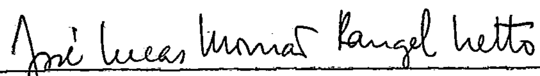
EXPAND UMA LINGUAGEM EXTENSÍVEL ATRAVÉS DE MACROS SINTÁTICAS: ANÁLISE LÉXICA E TRATAMENTO DAS MACROS

Paulo Cesar Kullock

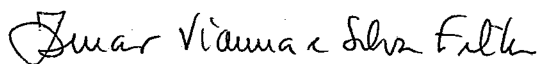
TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS (M.Sc.).

Aprovada por:



Prof.José Lucas M.Rangel Netto
(Presidente)

Prof.Paulo Augusto S.Veloso



Prof.Ysmar Vianna e Silva Filho

RIO DE JANEIRO, RJ - BRASIL
MARÇO DE 1981

KULLOCK, PAULO CESAR

Expand Uma linguagem Extensível Através de Macros Sintáticas: Análise Léxica e Tratamento das Macros Rio de Janeiro 1981.

VII, 94p. 29,7cm (COPPE-UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1981)

Tese - Univ. Fed. Rio de Janeiro. COPPE

1. Compiladores I. COPPE/UFRJ II. Título (série).

À Tamar

AGRADECIMENTOS

Ao Prof. José Lucas Mourão Rangel Netto pela orientação, acompanhada de apoio nos maus momentos.

À Profª Beatriz Zakimi Miyasato pelas informações que prestou-se a fornecer.

Ao Prof. Jean-Michel Nayrac pela incrível boa vontade demonstrada tôdas as vezes em que solicitei sua colaboração.

À SERVENCO, por tornar possível o mestrado, e aos numerosos colaboradores que lá dentro encontrei, como Airma Renner Vasconcelos, que trabalhou fora de seu horário para esta tese.

RESUMO

Macros e procedimentos foram desenvolvidos em várias direções e têm sido usados para estender as capacidades das linguagens de programação. A macro convencional é encontrada geralmente em conjunção com código de montagem.

Leavenworth e outros propuseram o uso da macro sob uma forma diferente. Esta macro, chamada de macro sintática, não tem uma estrutura fixa, como a macro convencional. Com isto, pode-se adicionar novas estruturas sintáticas às linguagens de programação. Os argumentos destas novas estruturas são elementos sintáticos como "comando", "identificador", "condição", por exemplo.

O objetivo desta tese e da de Beatriz Zakimi Miyasato é a definição de uma linguagem extensível através de macros sintáticas e da implementação de um compilador para esta linguagem no minicomputador MITRA-15 do Laboratório de Automação e Simulação de Sistemas do Programa de Engenharia de Sistemas e Computação da COPPE-UFRJ. A presente tese aborda a análise léxica e o tratamento das macros, enquanto que a de Beatriz Zakimi Miyasato trata da análise sintática e geração de código.

ABSTRACT

Macro and procedures have been developed in many directions and have been used to extend the capabilities of programming languages. The conventional macro is usually found in connection with assembly code.

Leavenworth and others have proposed the use of macros in a different form. This macro, called a syntax macro, does not have a fixed structure, like the conventional macro. In this way, it is possible to add new syntactic structure to programming languages. The arguments of these new structures are syntactic elements like "statement", "identifier", "condition".

The objetive of this thesis and that of Beatriz Zakimi Miyasato is the definition of a language capable of extension by syntax macros, and the implementation of a compiler for this language on the MITRA-15 minicomputer at COPPE'S Systems Laboratory. This thesis deals with the lexical analysis and the handling of the macros. Zakimi's thesis deals with the syntactic analysis and code generation.

ÍNDICE

	Página
CAPÍTULO I - INTRODUÇÃO	1
CAPÍTULO II - RECURSOS DISPONÍVEIS	5
II.1 - Equipamento	5
II.2 - Software Básico	5
CAPÍTULO III - DESCRIÇÃO DA LINGUAGEM	7
III.1 - Estrutura da Linguagem	7
III.2 - Especificação da Linguagem	10
III.2.1 - Programa	10
III.2.2 - Declaração	13
III.2.3 - Declaração de Procedimentos	15
III.2.4 - Bloco Função	17
III.2.5 - Comandos	18
III.2.6 - Comandos de entrada e saída	27
III.2.7 - Expressões Aritméticas	33
III.2.8 - Constantes e Representação Interna	35
III.2.9 - Identificadores	37
III.3 - Mecanismo de Extensão	38
III.4 - Formato dos Programas	41
III.5 - Comandos de Controle	43
CAPÍTULO IV - O COMPILADOR	44
CAPÍTULO V - O TRATAMENTO DAS MACROS	49
V.1 - A Definição da Macro	49
V.2 - Exemplo de Definição de Macro	53
V.3 - A Expansão da Macro	54
V.4 - Exemplos de Expansão de Macros	58

Páginas

V.5 - Expansão de Macros em Profundidades 62

V.6 - Limites de Expansão em Profundidade 70

V.7 - Exemplo de Expansão de Macros em Profundidade . 70

CAPÍTULO VI - OUTROS ASPECTOS DO COMPILADOR 75

VI.1 - A Análise Léxica 75

VI.2 - Uso da Tabela de Símbolos 80

VI.2.1 - Acesso à Tabela de Apontadores . . . 82

VI.2.2 - O Vetor de Informações 83

VI.3 - Tratamento de Erros 84

CAPÍTULO VII - CONCLUSÕES E CONSIDERAÇÕES FINAIS . . 87

BIBLIOGRAFIA 91

APENDICE I: PROGRAMA EXEMPLO 93

CAPÍTULO I

INTRODUÇÃO

Macros e procedimentos tem sido usados alternativamente para aumentar as facilidades oferecidas pelas linguagens de programação. A maneira usual de evitar repetição de texto é usar um procedimento, em que os parâmetros formais usados na declaração são substituídos pelos parâmetros reais em tempo de execução. Algumas vezes, entretanto, o tempo e espaço ocupado pelos mecanismos de ligação com o procedimento e passagem de parâmetros, sendo muito grandes em relação ao conjunto de instruções alcançado, justificam o uso, em seu lugar, da macro, que tem um mecanismo menor e mais simples. Tempo e espaço podem ser muito importantes em rotinas de sistemas operacionais, que são usualmente escritas em assembler. Como resultado, grande parte do uso inicial de macros ocorreu relacionado com linguagem assembler. Em 1966, |Cheatham[1]| escreveu um artigo sistematizando idéias sobre extensão de linguagens de programação de alto nível através do uso de macros. Este artigo e um outro, de |Leavenworth[2]|, introduziram o conceito da macro sintática como ferramenta de extensão de linguagens de alto nível.

Uma macro sintática é uma macro que apresenta as seguintes características:

- Formato livre, não estando restrito ao nome da macro seguido da lista de parâmetros separados por vírgulas;

- Possibilita múltiplos níveis de definição, isto é, dentro da definição de uma macro é possível fazer referência a macros já definidas;
- Não existem separadores específicos entre os parâmetros (como por exemplo, a clássica vírgula);
- Os parâmetros formais pertencem a categorias sintáticas como expressão, comando, condição, etc.

Um exemplo de declaração de macro deixará mais claro o conceito de macro sintática:

```
MACRO  FOR <variável> := <expressão1> TO  <expressão2> DO  <comando>
```

declara a macro *FOR* com quatro parâmetros formais, dois da categoria sintática *<expressão>*, um da categoria sintática *<variável>* e outro da categoria *<comando>*.

A declaração segue:

```
DEFINE  BEGIN  LABEL  L;
                  < variável > := <expressão1 >;
                    L:  IF  < variável > LE  <expressão2> THEN
                    BEGIN
                        < comando > ;
                        < variável > :=  < variável > +1;
                          GO  TO  L
                    END
        END
```

definindo o corpo da macro. O corpo da macro é definido usando-se recursos da linguagem original apenas, no exemplo um sub

conjunto do Algol 60 |Naur[3]|, e mais os parâmetros formais. As palavras ressaltadas em itálico são palavras reservadas da linguagem a ser extendida. É possível também uma referência a outra macro já definida, porém esta seria desenvolvida, resultando em elementos da linguagem original, mais os parâmetros. Como consequência, após a chamada de macro e a consequente substituição dos parâmetros formais pelos parâmetros reais, que são cadeias de literais da linguagem original, o resultado é um texto composto de elementos da linguagem original.

Por exemplo a chamada:

```
FOR  X:=1  TO  Y+10  DO  Z:=Z+2
```

é expandida para:

```
BEGIN  LABEL  L;
       X:=1;
       L:  IF  X  LE  Y+10  THEN
       BEGIN
            Z:=Z+2;
            X:=X+1
            GO  TO  L
       END
END
```

As macros sintáticas permitem ao usuário, em tempo de compilação, acrescentar novas estruturas sintáticas à linguagem original, também chamada de linguagem base. Isto é uma vantagem sobre as macros e procedimentos comuns, pois apresenta

maior compatibilidade com a idéia de extensão de linguagens,e, por serem mais flexíveis, as extensões através de macro sintática podem incluir estruturas de outras linguagens de programação. Com isto, dentro de certos limites, pode-se processar um programa de outra linguagem, mesmo sem um compilador para esta linguagem. O limite, é claro, está na capacidade de se definir a semântica de estruturas de outras linguagens através de macros, e através de linguagem base.

O presente trabalho, motivado por uma sugestão de |Aho[4]|, trata de desenvolver um compilador para uma linguagem cujo escopo pode ser aumentado com o auxílio de macros sintáticas.Observou-se que havia material suficiente para o desenvolvimento de duas teses, e como resultado o compilador foi dividido em duas partes. A presente tese contém o mecanismo de extensão (a definição e expansão de macros) e a análise léxica. A outra parte está contida na tese de |Zakimi[5]|, e contém a análise sintática e a geração de código.

Optou-se por uma linguagem base muito reduzida, que requer um compilador simples, podendo ter sua capacidade aumentada através do mecanismo de extensão. A linguagem base,apesar de pequena, apresenta recursos de programação similares aos de algumas linguagens bastantes difundidas (Algol, por exemplo).

Um compilador pequeno é útil no caso de minicomputadores, com pouca memória disponível em circuito. O compilador foi implementado no MITRA-15 do Laboratório de Automação e Simulação de Sistemas da COPPE-UFRJ, como parte do projeto de um Laboratório de Ensino de Computação. O MITRA-15 é um minicomputador, e também apresentava carência de uma linguagem de alto nível com estrutura de blocos e com alocação dinânica de memória.

