EXPAND UMA LINGUAGEM EXTENSÍVEL ATRAVÉS DE MACROS-SINTÁTICAS: COMPILADOR DA LINGUAGEM BASE

Beatriz Zakimi Miyasato

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS (M.Sc.).

Aprovada por:

Prof. José Lucas M. Rangel Netto
(Presidente)

Prof. Estevam De Simone

Prof. Jano Moreira de Souza

Prof. Sérgio Teixeira

RIO DE JANEIRO, RJ - BRASIL

MARÇO DE 1980

ZAKIMI, MIYASATO, BEATRIZ

EXPAND Uma Linguagem Extensível Através de Macros Sintáticas: Compilador da Linguagem Base

VIII, 107p. 29,7cm (COPPE-UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1980)

Tese - Univ. Fed. Rio de Janeiro. Fac. Engenharia

1. Assunto: Compiladores e Linguagens de Programação I. COPPE/UFRJ II.Título(série).

Aos meus pais,

a Gaby, Tati e Javier.

AGRADECIMENTOS

Ao Prof. José Lucas Mourão Rangel Netto pelas idéias, conhecimentos e paciente orientação ministrada.

Ao Prof. Estevam De Simone pelos conhecimentos, apoio e incentivo constantes dele recebidos.

Aos Profs. Jano Moreira de Souza e Ligia Barros Caúla pelas valiosas sugestões e pelo incentivo.

Aos Profs. Jean-Michel Nayrac e Gerhard Schwarz pelo interesse e colaboração.

Aos meus amigos Antonio Claudio, Antonio Carlos, Eliane, John e Luiz Carlos pelo apoio que deles recebi.

A COPPE e ao CNPq pelos recursos fornecidos na elaboração deste trabalho.

RESUMO

Procedimentos tem sido conhecidos e usados extensivamente como dispositivos para aumentar o poder declarativo de linguagens de programação. Um outro dispositivo que pode ser e tem sido usado de forma semelhante é a macro, seja a macro convencional, normalmente encontrada em conjunção com código de montagem, seja a macro sintática, proposta por Leavenworth e outros. Macros sintáticas admitem argumentos que não precisam ser delimitados por incomodas virgulas e parenteses, mas em vez disso são reconhecidos pelas categorias sintáticas a que devem pertencer: "comando", "primário", "identificador", etc.

O objeto desta tese e da tese de P.C. Kullock é a definição de uma linguagem extensível através de macros sintáticas e da implementação de um sistema compilador/macro-expansor no minicomputador MITRA-15 do Laboratório de Automação e Simulação de Sistemas (LASS) do Programa de Engenharia de Sistemas e Computação da COPPE-UFRJ. Em particular, esta tese descreve o compilador da linguagem básica, deixando para a tese de Kullock o projeto e a implmentação do macroexpansor.

ABSTRACT

Procedures have been known and used extensively as devices for increasing the declarative power of programming languages. Another device which can and has been used in that way is macros, either conventional macros, normally used with assembly code, or the so called syntax macros, proposed by Leavenworth and others. Syntax macros admit of arguments which do not have to be delimited by clumsy devices such as commas and parentheses, but instead are recognized by the syntactic cathegories to which they must belong, which way be "statement", "primary", "identifier", and so on.

The object of this thesis and of its companion by P.C. Kullock is the definition of a language capable of extension by syntax macros, and of the implementation of a compiler/macroexpander system on the MITRA-15 minicomputer at COPPE's Systems Laboratory. This thesis deals in particular with the compiler of the basic language; leaving the design and implementation of macroexpansion to Kullock's.

ÍNDICE

CAPÍTULO I

INTRODUÇÃO

É possível definir uma linguagem base simples (L_B), que poderia ter a sua sintaxe e semântica estendida usando o conceito de macro-sintática.

Uma macro-sintática é uma macro mais geral que apresenta as seguintes características:

- Formato de chamada livre. Não está restrito ao nome da macro seguido da lista de parâmetros separados por vírgulas;
- Os parâmetros são categorias sintáticas como expressão, comando, variável, etc (metavariáveis de L_B);
- Não existem separadores específicos entre os parâmetros (a clássica vírgula);
- Possibilita vários níveis de definição, isto é, dentro da definição de uma macro é possível fazer referência a macros já definidas.

Para deixar mais claro o conceito de macro-sintática daremos um exemplo.

MACRO

WHILE <*condição*> *DO* <*comando*>

declara a macro *WHILE* com dois parâmetros, o primeiro uma condição e o segundo um comando.

```
DEFINE
  BEGIN    LABEL  L;
        L: IF   <condição>
                THEN   BEGIN
                       <comando>;
                       GO TO L
                       END
  END
ENDMACRO
```

Define o corpo da macro usando os recursos de L_B, isto significa que cada chamada da macro *WHILE* será expandida ao bloco fornecido, com a correspondente substituição dos parâmetros. Por exemplo a chamada:

```
WHILE  A+B  LE  20  DO   BEGIN
                         M[A+B]:=C*0.45;
                         A:=A+1
                         END
```

é expandida para:

```
BEGIN   LABEL  L;
     L: IF  A+B  LE  20
            THEN  BEGIN
                  BEGIN
                  M[A+B]:=C*0.45;
                  A:=A+1
                  END ;
                  GO TO L
                  END
END
```

As macros sintáticas permitem ao usuário, em tempo de compilação, acrescentar características, não encontradas na linguagem base, escolhidas para seu uso individual. No caso de um usuário com uma certa experiência, essas macros poderão, de certa forma, utilizar definições "ótimas" para a aplicação em vista. Por exemplo poderia definir uma macro FOR1, mostrada no programa I.1, com incremento fixo igual a 1 e/ou definiria uma macro FOR, mostrada no programa I.2 com incremento variável, sendo este um dos parâmetros da macro.

```
MACRO     FOR1 <variável>:=<expressão>1  TO <expressão>2
          DO   <comando>
DEFINE    BEGIN LABEL #L;
             <variável>:=<expressão>1;
          #L : IF <variável> LE <expressão>2
                  THEN  BEGIN
                        <comando>;
                        <variável>:=<variável>+1;
                        GO TO #L
                        END
          END
ENDMACRO
```

PROGRAMA I.1

```
MACRO     FOR <variável>:=<expressão>1  STEP  <expressão>2
           TO <expressão>3   DO  <comando>
DEFINE    BEGIN  LABEL #L;
                 <variável>:=<expressão>1;
           #L :  IF <variável>  LE <expressão>3
                   THEN   BEGIN
                          <comando>;
                          <variável>:=<variável>+<expressão>2;
                          GO TO #L
                          END
          END
ENDMACRO
```

PROGRAMA I.2

Nota-se então, que este conceito de macro-sintática, nos permitiria definir uma L_B muito simples, o que implica num compilador pequeno, porém com as facilidades de programação de uma linguagem poderosa e sofisticada.

O nosso objetivo, foi construir um compilador pequeno para um computador também pequeno, porém fornecendo uma linguagem de alto nível adaptada ao usuário e com recursos de programação similares a alguns dos recursos, das linguagens que estamos acostumados a usar (Algol|[9]|por exemplo).

A escolha da máquina recaiu sobre o MITRA-15 do Laboratório de Automação de Sistemas e Simulação da COPPE-UFRJ, como parte do projeto de um Laboratório de Ensino de Computação.

A escolha foi baseada principalmente na carência no MITRA-15, de uma linguagem de alto nível com estrutura de blocos e com alocação dinâmica de memória.

O trabalho foi dividido, em duas partes a primeira o macro-expansor que faria o tratamento da definição e expansão das macros sintáticas; e a segunda responsável pelo analisador sintático e gerador de código.

A primeira parte está sendo desenvolvida em |Kullock[15]|

O presente trabalho descreve a construção do analisador sintático e gerador de código.

I.1. Recursos disponíveis

I.1.1. Equipamento

O compilador foi desenvolvido no MITRA-15|Gerhard[8]| que é um computador de tempo real, com uma capacidade de 16k bytes extensível a 32k bytes.

A memória principal do MITRA-15 é uma memória de núcleos de ferrite organizada em palavras de 16 bits, mais 1 de paridade e 1 de proteção. O tempo de ciclo é de 800 nanosegundos por palavra. A memória é endereçável por byte e alterável por byte, palavra (2 bytes) e dupla palavra (4 bytes).

Os periféricos disponíveis são:

- Uma leitora de cartões;
- Um teletipo de 72 caracteres;
- Tres linhas assíncronas;
- Um vídeo;

- Disco fixo e móvel.

São usados a leitora como unidade de entrada e o teletipo como unidade de saída.

I.1.2. Software básico

- Monitores de base, de tempo real e tempo real em disco |MITRA-15[11]|;
- Assembler;
- LP15
- FORTRAN
- BASIC

O compilador foi escrito na linguagem LP15E |MITRA-15[10]| e usa o monitor MCCOO, que é o menor monitor disponível no sistema (aproximadamente 10k bytes).

CAPÍTULO II

DESCRIÇÃO DA LINGUAGEM

II.1. Estrutura da Linguagem

EXPAND é uma linguagem de alto nível, com estrutura de blocos e alocação dinâmica de memória, que pode ser estendida usando as macros sintáticas.

A linguagem EXPAND pode ser dividida em duas partes. A primeira é a linguagem base e a segunda o mecanismo de extensão.

A linguagem base tem uma sintaxe similar à do ALGOL 60 |Naur[9]|.

Todo programa EXPAND está formado de:

- Um conjunto de declarações de macros (opcional);
- Um corpo, denominado bloco.

As declarações das macros |Kullock[15]| tem como objetivo definir as macros sintáticas que serão usadas no corpo do programa. Para cada macro sintática sua declaração fornece:

a) A estrutura da macro : que, além de dar o nome, descreve a forma de chamada e os parâmetros;

b) A definição da macro : que desenvolve a semântica correspondente à estrutura da macro, isto é, o conjunto de declarações e/ou instruções que compõem a macro sintática.

A chamada de macro (usada no corpo do programa) é uma estrutura de macro, com os parâmetros atuais. Cada chamada de macro será expandida à definição da macro, com a correspon-

dente substituição dos parâmetros formais pelos parâmetros reais. O usuário tem a possibilidade, além de definir as próprias macros, de usar a biblioteca de macros que será fornecida |Kullock[15]|. Nesta biblioteca estarão armazenadas macros que definem os comandos mais comuns não pertencentes à L_B (por exemplo DO, FOR, WHILE, etc).

O corpo do programa é um bloco; isto é, um conjunto de declarações seguido de um conjunto de comandos limitados por BEGIN e END.

- Das declarações

As declarações fornecem informações acerca dos dados que serão usados ao longo do programa (variável simples ou arranjo, tipo, além do escopo).

Os arranjos são unidimensionais (tipo vetor) com o limite inferior e superior definidos dinâmicamente.

Os tipos de variáveis são REAL, INTEGER e LABEL, este último para rótulos. Os caracteres alfanuméricos são tratados como sendo do tipo INTEGER (um caráter por palavra).

Os procedimentos podem ou não ter parâmetros (caso existam deverão ser variáveis simples, nunca arranjos), o corpo do procedimento é um bloco ou um comando composto. Dentro de um procedimento, só é permitida a declaração de variáveis simples (REAL, INTEGER) ou LABEL. Quando houver declarações de variáveis estas são locais ao procedimento, havendo verificação de escopo. Não é permitida a declaração de um procedimento no corpo de outro, porém é permitida a chamada desde que o procedimento chamado já tenha sido declarado ou no próprio bloco ou num bloco externo. A passagem de parâmetros é por referência |Gries[1]|. Na chamada os parâmetros formais são

carregados com o endereço dos parâmetros atuais .

Um procedimento só é válido no bloco no qual foi declarado e nos blocos internos a ele.

Existem procedimentos com e sem tipo, estes últimos retornam o resultado no próprio nome, em geral usados como operandos de uma expressão aritmética.

Na presente implementação, não é permitido o uso de procedimentos recursivos.

- Dos comandos

O conjunto de comandos do corpo do programa, define o algoritmo a ser executado usando os dados definidos nas declarações.

Os comandos podem ser simples ou compostos. Um comando composto é um conjunto de comandos encerrados entre BEGIN - END.

É possível ter um bloco como primário de uma expressão aritmética, usando o que denominamos de bloco-função. Um bloco-função é um conjunto, opcional, de declarações seguido de um conjunto de comandos encerrados com FBEGIN e FEND.

O comportamento é o mesmo que a chamada a um procedimento com tipo, isto é, após a execução de um bloco-função obter-se-á um resultado que será um primário da expressão aritmética que contém o bloco-função. O valor final está definido, pelo comando RESULT antes de sair do bloco-função.

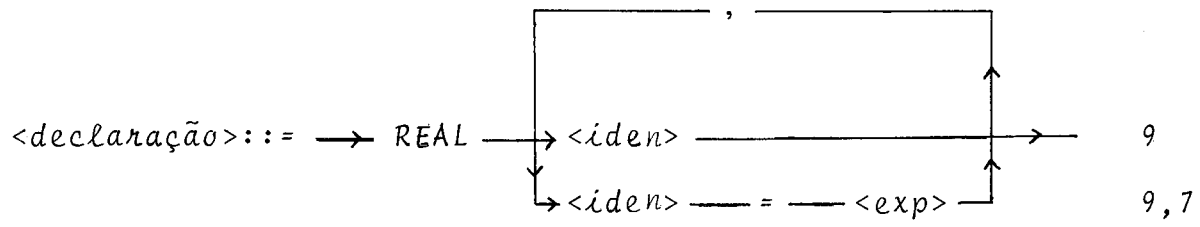
Os comandos da linguagem EXPAND são apresentados na sua versão mais simples e menos poderosa (por exemplo IF-THEN e não IF-THEN-ELSE) e não há muitos tipos de comandos. Isto foi feito com o objetivo de definir uma linguagem necessá

ria e suficiente, que seria a L_B que poderia ter seu conjunto de comandos aumentado, pelo uso das macro sintáticas.

II.2. Especificação da Linguagem

Notação usada para descrever a sintaxe:

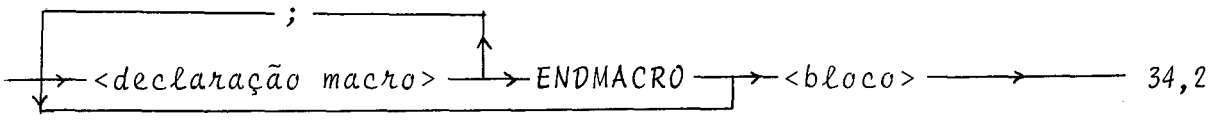
A descrição das categorias sintáticas da linguagem utiliza uma variação da forma normalizada de Backus. Sua principal característica é diminuir o número de categorias sintáticas. Exemplo:



- As categorias sintáticas são palavras delimitadas por " < " e " > ";
- As palavras são escritas com letras maiúsculas;
- O sinal ::= é lido como "é definido por";
- O sinal → é lido como "é seguido por";
- O sinal ↑ é lido como "ou seguido por";
- O(s) número(s) ao lado de cada linha indica (m) o(s) número (s) da(s) definição(ões) da(s) catagoria(s) sintática(s) referenciada(s) naquela linha.

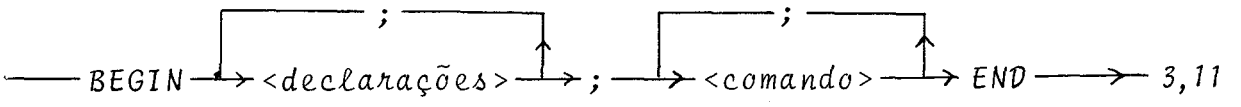
II.2.1. Programa

1 <programa>::=

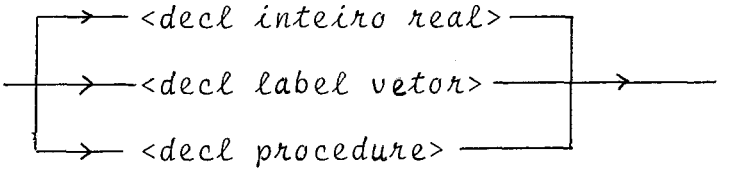


O programa está dividido em declarações de macros e um corpo chamado bloco. As declarações de macro fornecem as macros sintáticas (estrutura e definição) que serão usadas no corpo do programa.

2 <bloco>::=



3 <declarações>::=



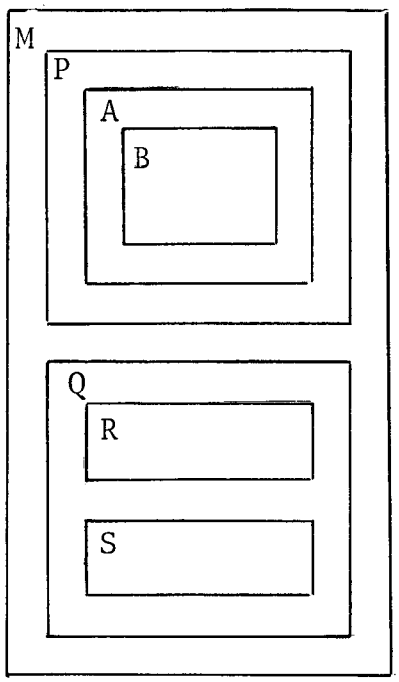
Um algoritmo ou programa para computador consiste de duas partes essenciais, uma descrição de ações, a serem executadas e uma descrição dos dados a serem manipulados pelas ações.

As ações são descritas pelos comandos e os dados são descritos pelas declarações.

Um bloco é definido como um conjunto de declarações seguido de um conjunto de comandos, encerrados entre BEGIN e END. Um ponto e vírgula (;) deve separar cada uma das declarações, cada um dos comandos e o conjunto de declarações do conjunto de comandos.

Como os blocos podem ser embutidos em outros, serão atribuídos níveis de profundidade a eles. Se o bloco mais externo tem nível zero, então um bloco definido nele terá nível 1.

Em geral um bloco definido no nível i será de nível i+1 . A figura II.1 ilustra uma estrutura de blocos.



Bloco	nível
M	∅
P,Q	1
A,R,S	2
B	3

Figura II.1

A validade de um identificador X, é todo o bloco no qual foi declarado, incluíndo os blocos definidos no mesmo bloco que X, isto é, ao BEGIN do bloco serão alocadas as posições de memória requeridas pelas declarações e ao END tais posições serão liberadas.

Identificadores definidos em	Válidos em
M	M,P,A,B,Q,R,S
P	P,A,B
A	A,B
B	B
Q	Q,R,S
R	R
S	S

É possível declarar no bloco B um identificador X já declarado no Bloco A. Tem o efeito de definir X como sendo local a B (não acessível em A) e pode ser de qualquer tipo. A última declaração de X é válida em todo o bloco B, a menos que seja redeclarado num bloco subordinado a B.

Não é permitido declarar o mesmo identificador mais de uma vez no mesmo nível.

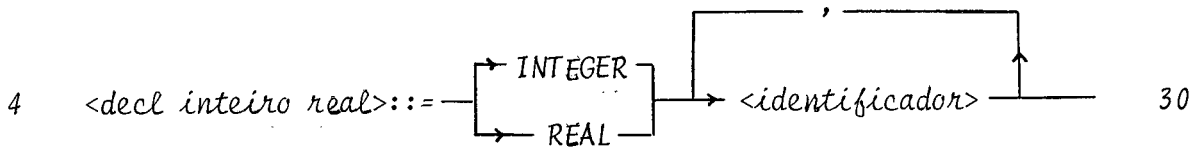
Todas as variáveis, rótulos e procedimentos devem ser declarados.

As palavras reservadas (Apêndice 1) não podem ser usadas como identificadores.

Os comandos descrevem as ações a serem executadas sobre os dados.

