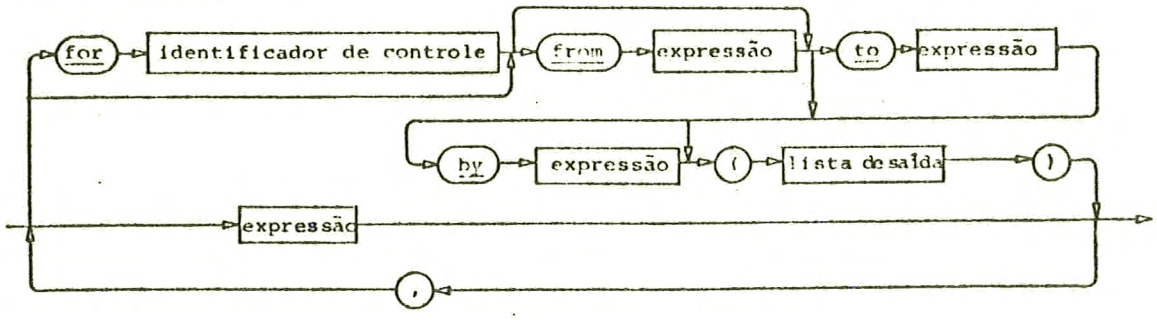
comando



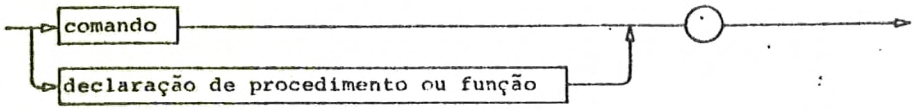
lista de entrada



lista de saída



programa



APÊNDICE 2 - DESCRIÇÃO DA LIMPA

1. INTRODUÇÃO

Neste apêndice, descrevemos os aspectos da LIMPA (Linguagem de Montagem do PADE) necessários à compreensão do código objeto indicado no texto da dissertação. Assim, não são descritas todas as funções disponíveis na linguagem.

2. ASPECTOS FORMAIS

Um algoritmo escrito em LIMPA, é composto de comandos, cada comando possuindo cinco campos:

i) rótulo : identifica o comando, possuindo significado especial em algumas pseudo-instruções.

ii) operação : especifica a ação a ser executada. É preenchido com um dos mnemônicos das instruções do PADE (ver tabela I) ou com uma das pseudo-instruções da LIMPA (ver tabela II).

iii) operando : contém informações complementares para especificação da ação.

iv) comentários : destina-se à documentação do algoritmo.

v) identificação: contém a numeração sequencial dos comandos.

3. OPERANDOS

3.1 - Elementos constituintes

Os operandos são expressões envolvendo os seguintes elementos:

i) Identificadores

Os identificadores são nomes dados a elementos de algo

ritmos, tais como registradores, posições de memória, etc. Podem ser de três tipos:

a) Registrador (ρ): designam um dos oito registradores do PADE. Identificadores reservados previamente pela LIMPA designam os registradores e suas diversas modalidades de operação:

   - acumuladores: R0, R1, R2, R3, R4, R5, R6, R7.
   - ponteiros estáticos: T0, T2, T4, T6.
   - ponteiros dinâmicos (índices decrementadores): P0,P2,P4,P6.

b) Memória (ε): designam uma posição de memória, seja da área de código ou da área de dados.

c) Número (ν): designam um número puro.

ii) Constantes

As constantes numéricas (μ), podem aparecer numa das seguintes representações:

- decimal, utilizando a notação convencional (ex.: 103, -2).
- binária, com a constante entre apóstrofos, precedida pela letra B (ex.: B'1011').

iii) Marcos de memória

São duas variáveis internas do tradutor da LIMPA,que podem ser utilizadas no campo de operandos de um comando. São representadas pelos símbolos:

. (ponto) - marco da área de código
$ (cifrão) - marco da área de dados

iv) Símbolos especiais

Os elementos anteriormente descritos, aparecem nos operandos separados pelos seguintes símbolos:

. ' ( ) @ + - * /

3.2 - Formato dos operandos

Os operandos das instruções possuem seis classes de formatos, descritas a seguir. O formato dos operandos das pseudo-instruções é descrito no item 4.2.

As sintaxes dos operandos são definidas em termos das entidades ρ, ε, μ, e ν (ver 3.1).

formato a) - instruções de referência à memória (área de dados ou código).

<op>::=ρ,<mem>
<mem>::= <end>|<end>'
<end>::= <exp>|<exp> (<índice>)
<exp>::= ε | ν
<índice>::= @ | ρ |@,ρ | ρ,@

Obs.: Para detalhes de <mem> ver a descrição da pseudo END.

formato b) - instruções imediatas

<op>::= ρ, <imed>
<imed> ::= <N> | -<N>
<N> ::= ν | ν (<índice>) (ver formato a)

formato c) - instruções para controle de sequência

instrução	formato
D	μ, ε
I, DF, DV	ρ, ε
DP	μ, <mem> (ver formato a)

formato d) - instruções privilegiadas

instrução	formato
BD BP	ρ

formato e) - instruções especiais

instrução	formato	
R	μ, <imed>	(ver formato b)
TR TRM	ρ, <N>	(ver formato b)
TEG	<mem>	(ver formato a)
FNP	expressão aritmética em notação polonesa	

4 - OPERAÇÕES

4.1 - Instruções do PADE

As instruções disponíveis no PADE (e usadas pelo compilador LAPA) são aquelas constantes da TABELA I.