CAPÍTULO II

RECURSOS DISPONÍVEIS

II.1 - Equipamento

O compilador foi desenvolvido no MITRA-15 |Schwarz[6]|, que é um computador de tempo real, com capacidade de 32 Kbytes extensível a 64 Kbytes.

A memória principal do MITRA-15 é 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 e dupla palavra.

Os periféricos disponíveis são:

a) Uma leitora de cartões

b) Um teletipo de 72 caracteres, usado como console

c) Quatro linhas assíncronas, de velocidade máxima de 1200 bauds, que podem ser divididas entre:
   - três teletipos, de velocidade 110 bauds
   - dois vídeos, um de 600 bauds, outro de 9600 bauds

d) Disco móvel e disco fixo, de capacidade de 5 megabytes.

O compilador usa a leitora como unidade de entrada e o teletipo principal (console) como unidade de saída.

II.2 - Software básico

- Monitor de base, de tempo real e tempo real em disco |MITRA-15[7]|.

- Assembler
- LP15E
- Fortran IV
- Basic

O compilador foi escrito na linguagem LP15E |MITRA-15[8]|. Pode usar qualquer monitor que tenha todas as rotinas de ponto flutuante. Atualmente o menor monitor (10644 bytes) nestas condições é o MCCOPF.

CAPÍTULO III

DESCRIÇÃO DA LINGUAGEM

Grande parte do material exposto neste capítulo pode ser encontrado na tese de |Zakimi[5]|, publicada no princípio de 1980.

III.1 - Estrutura da Linguagem

EXPAND é uma linguagem de alto nível, com estrutura de blocos e alocação dinâmica de memória, que pode ser estendida através do uso de 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.

Todo programa EXPAND está formado de:

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

As declarações das macros 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 reais. Cada chamada de macro será expandida à definição da macro, com a correspondente substituição dos parâmetros formais pelos parâmetros reais.

O corpo do programa é um bloco; isto é, um conjunto de declarações seguido de um conjunto de comandos encerrados entre 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 parâmetro é por referência |Gries[9]|. Na chamada os parâmetros formais são carregados com o endereço dos parâmetros reais.

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 de um conjunto de comandos, tudo encerrado entre 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.

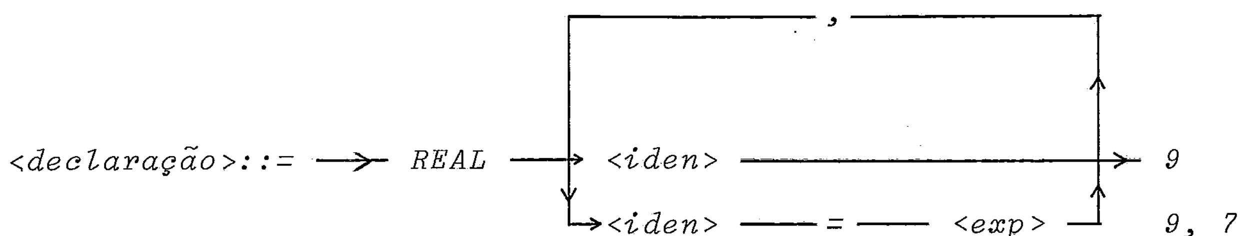
Os comandos da linguagem EXPAND são apresentados na sua versão mais simples e menos poderosa (por exemplo IF-THEN e não IF-THEN-ELSE) e não há muitos tipos de comandos. Isto foi feito com o objetivo de definir uma linguagem necessária e suficiente, que seria a linguagem de base, e que poderia ter seu

conjunto de comandos aumentado pelo uso das macros sintáticas.

III.2 - Especificação da Linguagem

Notação usada para descrever a sintaxe:

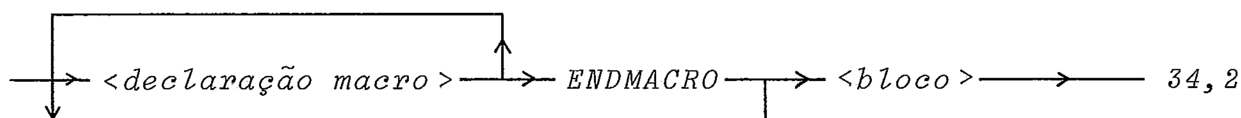
A descrição das categorias sintáticas da linguagem utiliza uma variação da forma normalizada de Backus. Sua principal característica é diminuir o número de categorias sintáticas. Exemplo:



- 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) categoria(s) sintática(s) referenciada(s) naquela linha.

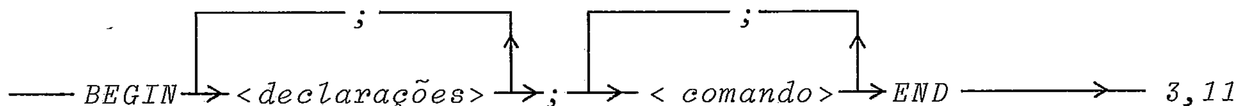
III.2.1 - Programa

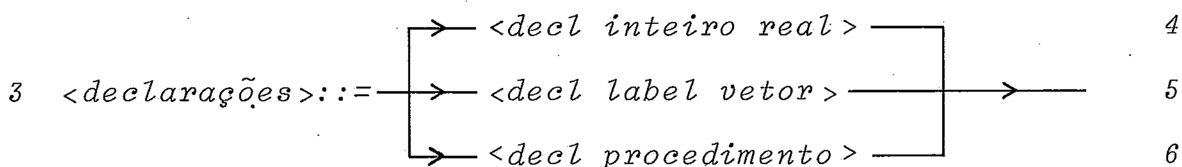
1 <programa>::=



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

2 <bloco>::=





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.

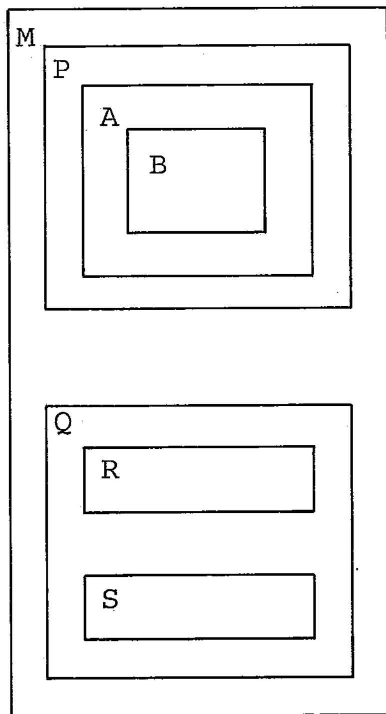
As ações são descritas pelos <u>comandos</u> e os dados são descritos pelas <u>declarações</u>.

Um bloco é definido como um conjunto de declarações seguido de um conjunto de comandos, tudo encerrado entre <u>BEGIN</u> e <u>END</u>. 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 o bloco definido nele terá nível 1.

Em geral um bloco definido no nível <u>i</u> será de nível

i+1. A figura III.1 ilustra uma estrutura de blocos.



Bloco	nível
M	0
P, Q	1
A, R, S	2
B	3

Figura III.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.

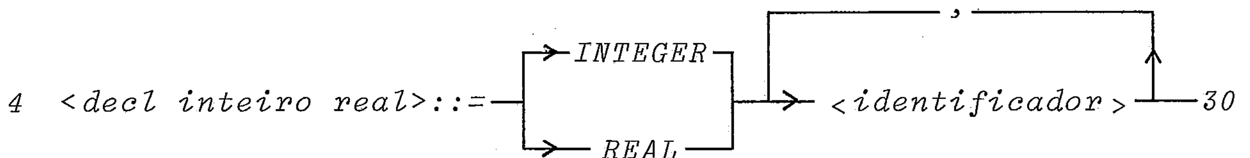
Todas as variáveis, rótulos e procedimentos devem ser declarados.

As palavras reservadas 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).

III.2.2 - Declarações

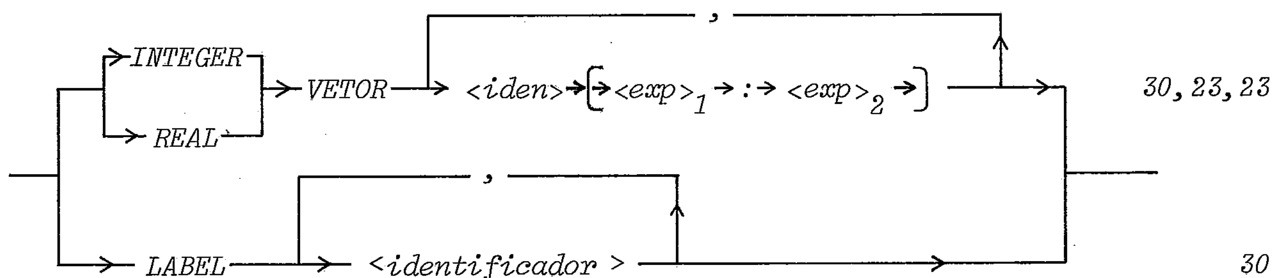


A declaração com INTEGER é usada para definir identificadores que representarão valores inteiros. (Cada variável inteira ocupa 2 bytes)

Para declarar identificadores que representem valores

reais é usada a palavra REAL. (Cada variável real ocupará 4 bytes).