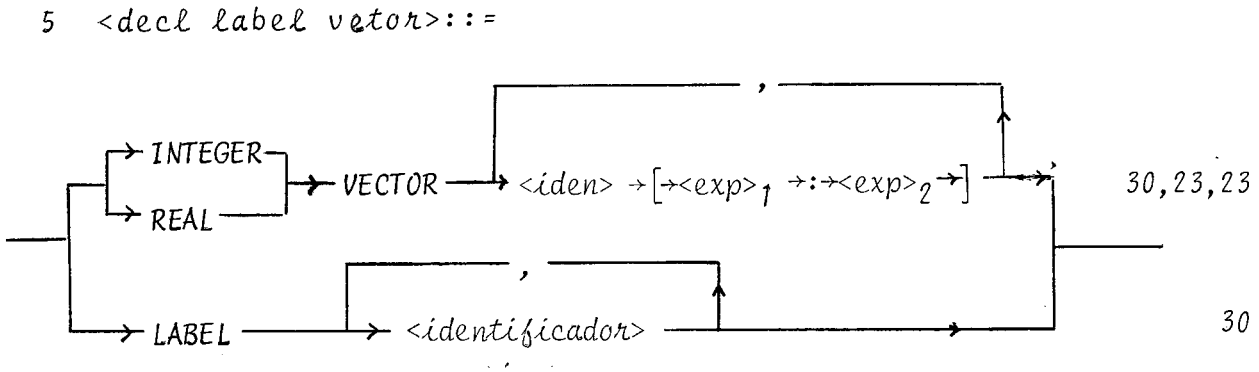
Normalmente os comandos são executados sequencialmente (na ordem em que foram escritos), podendo esta ordem ser alterada por uma ordem de desvio condicional (comando IF) ou incondicional (GO TO).

II.2.2. Declarações



A declaração INTEGER é usada para definir identificadores que representarão valores inteiros.(Cada variável inteira ocupa 2 bytes).

Para declarar identificadores que representem valores reais é usada a palavra REAL. (Cada variável real ocupará 4 bytes).



Declaração LABEL

Os comandos de um programa podem receber nomes, para que se possa referencia-los. Estes nomes são chamados de rótulos, e sintáticamente são identificadores, que devem ser declarados na declaração LABEL.

Os rótulos são usados no comando incondicional GO TO.

Declaração de Arranjos

O arranjo é uma estrutura que consiste de um número fixo de componentes, os quais são todos do mesmo tipo, chamado tipo do componente.

A linguagem EXPAND permite definir arranjos unidimensionais (vetor) com limites dinâmicos.

<identificador> é o nome do arranjo.

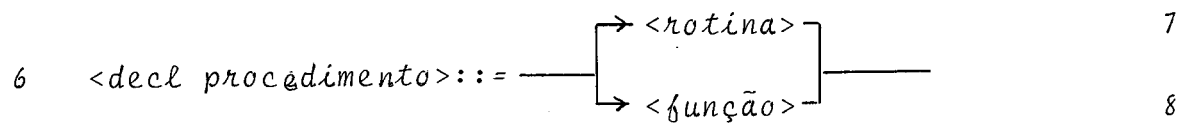
$<exp>_1$ determina o valor do limite inferior do arranjo.

$<exp>_2$ determina o valor do limite superior do arranjo.

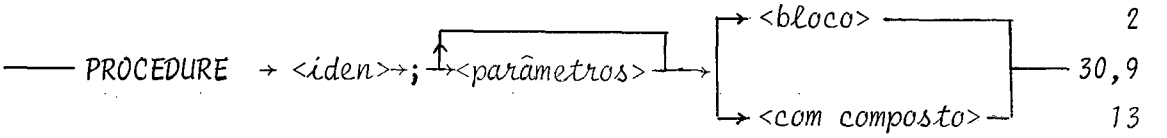
$<exp>_1$ e $<exp>_2$ devem ser inteiras e os identificadores nelas usados, devem estar declarados nos blocos mais externos ao bloco da declaração de arranjo, isto é, se um arranjo é declarado no nível i , os identificadores, usados nas expressões que determinam seus limites, devem estar declarados nos blocos i-1, i-2,...,0 , nunca no próprio bloco de nível i.

Os componentes dos arranjos podem ser do tipo REAL (4 bytes por posição) ou INTEGER (2 bytes por posição).

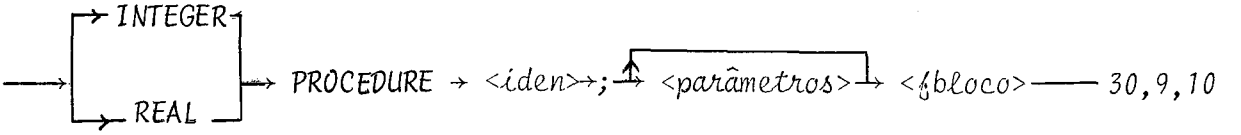
II.2.3. Declaração de Procedimentos



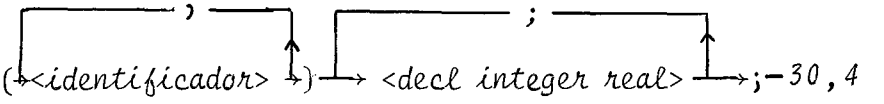
7 <rotina>::=



8 <função>::=



9 <parâmetros>::=



A declaração de procedimento serve para definir um segmento de programa e dar-lhe um nome, de modo que esse segmento de programa possa ser ativado por uma chamada.

Uma declaração de procedimento é um bloco ou comando composto com um cabeçalho.

Dentro do bloco de um procedimento não são permitidas declarações de arranjos, nem de outros procedimentos. Os identificadores declarados no procedimento serão locais a ele. É permitido usar dentro do procedimento identificadores já declarados no bloco em que o procedimento é declarado,ou em blocos mais externos.

Não é permitido o uso recursivo de procedimentos, nesta implementação.

No corpo de um procedimento é possível fazer referência a procedimentos declarados em blocos externos.

O escopo de um procedimento é determinado da mesma forma que para os identificadores (válido no próprio bloco e

nos blocos mais internos).

O objetivo dos parâmetros (que são opcionais) é tornar o procedimento genérico, permitindo que ele seja usado em diferentes situações, bastando para isso chamá-lo com os parâmetros adequados.

Caso existam os parâmetros, eles só podem ser do tipo REAL ou INTEGER simples, nunca arranjos.

Todos os <identificador>es devem aparecer uma e só uma vez em <decl integer real>, definindo os parâmetros formais.

A passagem de parâmetros é feita por referencia, isto é, no momento da chamada os parâmetros formais são carregados com os endereços dos parâmetros atuais.

Deve existir uma correspondencia biunivoca, em número e tipo, entre parâmetros formais e atuais.

Procedimento sem tipo (rotina)

São procedimentos cujo corpo é um bloco ou comando composto , tem como função realizar uma tarefa específica e os resultados são retornados via parâmetros.

São ativados com comandos de chamadas.

Procedimento com tipo (função)

São procedimentos que calculam e retornam um valor no próprio nome da função.

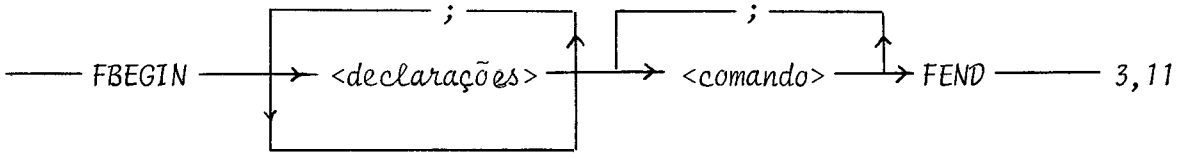
São ativados por chamadas que fazem parte de uma expressão aritmética (<fchamada>).

O valor retornado será real ou inteiro dependendo do tipo de procedimento.

O corpo deste tipo de procedimento é um <fbloco>, justamente, um bloco que retorna um valor após a sua execução, valor este definido, num comando RESULT .

II.2.4. Bloco Função

10 <fbloco>::=



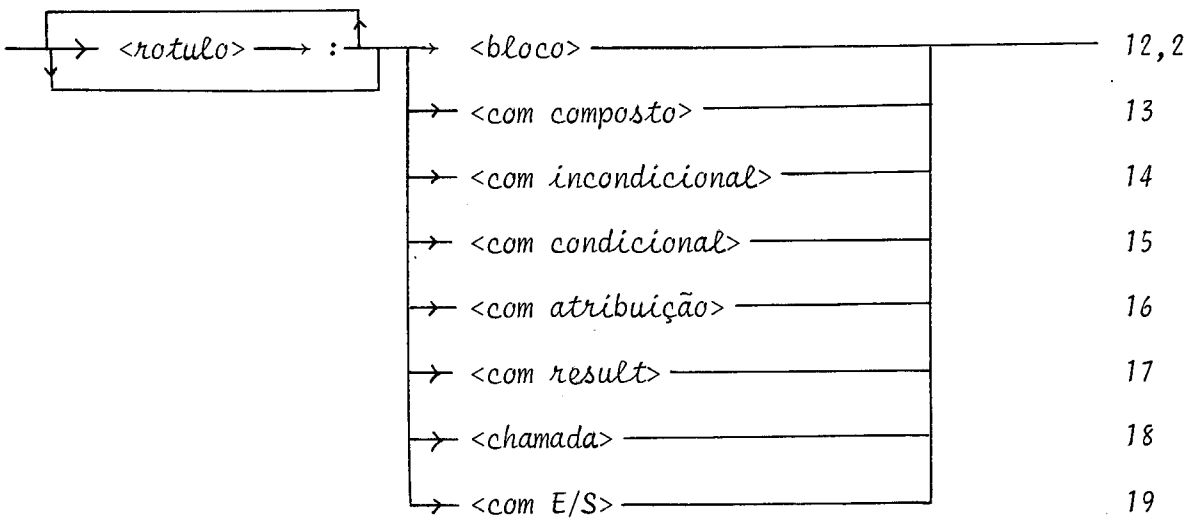
Um bloco é um conjunto de declarações (opcional) e um conjunto de comandos encerrados por FBEGIN e FEND.

É usado como o corpo de um procedimento com tipo, ou como primário de uma expressão aritmética.

A diferença principal com o <bloco> é que após a sua execução retorna um valor, definido num comando RESULT.

II.2.5. Comandos

11 <comando>::=



12 <rótulo>::= —→— <identificador> ———————— 30

Qualquer comando de um programa pode ser marcado, prefixando o comando por um rótulo seguido de dois pontos (:) (isto torna possível a referência a este comando no comando GO TO).

O rótulo deve estar definido numa declaração LABEL, no próprio bloco em que está sendo usado para nomear um comando. A figura II.2 mostra dois exemplos.

```
BEGIN
 LABEL  X,Y,Z ;
 _____
 _____
 _____
 X: _____
    _____
 Y:Z: ____
END
```

correto

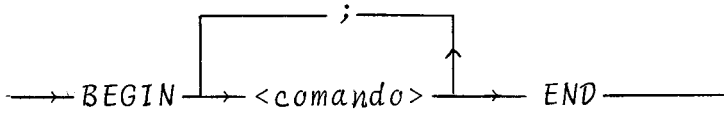
```
BEGIN
 LABEL X;
   BEGIN
   _____
   _____
   _____
   X: _____
      _____
      _____
   END
END
```

errado (label X declarado num bloco mais externo ao seu uso)

Figura II.2

Como mostrado no exemplo, um comando pode ter mais de um rótulo.

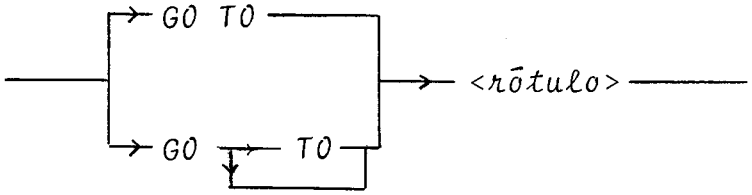
13 <com composto>::=



11

Em certos casos quando um grupo de comandos foi criado para atingir certo objetivo, eles devem constituir um único comando denominado comando composto. A composição é feita agrupando-se uma série de comandos entre BEGIN e END, de modo a serem tratados como um só comando.

14 <com incondicional>::=



12

O comando GO TO gera um desvio incondicional para a instrução rotulada com <rótulo>.

<rótulo> deve estar definido numa declaração LABEL, antes de sua referência no programa.

Só são válidos desvios dentro do mesmo bloco ou a blocos mais externos (Fig.II.3).

Não são permitidos desvios para blocos mais internos (Fig.II.4).

Exemplo 1:

```
BEGIN
LABEL A,B,C ;
  GO C;
A:___
  ===
  ===
  GO B;
  C :   BEGIN INTEGER X;
        ___
        ===
        ===
        GO B;                          VÁLIDO
        ___
        ===
        ===
        GO A;
        ___
        ===
        ===
        END;
  ___
  ===
  ===
B:___
  GO A
END
```

Figura II.3

Exemplo 2:

```
BEGIN
 ___
 ===
 ===
  GO A;
    BEGIN
    LABEL A;
     ___
     ===
     ===
    A: ___
    END
___
===
===
END
```

No bloco mais externo A não está definido, e sua referência nele será inválida.

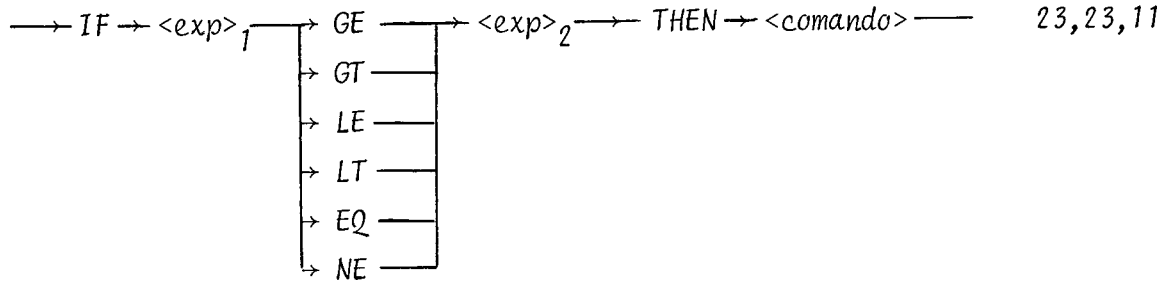
Figura II.4

Exemplo 3:

```
BEGIN LABEL A;
   BEGIN
      ===
      ===
   GO A;
     BEGIN LABEL A;                    O GO TO é executado desviando
        ===                            o controle ao LABEL A do bloco
        ===                            mais externo
     A: ===
        ===
     END
   END
 A: ===
    ===
END
```

Figura II.5

15 <com condicional>::=



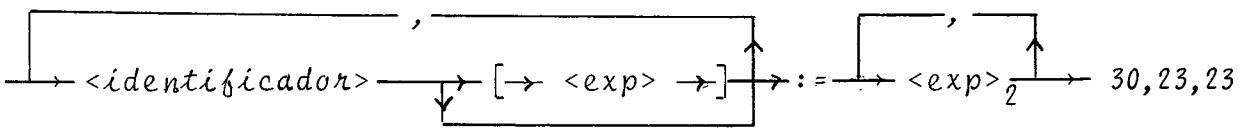
O comando IF especifica que o <comando> será executado só se $<exp>_1$ e $<exp>_2$ satisfizerem a relação especificada, caso contrário, será executado o comando seguinte ao IF.

Relações válidas	<exp>$_1$ tem que ser ... que <exp>$_2$
GE	maior igual
GT	maior
LE	menor igual
LT	menor
EQ	igual
NE	não igual

Caso as expressões sejam de tipos diferentes, será feita uma conversão e o IF será executado entre duas expressões do tipo real.

Cabe notar que a relação = (igual) entre valores reais não faz muito sentido devido aos problemas de aproximação. Isto é, sabe-se que 1.0 não é igual a 0.45 + 0.55.

16 *<com atribuição>:::=*



O comando de atribuição especifica que um ou vários valores de expressões aritméticas recentemente calculados serão atribuidos a uma ou várias variáveis.

:= é o operador de atribuição.

O número de expressões ao lado direito do operador de atribuição deve ser igual ao número de variáveis à esquerda do mesmo.

Se o tipo da expressão difere do tipo da variável, será feita uma conversão da expressão ao tipo da variável.

Caso tenha mais de uma variável/expressão, a atribuição será feita da esquerda para direita (primeira variável recebe primeira expressão, segunda variável recebe segunda expressão etc) e deve entender-se que as atribuições são feitas em paralelo. (Fig.II.6)

Exemplo 1:

I:=Ø	a variável I recebe o valor Ø

Exemplo 2

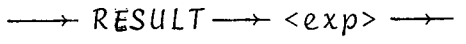
A,B:=B,A	Especifica uma troca dos valores de A e B Equivalente a: T1:=A; A:=B;B:=T1;

Exemplo 3

I:=2	
I,A[I]:= Ø,20	Ao fim da atribuição: I é zero A[0] não muda o valor A[2] é 20

Figura II.6

17 <com result>:::=



23

Define uma expressão aritmética (<exp>) como resultado de um <fbloco>.

O comando RESULT deve aparecer sempre dentro de um <fbloco> e vários RESULT podem aparecer no mesmo <fbloco>.

Caso o <fbloco> seja o corpo de um procedimento com tipo, o comando RESULT define o valor resultado que será armazenado no nome do procedimento . Se o tipo da expressão é diferente ao tipo do procedimento, será feita uma conversão.(Fig.II.7)

Caso o <fbloco> não seja o corpo de um procedimento, o comando RESULT define o valor resultante do <fbloco> cujo tipo será dado pela expressão do primeiro comando RESULT que aparece no <fbloco>. Se o tipo das expressões dos vários RESULT s (do mesmo fbloco) são diferentes será feita uma conversão ao tipo da expressão do primeiro RESULT do bloco (Fig.II.8).

```
INTEGER PROCEDURE TAG (A , B); INTEGER A,B ;
FBEGIN
 LABEL FIM;
  IF ( A GT B)  THEN   BEGIN
                       RESULT  0.0;
                       GO TO  FIM
                       END;
  RESULT 1;            % conversão implícita
 FIM:
FEND
```

O valor retornado pelo procedimento será:

= 0 se A > B Resultado inteiro (tipo do procedimento) embora a expressão seja real.

= 1 se A ≤ B

Figura II.7

```
A:=1 +    FBEGIN
          LABEL FIM, SEGUE;
          REAL SOMA; INTEGER X;
          IF (I EQ Ø) THEN   BEGIN
                             RESULT 1;
                             GO FIM
                             END;
           X:=1;
          SEGUE:
           SOMA:=SOMA + B[X];
           X:=X+1;
           IF  (X LE I) THEN GO SEGUE;
           RESULT  SOMA * 0.5      % conversão implícita
          FEND
```

O valor de <fbloco> será inteiro e

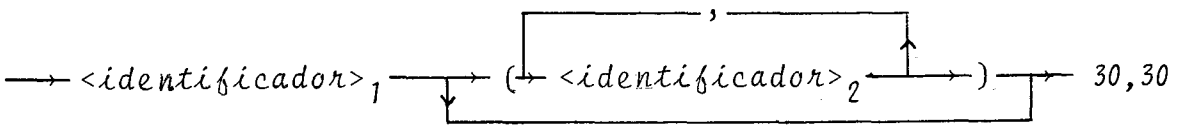
= 1 se I = Ø

= ⌊SOMA*0.5⌋ se I ≠ Ø

Neste caso o tipo de <fbloco> é dado pela expressão "1" do primeiro RESULT. Se o "RESULT SOMA*0.5" fosse o primeiro do fbloco, o resultado seria do tipo real.

Figura II.8

18 <chamada>::=



Especifica a ativação do procedimento <identificador>$_1$ que deve ser necessariamente sem tipo. A chamada só é válida dentro do escopo do procedimento a ser usado.

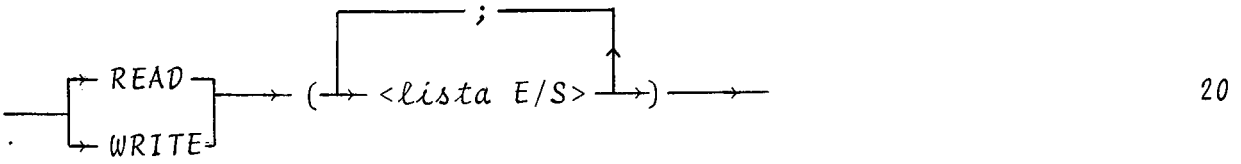
<identificador>$_2$ representa as variáveis (parâme-

tros atuais)que serão transmitidos ao procedimento através dos correspondentes parâmetros formais. A correspondência é obtida tomando-se os parâmetros das duas listas na mesma ordem.