TABELA I

Mnemônico	Ação	Formato do operando
A	Adição	a
AI	Adição Imediata	b
Am	Adição (Resultado na Memória - RM)	a
AP	Adição (operando na área de código-AC)	a
AR	Adição Real	a
ARM	Adição Real (RM)	a

Mnemônico	Ação	Formato do operando
ARP	Adição Real (AC)	a
BD	Carga de β (reg. base da área de dados)	d
BP	Carga de β' (reg. base da área de código)	d
C	Comparação	a
CI	Comparação Imediata	b
CP	Comparação (AC)	a
D	Desvio condicional	c
DF	Desvio, se condição falsa	c
DP	Desvio condicional (AC)	c
DS	Desvio para subrotina	a
DV	Desvio, se condição verdadeira	c
E	Lógico E	a
EP	Lógico E (AC)	a
FNP	Polaca	e
I	Itera	c
M	Multiplicação	a
MI	Multiplicação Imediata	b
MM	Multiplicação (RM)	a
MP	Multiplicação (AC)	a
MR	Multiplicação Real	a
MRM	Multiplicação Real (RM)	a
MRP	Multiplicação Real (AC)	a
O	Lógico OU	a
OP	Lógico OU (AC)	a

Mnemônico	Ação	Formato do operando
Q	Divisão	a
QI	Divisão Imediata	b
QM	Divisão (RM)	a
QP	Divisão (AC)	a
QR	Divisão Real	a
QRM	Divisão Real (RM)	a
QRP	Divisão Real (AC)	a
R	Repetição	e
S	Subtração	a
SI	Subtração Imediata	b
SM	Subtração (RM)	a
SP	Subtração (AC)	a
SR	Subtração Real	a
SRM	Subtração Real (RM)	a
SRP	Subtração Real (AC)	a
T	Transferência	a
TE	Transferência de Endereço	a
TEG	Transferência de Endereço para Y	e
TI	Transferência Imediata	b
TM	Transferência para Memória (RM)	a
TP	Transferência da Memória (AC)	a
TR	Transferência para Registradores	e
TRM	Armazenamento de registradores	e
X	OU Exclusivo	a
XP	OU Exclusivo (AC)	a
ZAD	Deslocamento Aritmético Duplo	b
ZAS	Deslocamento Aritmético Simples	b

Mnemônico	Ação	Formato do operando
ZLD	Deslocamento Lógico Duplo	b
ZLS	Deslocamento Lógico Simples	b

4.2 - Pseudo-instruções

As pseudo-instruções definem ações especiais para o tradutor da LIMPA. São descritas neste apêndice as pseudos constantes da Tabela II.

TABELA II

Mnemônico	Ação
C10	Definição de constante numérica decimal
END	Definição de constante de endereço
EQV	Equivalência
ORD	Controle da alocação de dados
MEM	Reserva de memória
FIM	Fim de programa fonte

Cada uma das pseudos é descrita a seguir. Quando o campo de rótulo aparecer entre barras, significa que ele é opcional.

i) C10 - Definição de constante numérica decimal

Esse tipo de pseudo possui o seguinte formato:

|Rótulo| C10 { v'<cte>'| '<cte>'}

onde <cte> ::= {0+1+...+9}$^{+}$

O campo ν no operando representa um fator de duplicação; quando é omitido,seu valor implícito é 1.

Exemplo:

```
J        C10      20'12'
```

No exemplo acima, o tradutor da LIMPA alocará 20 palavras de conteúdo igual a 12, em endereços consecutivos, principiando em J.

ii) END - Definição de constante de endereço

O formato do comando é:

|Rótulo| END <mem>

<mem> ::= <loc> | <loc>'
<loc> ::= <exp> | <exp> (<índice>)
<exp> ::= ε | ν
<índice> ::= @| ρ |@ ,ρ | ρ,@

OBS.: O campo <mem> tem a mesma sintaxe do seu correspondente nas instruções de referência à memória (ver formato a) de 3.2). A interpretação de seus constituintes é a mesma daqueles.

Exemplo: Se A representa um identificador associado a um endereço, então:

	Endereçamento Direto	Endereçamento Indireto
Simples	A	A'
Relativo ao γ	A (@)	A (@)'
Indexado	A (R5)	A (T2)'
Indexado Relativo ao γ	A (P2,@)	A (@, R4)'

iii) EQV - Equivalência

Essa pseudo-instrução associa ao identificador do campo de rótulo, a entidade do campo de operando.

Todo identificador que figurar no operando, deverá ter sido previamente definido.

Exemplo:

```
I       EQV     RO
J       EQV     I
```

iv) ORD - Controle da alocação dos dados

Esta pseudo é utilizada para instruir o tradutor da LIMPA a respeito da alocação de constantes (pseudos END, C10) e da reserva de memória (pseudo MEM).

Por exemplo, se desejarmos passar a alocar os dados na área de código, as pseudos de alocação devem ser precedidas por:

```
ORD     .     (marco da área de código)
```

Se desejarmos retornar à área de dados, devemos especificar:

```
ORD     $     (marco da área de dados)
```

O operando dessa pseudo deve ser uma expressão aritmética do tipo ν ou ε.

Em nenhum caso as instruções do PADE são editadas na área de dados.

Se a pseudo não for especificada em um programa, o tradutor da LIMPA utiliza a definição padrão: ORD $.

v) MEM - Reserva de memória

A pseudo MEM determina ao programa carregador que as

posições de memória por ela definidas permaneçam inalteradas na carga do programa.

O formato do comando é o seguinte:

|Rótulo| MEM v

vi) FIM - Fim de programa fonte

Esta pseudo indica ao tradutor da LIMPA que terminaram os cartões do programa fonte.