5 *<decl label vetor> ::=*



Declaração LABEL

Os comandos de um programa podem receber nomes, para poder fazer referência a eles. 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

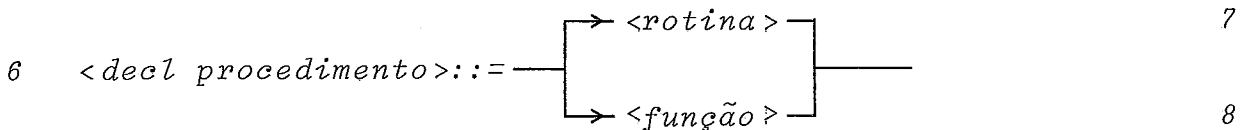
$<exp>_2$ determina o valor do limite superior do arranjo

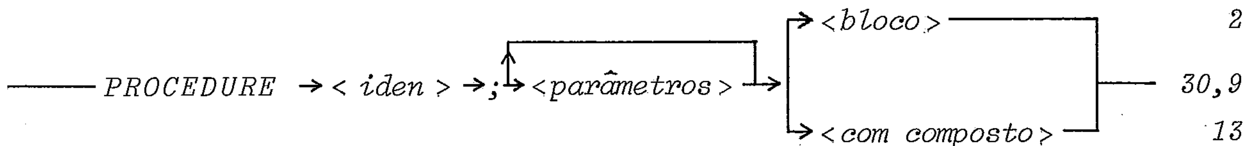
$<exp>_1$ e $<exp>_2$ devem ser inteiras e os identificadores porventura nelas usados devem estar declarados nos blocos

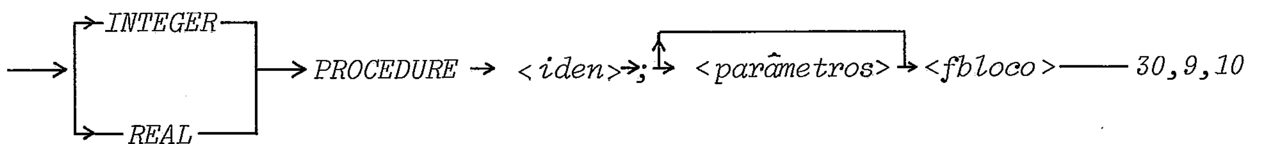
mais externos ao bloco da declaração de arranjo, isto é, se um arranjo é declarado no nível i, os identificadores, usados nas expressões que determinam seus limites, devem estar declarados nos blocos i-1, i-2,...,0 , nunca no próprio bloco de nível i.

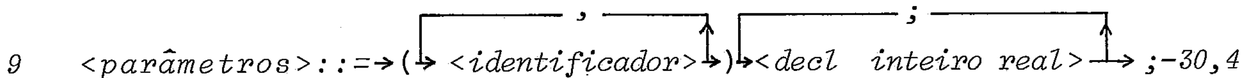
Os componentes dos arranjos podem ser do tipo REAL (4 bytes por posição) ou INTEGER (2 bytes por posição).

III.2.3 - Declaração de Procedimentos









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 referência, isto é, no momento da chamada os parâmetros formais são carregados com os endereços dos parâmetros reais.

Deve existir uma correspondencia bi-univoca, em número e tipo, entre parâmetros formais e reais.

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.

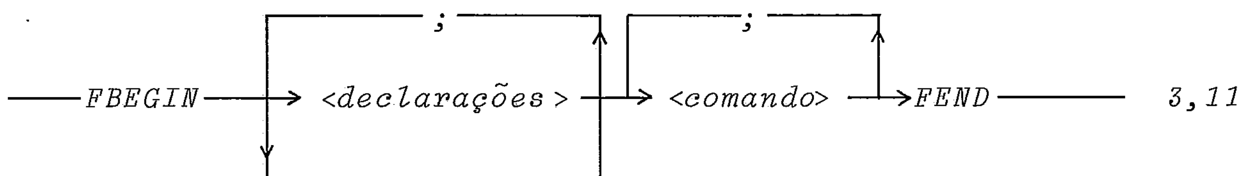
São ativados por chamadas que fazem parte de uma expressão aritmética (<fchamada>).

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.

III.2.4 - Bloco Função

10 <fbloco>::=



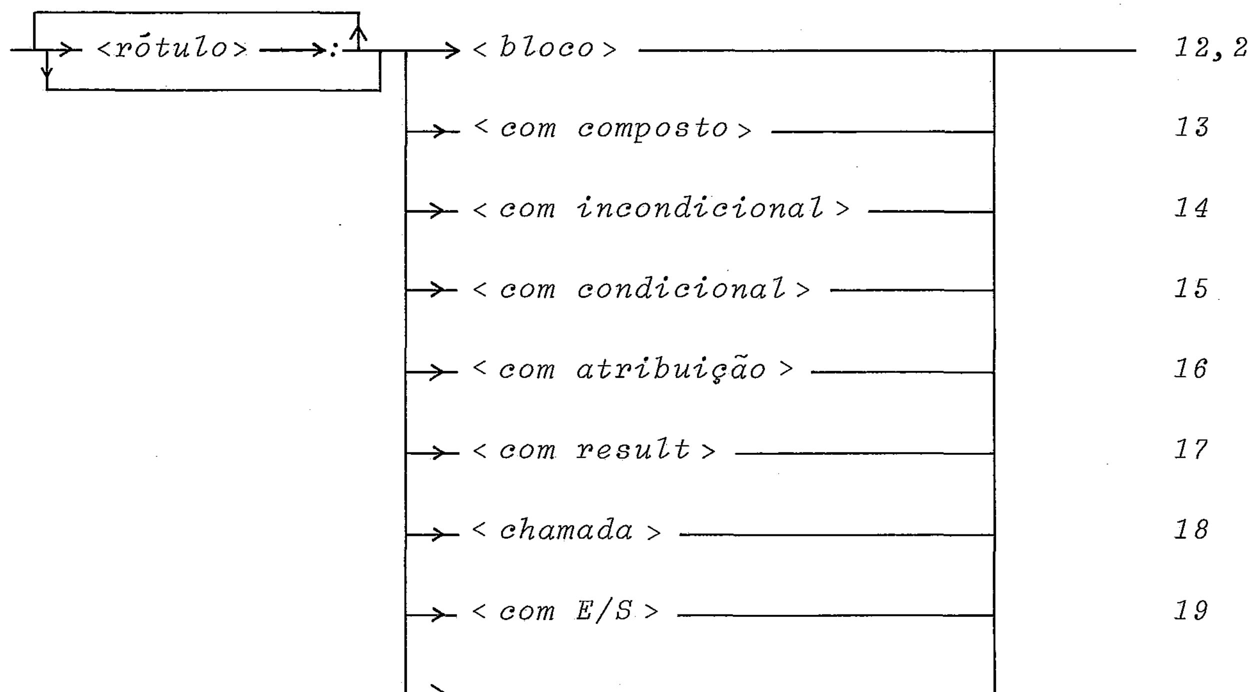
Um bloco é um conjunto de declarações (opcional) e um conjunto de comando, tudo encerrado entre FBEGIN e FEND.

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.

III.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 III.2 mostra dois exemplos:

```
BEGIN
 LABEL  X,Y,Z;
 ______
 ______
 ______
 X:  ______
     ______
 Y:Z: ___
END
```

correto

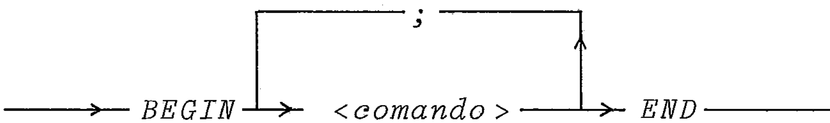
```
BEGIN
 LABEL  X;
  BEGIN
  ______
  ______
  ______
  X:  ______
      ______
      ______
  END
END
```

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

Figura III.2

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

13 <com composto>::=

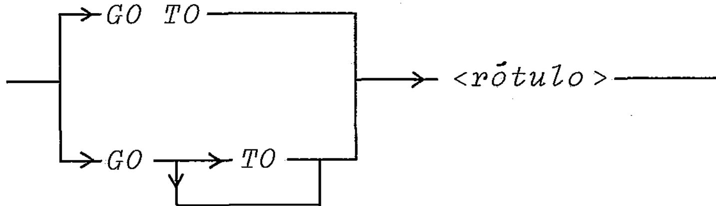


11

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 se-

rem tratados como um só comando.

14 <com incondicional>::=



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. III.3).

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

Exemplo 1:

```
BEGIN
LABEL A,B,C;
  GO C;
A: ___
   ___
   ___
  GO B;
  C :  BEGIN INTEGER X;                                   válido

       ___
       ___
       ___
       GO B;

       ___
       ___
       ___
       GO A;

       ___
       ___
       ___
       END;

 ___
 ___
 ___
B: ___
 GO A
END
```

Figura III.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 III.4

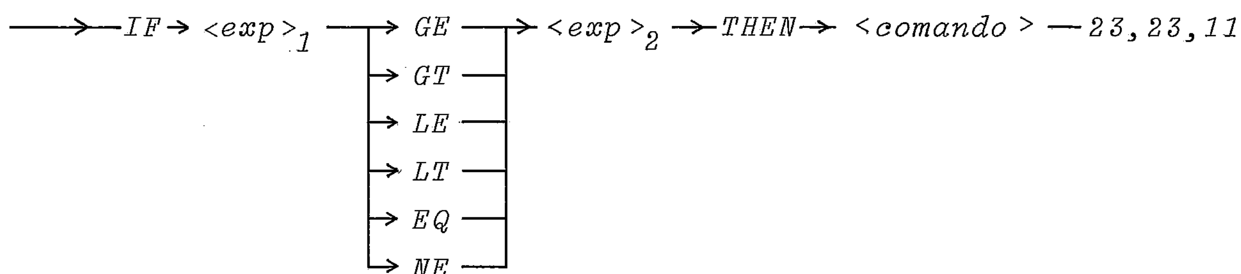
Exemplo 3:

```
BEGIN LABEL A;
    BEGIN

     ___
     ___
     ___
    GO A;
       BEGIN LABEL A;

        ___
        ___
        ___
       A: ___
          ___
          ___
       END
     END
  A: ___
     ___
     ___
  END
```

O GO TO é executado desviando o controle ao LABEL A do bloco mais externo.

Figura III.5

15 <com condicional>::=



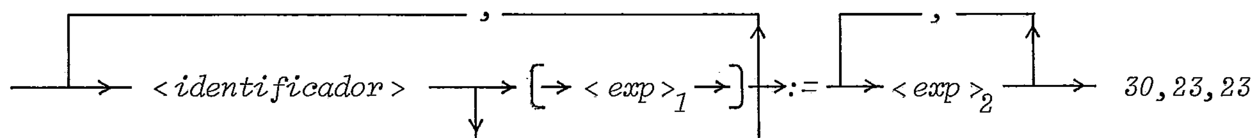
O comando IF especifica que o <comando> será executado só se $\langle exp\rangle_1$ e $\langle exp\rangle_2$ satisfazem a relação especificada, caso contrário, será executado o comando seguinte ao IF.

Relações válidas	$\langle exp\rangle_1$ tem que ser ... que $\langle exp\rangle_2$
GE	maior ou igual
GT	maior
LE	menor ou 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 pode não ser 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

atribuídos a uma ou várias variáveis.