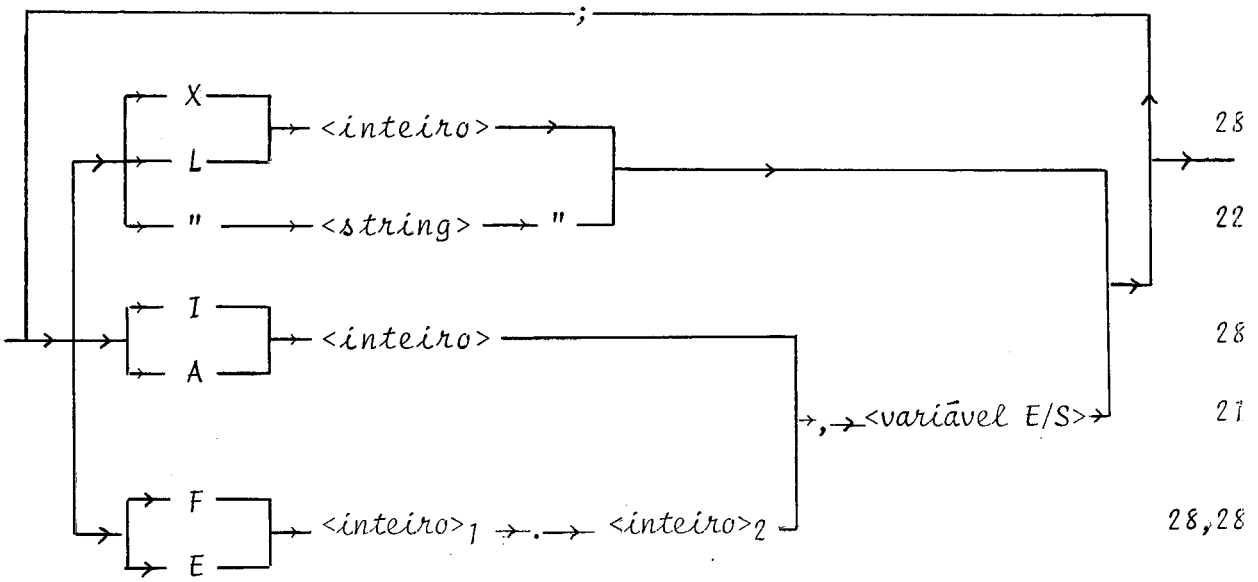
Os parâmetros atuais da chamada do procedimento, devem concordar em número e tipo com os parâmetros formais da declaração do procedimento.

II.2.6. Comandos de entrada e saída

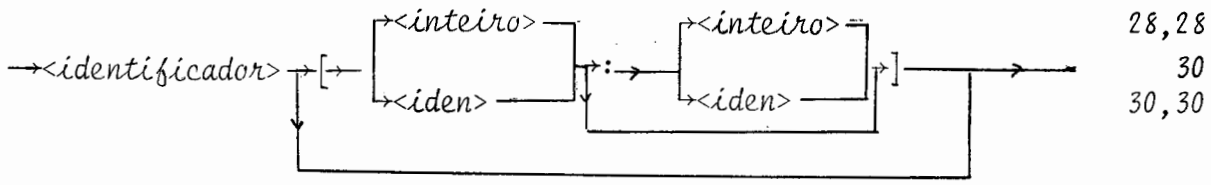
19 <com E/S>::=



20 <lista E/S>::=



21 <variável E/S>:::=



Os dispositivos de entrada/saída permitidos nesta implementação são:

- Uma leitora de cartões;
- Um teletipo de 72 caracteres/linha.

Os comandos READ e WRITE fornecem informações acerca da disposição dos dados de entrada (nos cartões) ou da maneira como os resultados devem ser impressos.

Toda variável num comando READ/WRITE deve ter associado um formato, podendo ser uma variável simples, uma determinada posição de um arranjo ou um subconjunto de um arranjo. Neste último caso todas as posições do arranjo serão lidas/impressas com o mesmo formato.

Exemplos:

A variável simples

B[5] posição 5 do arranjo B

B[I:9] posições de I a 9 do arranjo B.

Os índices devem ser constantes ou identificadores inteiros. No caso de um identificador do índice ser real, será feita uma conversão. Se forem usados os dois índices o primeiro deve ser menor ou igual ao segundo. Sempre é feito um teste de validade de índice.

Formato Iw

Usado para leitura/impressão de números inteiros.

w conta os dígitos do número e o sinal, se tiver.

Entrada: se existe sinal deve vir imediatamente antes do número, sem brancos. O número a ler pode aparecer alinhado pela direita. O branco não é considerado zero.

Saída: Se o número for positivo será impresso sem sinal. Se o número a ser impresso tem um número de dígitos menor que w , os espaços serão deixados à esquerda. Caso contrário, se w for subdimensionado, serão impressos, no lugar do número, w asteríscos.

As variáveis a serem lidas ou impressas devem ser tipo INTEGER .

Formato Fw.d

Usado para leitura/impressão de números reais.

w especifica o comprimento total do campo incluíndo do sinal, ponto fracionário, dígitos inteiros e dígitos fracionários.

d especifica o número de dígitos fracionários.

Uma das duas partes, parte inteira ou fracionária pode ser omitida.

Leitura

O ponto nunca deve ser perfurado e a parte fracionária fica subentendida como sendo os d dígitos mais à direita no campo.

O sinal (se existe) deve ir imediatamente antes do número,sem brancos. O número pode ser perfurado em qualquer posição dentro do campo especificado.

Saída

O número é arredondado a possuir d dígitos fracionários, e é impresso na forma sii...i.fff..., o sinal só aparece para números negativos. Se o número não ocupa todo o campo será impresso à direita. Se w for subdimensionado, serão impressos,no lugar do número, w asteriscos.

As variáveis a serem lidas/impressas devem ser do tipo REAL.

Formato Ew.d

Usado para leitura/impressão de números reais com expoente.

w especifica o comprimento total do campo incluindo do sinal do número e do expoente, ponto fracionário, letra E, dígitos do expoente, parte inteira e parte fracionária.

d especifica o número de dígitos fracionários.

Uma das duas partes, inteira ou fracionária pode ser omitida. A presença do expoente é obrigatória.

Leitura

O ponto não deve ser perfurado e a parte fracionária ocupa d colunas à esquerda da letra E. O expoente deve ter no máximo dois algarítmos e um sinal opcional, tanto para o número quanto para o expoente.

O sinal do número deve vir imediatamente antes de le, sem brancos. O número pode ser perfurado em qualquer posição do campo especificado.

Saída

O valor da variável é arredondado para se obter d algaritmos na parte fracionária. O valor arredondado é, então, escrito no campo da seguinte forma:

$$s_1 i.fff...f\ E\ s_2\ ee$$

s_1 sinal do número

i parte inteira

f parte fracionária

s_2 sinal do expoente

ee expoente

se o número não ocupa todo o campo será posicionado à direita. Se w for subdimensionado, serão impressos asteriscos no lugar do número.

As variáveis a serem lidas/impressas devem ser do tipo REAL.

Formato Aw

Usado para leitura/impressão de caracteres EBCDIC.

Leitura

w deve ser obrigatóriamente 1. Será lido um carácter EBCDIC.

Saída

Se w for maior que 1, serão impressos w-1 brancos antes do carácter contido na variável a ser impressa.

As variáveis a serem lidas/impressas devem ser do tipo INTEGER. A informação alfanumérica é armazenada a um carácter por palavra.

Formato Xw

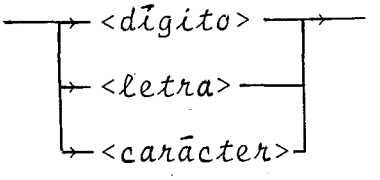
Usado para ignorar w colunas no cartão, ou deixar w espaços brancos na impressão.

Formato Lw

Especifica w cartões (linhas) a serem pulados(as) na leitura (impressão).

Formato "<string>"

22 <string>:::=



31
32
33

só pode ser usado em WRITE.

Causa a impressão da <string> ; é usado para imprimir títulos.

O número máximo de caracteres de um string é 64, os restantes serão ignorados.

Um carácter aspas (") pertencendo à string deve aparecer duplicado.

Exemplo:

WRITE (" " "FORMATO CADEIA" " ") será impresso

"FORMATO CADEIA"

Não serão considerados campos divididos, isto é, se um campo ultrapassa o limite do registro (cartão ou linha) será totalmente incluído no registro seguinte, sendo que a mudança de registro será automática.

Exemplo:

READ(I5,A[0:16]);

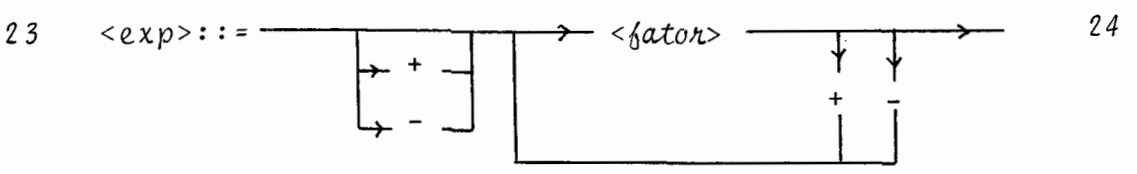
O valor A[16] será lido no segundo cartão.

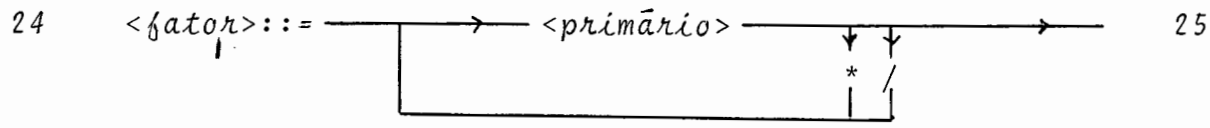
WRITE(X70;"TITULO");

TITULO será impresso na segunda linha.

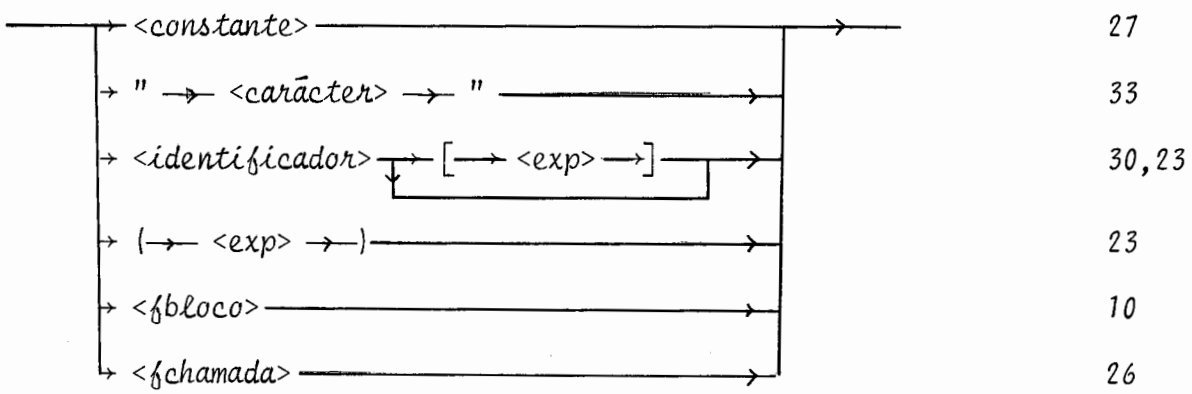
Cada WRITE/READ indica o começo de um novo registro (linha/cartão).

II.2.7. Expressões Aritméticas





25 <primário>::=



Uma expressão consiste de operandos (constantes, variáveis, blocos função, chamadas a procedimento com tipo) e operadores combinados cumprindo as regras descritas nos diagramas. Na avaliação das expressões são observadas as regras de avaliação da esquerda à direita e precedência de operadores.

Prioridade dos operadores:

1) - unário

2) *,/

3) +,-

Numa expressão aritmética primeiro são calculados os blocos função, chamada de procedimento e expressões entre parênteses.

Na expressão A/B*C, primeiro é executada a divisão e logo a multiplicação.

Não é permitido operações implícitas, isto é:

A(B+C), deve ser escrito A*(B+C).

Se os dois operandos de uma determinada operação não são do mesmo tipo, será feita uma conversão e a operação será executada entre números reais.

O primário "<caracter>" é considerado como uma constante inteira.

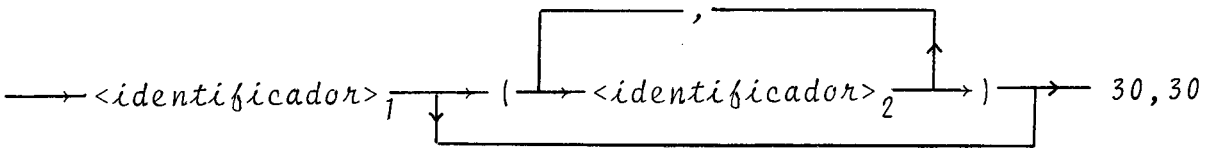
Exemplo:

```
A:= "Z";
WRITE (A1,A);
```

imprime o carácter Z.

<identificador>[*<exp>*] é usado para referenciar arranjos. *<identificador>* deve estar declarado como arranjo. *<exp>* deve ser um valor inteiro, caso contrário será feita uma conversão, sempre é feito um teste de validade do índice.

26 *<chamada>*



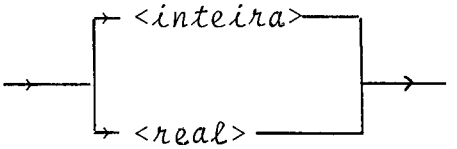
Especifica a ativação do procedimento <identificador>$_1$, que deve ser necessáriamente com tipo. A chamada só é válida dentro do escopo do procedimento a ser usado.

<identificador>$_2$ representam as variáveis (parâmetros atuais) que serão transmitidas ao procedimento através dos correspondentes parâmetros formais. A correspondência é obtida tomando-se os parâmetros das duas listas, na mesma ordem.

Os parâmetros atuais da chamada do procedimento devem concordar em número e tipo com os parâmetros formais da declaração.

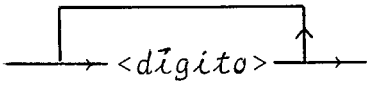
II.2.8. Constantes e Representação Interna

27 <constante>::=



28 29

28 <inteira>::=

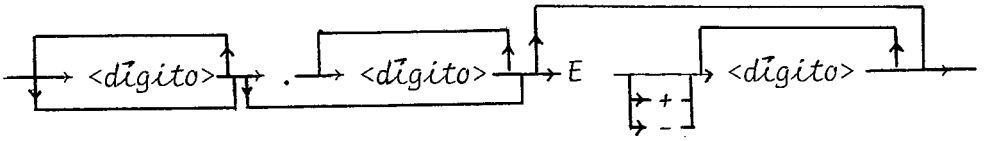


31

A constante inteira é armazenada em 2 bytes. O bit Ø é zero se o número for positivo, e 1 se for negativo. O negativo de um número é representado pelo complemento a 2 do número positivo.

Limites: -32768 a +32767

29 <real>::=

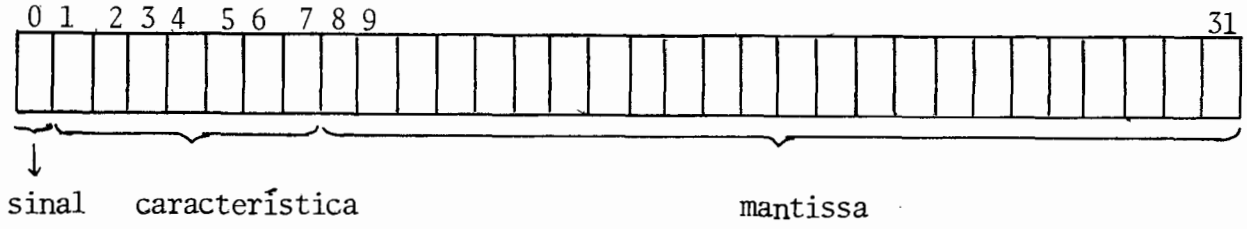


31,31,31

A constante real é armazenada em 4 bytes, com a seguinte estrutura:

- sinal (posição zero)

- Expoente na base 16 com o valor aumentado de 64, chamado de característica.
- Uma mantisa de 6 dígitos hexadecimais



Um número negativo é representado pelo complemento a dois de seu valor absoluto.

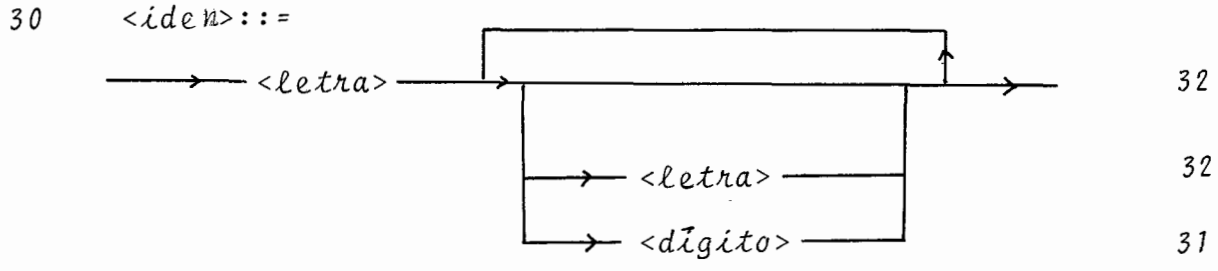
O ponto fracionário não pode aparecer só com o expoente, deve ser precedido de uma parte inteira ou seguida de uma parte fracionária ou todas as duas. O expoente não pode aparecer só, deve ser precedido de uma parte inteira ou seguido de uma parte fracionária, ou todas as duas.

Limites:

$$0{,}69090 \times 10^{-76} < N < 0{,}72310 \times 10^{76}$$

É fornecida uma precisão de 7 dígitos fracionários.

II.2.9. Identificadores

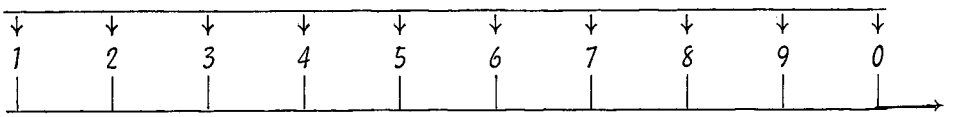


Os identificadores são comumente usados para representar valores numéricos e são denominados variáveis.

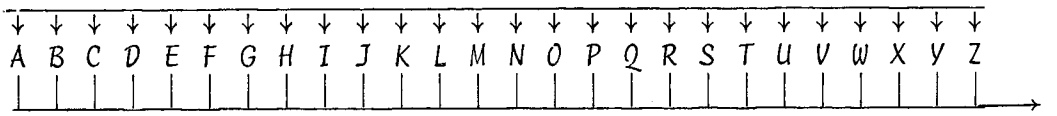
Um identificador deve sempre ser declarado antes de seu uso e o valor inicial não está determinado. É responsabilidade do programador atribuir-lhe um valor inicial.

Os identificadores poderão ser formados com um máximo de 64 caracteres, os restantes são ignorados.

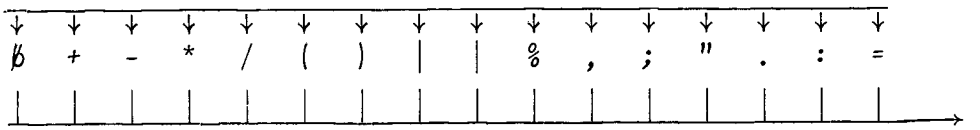
31 <dígito>::=



32 <letra>::=

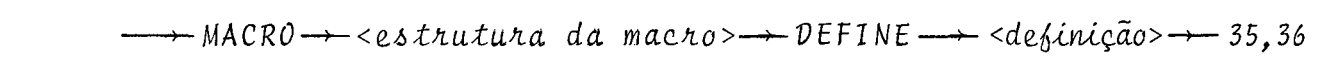


33 <caracter>::=



II.3. Mecanismo de Extensão

34 <decl de macro> : : =



E4

<comando> : : = <estrutura da macro> 35

E25

<primário> : : = <estrutura da macro> 35

O mecanismo de extensão permite que novas formas de comandos e funções sejam incluidas na linguagem usando a declaração de macro sintática (diagrama 34).

A <estrutura da macro> define a forma da nova construção sintática e a <definição>, a tradução que será associada à nova construção sintática.

35 <estrutura da macro>

é uma cadeia de metavariáveis e símbolos terminais. O primeiro símbolo da cadeia deve ser um terminal único chamado delimitador ou nome da macro. As metavariáveis são os parâmetros da macro e devem ser cadeias pertencentes às categorias sintáticas da linguagem. Nesta implementação as categorias sintáticas a que podem pertencer os parâmetros são: <variável> , <expressão> , <condição> e <comando>. Detalhes da implementação podem ser encontrados em |Kullock[15]|.

Nomenclatura para as metavariáveis:

$VARn	para variável
$EXPn	para expressão
$CONDn	para condição
$STATn	para comando

com n = branco ou 1,2,3... ; usado no caso de ter mais de uma metavariável do mesmo tipo.

36 *<definição>*

É também uma cadeia de metavariáveis e símbolos terminais. Cada metavariável na definição deve aparecer na estrutura. Os nomes dos identificadores locais às macros, isto é, nelas declarados, devem começar sempre com o carácter "#".

A macro sintática pode ser interpretada como uma função cujo domínio é a estrutura da macro e a imagem é a definição da macro.

Exemplos 1:

```
MACRO
 FOR $VAR:=$EXP1      TO $EXP2 DO $STAT
DEFINE
  BEGIN LABEL  #L1;
    $VAR:=$EXP1;
   #L1:IF $VAR LE $EXP2
          THEN   BEGIN
                 $STAT;
                 $VAR:=$VAR+1;
                 GO #L1;
                 END
  END
ENDMACRO
```

Declara a macro FOR com os seguintes parâmetros: Uma variável, duas expressões e um comando. A definição da macro é um bloco com um rótulo próprio que tem como carácter inicial "#" para notar que é rótulo interno à macro.

O delimitador da macro é FOR. Cada vez que um delimitador de macro aparece numa declaração de macro, este é considerado palavra reservada e só pode ser usado na chamada a essa macro.

Note que a estrutura e a definição da macro, têm o mesmo tipo sintático, isto é, comando.

Uma chamada de macro é uma estrutura de macro cujas metavariáveis (parâmetros formais) são substituídas por literais (parâmetros atuais).

Quando a chamada de macro é examinada, cada literal deve estar de acordo com o tipo sintático da metavariável correspondente. Depois que as metavariáveis da definição foram substituidas pelos literais, a cadeia resultante deve ser uma cadeia sintáticamente correta na linguagem base.

Segundo os diagramas E4 e E25 podemos notar que uma chamada de macro é ou um comando (similar a <chamada> (17) de procedimento) ou um primário (similar a <fchamada> (26) de procedimento).

Por exemplo, uma chamada para a macro FOR será:

```
FOR K:=1 TO N+1 DO J:=J+M[K];
```

A correspondência entre metavariáveis e literais é:

Metavariável	literal
$VAR	K
$EXP1	1
$EXP2	N+1
$STAT	J:=J+M[K]

A chamada da macro será expandida resultando na seguinte cadeia:

```
BEGIN     LABEL #L1;
 K:=1;
#L1   : IF K LE N+1
        THEN   BEGIN
               J:=J+M[K];
               K:=K+1;
               GO #L1
               END
END
```

Pode-se notar que esta macro gerará um bloco, logo a chamada da macro FOR deve pertencer à categoria sintática <comando> , para que o programa expandido seja sintáticamente correto.

Exemplo 2:

```
MACRO
 SOMA $EXP1 COM $VAR:=$EXP2  A $EXP3
DEFINE
 FBEGIN
   INTEGER  #TEMP;
    #TEMP:=0;
    FOR $VAR:=$EXP2 TO $EXP3 DO #TEMP:=#TEMP + $EXP1;
    RESULT #TEMP
 FEND
ENDMACRO
```

Declara a macro SOMA que tem como parâmetros tres expressões e uma variável.

Note que na definição da macro é usada uma chamada à macro FOR anteriormente declarada. Podemos dizer então, que na definição de uma macro podem aparecer chamadas a macros já declaradas.

Esta macro gerará um bloco função, logo a sua chamada deve pertencer à categoria sintática <primário> para que o programa expandido seja, sintáticamente correto.

Exemplo de chamada:

C:= T*SOMA B[K] COM K:=1 A 10.

Esta expressão aritmética contém uma chamada à macro SOMA.

Metavariáveis	Literais
$EXP1	B[K]
$VAR	K
$EXP2	1
$EXP3	10

A expansão da macro SOMA resultará em:

```
C:= T* FBEGIN
       INTEGER #TEMP;
       #TEMP:=0;
       FOR K:=1 TO 10 DO #TEMP:=#TEMP + B[K];
       RESULT $TEMP
      FEND
```

e a expansão da macro FOR resultará em:

```
C:=T*   FBEGIN
          INTEGER #TEMP;
           #TEMP:=Ø;
            BEGIN  LABEL #L1;
             K:=1;
            #L1:    IF K LE Ø
                      THEN   BEGIN
                             #TEMP:=#TEMP + B[K];
                             K:=K+1;
                             GO #L1
                             END
             END;
           RESULT #TEMP
         FEND
```

II.4. Formato dos Programas

Os programas em EXPAND são fornecidos ao computador usando como, meio de entrada o cartão perfurado.

Utilizam-se as colunas 1 a 72, inclusive, na perfuração dos programas. As colunas 73 a 80 podem ser usadas para identificação e enumeração dos cartões ou simplesmente deixadas em branco.

Os programas podem ser perfurados continuamente,ou seja, um mesmo cartão pode conter diversos comandos ou declarações. Como alternativa podem, também, os cartões não precisarem ser preenchidos em todo o campo útil, em qualquer caso, um cartão é continuação do campo útil do precedente.

Os comentários são introduzidos no programa, com o uso do símbolo percentagem (%) que indica final do campo útil do cartão.

O fim do programa fonte é marcado pelo cartão contendo "%EOD" nas primeiras colunas assim como o fim dos dados. Ver Fig.II.9

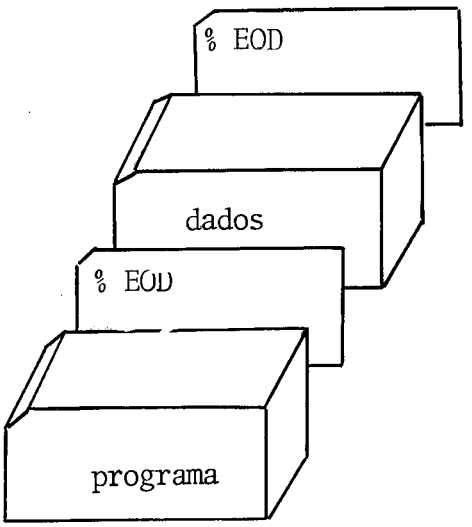


Figura II.9

II.5. Comandos de Controle

% C/EXPAND/

Na chamada do compilador estão previstas duas opções: listagem do programa fonte e listagem do programa fonte expandido.

Se nenhuma opção é usada, o programa fonte não será listado. Detalhes em |Kullock[15]|.

Ao fim da execução do compilador, se o programa gerado não tiver erros, segue-se os seguintes comandos de controle:

% C/LINKD/ (nome do programa), SL, UL

Chama o link-editor

%L

Carrega o programa na memória

%R

Para começar a execução do programa.

CAPÍTULO III

O COMPILADOR

O compilador EXPAND está dividido em duas partes:

- Macro-expansor;
- Análise sintática e gerador de código.

O macro expansor |Kullock[15]| examina todo o programa fonte e suas funções principais são:

a) Analisar as declarações das macros sintáticas;

b) Expansão das macros chamadas pelo programa, substituindo os parâmetros formais pelos parâmetros atual;

c) Análise léxica gerando, na zona de trabalho do disco, um programa codificado em tokens |Gries[1]|. No que se segue este programa será chamado PT.

O programa PT gerado pelo macro expansor é um subconjunto da linguagem base, que é obtido da aplicação das produções da gramática definida em II.2.

A análise sintática e gerador de código tem como entrada o programa gerado pelo macro expansor e como saída um programa BT (Binary Translatable) |Mitra[6]|, que por sua vez será a entrada do link-editor do MITRA-15.

O analisador sintático vai tomando, sequencialmente, os tokens do programa através de um módulo (chamado Miniscanner), cuja única função é pegar o próximo token disponível de PT.

Funções do analisador sintático:

- Verificar a conformidade sintática e semântica, do PT, às regras da linguagem definidas em II.2;
- Emitir as mensagens de erro, quando necessário ;
- Tratamento da tabela de símbolos com inserção, consulta e retirada dos identificadores;
- Chamar as rotinas de geração de código, caso não existam erros.

O gerador de código gera o programa BT, a ser tratado pelo link-editor para sua posterior execução.

Um programa BT, além de conter o código objeto, contém informações dirigidas ao link-editor como por exemplo definição da zona de dados, tanto CDS quanto LDS, definição de referências para frente, chamada a módulos externos etc |Mitra[6]|.

Cada vez que uma construção gramatical é reconhecida como válida, o analisador sintático chama ao gerador de código para a montagem do programa BT.

As fases de análise sintática e gerador de código (segunda parte do compilador) são realizadas em paralelo, de modo intercalado, pois esta segunda parte é de um só passo.

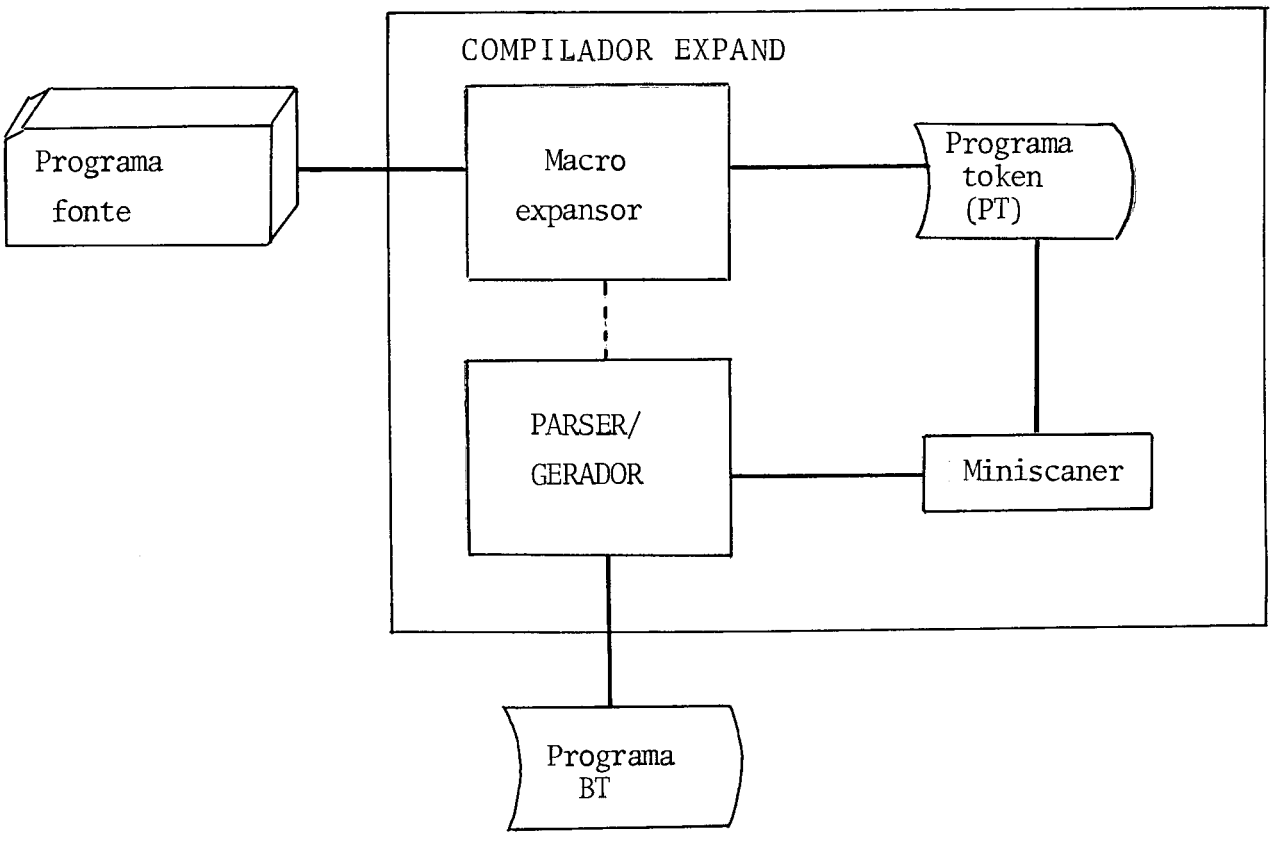


Figura III.1 - Esquema do Compilador EXPAND

III.1. Análise Sintática

O analisador sintático usa o método de matriz de transição |Gries[1]| para fazer a análise sintática do programa. É um método ascendente (bottom-up) com análise feita determinando-se repetidamente o pivô (u) da forma sentencial que está sendo analisada e reduzindo-o a um não terminal U, usando a regra U::=u.

O método exige que a gramática original esteja na forma de gramática de operadores |Gries[1]| e que pertença a classe de gramáticas tratáveis por Matriz de transição |Caúla[3]|.

III.1.1. Cálculo da Matriz de Transição

O cálculo da matriz de transição envolve a transformação inicial da gramática, de modo a reduzir o comprimento dos lados direitos das produções a limites inferiores ou iguais a 3 símbolos, através da adição de novos não-terminais denominados "não-terminais estrelados" (U*). A gramática transformada, chamada gramática aumentada de operadores é equivalente à gramática original.

Usamos para o cálculo da matriz de transição o gerador de analisadores sintáticos "NHÃONHÃO" |Simone[4]|, que nos fornece a gramática aumentada e a matriz de transição compactada.

III.1.2. Algoritmo de Análise Sintática

O algoritmo de análise sintática utiliza uma pilha sintática para os não-terminais estrelados, uma variável para não terminal simples e a matriz de transição. A cada momento tem-se o estrelado (U*) no topo da pilha, a variável que representa a última redução efetuada (U) e o símbolo obtido através do Miniscanner (T). Fazendo-se uma entrada na matriz pela linha correspondente a U* e a coluna equivalente a T, obtem-se a redução a ser usada. Para cada redução tem-se uma rotina semântica associada e o teste para o terminal U, isto é o que chamaremos de rotina para a tripla (U*,U,T).

A decomposição do analisador em diversas rotinas, cada uma encarregada do tratamento de uma situação (U*,U,T) específica, facilita a determinação de erros, gerando mensagens

de erro adequados e procedimentos de recuperação de erros convenientes.

O compilador recebe do gerador NHÃONHÃO |Simone[4]| a matriz de transição com as informações sobre a relação entre cada tripla (U*,U,T) e a redução a ser efetuada. Além disso o gerador fornece uma indicação do tipo de redução a ser usada. Menor (<), igual (=) ou maior (>) |Caúla[3]|

A compilação é iniciada com o delimitador de programa (%) na pilha-sintática. Este delimitador é usado na produção extra da gramática <S> → % <programa>% para permitir informar ao compilador a ocorrência do final físico do programa. A compilação se desenvolve token a token e quando encontrado o token "%" a compilação é encerrada, estando o código BT pronto para seu processamento pelo link-editor, caso o programa esteja correto.

III.1.3. Estrutura da Matriz de Transição

A gramática da linguagem, apresentada em II.2, teve que ser adaptada para satisfazer os requerimentos de gramáticas de precedência de operadores tratáveis por matriz de transição |Caúla[3]|, para ser aceita como entrada do programa NHÃONHÃO |Simone[4]|. A gramática modificada é mostrada no Apêndice 2.

Como saída do programa NHÃONHÃO obtemos a lista dos terminais estrelados que foram criados (apêndice 3) além da matriz de transição compactada.

A matriz de transição é apresentada em forma de

tuplas (U*,U,T).

Para cada tupla a matriz fornece a redução e a relação (< , > , =) a ser usada, isto é cada tupla está, na realidade, constituída de 5 elementos (U*,U,T,RED,REL).

Considerando que das 806 tuplas obtidas, 378 possuiam U=ε (meio=vazio) a matriz foi dividida em duas submatrizes, uma com U≠ ε denominada MATRIZ e a outra com U=ε de nome MATEPS.

As matrizes são armazenadas em vetores considerando:

a) O código relativo a U* não é armazenado;

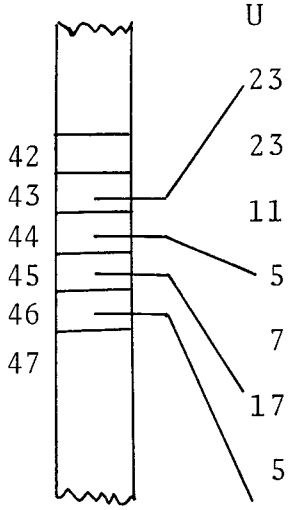
b) Para tuplas (U*,U,T) com U≠ε armazenamos U,T, RED e REL;

c) Para tuplas (U*,U,T) com U=ε são armazenados T,RED,REL.

Como as informações são armazenadas ordenadas pelo código de U*, foi criada uma tabela de apontadores para dispensar o armazenamento dos U*. (Fig III.2)

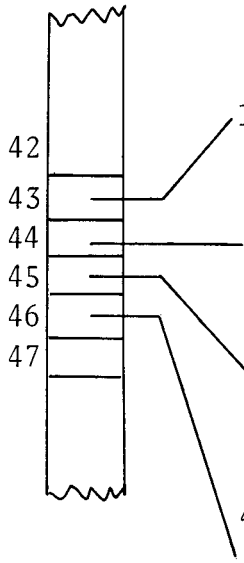
A seguir mostramos um segmento da matriz de transição que é usada (figura III.2.)

U*	U	T	REL	RED
43	23	4	2	4
43	23	12	1	56
43	11	13	1	60
43	0	16	1	63
44	0	2	1	35
44	5	4	2	24
44	7	4	2	3
44	0	5	1	39
44	0	6	1	40
45	17	4	2	3
45	5	4	2	25
45	0	5	1	39
45	0	6	1	40
45	0	46	1	110



U	T	REL	RED
23	4	2	4
23	12	1	56
11	13	1	60
5	4	2	24
7	4	2	3
17	4	2	5
5	4	2	25

MATRIZ



	T	REL	RED
42	16	1	63
43	2	1	35
44	5	1	39
45	6	1	46
46	5	1	39
47	6	1	40
	46	1	110

MATEPS

Figura III.2

Em ambas matrizes, as tuplas de relação 1 e 2 (< e >) são separadas das que tem relação 3 (=), isto permite armazenar a informação da relação (1 bit) no mesmo byte da redução (7 bits). Com esta consideração a MATRIZ gasta 3 bytes por entrada (1 byte para U, 1 byte para T e 1 byte para REL,RED), e a MATEPS gasta 2 bytes por entrada (1 para T e outro para REL e RED).

A matriz de transição é declarada no compilador como um vetor com valor pré-definido e seu tamanho é de 2040 bytes.

III.2. Geração de Código

A geração de código é a última parte da compilação,dando como saída um programa BT |Mitra 15[6]|.

O gerador de código está intimamente ligado ao analisador sintático. Para cada tripla (U*,U,T) além da re-

dução e relação a ser usada, o módulo TRIPLA fornece a rotina semântica associada a essa redução e cada rotina semântica toma as ações semânticas necessárias à geração de código, isto é, a entrada ao gerador é uma forma implícita |Simone[5]|.

Para auxiliar a geração de código foi criada uma pilha-semântica associada a pilha-sintática. Nesta pilha-semântica é armazenada informação semântica relativa ao estrelado na pilha-sintática.

As informações semânticas, são em geral, apontadores à tabela de símbolos ou à memória. Por exemplo se entra na pilha sintática um operando de uma expressão aritmética, a informação semântica será um ponteiro à tabela de símbolos, à entrada correspondente ao identificador que está sendo analisado. No caso de um comando IF , a informação semântica armazenada é o endereço de geração do "branch" (valor do "programa counter"), pois será gerado um branch com endereço de desvio igual a zero. Após a análise do comando que segue ao THEN, o endereço do desvio será conhecido e neste momento podemos gerar tal endereço na instrução branch, cujo endereço de geração está armazenada na pilha semântica.

Devido à extensão do gerador de código (45 rotinas) não é feita uma exposição das rotinas semânticas. O compilador, já implementado, está disponível no MITRA-15 do Laboratório de Automação e Simulação de Sistemas da COPPE-UFRJ.

O módulo BT gerado, é a entrada para o link-editor do MITRA-15. O módulo é relocável e permite a chamada aos módulos disponíveis na biblioteca do sistema.

O código BT, além de conter um programa em lingua

gem máquina, possui informações dirigidas ao link-editor, como por exemplo chamadas a módulos externos, definição e uso de referências para frente, definição de zona de dados etc, |Mitra 15[6]|.

O programa gerado será link-editado para sua posterior execução.

O método básico, usado no gerador é achar os operandos através da pilha-semântica e aplicar a operação indicada pela pilha-sintática.