<u>:=</u> 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 d o 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 e m paralelo. (Fig. III.6)

<u>Exemplo 1:</u>

I:=0 — a variável I recebe o valor 0

<u>Exemplo 2:</u>

A,B:=B,A — especifica uma troca de valores de A e B equivalente a: T1:=A; A:=B; B:=T1

<u>Exemplo 3:</u>

I:=2

I,A[I]:= 0,20 — ao fim da atribuição:

I é Zero

A[0] não muda o valor

A[2] é 20

<u>Figura III.6</u>

17 <com result>::=

——→*RESULT* ——→ *<exp>* ——→ *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. III.7) .

Caso o <fbloco> não seja o corpo de um procedimento, o comando RESULT define o valor resultante do <fbloco> cujo o 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. III.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
```

Figura III.7

O valor retornado pelo procedimento será:

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

= 1 se A ≤ B

```
A:=1 +  FBEGIN
        LABEL FIM, SEGUE;
        REAL SOMA; INTEGER X;
        IF (I EQ 0) 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
        FIM:
       FEND
```

O valor de fbloco será inteiro e

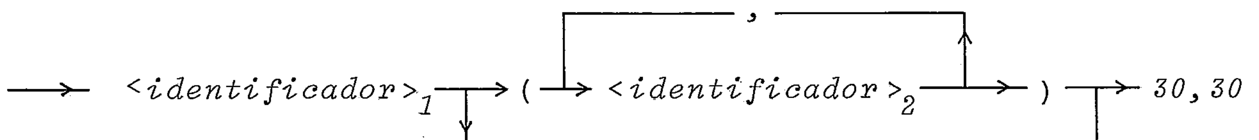
= 1 se I = 0

= ⌊SOMA * 0.5⌋ se I ≠ 0

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 III.8

18 <chamada>::=



Especifica a ativação do procedimento <identificador> $_1$

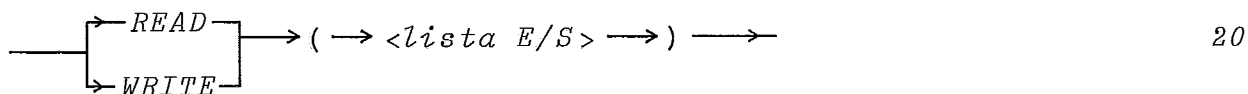
que deve ser necessariamente sem tipo. A chamada só é válida dentro do escopo do procedimento a ser usado.

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

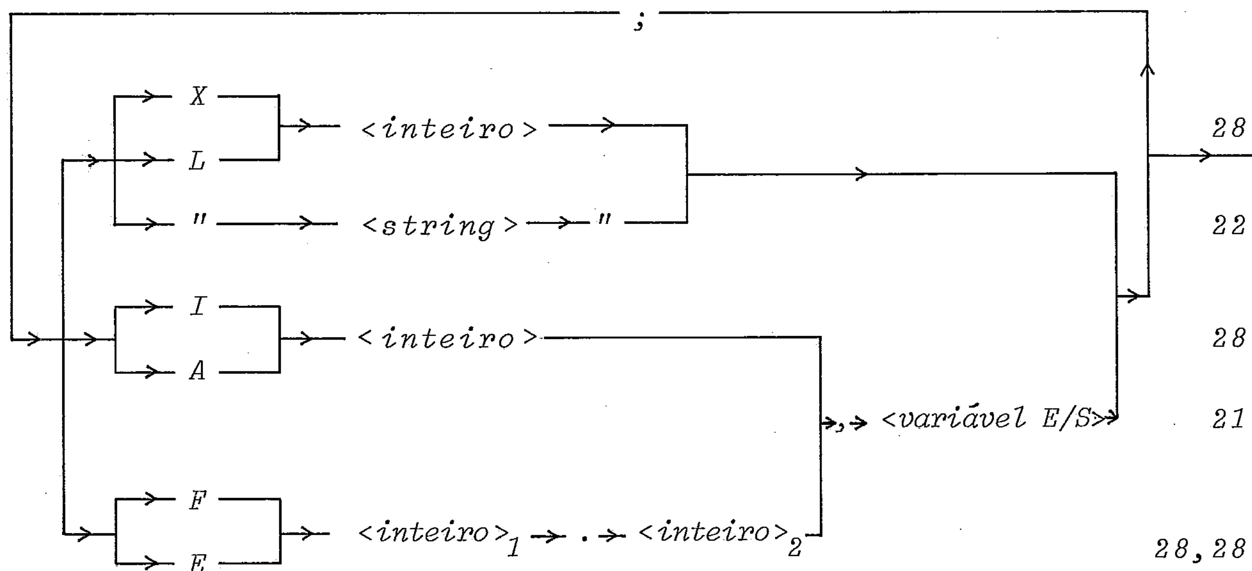
Os parâmetros reais da chamada do procedimento, devem concordar em número e tipo com os parâmetros formais da declaração do procedimento.

III.2.6 - Comandos de Entrada e Saída

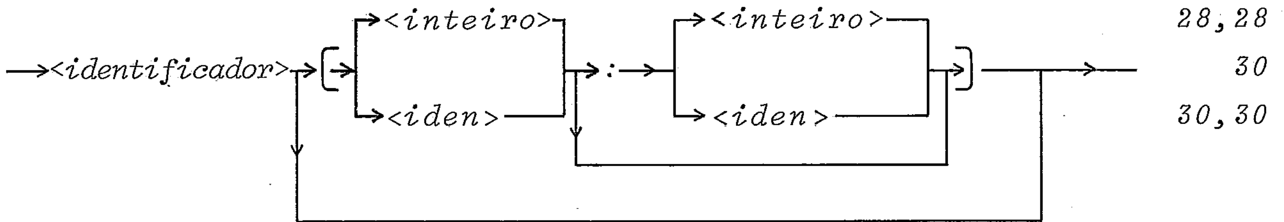
19 <com E/S>::=



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 um conversão. Se são 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. O 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 asteriscos.

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

Formato Fw.d

Usado para leitura/impressão de números reais. O w especifica o comprimento total do campo incluindo sinal, ponto fracionário, dígitos inteiros e dígitos fracionários. O 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 arrendondado 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.

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

O 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 algarismos e o sinal é opcional, tanto para o número quanto para o expoente.

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

Saída

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

s_1i.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

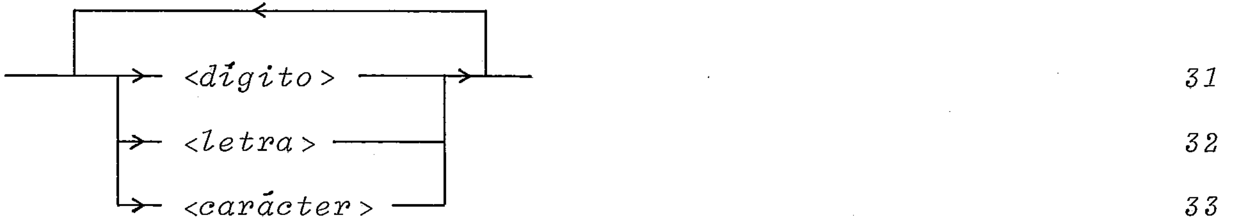
Usado para ignorar w colunas no cartão, ou deixar w espaços brancos na impressão.

Formato Lw

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

Formato "<string>"

22 <string>::=



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 são 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]);

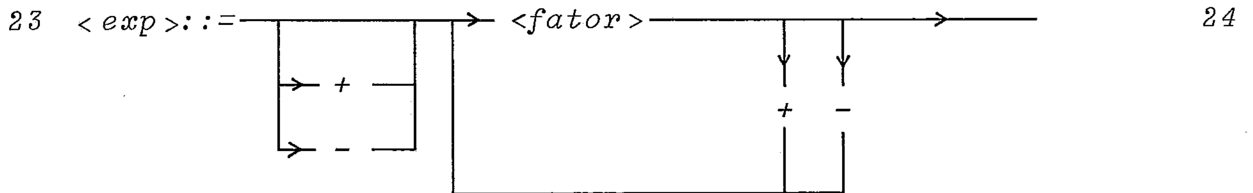
O valor A[16] será lido no segundo cartão.

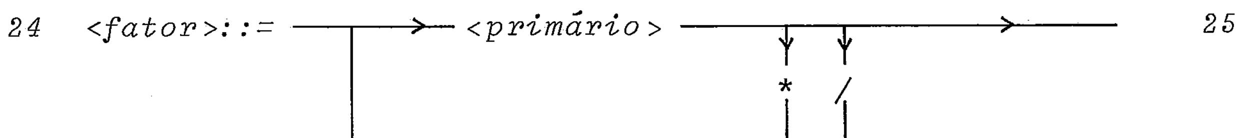
WRITE(X70;"TITULO");

TITULO será impresso na segunda linha.

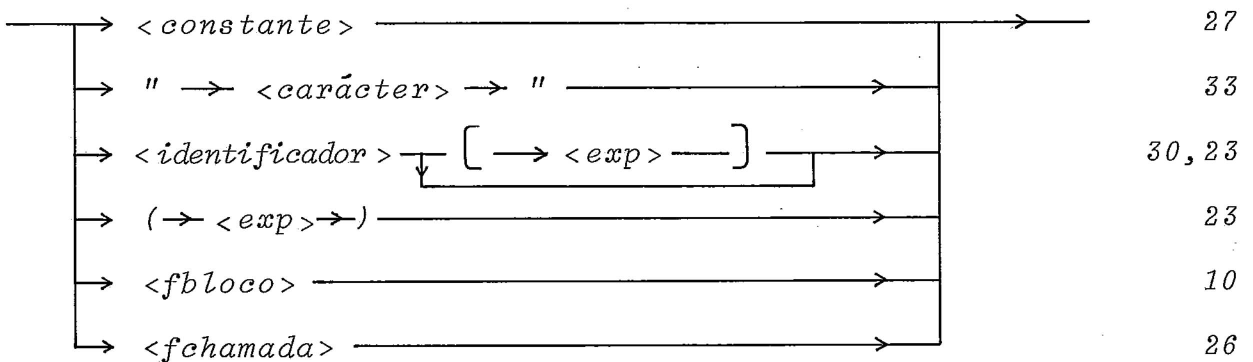
Cada WRITE/READ indica o começo de um novo registro (linha/cartão).

III.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 "<carácter>" é considerado como uma constante inteira.

<u>Exemplo</u>:

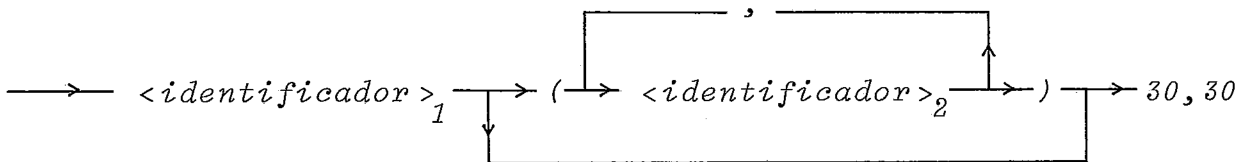
A:= "Z"

WRITE (A1,A);

imprime o carácter <u>Z</u>.

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

26 <fchamada>

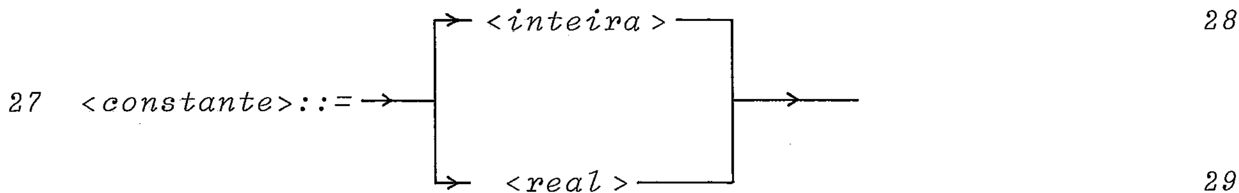


Especifica a ativação do procedimento <identificador>$_1$, que deve ser necessàriamente com tipo. A chamada só é válida dentro do escopo do procedimento a ser usado.

<identificador>$_2$ representam as variáveis (parâmetros reais) 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 reais da chamada do procedimento devem concordar em número e tipo com os parâmetros formais da declaração.

III.2.8 - Constantes e Representação Interna



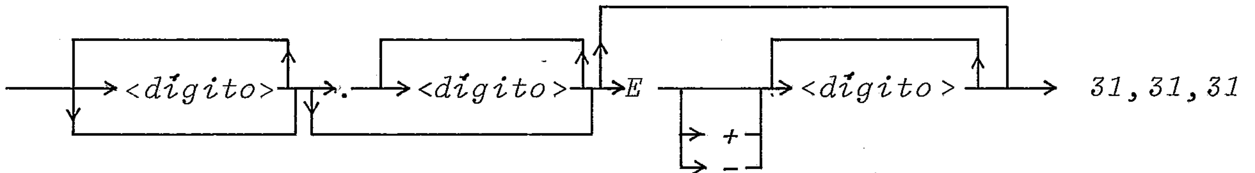
28 <inteira>::=

A constante inteira é armazenada em 2 bytes. O bit 0 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.

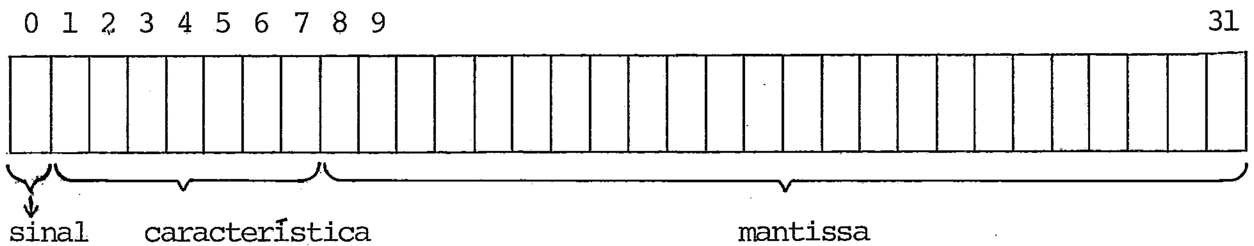
Limites: -32768 a +32767

29 <real>::=



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 mantissa 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 seguido de uma parte fracionária ou todas as duas. O expoente não pode aparecer só, deve ser precedido de uma parte inteira ou de uma parte fracionária, ou todas as duas.