O código gerado possui as seguintes características:

- Evita carregar desnecessáriamente.
  Usa valores da memória sempre que possível;
- Procura evitar salvar valores desnecessáriamente;
- Procura evitar a troca de registros;
- Usa as instruções adequadas para cada comando.
  A correspondência entre os comandos da linguagem fonte e a linguagem máquina foi definida na confecção do projeto do gerador de código.

A alocação das variáveis simples é feita em tempo de compilação (seção V.4.1) e a alocação de arranjos é feita em tempo de execução. Ao aparecer uma declaração de vetor é gerada uma chamada ao módulo ALOCA cuja execução resultará na alocação de memória necessária ao vetor.

III.3. Esquema do Compilador

Mostraremos a seguir um esquema simplificado do compilador, usando linguagem ALGOL para representá-lo.

```
BEGIN
   MINISCANNER(T);
   WHILE  NAOACABOU DO
      BEGIN
       TRIPLA(U*,U,T,REL,URED,ROTINA);
       GERA(ROTINA);
       CASE   REL   OF
          BEGIN
           1: % relação menor
              PUSH;
              U:=ESP;
              MINISCANNER(T);
           2: % relação maior
              U:=URED;
              POP;
           3: % relação igual
              POP;
              PUSH;
              U:=EPS;
              MINISCANNER(T);
          END
      END
END
```

MINISCANNER fornece (em T) o próximo símbolo e informa quando termina a cadeia de entrada.

A rotina TRIPLA fornece a relação (REL) entre U*, U e T , e a redução URED; além do número da rotina semântica (ROTINA) a executar.

Para a relação MENOR, URED:=UT|T; para MAIOR, URED:=U*U|U* e para IGUAL, URED:=U*UT|U*T.

GERA é o módulo que contém as rotinas semânticas que geram o código BT.

A estrutura de GERA se reduz a:

```
CASE ROTINA OF
   BEGIN
      1:
      2:
      3:
      !
      !
   END
```

III.4. Recuperação de erros

A Recuperação de erros é o processo que permite continuar a analisar o programa fonte após ter sido encontrado um erro.

Os erros podem ser classificados em erros léxicos, sintáticos ou semânticos, dependendo da fase da análise em que ocorreram.

Em nosso caso só interessam os erros sintáticos e semânticos. Os erros léxicos são tratados no macro-expansor descrito em |Kullock[15]|.

III.4.1. Erros sintáticos

Em geral a maior parte de deteção e recuperação de erros num compilador, está localizada na análise sintática. A razão é o alto grau de precisão que podemos alcançar

na especificação sintática das linguagens de programação. Dada uma gramática podemos gerar um analisador sintático que reconhece exatamente a linguagem especificada pela gramática. Qualquer violação da especificação sintática pode ser automáticamente detectada pelo analisador sintático. Apesar de se ter investido muito esforço em estudar técnicas de recuperação para erros sintáticos, a estrategia ótima para qualquer linguagem de programação é ainda uma questão em aberto.

O parser detecta um erro quando acha uma configuração, no programa fonte, não permitida pela gramática. Para recuperar o erro, o parser deve idealmente,localizar a posição do erro, corrigir o erro, verificar a configuração atual e continuar o parser. Todos os métodos existentes se aproximam deste ideal e permitem a continuação da análise sintática mas não garantem que o erro tenha sido corrigido com sucesso.

O método usado neste compilador é o denominado "panic mode" |Gries[1], Caúla[3]| que embora sendo imperfeito, é um método sistemático e efetivo de recuperação de erros em qualquer tipo de gramática |Aho[2]|.

O método básicamente, descarta elementos da entrada até encontrar um token considerado relevante, por exemplo; (ponto e vírgula) ou END. Neste momento o analisador sintático apaga as entradas da pilha sintática, até achar uma entrada tal que possa continuar a análise sintática, com o token relevante da entrada.

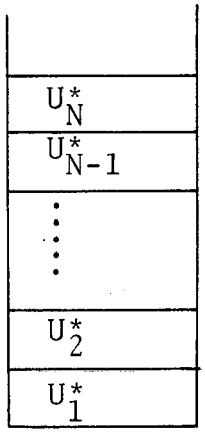
As qualidades do método são: ser simples e fácil de implementar e não usa esquema de inserção |Aho[2]| o que

garante que o método nunca entrará em laço infinito.

O método "panic mode" usado,tem uma pequena variação, com relação à definição. Tenta-se achar símbolos relevantes tanto na entrada como na pilha sintática.

Na entrada são considerados símbolos relevantes END e FEND , que são aceitos após um fim de comando. Na pilha, os estrelados BEGIN* e FBEGIN* (cujos códigos são 2 e 46), são considerados os elementos relevantes, que são começos de blocos e comandos compostos.

Supondo que o analisador sintático ache um erro, na seguinte situação:



pilha sintática

U T_N T_{N-1} T_{N-2} .... T_2T_1

MEIO CADEIA DE ENTRADA

Figura III.3

Isto é, a tripla (U_N^*,U,T_N) é inválida.

A ação a ser tomada é fazer $U=\varepsilon$ e eliminar alternativamente elementos da pilha e da entrada, até achar uma tripla (U_i^*,U,T_j) válida. A eliminação dos elementos é feita tendo-se em consideração o seguinte:

a) Caso apareça um estrelado BEGIN* ou FBEGIN* na pilha sintática, eliminar só os elementos da entrada.

b) Caso apareça um END ou FEND na entrada eliminar só os elementos da pilha.

Tendo em consideração os pontos antes mencionados sempre chegaremos a uma construção válida, que poderia ser BEGIN* na pilha com um começo de comando ou declaração na entrada, ou um fim de comando na pilha e um END na entrada. Na pior das hipóteses teríamos BEGIN* na pilha e END na entrada que é considerado como bloco vazio e permite continuar a análise sintática.

Na Fig.III.3, tentaríamos as seguintes triplas:

U^*_{N-1}, U, T_N

U^*_{N-1}, U, T_{N-1}

U^*_{N-2}, U, T_{N-1}

U^*_{N-2}, U, T_{N-2}

se T_{N-2} é igual a END e U^*_{N-2} diferente de BEGIN* , as tuplas seguintes seriam:

U^*_{N-3} , U, T_{N-2}

U^*_{N-4} , U, T_{N-2}

se U_{N-2} fosse igual a BEGIN* e T_{N-2} diferente de END, teríamos as triplas:

U^*_{N-2}, U, T_{N-3}

U^*_{N-2}, U, T_{N-4}

⋮

O módulo de recuperação de erro emite uma mensagem só para a situação que motiva o erro (Tripla (U^*_N, U, T_N) do

exemplo) e não para as triplas que surgem da própria recuperação (Triplas(U^*_{N-k}, U, T_{N-j}) do exemplo).

III.4.2. Erros Semânticos

Há alguns erros semânticos que podem ser detectados em tempo de compilação e outros que só são detetados em tempo de execução |Aho[2]|.

Os erros semânticos mais comuns detectados em tempo de compilação são:

- Identificador não declarado;
- Uso inválido de um identificador. Por exemplo, nome de arranjo ou procedimentos usados como identificadores de variáveis simples;
- Uso de rótulo como variável, e vice-versa;
- Declaração múltipla de um identificador;
- Incompatibilidade entre parâmetros formais e reais;
- Incompatibilidade entre a especificação de formato e o tipo da variável a ler/imprimir.

Os erros mais comuns detetados em tempo de execução são:

- Índice de um arranjo inválido;
- Dados de leitura não compatíveis com o formato fornecido;
- Tentativa de alocar mais memória do que a disponível.

No compilador EXPAND o tratamento de erros semânticos, em tempo de compilação, é muito simples, considerando

que ainda existindo um erro semântico no programa, ele está sintáticamente correto. Ao reconhecer a existência de um erro semântico, o compilador EXPAND, emite a mensagem correspodente, interrompe a geração de código e continua a análise sintática normalmente.

Os erros semânticos, em tempo de execução, são detectados ou pelos módulos externos chamados, ou pelo próprio programa gerado que incluirá código correspondente para a deteção desses erros.

CAPÍTULO IV

TRATAMENTO DE IDENTIFICADORES E TEMPORÁRIAS

IV.1. Tabela de Símbolos

A tabela de símbolos contém todas as informações relativas aos identificadores usados no programa fonte. Está constituida de 256 entradas de 2 bytes cada, e seu acesso é feito usando "hashing".

IV.1.1. Função de "hashing"

A função de hashing será obtida usando o método de dobramento seguido do meio-quadrado |Souza[22]|.

Dado o identificador: $K = a_1 a_2 \ldots a_n$ com $1 \leq n \leq 64$

a_i representa os caracteres do identificador, ocupando 1 byte cada.

Definimos:

$$R.[15:16] = \begin{cases} a_1a_2 \oplus a_3a_4 \oplus \ldots \oplus a_{n-1}a_n & \text{se n é par} \\ a_1a_2 \oplus a_3a_4 \oplus \ldots \oplus a_n a_{n+1} & \text{se n é impar com } a_{n+1} = 0 \end{cases}$$

$T.[21:8] = R*R$

$T1.[4:5] = R$

$i(K) = T1*2$

Nota: x.[m:n] deve ser entendido como os n bits à direita do bit m, de x considerando que o bit zero é o bit de menor peso.

A função hashing do identificador K é definida:

$$h_j(k) = \begin{cases} T*2 & \text{se } j = 0 \\ (h_{j-1}(k)+i(k)) \bmod 512 & \text{se } j > 0 \end{cases}$$

Esta função hash obtem um endereço inicial $h_o(k)$ (home-address), e um incremento i(k) usado para tratamento de colisões pelo método de endereçamento aberto com hashing duplo |Souza[22]|.

O endereço inicial obtido é um número par, entre 0 e 510 (9 bits), que garante um acesso dentro dos limites da tabela.

Chegou-se a esta função através de pesquisa, entre funções de igual simplicidade e rapidez de cálculo, feita por programa auxiliar. Escolheu-se a função que apresentou valores de endereços mais uniformemente distribuídos, ou seja, com o menor número médio de colisões primárias.

IV.1.2. Tratamento de Colisões

Se $h_o(K_1) = h_o(K_2)$ com $K_1 \neq K_2$ procurar-se-á um lugar vago na Tabela de símbolos usando hashing duplo |Souza[22]|, considerando a Tabela como sendo circular |Knuth[19]|.

Em linguagens com estrutura de blocos existe a possibilidade de termos, na tabela de símbolos, identificadores com o mesmo nome declarados em vários blocos. Neste caso as

diferentes informações relativas ao identificador serão encadeadas, sendo a inserção feita na cabeça da lista, de tal modo que a informação a ser acessada será sempre a relativa ao identificador do bloco mais interno. (Fig.IV.1)

```
BEGIN
 INTEGER A,B;
   ===
   BEGIN
    REAL B,C;
      ===
    END
END
```

Seja $h_o(A) = 20$, $h_o(B) = 20$, $h_o(C) = 400$

$i(B) = 30$

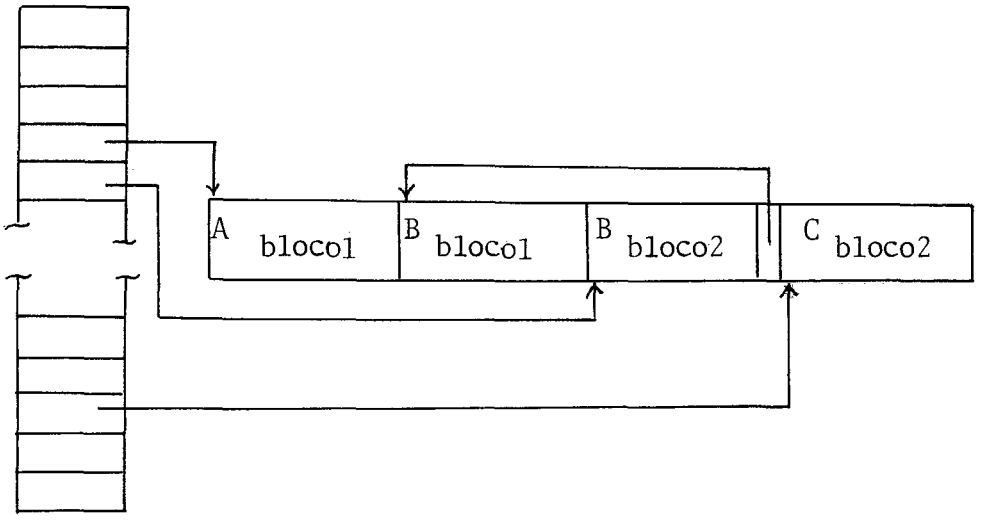
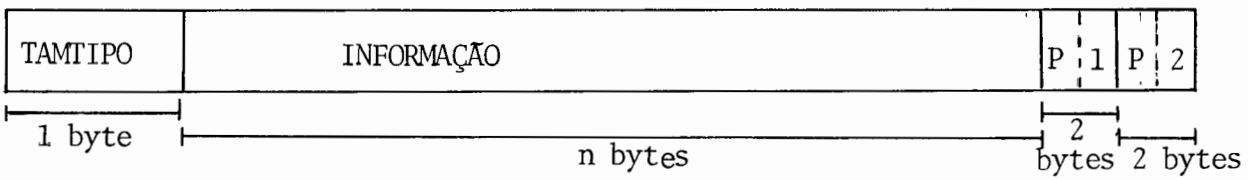


Figura IV.1

Cada entrada da tabela de símbolos contém ou o valor -1, se a entrada está vazia, ou um número maior ou igual a zero, se a entrada está ocupada. Neste último caso, o conteúdo da tabela hash é um ponteiro para um vetor, de nome SIMBOL, que conterá todas as informações relativas ao identificador.

IV.1.3. Estrutura Lógica

Lógicamente SIMBOL está dividido em nós (de comprimento variável) com a seguinte estrutura

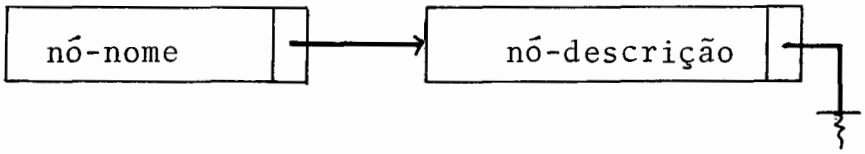


Existem dois tipos de nós:

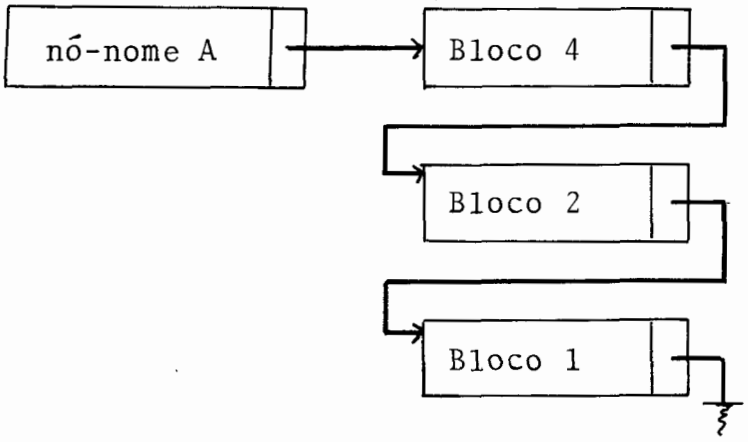
a) Nó-nome. O campo informação contém o nome do identificador;

b) Nó-descrição. O campo informação contém todos os dados relativos ao identificador como tipo, bloco ao qual pertence, posição alocada etc.

Todo identificador terá associado um nó-nome e um nó-descrição.



Para vários identificadores com o mesmo nome (em blocos ou níveis diferentes) só existe um único nó-nome, que será a cabeça de lista, dos nós descrição.



Neste exemplo o identificador A, que aparece nos blocos 1,2 e 4, possui um nó-nome e 3 nós-descrição (um para cada bloco) todos eles encadeados.

Conteúdo do NÓ-NOME

TAMTIPO: número de caracteres do identificador. (1 byte)

INFORMAÇÃO: caracteres do identificador. (TAMTIPO bytes)

P1: Ponteiro de volta à tabela hash correspondente ao identificador, é usado para remover o nó e liberar a respectiva entrada na tabela de símbolos. (2 bytes)

P2: Ponteiro para o nó-descrição do identificador a ser acessado (informação do identificador declarado no bloco mais interno). (2 bytes)

Conteúdo do NÓ-DESCRIÇÃO

TAMTIPO: Indica o tipo do identificador e, implícitamente, o comprimento do nó. (1 byte)

INFORMAÇÃO: Dados relativos ao identificador (bloco de declaração, posição alocada, descrição de parâmetros se for procedimento etc.). (Tamanho variável, definido em IV.1.6).

P1: Aponta para o nó-nome ligado a este nó-descrição. (ou seja a cabeça da lista). (2 bytes)

P2: Aponta para o próximo nó-descrição da lista (se existe), ou é vazio (-1). (2 bytes)
Se não houver identificadores de mesmo nome P2 = -1, caso contrário P2 aponta para o nó-descrição relativo ao identificador, de mesmo nome, declarado no bloco externo de nível mais próximo.

Pode-se ver na figura IV.3 a estrutura correspondente ao programa IV.2.

```
BEGIN  INTEGER B;
         ===
       BEGIN  REAL  A,B;
                ===
              BEGIN  LABEL  B,A;
                       ===
              END
       END
END
```

Programa IV.2

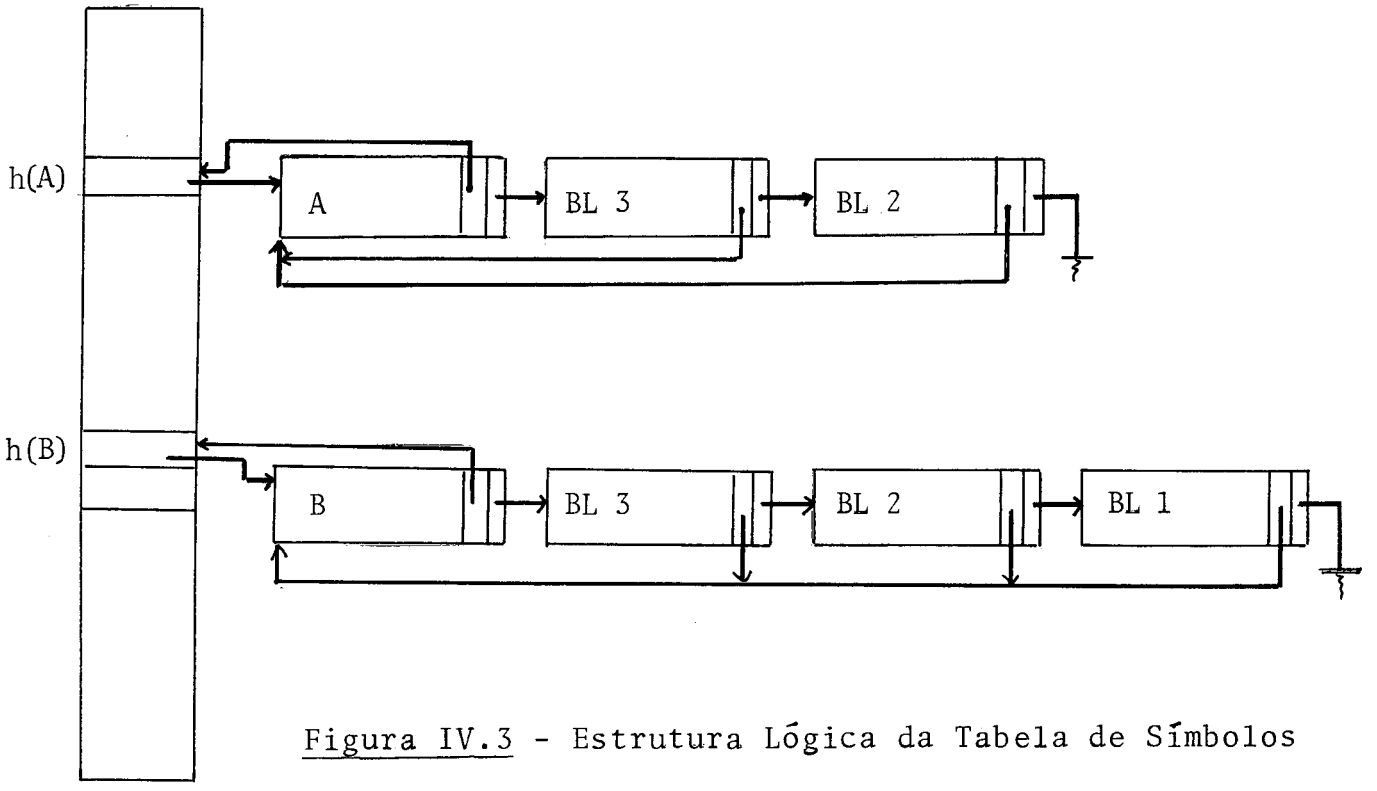


Figura IV.3 - Estrutura Lógica da Tabela de Símbolos

IV.1.4. Estrutura Física da Tabela de Símbolos

O vetor SIMBOL, fisicamente, é uma sequência de bytes que serão preenchidos das posições mais baixas às mais altas. As informações relativas aos identificadores serão armazenadas sequencialmente, na mesma ordem em que aparecem no programa fonte.