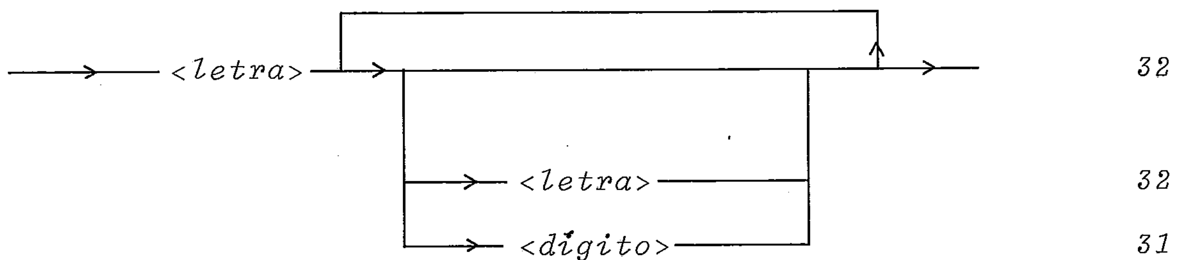
Limites:

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

É fornecida uma precisão de 7 dígitos fracionários.

III.2.9 - Identificadores

30 <iden>::=

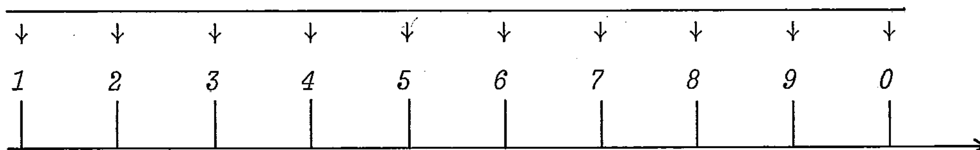


Os identificadores são comumente usados para representar valores numéricos e são denominados variáveis.

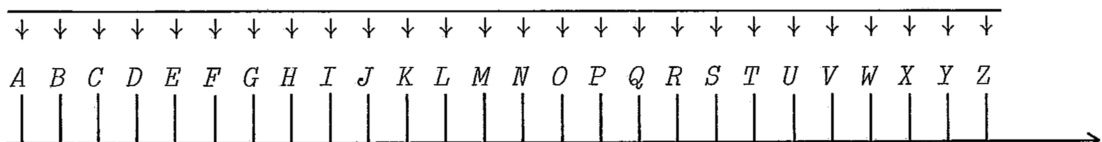
Um identificador deve sempre ser declarado antes de seu uso e o valor inicial não está determinado. É responsabilidade do programador atribuir-lhe um valor inicial.

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

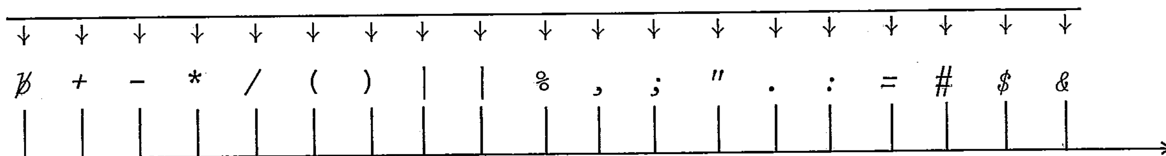
31 <dígito>::=



32 <letra>::=



33 <carácter>::=



III.3 - Mecanismo de Extensão

34 <decl de macro>::=

——→——MACRO—→—<estrutura da macro>—→—DEFINE—→—<corpo da macro>—→— 35, 36

Extensão do diagrama 11

11 <comando>::=——→——<chamada de macro>——→—— 37

Extensão do diagrama 25

25 <primário>::=——→——<chamada de macro>——→—— 37

O mecanismo de extensão permite que novas formas de comando e primário sejam incluídas na linguagem usando a declaração de macro sintática (diagrama 34).

A <estrutura da macro> define a forma da nova construção sintática e o <corpo da macro>, 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 <identificador>, e a ele estará associado o nome da macro. As metavariáveis são os parâmetros da macro, e pertencem a uma das quatro categorias sintáticas: <variável>, <expressão>, <condição> e <comando>.

Nomenclatura para as metavariáveis:

$VAR ou $VARn para variável

$EXP ou $EXPn para expressão

$COND ou $CONDn para condição

$STAT ou $STATn para comando

com n = 1,2,3,..., para possibilitar o uso de mais de uma metavariável do mesmo tipo. Cada metavariável distinta deve aparecer apenas uma vez.

Os símbolos terminais podem ser palavras reservadas, identificadores, caracteres, constantes. Os identificadores que não forem também palavras reservadas serão consideradas palavras reservadas da macro.

36 <corpo da macro>

É também uma cadeia de metavariáveis e símbolos terminais. Cada metavariável no <corpo da macro> deve aparecer na <estrutura da macro>, embora o oposto não precise ser verdade. Cada metavariável distinta pode aparecer mais de uma vez no <corpo da macro>.

Os símbolos terminais podem ser palavras reservadas, identificadores, caracteres, constantes. Os identificadores que não forem também palavras reservadas não terão tratamento especial. Entretanto, é permitido começar os identificadores de um <corpo de macro> com o carácter "#".

37 <chamada de macro>

É uma cadeia de palavras reservadas, identificadores, caracteres, constantes. O primeiro símbolo deve ser um identificador associado a um nome de macro. Devem aparecer na cadeia os mesmos símbolos que existem na <estrutura da macro> que tem o referido nome, e na mesma ordem. Os símbolos restantes da cadeia

devem estar reunidos em pedaços de cadeia, denominados parâmetros reais, em uma maneira tal que, ao serem substituídos (os parâmetros reais) pelas metavariáveis da estrutura da macro , obtenha-se, da cadeia, a <estrutura da macro> . Além disso, se forem substituídas as metavariáveis do <corpo da macro> pelos parâmetros reais correspondentes da <chamada de macro>, deve ser obtida uma cadeia sintáticamente correta na linguagem base. Cabe ao usuário do compilador garantir que isto ocorra, assim como é de sua responsabilidade que o fim dos parâmetros reais seja reconhecível pelo compilador, conforme comentário no Capítulo V.

Exemplo de declaração de macro:

```
MACRO WHILE $COND DO $STAT
DEFINE BEGIN LABEL #L;
              #L: IF $COND THEN
               BEGIN
                   $STAT;
                   GOTO #L
               END
      END
```

A <estrutura da macro> constitui-se de WHILE $COND DO $STAT. O nome da macro é WHILE. O identificador DO será considerado como uma palavra reservada de macro. As metavariáveis são $COND e $STAT. O <corpo da macro> é tudo que vem após DEFINE. Note-se que cada metavariável do <corpo da macro> aparece na <estrutura da macro>.

Exemplo de chamada de macro:

WHILE I GT 10 DO I:=I+1

A correspondencia entre os parâmetros formais e os parâmetros reais é:

parâmetros formais	parâmetros reais
$COND	I GT 10
$STAT	I:=I+1

O uso do mecanismo de extensão implica, a cada <chamada de macro>, na substituição da chamada pelo <corpo da macro> correspondente, tendo neste último sidos substituídos os parâmetros formais pelos parâmetros reais.

A partir do exemplo de chamada acima, seria alcançada a seguinte cadeia:

```
BEGIN LABEL #L;
      #L: IF I GT 10 THEN
      BEGIN
            I:=I+1;
            GOTO #L
      END
END
```

Pode-se notar que a cadeia gerada consiste em um bloco, e portanto será reconhecida sintáticamente através da extensão do diagrama 11 referida anteriormente.

III.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, os cartões não precisam 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 "&" que indica final do campo útil do cartão.

O usuário do compilador tem a opção de sustar a listagem dos programas fonte e fonte expandido, inserindo a palavra reservada NOLIST antes do programa. Neste caso, serão listados apenas as mensagens de erro porventura existentes.

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

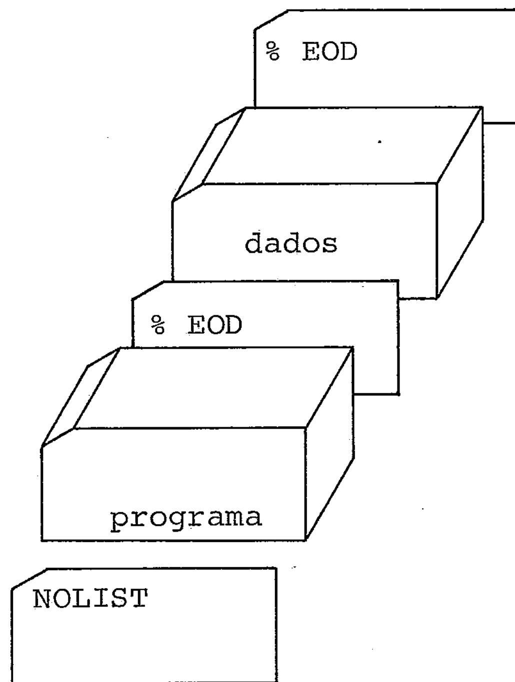


Figura III.9

III.5 - Comandos de Controle

% C/EXPAND/

(chama o compilador)

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

% C/LINKDX/ (nome do porgrama), SL, UL

(Chama o link-editor)

%L

(Carrega o programa na memória)

%R

(Ordena a execução do programa)

CAPÍTULO IV

O COMPILADOR

O compilador EXPAND está dividido em duas partes:

1) Tratamento de macros e análise léxica

2) Análise sintática e geração de código

Tendo sido cada uma das partes alvo de um trabalho independente, optou-se por fazer um compilador de dois passos. Assim, o resultado da primeira parte é gravado em disco, e lido para execução da segunda parte. Um pequeno programa principal executa o trabalho de ligação entre as partes.

O primeiro passo do compilador está todo contido neste trabalho. Suas principais funções são:

- Análise das declarações das macros sintáticas
- Expansão das macros chamadas pelo programa, substituindo os parâmetros formais pelos parâmetros reais
- Análise léxica

Durante cada uma destas atividades há um tratamento para erros, podendo inclusive ser interrompida a compilação. Como resultado do primeiro passo, será gerado um programa codificado em tokens |Gries[9]|, que será gravado em disco. Este programa, se correto, pertence à linguagem base, podendo ser obtido da aplicação das produções da gramática definido em III.2 .

O primeiro passo do compilador pode ser descrito pelo programa da Figura IV.1, escrito em linguagem similar a ALGOL.

```
rotina SCANNER/MACRO;
BEGIN
    LER(token);
    IF token=nolist THEN                        % opção de não listar
    BEGIN
        NAOLISTAR;
        LER(token)
    END;
    IF token=macro THEN
    BEGIN
        MACRODEF;
        LER(token)
    END;
    WHILE token NE %eod DO
    BEGIN
        IF class(token)=nomemacro
           THEN MACROCALL
           ELSE GRAVA(token);
        LER(token)
    END
END
```

Figura IV.1

O segundo passo do compilador pode ser encontrado no trabalho de |Zakimi[5]|, a menos do módulo de leitura de disco, que faz parte do presente trabalho, e cuja função é, tôda vez que solicitado, ler um token do disco e passá-lo ao analisador

sintático. Este módulo imprime, também, o programa fonte com as alterações causadas pela expansão de macros.

As funções do segundo passo são as seguintes:

- Análise sintática, verificando a conformidade sintática e semântica do programa expandido às regras de linguagem definidas em III.2 .
- Geração de código.

A análise sintática tem um tratamento de erros que pode impedir a geração de código. O resultado do segundo passo é um programa BT (Binary Translatable) |Mitra[10]|, que contém, além do código objeto, informações dirigidas ao link-editor, como definição de zonas de dados (tanto CDS com LDS), definição de referências para frente, chamada a módulos externos, etc. Após o fim da compilação, o programa BT servirá de entrada para o link-editor do MITRA-15.

A Figura IV.2 mostra um esquema do compilador.

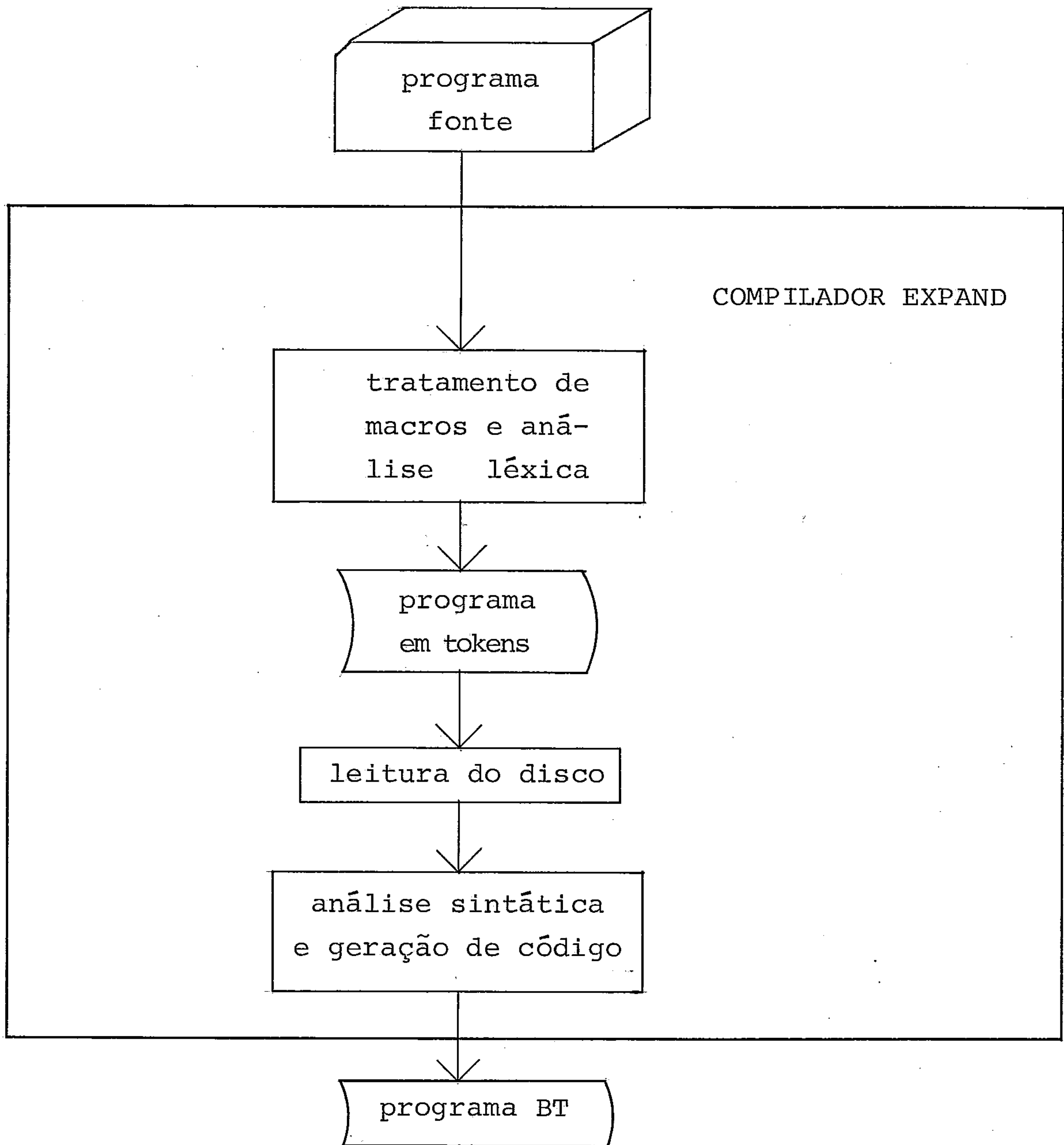


Figura IV.2

Cada um dos trabalhos gerou um programa em LP-15; cada programa tem uma zona de dados globais. Como algumas seções, compiladas separadamente, já estavam com o limite máximo de dados, foi constatada a impossibilidade de serem unidas as duas zonas de dados globais. Como resultado, o programa principal, que começa colocando em atividade a parte de análise léxica e tratamento de macros, tem a zona de dados referente a esta parte. Após esta parte, é chamada uma rotina cuja única tarefa é re-inicializar a zona de dados para a parte da análise sintática e geração de código. Após isto o programa principal chama esta última parte.

CAPÍTULO V

O TRATAMENTO DAS MACROS

As informações referentes às macros estão dispersas em quatro vetores: S, S_p, S_1 e S_2. Nestes vetores teremos informações de características variadas, tais como apontadores, representação interna de tokens, e informações referentes ao estado de desenvolvimento de expansões de macros. Portanto, muita atenção deve ser dedicada ao estudo destes vetores.

Existe uma rotina para tratar a definição das macros, e outra para tratar a expansão das macros.

V.1 - A Definição da Macro

A rotina de definição de macro trabalha apenas com a pilha S. Esta pilha será gravada em disco, durante a execução desta rotina ou da rotina de expansão de macros. Se o corpo da macro fizer menção a uma outra macro já definida, será chamada a rotina de expansão de macros. A definição da macro deve ser feita antes do aparecimento da palavra reservada ENDMACRO.

A rotina de definição será chamada somente se a primeira palavra do programa for a palavra reservada MACRO; a partir daí ela permanece em atividade, podendo, após o fim de uma definição, retornar ao início, ao ser encontrada outra vez a palavra MACRO (significando outra macro a ser definida), ou ser encerrada, ao ser encontrada a palavra ENDMACRO.

Esta rotina espera encontrar a seguinte estrutura:

MACRO < nome da macro > <estrutura da macro > *DEFINE* <corpo da macro>

O nome da macro deve ser um identificador, e deve ter o tamanho menor que o de um identificador comum. A razão para isto ficará clara mais adiante. Ao ser encontrado, o nome da macro é inserido na Tabela de Símbolos com a representação interna de nome de macro. Assim, qualquer referência futura a este nome, se correta, desencadeará a chamada à rotina de expansão de macros. É colocado, com o nome da macro, um apontador para a posição do vetor S onde começa a definição da macro que tem este nome.

Em seguida, a representação interna dos tokens que compõem a estrutura da macro é analisada e inserida no vetor S.

Ao ser encontrado um parâmetro formal, é feita uma pesquisa para verificar se ele se encaixa em uma das categorias sintáticas usadas neste compilador, e, estando tudo correto, ele é inserido na Tabela de Símbolos, junto com o número de ordem de sua aparição entre os parâmetros.

Na estrutura de macro os identificadores encontrados são considerados palavras reservadas de macro, podendo funcionar, entre outras coisas, como separadores de parâmetros formais.

Ao ser encontrada a palavra DEFINE, uma marca é inserida no vetor S, caracterizando o fim da estrutura da macro. Na implementação foi usado o número 99 para esta marca, mas nos exemplos aqui fornecidos ela estará representada por "§", nesta ou em qualquer outra situação em que se necessite de uma marca.

Nesta altura é colocado na tabela de símbolos, junto ao nome da macro, um apontador para o começo das próximas posições do vetor S, onde será inserida a representação interna dos tokens que compõem o corpo da macro; e o número de parâmetros, contados durante a análise da estrutura da macro.

Em seguida, o corpo da macro é analisado, se necessário aumentado, e sua representação interna colocada em S.

Neste momento, apenas, é correto o aparecimento do nome de outra macro (dentro do corpo da primeira), caracterizando uma chamada de macro. O resultado seria que o corpo da outra macro seria encaixado no corpo da primeira, em substituição àquela chamada. Assim alterado, o corpo da primeira macro é colocado em S. Nesta implementação optou-se por não se permitir o uso da própria macro em seu corpo, ou seja, as macros não podem ser recursivas.

Quando um parâmetro formal é encontrado no corpo da macro, apenas o número de ordem de sua aparição, entre os parâmetros, é colocado em S, pois esta é a única característica que será usada posteriormente.

Após a representação do corpo da macro, será colocada em S outra marca, caracterizando o fim do corpo no vetor S.

A rotina de definição de macro pode ser descrita pelo algoritmo V.1 .