Dentro do mesmo vetor, além das informações correspondentes à tabela de símbolos, serão armazenadas informações relativas aos blocos/comandos compostos que serão usados pelo compilador para a geração de código (Fig.IV.4).

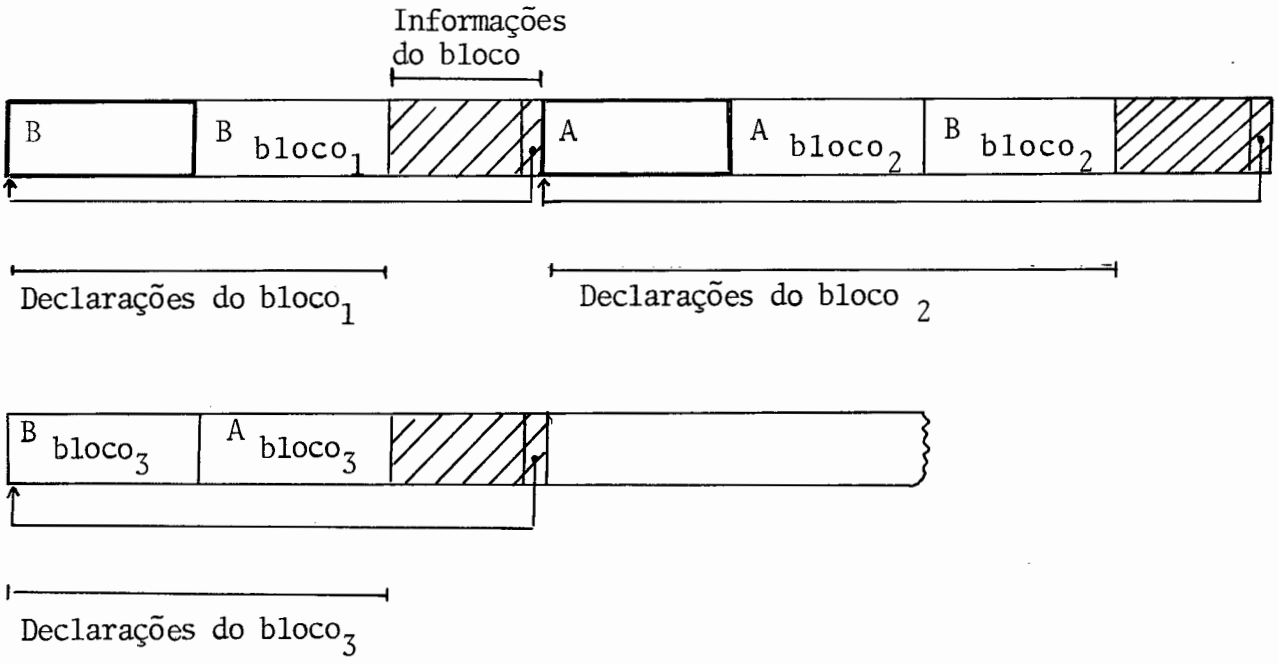


Figura IV.4 - Estrutura física da tabela de símbolos correspondente à estrutura lógica da Figura IV.3. O desenho está simplificado, não mostra os ponteiro P1 e P2 de cada nó.

Para os blocos, depois do tratamento de todas as declarações, será armazenada a seguinte informação:

- Primeira posição livre da memória ao começar o bloco, usada para desalocar a memória ao fechar o bloco;
- Posição mais alta da memória usada no bloco, utilizada para gerar o teste de controle de limite de memória;
- Apontador para o começo das declarações do bloco, ao próprio vetor SIMBOL, usado para remover todos os identificadores do bloco ao achar o END(FEND);

- Código 1, para reconhecer que o END(FEND) fecha um bloco e será tomada a ação semântica de atualizar a Tabela de Símbolos e desalocar a memória alocada a este bloco.

Para os comandos compostos só é armazenado o código Ø (zero), para reconhecer que o END/FEND que aparece, fecha um comando composto e nenhuma ação semântica será executada.

As informações relativas aos blocos/comandos compostos são inseridas, das posições mais baixas as mais altas, ao aparecer o primeiro comando do bloco/comando composto.

A remoção das informações em SIMBOL (ao fechar um bloco) será feita no sentido contrário ao da inserção (isto é das posições mais altas as mais baixas). (Fig.IV.5)

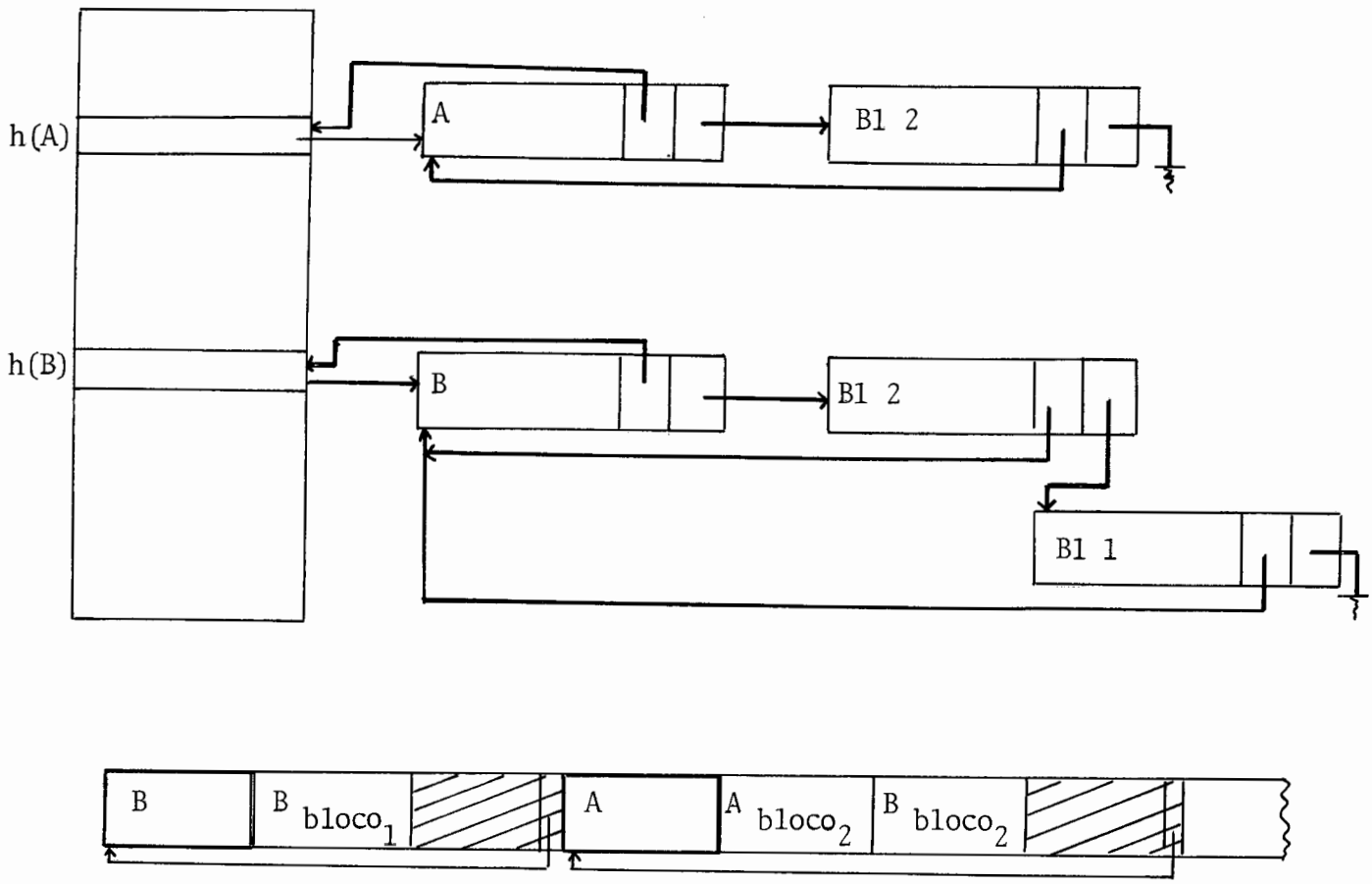


Figura IV.5

Estrutura lógica e física do programa.

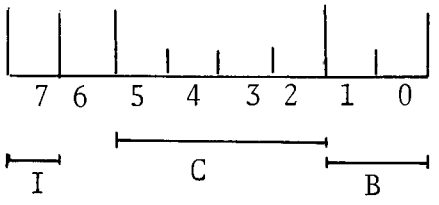
Programa IV.2 ao fechar o bloco 3.

IV.1.5. Informações da Tabela de Símbolos

Na tabela de símbolos, além do nome do identificador que está armazenado no nó-nome, temos outros tipos de informações armazenadas no nó-descrição.

A estrutura do nó-descrição foi descrita em IV.1.4; analisaremos a seguir o conteúdo.

Byte código: contém o código do tipo do identificador. Este byte está dividido em 3 campos.



I Ø variável não referenciada

1 variável referenciada

B 01 label

10 inteiro

11 real

C 0001 variável simples

0010 temporária

0011 parâmetro

0100 vetor

0110 procedimento

1010 temporária vetor

Apresentamos a seguir as combinações válidas dos campos C e B, sem considerar o campo I.

000000 (0) indefinido

000001 (1) label

000110 (6) inteiro simples

001010 (10) temporária inteira

001110 (14) parâmetro inteiro

010010 (18) vetor inteiro

011010 (26) procedure inteira

000111 (7) real simples

001011 (11) temporária real

001111 (15) parâmetro real

010011 (19) vetor real

011011 (27) procedure real

011000 (24) procedure sem tipo

101010 (42) temporária vetor inteiro

101011 (43) temporária vetor real.

Informação associada ao tipos 6 e 7(simples)

- Posição da memória alocada ao identificador (2 bytes). Para inteiros são alocados 2 bytes e para reais 4 bytes;
- Bloco ao qual pertence o identificador (1 byte).

Informação associada ao tipo 1 (rótulo)

- Posição alocada ao rótulo (2 bytes), tem valor positivo para rótulos já definidos e valor negativo para rótulos ainda não definidos. A cada rótulo são alocados 2 bytes;
- Bloco ao qual pertence o rótulo (16 bytes).

Informação associada aos tipos 18 e 19 (vetor)

- Posição alocada para o descritor (2 bytes). Todo descritor tem 2 bytes alocados;
- Bloco ao qual pertence o vetor.

Informação associada aos tipos 24,26 e 27(procedimento)

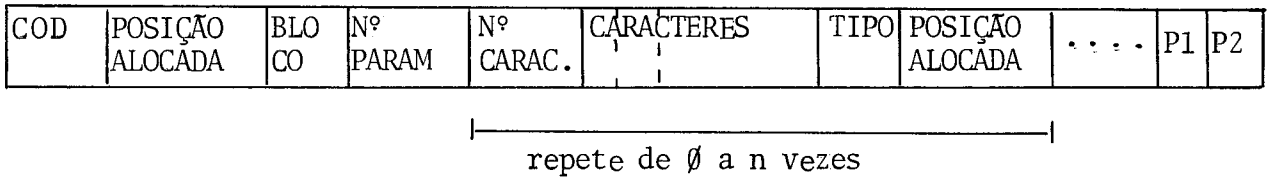
a) Durante o processamento da declaração do procedimento:

- Posição alocada ao procedimento. (2 bytes);
- Bloco ao qual pertence o procedimento. (1 byte);

- Número (n) de parâmetros do procedimento.
  (1 byte).

Para cada parâmetro:

- Número (m) de caracteres do identificador (1 byte);
- Caracteres do identificador (m bytes);
- Tipo do parâmetro (1 byte)
  14 inteiro
  15 real
- Posição alocada ao parâmetro (2 bytes)



Para os procedimentos são alocados 4,6, ou 8 bytes dependendo do tipo. Os primeiros dois bytes são usados para armazenar o endereço de geração do procedimento,os dois bytes seguintes são usados para endereço de retorno.Se o procedimento é com tipo terá alocados 2 ou 4 bytes para o resultado (inteiro ou real). O procedimento sem tipo não tem bytes alocados para resultado.

Aos parâmetros (reais ou inteiros) são alocados 2 bytes, que em tempo de execução conterão o endereço dos parâmetros reais.

b) Depois de processada a declaração de procedimento não é necessário guardar os nomes dos parâmetros, interessa só o tipo e a posição alocada a eles. A tabela de símbolos é atualizada e conterá:

- Posição alocada ao procedimento (2 bytes);
- Bloco ao qual pertence o procedimento (1 byte);
- Número de parâmetros (1 byte);

Para cada parâmetro:

- Tipo (1 byte);
- Posição alocada ao parâmetro (2 bytes);

IV.2. Temporárias

As posições temporárias são necessárias principalmente para guardar resultados parciais de expressões. Também se usam temporárias para armazenar endereços de variáveis com índices |Gries[1]|, e cada bloco função (II.2. diagrama 10) tem igualmente associada uma temporária.

Estas variáveis temporárias são alocadas no momento que são necessárias à geração de código e imediatamente após seu uso são desalocadas.

Define-se como escopo de uma temporária T_i a sequência de operações entre sua definição inicial e sua última referência |Gries[1]|.

É importante notar que os escopos de duas temporárias são ou exteriores ou aninhados |Gries[1]| o que permite usar as posições atribuidas as temporárias como uma pilha (a última posição alocada será a primeira a ser desalocada).

As temporárias não ocupam entradas da tabela de símbolos, porém a informação relativa a elas é armazenada no vetor SIMBOL, das posições mais altas as mais baixas.

Para cada temporária armazenar-se-á a seguinte informação:

- Tipo (1 byte)
  - 10 temporária inteira
  - 11 temporária real
  - 42 temporária-vetor inteira
  - 43 temporária-vetor real.
- Posição alocada à temporária (2 bytes).
  Para os tipos 10,42,43 são alocados 2 bytes e para o tipo 11, 4 bytes.

Os tipos 42 e 43 são usados para armazenar endereços reais das variáveis com índice |Gries[1]|. Por exemplo na expressão

A[I] + B[C + D]

com A real e B inteiro, serão alocados duas temporárias-vetor a primeira para armazenar o endereço A[I] e a segunda para o endereço B[C + D]. Estas temporárias-vetor são usadas com endereçamento indireto (a temporária contém o endereço da variável) a diferença das outras temporárias que são usadas com endereçamento direto (a temporaria contém o valor).

CAPÍTULO V

GERÊNCIA DE MEMÓRIA EM TEMPO DE EXECUÇÃO

V.1. Tratamento de Constantes

São considerados tres tipos de constantes:

1 - Constante inteira $\leq$ 255

2 - Constante inteira > 255

3 - Constante real.

Constante inteira $\leq$ 255

É alocada uma temporária inteira para esta constante e o compilador gera:

```
LOAD    =  constante
STORE      TEMPORÁRIA.
```

Isto porque o MITRA 15 permite definir valores, menores que 256, na própria instrução de "carrege acumulador".

Constantes inteiras > 255 e reais

Para o tratamento destas constantes é criada uma tabela de constantes (TABCON), em tempo de execução, (no programa gerado) com capacidade de 512 bytes.

Ao aparecer uma constante inteira (real), no programa, ela é armazenada em TABCON, e é alocada uma temporária inteira (real) para ela.

Se a constante foi armazenada na posição X de TABCON, é gerado

```
LOAD   TABCON(X)
STORE  TEMPORÁRIA
```

para constantes inteiras, e

```
LOAD    DOUBLE    TABCON(X)
STORE   DOUBLE    TEMPORÁRIA
```

para constantes reais.

V.2. Tratamento de Cadeias

Para o tratamento de cadeias (ver WRITE - FORMATO II.2.7) é criada uma tabela de cadeias (TABCAD) em tempo de execução (no programa gerado). Sua capacidade é de 256 bytes.

As cadeias, na linguagem EXPAND, só podem aparecer em formatos da instrução WRITE.

Ao aparecer uma cadeia ela é armazenada em TABCAD e o endereço de armazenamento será dado como parâmetro ao módulo de impressão, encarregado das operações de saída.

Por exemplo se uma cadeia é armazenada na posição X de TABCAD

```
LOAD  = X
STORE   PAR1
CLS     SAÍDA

No módulo SAÍDA
MOVER     TABCAD(PAR1) a BUFFER-SAÍDA
```

V.3. Tabela de Desvios

As instruções de GO TO, IF, chamada de procedimento, geram instruções de desvio do MITRA (BRANCH).

Para gerar estes desvios o MITRA tem duas opções:

a) Desvio direto, que limita a distância de desvio a 512 bytes;

b) Desvio indireto, que não tem distância máxima de desvio, porém exige a definição de uma palavra na zona de dados.

Foi escolhida a segunda opção por razões evidentes, e foi necessário criar uma tabela de desvios (TABRAN) com uma dimensão de 512 bytes. Portanto, toda instrução BRANCH gerada pelo compilador será feita usando indireção numa determinada posição da tabela.

Isto implica que todo identificador declarado como LABEL tem como endereço de alocação uma posição de TABRAN.

Por exemplo, se o LABEL FORA tem atribuido a posição 20 de TABRAN, uma instrução: GO TO FORA, geraria:

BRANCH #TABRAN(20)

O símbolo # indica indireção.

Este esquema usado para desvios, deixa para o LINK-EDITOR o problema de resolver as referências para frente, pois em TABRAN cada posição é declarada como uma referência do programa, e ao aparecer a definição do rótulo no programa tal referência é definida com o valor do "program counter" |Mitra 15^6|

Por exemplo se o LABEL T, foi alocada à posição Ø de TABRAN, essa posição fica declarada como sendo a referência Ø do programa (artigo ØE - TR |Mitra 15^6|).

Ao aparecer T: (definição de rótulo) no programa fonte, será inserido no programa gerado uma definição da referência Ø (artigo ØB-BT |Mitra 15^6|), com o valor, nesse momento, do "program counter".

V.4. Administração da Memória em Tempo de Execução

Todo programa, no MITRA 15, possui duas áreas de dados, a primeira denominada COMMON DATA SECTION (CDS) contem os dados que serão usados tanto pelo programa principal, como pelos módulos por ele chamados. A segunda área denominada LOCAL DATA SECTION (LDS) contém os dados locais ao programa (não são acessíveis pelos módulos chamados pelo programa principal).

Dado que o programa gerado pelo compilador EXPAND, chama módulos externos (rotina de entrada/saída, alocação de vetores etc) foi necessário definir uma CDS e uma LDS.

A CDS contém principalmente a tabela de cadeias (V.2) e a "memória", denominada memória-usuário, usada pelo programa gerado. A LDS contém a tabela de constantes (V.1) e a tabela de desvios (V.3).

A figura V.1 mostra a organização da memória do MITRA 15, na execução do programa gerado pelo compilador EXPAND.

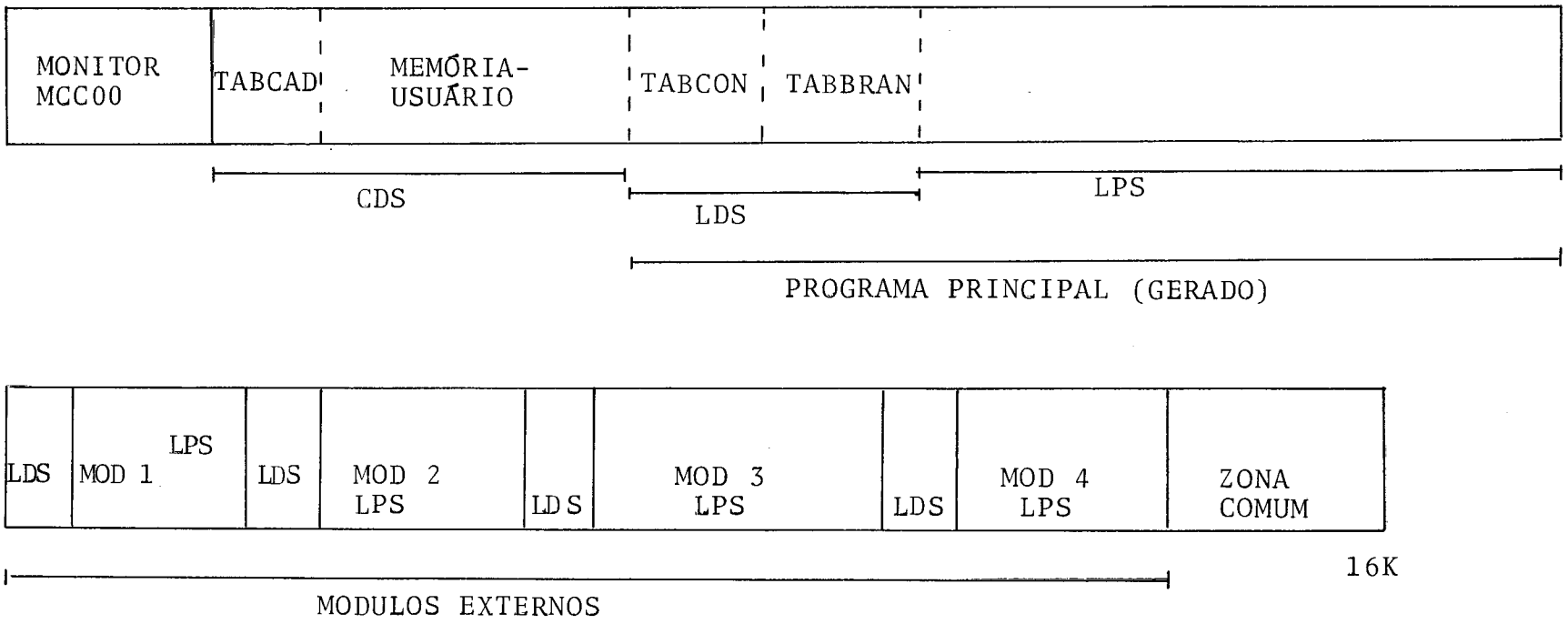


Figura V.1 - Esquema da memória total do MITRA-15 na execução do programa gerado.

A memória-usuário tem uma capacidade máxima de 13824 bytes, sendo que o tamanho real depende do programa do usuário, que deve ocupar no máximo 3K bytes para poder ter toda a memória-usuário disponível. Isto é, um programa de 3K bytes conta com 13824 bytes de memória-usuário; no entanto um programa de 6K bytes terá disponível 13824-3K bytes de memória-usuário.

É importante notar que estes cálculos de memória foram feitos em base a ocupação do monitor MCC00 que é de aproximadamente 10K bytes.

Devido a restrições de endereçamento do MITRA-15 |Mitra-15[14]| a acesso à memória-usuário é feito usando indireção e indexação.

A memória-usuário foi organizada em 54 blocos de 256 bytes cada (Figura V.2)

A organização mostrada na figura V.2, garante o endereçamento da memória-usuário com duas instruções de máquina.

No que se segue denominaremos a memória-usuário como sendo uma pilha de nome MEMÓRIA.

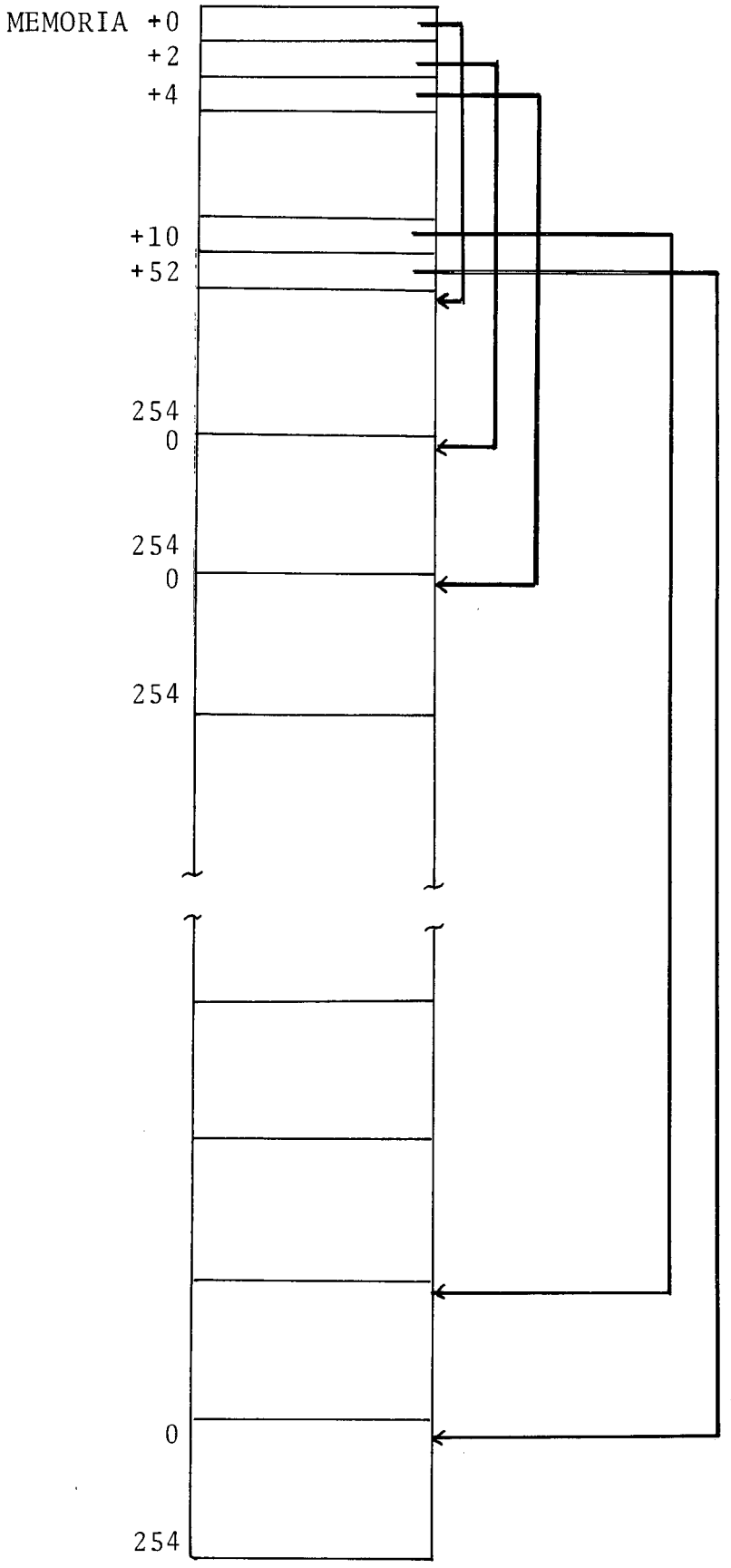


Figura V.2 - Organização da memória-usuário

V.4.1. Alocação de Variáveis

A pilha MEMÓRIA é considerada como uma pilha de duas entradas: das posições mais baixas às mais altas são alocadas as variáveis que não são arranjos e das posições mais altas às mais baixas, são alocadas as posições para os arranjos.

A alocação dos arranjos é feita em tempo de execução a diferença da alocação das variáveis simples que é feita em tempo de compilação.

Para os seguintes tipos de variáveis são alocados 2 bytes:

- inteiro simples
- descritor de arranjos
- endereço de geração de procedimento
- endereço de retorno do procedimento
- parâmetros.

Só as variáveis reais tem alocados 4 bytes.

Ao entrar a um novo bloco serão alocados 2 bytes, que serão usados como ponteiro para a próxima posição livre da MEMÓRIA.

A seguir mostramos um programa (Figura V.3) e a situação da pilha MEMÓRIA em tempo de compilação (Fig.V.4) e em tempo de execução (Fig.V.5 e V.6).

```
BEGIN
   INTEGER     A,B;
   INTEGER     VECTOR  X[2:4], T[1:2];

    ==
    ==
   BEGIN
      REAL C,D,E;
      REAL  VECTOR  Y[A:B]

       =
       =
   END
END
```

Figura V.3

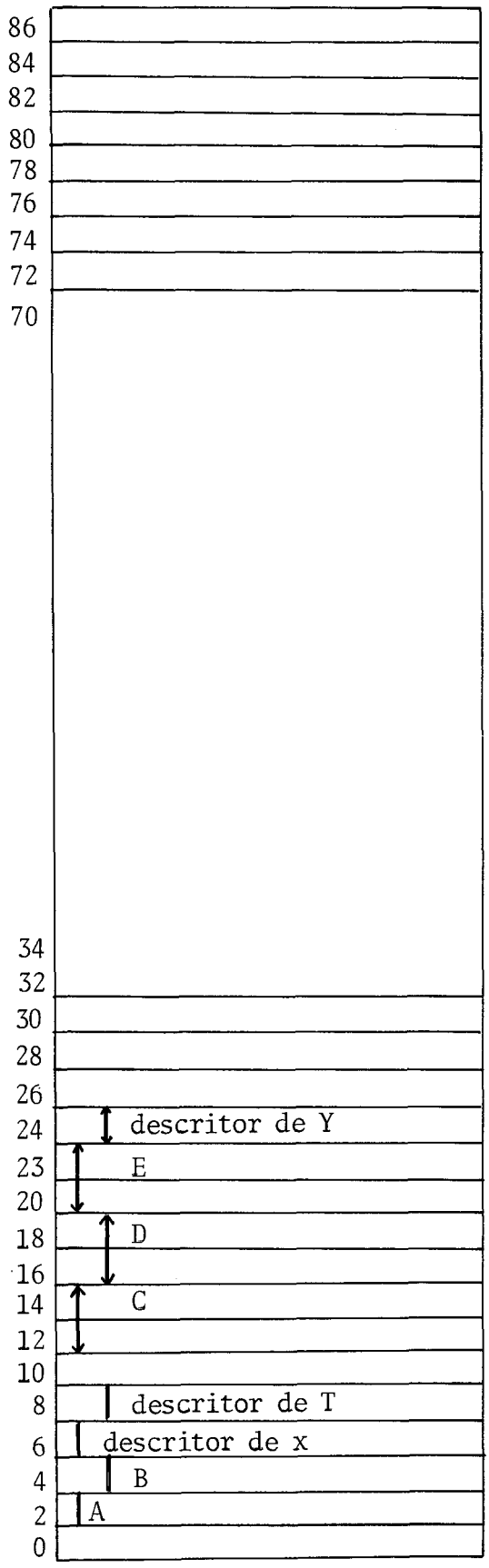
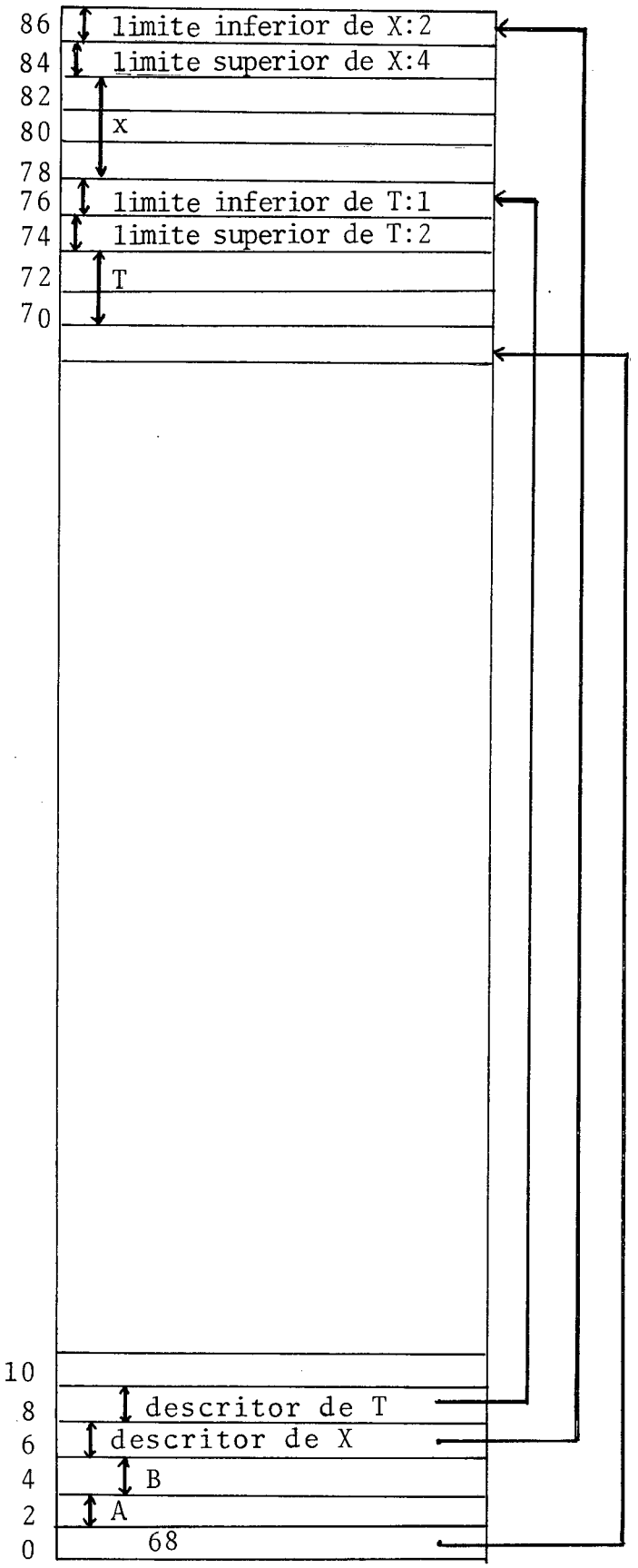


Figura V.4 - MEMÓRIA antes de fechar o segundo bloco em tempo de compilação.

↕ indica alocado a.



Os limites superior e inferior são usados para testar índice inválido na referencia a um arranjo.

← posição livre do bloco na parte superior de MEMÓRIA

Figura V.5 - MEMÓRIA depois de alocar memória para X e Y em tempo de execução.

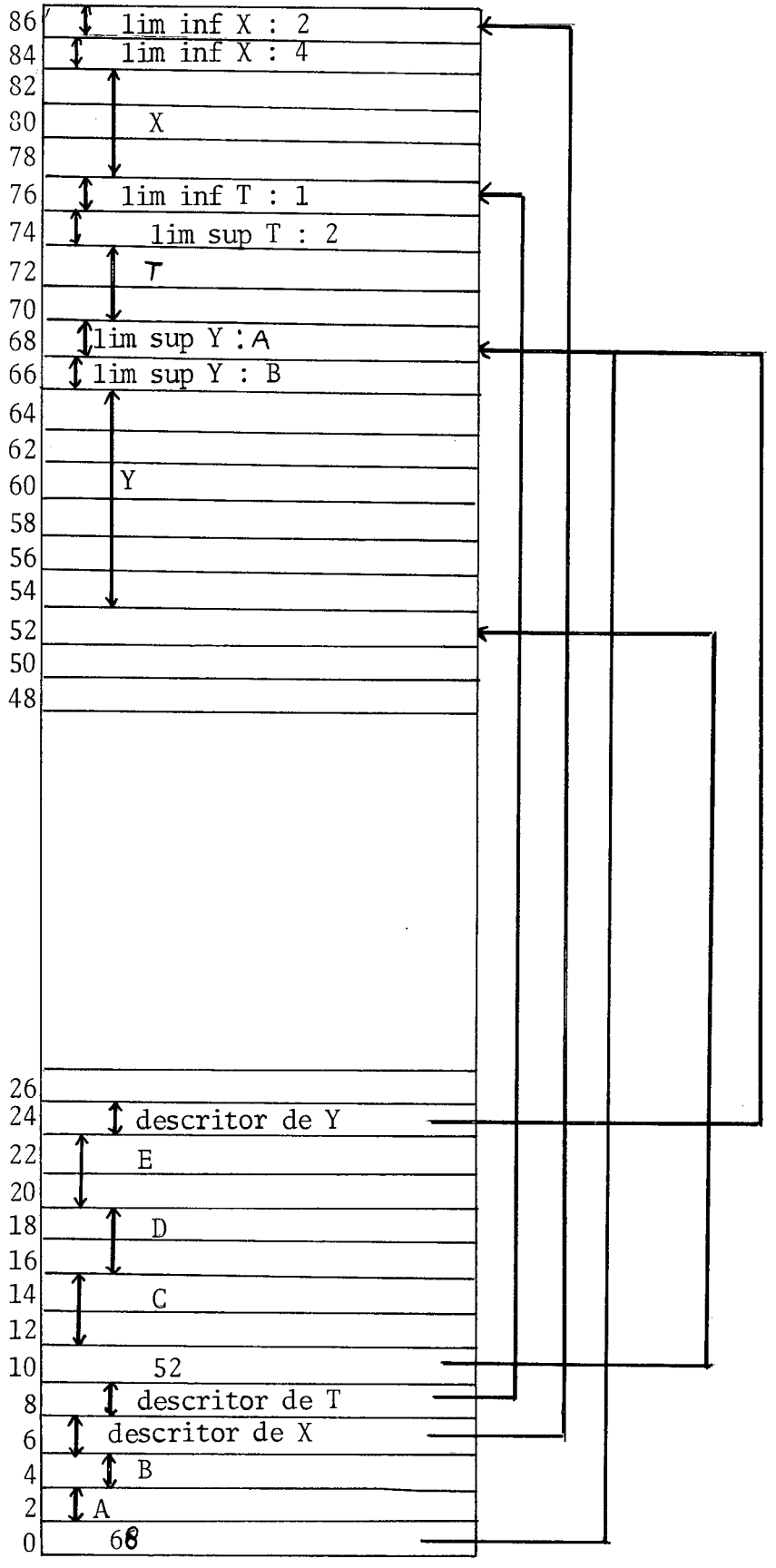


Figura V.6 - MEMÓRIA antes de fechar o segundo bloco em tempo de execução.

V.4.2. Alocação de Variáveis Temporárias

As variáveis temporárias são alocadas em qualquer ponto do programa objeto.

Para o seguinte programa:

```
BEGIN   INTEGER   A[1:20];
        INTEGER   C,D,E;

        A[1]:=C + FBEGIN
                    ==
                  FEND;
        BEGIN  INTEGER B;

          ==

END
```

Programa V.7

Ao processar a declaração do arranjo A, alocamos uma temporária para cada limite. Estas temporárias serão desalocadas imediatamente após a chamada do módulo de alocação de memória, como mostrado nas figuras V.8 e V.9.

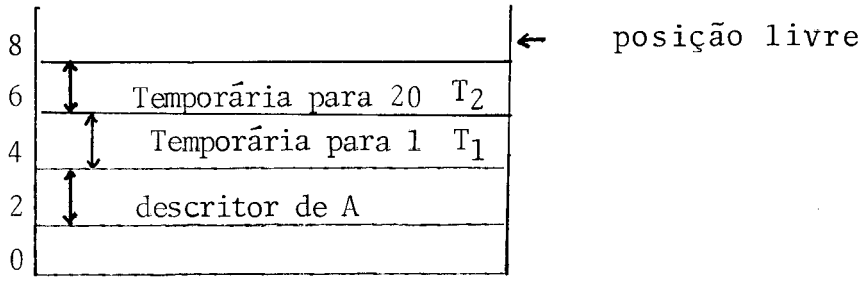


Figura V.8 - Alocação da memória ao processar a 1a. declaração do programa V.8.

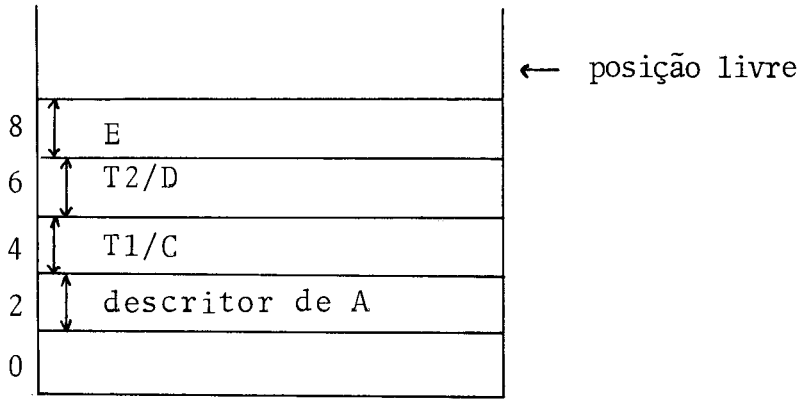


Figura V.9. Alocação da memória ao processar a segunda declaração do programa V.7. As temporárias T1 e T2 já foram desalocadas a barra (/) indica que a mesma posição da memória foi alocada para várias variáveis. No caso T1/C a posição 4 foi alocada à temporária T1 e logo à variável C.