```
rotina MACRODEF;

BEGIN
    LER(token);
    INSERE-NA-TABSIMB(token, nomemacro);
    LER(token);
    WHILE token NE define DO
    BEGIN
        INSERE-EM-S(token);
        IF token EQ param
           THEN INSERE-NA-TABSIMB(token, param);
        LER(token)
    END;
    INSERE-EM-S(marca);
    LER(token);
    WHILE token NE macro AND token NE endmacro DO
    BEGIN
        IF token EQ nomemacro
           THEN MACROCALL
           ELSE BEGIN
                     IF token EQ param
                        THEN PESQUISA-NA-TABSIMB(token, ordemparam);
                        INSERE-EM-S(token)
                END
    END;
    INSERE-EM-S(marca)
END
```

Algoritmo V.1

V.2 - Exemplo de Definição de Macro

Sendo usada a macro definida na Figura V.1, teria sido desenvolvida a representação interna mostrada na Figura V.2 .

```
MACRO WHILE $COND DO $STAT
DEFINE BEGIN LABEL #L;
            #L: IF $COND THEN
                  BEGIN
                     $STAT;
                     GOTO #L
                  END
     END
```

Figura V.1

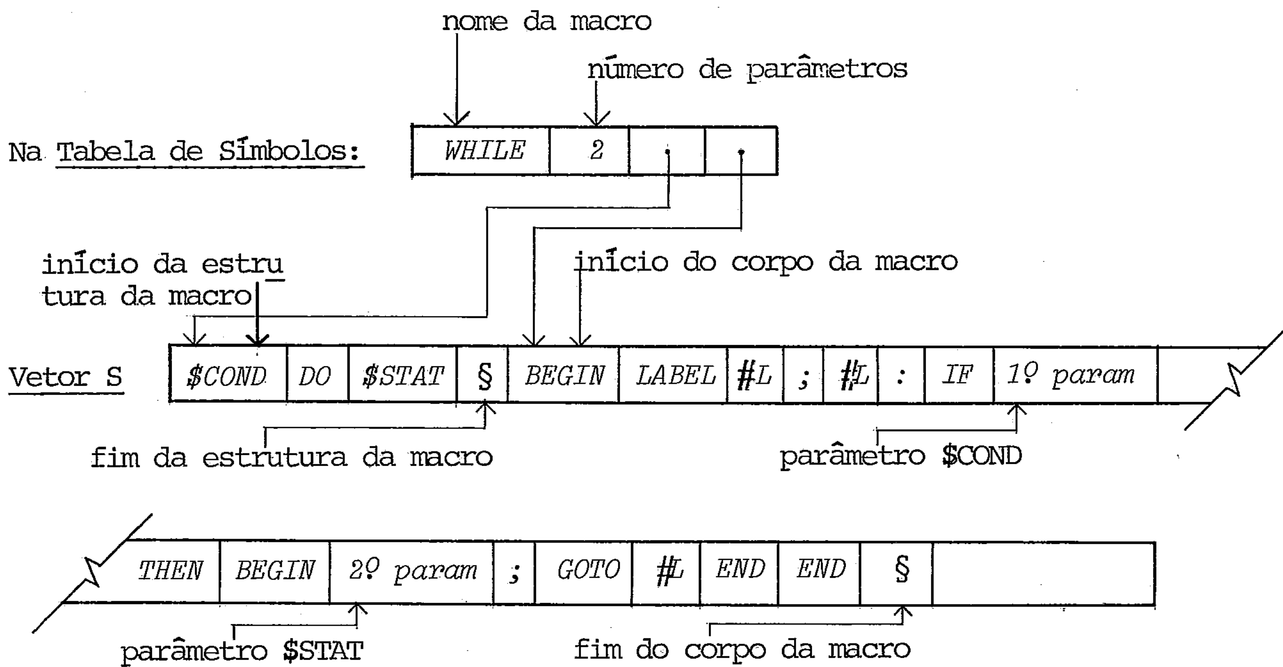


Figura V.2

A Tabela de Símbolos será abordada minuciosamente mais adiante. No lugar de cada símbolo ($COND, DO, $STAT, BEGIN, etc.), estaria a sua representação interna. Foi escolhido mostrar o próprio símbolo, como na Figura V.2, para permitir uma melhor visualização.

V.3 - A Expansão da Macro

Ao ser encontrado um identificador, este será pesquisado junto à Tabela de Símbolos. Se for constatado que ele é o nome de uma macro, será chamada a rotina de expansão de macros. É possível se encontrar uma chamada de macro durante o programa, depois da palavra ENDMACRO, ou em uma definição de macro, no corpo da macro. Em uma chamada de macro, é possível que se encontrem outras chamadas de macro, em seus parâmetros ou nos parâmetros das novas chamadas. Neste caso a rotina de expansão trabalhará com todas as macros ao mesmo tempo, substituindo as chamadas pelos corpos, e só será desativada ao expandir todas as macros. Após isto, pode ser chamada de novo, ao ser encontrado novo nome de macro (a rotina de definição de macro, ao contrário, é chamada no máximo uma vez). Só serão dados detalhes a respeito de procedimentos para expansão de macros em profundidade no Capítulo V.5.

A rotina de expansão de macros é dividida lógicamente em duas etapas distintas e independentes. Na primeira, todas as chamadas de macro, a primeira e outras que porventura apareçam em seus parâmetros, são analisadas. É verificado se es

tão de acôrdo com as estruturas de macro definidas anteriormente. Seus parâmetros são analisados e guardados. Na segunda etapa estes parâmetros são usados para se obter o corpo da macro expandida.

Na primeira etapa, obtém-se, a partir da Tabela de Símbolos, junto ao nome da macro, o número de parâmetros e a posição da estrutura da macro e (para uso na segunda etapa) do corpo da macro no vetor S. A seguir lê-se em paralelo a estrutura da macro, em S, e o restante da chamada. As palavras reservadas, os números e caracteres especiais encontrados em S devem ser encontrados simultâneamente na chamada. Neste caso, a sintaxe da chamada está correta, mas não há nenhuma atitude a ser tomada.

Quando é encontrado um parâmetro formal, em S, deve-se procurar o parâmetro real correspondente, na chamada. A representação interna dos tokens que compõem o parâmetro real é inserida no vetor S_p (S_p é gravado em disco). Cada token é analisado para ver se pode ou não fazer parte do parâmetro; caso possa, será inserido em S_p. A análise é feita para que seja aceita tôda configuração válida para o tipo do parâmetro real em questão. Por ser simples, a análise também aceita configurações sintáticamente inválidas; entretanto, o erro será descoberto durante a análise sintática. Outra observação a ser feita é que a análise aceita sempre o maior número de tokens que possa fazer parte do parâmetro. (Na verdade, aceita todos os tokens até o primeiro que não possa fazer parte). Isto requer um certo cuidado da parte do programador. Um erro

possível seria a definição na estrutura da macro da sequência $\$EXP_1 + \$EXP_2$ (ou outra similar). Neste caso, durante a análise dos parâmetros reais, seriam lidos, inicialmente, todos os elementos que o programador imaginou pertencerem à primeira expressão, e efetivamente inseridos nela. Em seguida, seria lido o sinal "+", e também inserido como fazendo parte da primeira expressão, e da mesma forma todos os elementos que deveriam fazer parte da segunda expressão.