Ao processar o comando de atribuição serão alocadas as seguintes temporárias:

T3 posição 10. Para a constante Ø que será desalocada após a chamada do módulo que faz o cálculo do índice de A.

T4 posição 10. Para armazenar o endereço de A[Ø].

T5 posição 12. Temporária associada ao Bloco função, que depois será usada para a geração da soma.

As temporárias T4 e T5 serão desalocadas só depois

de gerada a atribuição.

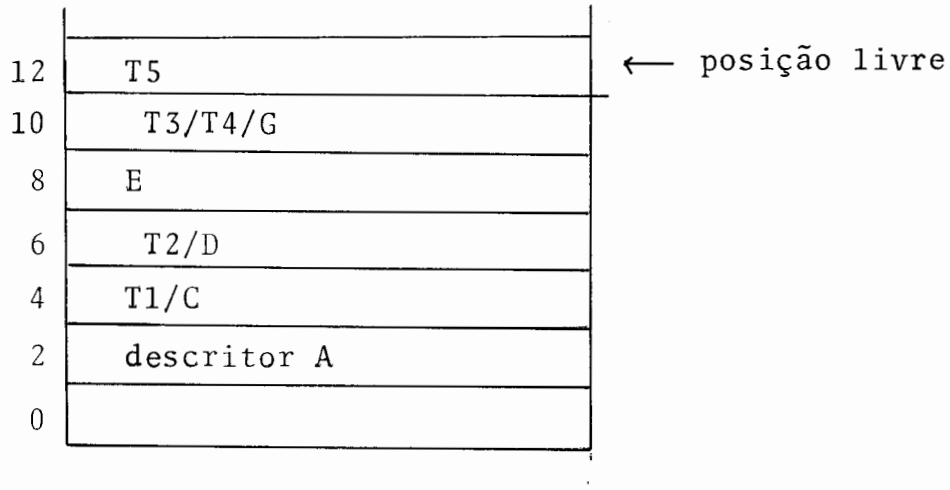


Figura V.10. Alocação da memória após o processamento da declaração do segundo bloco.

V.5. Módulos Externos

O programa gerado, chama alguns módulos externos para a execução de tarefas específicas e independentes da estrutura do programa a ser executado.

V.5.1. Módulo ERRO

Emite todas as mensagens de erro de execução, como por exemplo índice inválido, erro de formato, estouro de memória etc.

V.5.2. Módulo LESCR

Executa todas as operações de entrada e saída.

Tem como parâmetros:

- Formato
- Endereço da variável a ler/escrever
- Número de posições inteiras e decimais a ler/escrever
- Em caso de arranjo, os valores dos índices.

V.5.3. Módulo ALOCA

Encarregado da alocação de memória para arranjos.

Parâmetros:

- Endereço do descritor
- Tipo de arranjo (real ou inteiro)
- Valor do limite inferior
- Valor do limite superior.

V.5.4. Módulo SUBIND

Calcula o endereço real de uma variável indexada com teste de índice inválido.

Parâmetros:

- Endereço do descritor do arranjo
- Valor do índice
- Retorna o endereço real

CAPÍTULO VI

CONCLUSÕES E CONSIDERAÇÕES FINAIS

O trabalho a que se propos esta tese, isto é, a construção do compilador da linguagem base EXPAND foi completado e encontra-se disponível no MITRA 15 do Laboratório de Automação e Simulação de Sistemas. A totalidade do trabalho, isto é, macro expansor e compilador, estará disponível ao término do trabalho que está sendo desenvolvido por |Kullock[15]|.

Como mencionado anteriormente, a linguagem EXPAND é uma linguagem experimental cujas características parecem adequadas à implementação em minicomputador.

O uso de macros sintáticas, embora ofereça algumas características semelhantes às linguagens de alto nível, envolve também algum desconforto para o usuário, principalmente no cuidado necessário para o uso de macros tão poderosas.

Este trabalho poderia ser extendido, principalmente, pela inclusão de um módulo otimizador de código, que certamente se justifica uma vez que algumas construções implementadas através de macros sintáticas, podem revelar-se ineficientes.

Outras extensões possíveis incluiriam, por exemplo, adicionar à linguagem a capacidade de abstração de dados, que acoplada ao uso de macro-sintática permitiria que a linguagem adquira maior flexibilidade e potência.

BIBLIOGRAFIA

|1| Gries, David - Compiler Construction for digital Computers, John Wiley & Sons, Wiley, 1971.

|2| Aho, Alfred e Ullman, Jeffrey - Principles of Compiler Design, California, Addison-Wesley, 1978.

|3| Caúla, Lígia Barros - PLMIX - Um modelo de linguagem de médio nível e seu compilador - Rio de Janeiro, Tese de M.Sc., COPPE/UFRJ, 1978.

|4| De Simone, Estevam Gilberto - Tese de doutoramento, a ser publicada- Rio de Janeiro, COPPE/UFRJ.

|5| De Simone, Estevam Gilberto - Notas de aulas do curso Compiladores III, COPPE-UFRJ, 1979.

|6| MITRA 15, Editeurs Chargeur ECH, Editeurs de liens EDL, EDL-E, EDL-EX, EDL-D, EDL-DX, França, CII, 1975.

|7| Leavenworth, B.M. - Syntax Macros and Extended Translation, CACM, USA, 9(11):790-793, 1966.

|8| Schwarz, G. - O laboratório de automação e simulação de sistemas (LASS):descrição e sua atuação desde a origem até 1978- Rio de Janeiro, COPPE/UFRJ, 1978.

|9| Naur, Peter - Revised Report on the Algorithms Languages Algol 60 - Communications of the ACM, USA, 6(1):65-87, 1960.

|10| MITRA-15 Langage LP15, LP15E, França, Compagnie Internacionale pour L'Informatique, 1975.

|11| MITRA-15 Moniteur temps réel MTR - França, CII, 1974.

|12| Aho, Alfred e Ullman, Jeffrey - Theory of parsing, translation and compiling - N.J., Prentice Hall, 1978.

|13| Mitra 15 Module d'enchaînement Batch, França, CII, 1973.

|14| Mitra 15, Manuel de reference , França, CII, 1972.

|15| Kullock, Paulo - Tese de mestrado a ser publicada, Rio de Janeiro, COPPE/UFRJ.

|16| Mitra 15, Manuel de references Entrées/Sorties , França, CII, 1975.

|17| Mitra 15, Bibliothèque Mathématique FLSD, Franca, CII, 1975.

|18| Mitra 15, Bibliotecaire BIB , França, CII, 1975.

|19| Knuth, D.E. - The art of computer programming vol 1 : Fundamental Algorithm , Reading, Mass., Addison-Wesley, 1968.

|20| Knuth D.E.- The art of computer programming vol 3 : sorting and searching - Reading, Mass., Addison-Wesley, 1968.

|21| Gries, D. - Error recovery and correction - an introduction to the literatura-compiler construction-an advances course - Springer-Verlag, 1976.

|22| De Souza, Jano Moreira - Algoritmos de hashing para problemas específicos, Rio de Janeiro, Tese de M.Sc., COPPE-UFRJ, 1978.

APÊNDICE 1

PALAVRAS RESERVADAS

BEGIN

END

ENDMACRO

EQ

FBEGIN

FEND

GE

GO TO

GO

GT

IF

INTEGER

LABEL

LE

LT

MACRO

NE

PROCEDURE

READ

REAL

RESULT

THEN

TO

VECTOR

WRITE

APÊNDICE 2

GRAMÁTICA MODIFICADA

```
@<PRØGRAMA>#<BLØCØ>@
@<BLØCØ>#BEGIN<CØRPØ>END@
@<CØRPØ>#<CØMANDØ>@
@<CØRPØ>#<CØRPØ>;<CØMANDØ>@
@<CØRPØ>#<LISTA DECL>;<CØMANDØ>@
@<LISTA DECL>#<LISTA DECL>;<DECL>@
@<LISTA DECL>#<DECL>@
@<DECL>#<DECE>@
@<DECL>#<DECV>@
@<DECL>#<DECP>@
@<DECE>#INTEGER<LISTA IDEN>@
@<DECE>#REAL<LISTA IDEN>@
@<DECV>#LABEL<LISTA IDEN>@
@<DECV>#REAL&VECTØR<LISTA VETØRES>@
@<DECV>#INTEGER&VECTØR<LISTA VETØRES>@
@<DECP>#<CABPRØC>;<BLØCØ>@
@<DECP>#<FCABPRØC>;<FBLØCØ>@
@<CABPRØC>#PRØCEDURE<IDEN>@
@<CABPRØC>#PRØCEDURE<IDEN>(<LISTA IDEN>)@
@<CABPRØC>#<CABPRØC>;<DECE>@
@<FCABPRØC>#INTEGER&PRØCEDURE<IDEN>@
@<FCABPRØC>#INTEGER&PRØCEDURE<IDEN>(<LISTA IDEN>)@
@<FCABPRØC>#REAL&PRØCEDURE<IDEN>@
@<FCABPRØC>#REAL&PRØCEDURE<IDEN>(<LISTA IDEN>)@
@<FCABPRØC>#<FCABPRØC>;<DECE>@
@<LISTA IDEN>#<LISTA IDEN>,<IDEN>@
@<LISTA IDEN>#<IDEN>@
@<LISTA VETØRES>#<LISTA VETØRES>,<IDEN>[<EXP>:<EXP>]@
@<LISTA VETØRES>#<IDEN>[<EXP>:<EXP>]@
@<IDEN>#ID@
@<CØMANDØ>#<ATRIBUICAØ>@
@<ATRIBUICAØ>#<VARIAVEL>,<ATRIBUICAØ>@
@<ATRIBUICAØ>#<VARIAVEL>:=<LISTA EXP>@
@<LISTA EXP>#<EXP>,<LISTA EXP>@
@<LISTA EXP>#<EXP>@
@<CØMANDØ>#<CLAUSULA IF>LT<CLAUSULA THEN>@
@<CØMANDØ>#<CLAUSULA IF>LE<CLAUSULA THEN>@
@<CØMANDØ>#<CLAUSULA IF>EQ<CLAUSULA THEN>@
@<CØMANDØ>#<CLAUSULA IF>NE<CLAUSULA THEN>@
@<CØMANDØ>#<CLAUSULA IF>GT<CLAUSULA THEN>@
@<CØMANDØ>#<CLAUSULA IF>GE<CLAUSULA THEN>@
@<CLAUSULA IF>#IF<EXP>@
@<CLAUSULA THEN>#<EXP>THEN<CØMANDØ>@
@<CØMANDØ>#RESULT<EXP>@
@<CØMANDØ>#<RØTULØ>:<CØMANDØ>@
@<CØMANDØ>#<RØTULØ>:@
@<CØMANDØ>#<CHAMADA>@
@<CHAMADA>#<IDEN>(<LISTA IDEN>)@
@<CØMANDØ>#<IDEN>@
@<CØMANDØ>#<BLØCØ>@
@<CØMANDØ>#GØ&TØ<RØTULØ>@
@<CØMANDØ>#GØTØ<RØTULØ>@
@<CØMANDØ>#GØ<RØTULØ>@
@<CØMANDØ>#WRITE<(LIST)>@
@<CØMANDØ>#READ<(LIST)>@
@<(LIST)>#(<LISTA I/Ø>)@
@<LISTA I/Ø>#<LISTA I/Ø>;<FØRMATØ>@
@<LISTA I/Ø>#<LISTA I/Ø>;<DUPLA>@
@<LISTA I/Ø>#<DUPLA>@
@<DUPLA>#<FØRMATØ>,<IDEN>@
@<DUPLA>#<FØRMATØ>,<IDEN>[<INDICE>]@
@<DUPLA>#<FØRMATØ>,<IDEN>[<INDICE>:<INDICE>]@
```

```
@<LISTA I/Ø>#<FØRMATØ>@
@<INDICE>#<INTEIRØ>@
@<INDICE>#<IDEN>@
@<FØRMATØ>#F<INTEIRØ>.<INTEIRØ>@
@<FØRMATØ>#E<INTEIRØ>.<INTEIRØ>@
@<FØRMATØ>#I<INTEIRØ>@
@<FØRMATØ>#A<INTEIRØ>@
@<FØRMATØ>#X<INTEIRØ>@
@<FØRMATØ>#L<INTEIRØ>@
@<FØRMATØ>#CAD@
@<INTEIRØ>#NUM@
@<RØTULØ>#<IDEN>@
@<EXP>#<EXP>+<TERMØ>@
@<EXP>#<EXP>-<TERMØ>@
@<EXP>#<TERMØ>@
@<EXP>#-<TERMØ>@
@<EXP>#+<TERMØ>@
@<TERMØ>#<TERMØ>*<PRIMARIØ>@
@<TERMØ>#<TERMØ>/<PRIMARIØ>@
@<TERMØ>#<PRIMARIØ>@
@<PRIMARIØ>#<VARIAVEL>@
@<PRIMARIØ>#CØNS@
@<PRIMARIØ>#(<EXP>)@
@<PRIMARIØ>#<FBLØCØ>@
@<PRIMARIØ>#<CHAMADA>@
@<FBLØCØ>#FBEGIN<CØRPØ>FEND@
@<VARIAVEL>#<IDEN>[<EXP>]@
@<VARIAVEL>#<IDEN>@
```

APÊNDICE 3

SÍMBOLOS TERMINAIS, NÃO TERMINAIS E ESTRELADOS

Simbolos Terminais

```
 1 %
 2 BEGIN
 3 END
 4 ;
 5 INTEGER
 6 REAL
 7 LABEL
 8 VECTØR
 9 PRØCEDURE
10 (
11 )
12 ,
13 [
14 :
15 ]
16 ID
17 :=
18 LT
19 LE
20 EQ
21 NE
22 GT
23 GE
24 IF
25 THEN
26 RESULT
27 GØ
28 TØ
29 GØTØ
30 WRITE
31 READ
32 F
33 .
34 E
35 I
36 A
37 X
38 L
39 CAD
40 NUM
41 +
42 -
43 *
44 /
45 CØNS
46 FBEGIN
47 FEND
```

Simbolos Não Terminais

0
1 DUPLA
2 FØRMATØ
3 DECP
4 DECV
5 DECE
6 DECL
7 BLØCØ
8 CØMANDØ
9 CHAMADA
10 ATRIBUICAØ
11 IDEN
12 INTEIRØ
13 VARIAVEL
14 PRIMARIØ
15 TERMØ
16 EXP
17 FBLØCØ
18 PRØGRAMA
19 SLINHA
20 CØRPØ
21 LISTA DECL
22 LISTA IDEN
23 LISTA VETØRES
24 CABPRØC
25 FCABPRØC
26 LISTA EXP
27 CLAUSULA IF
28 CLAUSULA THEN
29 RØTULØ
30 (LIST)
31 LISTA I/Ø
32 INDICE

Estrelados

```
33 %*
34 %*-PRØGRAMA-%*
35 BEGIN*
36 BEGIN*-CØRPØ-END*
37 CØRPØ-;*
38 LISTA DECL-;*
39 INTEGER*
40 REAL*
41 LABEL*
42 REAL*- -VECTØR*
43 INTEGER*- -VECTØR*
44 CABPRØC-;*
45 FCABPRØC-;*
46 PRØCEDURE*
47 PRØCEDURE*-IDEN-(*
48 PRØCEDURE*-IDEN-(*-LISTA IDEN-)*
49 INTEGER*- -PRØCEDURE*
50 INTEGER*- -PRØCEDURE*-IDEN-(*
51 INTEGER*- -PRØCEDURE*-IDEN-(*-LISTA IDEN-)*
52 REAL*- -PRØCEDURE*
53 REAL*- -PRØCEDURE*-IDEN-(*
54 REAL*- -PRØCEDURE*-IDEN-(*-LISTA IDEN-)*
55 LISTA IDEN-,*
56 LISTA VETØRES-,*
57 LISTA VETØRES-,*-IDEN-[*
58 LISTA VETØRES-,*-IDEN-[*-EXP-:*
59 LISTA VETØRES-,*-IDEN-[*-EXP-:*-EXP-]*
60 IDEN-[*
61 IDEN-[*-EXP-:*
62 IDEN-[*-EXP-:*-EXP-]*
63 ID*
64 VARIAVEL-,*
65 VARIAVEL-:=*
66 EXP-,*
67 CLAUSULA IF-LT*
68 CLAUSULA IF-LE*
69 CLAUSULA IF-EQ*
70 CLAUSULA IF-NE*
71 CLAUSULA IF-GT*
72 CLAUSULA IF-GE*
73 IF*
74 EXP-THEN*
75 RESULT*
76 RØTULØ-:*
77 IDEN-(*
78 IDEN-(*-LISTA IDEN-)*
79 GØ*
80 GØ*- -TØ*
81 GØTØ*
82 WRITE*
83 READ*
84 (*
85 (*-LISTA I/Ø-)*
86 LISTA I/Ø-;*
87 FØRMATØ-,*
88 FØRMATØ-,*-IDEN-[*
89 FØRMATØ-,*-IDEN-[*-INDICE-]*
90 FØRMATØ-,*-IDEN-[*-INDICE-:*
91 FØRMATØ-,*-IDEN-[*-INDICE-:*-INDICE-]*
92 F*
93 F*-INTEIRØ-.*
94 E*
```

```
 95 E*-INTEIRØ-.*
 96 I*
 97 A*
 98 X*
 99 L*
100 CAD*
101 NUM*
102 EXP-+*
103 EXP--*
104 -*
105 +*
106 TERMØ-**
107 TERMØ-/*
108 CØNS*
109 (*-EXP-)*
110 FBEGIN*
111 FBEGIN*-CØRPØ-FEND*
112 IDEN-[*-EXP-]*
```

APÊNDICE 4

PROGRAMAS EXEMPLO

```
% PROGRAMA EXEMPLO  (COM DEFINIÇÃO E USO DE MACRO)
% MACRO IFF-THEN-ELSE
MACRO IFF $COND THEN $EXP1 ELSE $EXP2
DEFINE   FBEGIN  LABEL #L;
          IF $COND THEN   BEGIN
                          RESULT $EXP1;
                          GO TO #L
                          END;
          RESULT $EXP2;
          #L:
         FEND ;
% MACRO FOR
MACRO FOR $VAR :=$EXP1 UNTIL $EXP2 DO $STAT
DEFINE   BEGIN LABEL #L1, #L2;
          $VAR:=$EXP1;
          #L1:
          IF $VAR GT $EXP2 THEN GO TO #L2;
          $STAT;
          $VAR:=$VAR+1;
          GO TO #L1 ;
          #L2:
         END
% FIM DE DECLARAÇÕES DE MACRO
% PROGRAMA QUE CALCULA O N-ESIMO NUMERO DE FIBONACCI
BEGIN INTEGER N,F ;
READ (I2,N);
F:=IFF N EQ Ø
    THEN Ø
       ELSE  IFF N EQ 1
                THEN 1
                ELSE
                 FBEGIN INTEGER ULT,J,FIB;
                 ULT, FIB:=Ø,1;
                 FOR J:=2 UNTIL N DO ULT,FIB:=FIB,ULT+FIB;
                 RESULT FIB
                 FEND;
WRITE("NUMERO"; I2, N; "DA SERIE DE FIBONACCI";I6,F);
END
```

```
% PROGRAMA EXPANDIDO
BEGIN INTEGER N,F ;
READ (I2,N);
F:=  FBEGIN LABEL #L;
     IF N EQ  0 THEN   BEGIN
                       RESULT Ø; GO TO #L
                       END;
     RESULT
       FBEGIN  LABEL #L;
       IF N EQ  1 THEN   BEGIN
                         RESULT 1; GO TO #L
                         END;
       RESULT
        FBEGIN INTEGER ULT,J,FIB;
        ULT,FIB:=Ø,1;
         BEGIN LABEL #L1,#L2;
         J:=2;
         #L1:
         IF J GT N THEN GO TO #L2;
         ULT,FIB:=FIB,ULT+FIB;
         J:=J+1;
         GO TO #L1;
         #L2:
         END;
       RESULT FIB
       FEND;
     #L:
     FEND;
WRITE ("NUMERO; I2,N; "DA SERIE DE FIBONACCI";I6,F);
END
```