A análise dos parâmetros preocupa-se em aceitar comandos compostos e blocos função, onde válidos. Como podem haver comandos compostos e blocos função inseridos uns nos outros, torna-se necessário uma contagem dos BEGIN, FBEGIN, END e FEND para descobrir o fim do parâmetro em questão.

No caso da chamada de macro estar no corpo de uma outra macro (que está sendo definida), pode-se encontrar em um parâmetro, X, da macro chamada uma referência a um parâmetro, Y, da macro sendo definida. Neste caso é feita uma pesquisa na Tabela de Símbolos para descobrir o número de ordem de aparição do parâmetro Y. O que será colocado no corpo da macro chamada será o número de ordem de Y, não de X. Para ilustração veja exemplo b, Capítulo V.4.

Quando o fim de um parâmetro é encontrado, é colocada uma marca no vetor S_p, e no começo da análise de cada parâmetro é inserido em S_1 um apontador para a posição de S_p onde começa o parâmetro.

A Figura V.3 mostra a situação dos vetores S_1 e S_p,

após o fim da primeira etapa, no caso de uma macro M_1 de três parâmetros.

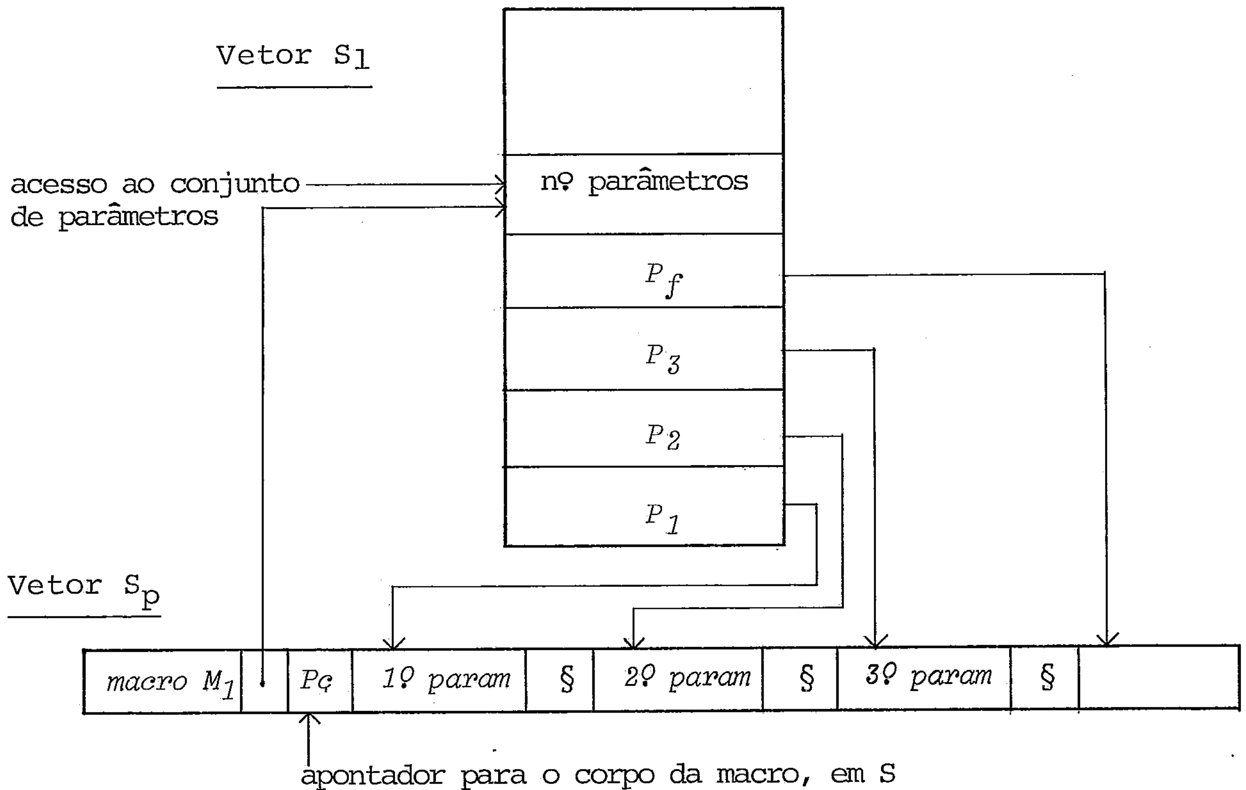


Figura V.3

A posição, no vetor S_1, do apontador para o começo do parâmetro desejado, em S_p, pode ser obtido através da fórmula : P_p - 3 (que é o número de parâmetros, obtido na própria posição apontada por P_p) - 2 + <nº de ordem da aparição do parâmetro desejado>.

Ao fim da análise dos parâmetros de uma macro, insere-se em S_1 um apontador, representado na Figura V.3 por P_f, para a continuação de S_p (após os parâmetros da macro). A utilização deste apontador aparece no Capítulo V.5 .

Na segunda etapa, constrói-se o corpo da macro expandida, incluindo-se os parâmetros. São lidos do vetor S os elementos do corpo da macro (não expandida), que farão parte, também, do corpo da macro expandida. Quando, porém, é lido de S um parâmetro formal, este é substituído, usando-se seu número de ordem de aparição (de S) na fórmula acima descrita, para se chegar ao parâmetro formal, em S_P, que será lido no lugar do parâmetro formal.

O resultado das duas etapas, o corpo da macro expandido, pode ter um de dois destinos:

1) Pode ser gravado em S, se for o caso de uma definição de macro fazer referência a outra. Neste caso o corpo da macro (expandido) da outra seria encaixado no corpo da macro (não expandido) da primeira, em S.

2) Pode ser gravado em disco, para posterior análise sintática, no caso de uma chamada na parte "executável" do programa. (após o ENDMACRO)

A rotina de expansão de macro pode ser descrita pela figura V.11 , ao fim do Capítulo V.5 .

V.4 - Exemplos de Expansão de Macros

a) Chamada de macro após o ENDMACRO

Tomemos como exemplo uma chamada da macro WHILE, cuja definição está na Figura V.1 e cuja disposição em S está na Figura V.2 .

Vejamos o resultado da chamada WHILE I LT 1 DO I:=I+1 .

Após a primeira etapa, teríamos a situação dos vetores S_p e S_1 mostrada na Figura V.4 .

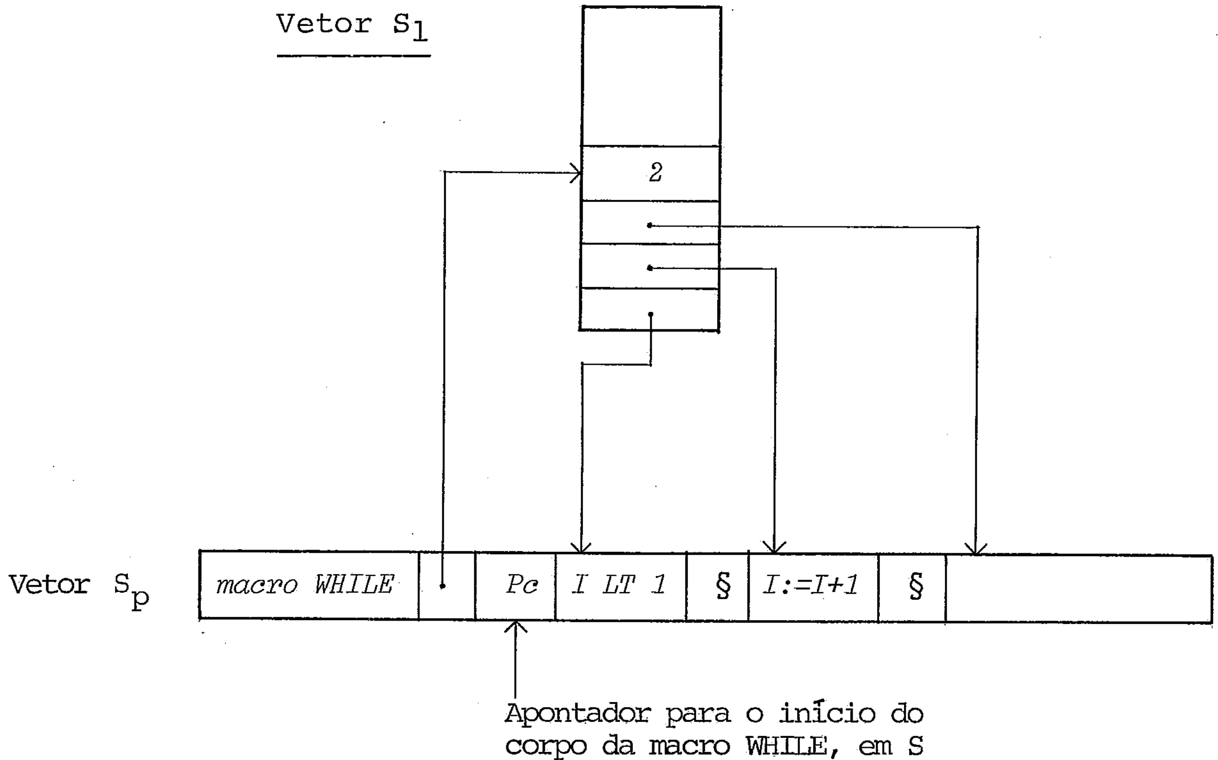


Figura V.4

Deve-se observar que a chamada acima referida não é suficiente para que a situação da Figura V.4 seja alcançada. É necessário, ainda, algum elemento logo após a chamada, que, ao ser lido, caracterize o fim do último parâmetro (na definição, $STAT).

Após a segunda etapa, teremos o corpo de macro expandido mostrado na Figura V.5 .

```
BEGIN LABEL #L;
      #L: IF
            I LT 1                % primeiro parâmetro
          THEN
             BEGIN
                 I:=I+1;          % segundo parâmetro (sem o ";")
                 GOTO #L
             END
END
```

Figura V.5

Os parâmetros são lidos de S_p, o resto de S.

b) Chamada de macro antes do ENDMACRO

Neste exemplo estaremos trabalhando com as rotinas de definição e expansão de macro, ao mesmo tempo; entretanto o interesse do exemplo está na parte relacionada com a expansão.

Seja a macro definida na Figura V.6 .

```
MACRO FOR $VAR := $EXP1 STEP $EXP2 UNTIL $EXP3 DO $STAT
DEFINE BEGIN
            $VAR := $EXP1;
            WHILE $VAR LE $EXP3 DO
                BEGIN
                    $STAT;
                    $VAR := $VAR + $EXP2
                END
       END
```

Figura V.6

Teríamos nos vetores S e S_p, após a primeira etapa da rotina de expansão (chamada pela de definição), o mostrado na Figura V.7 .

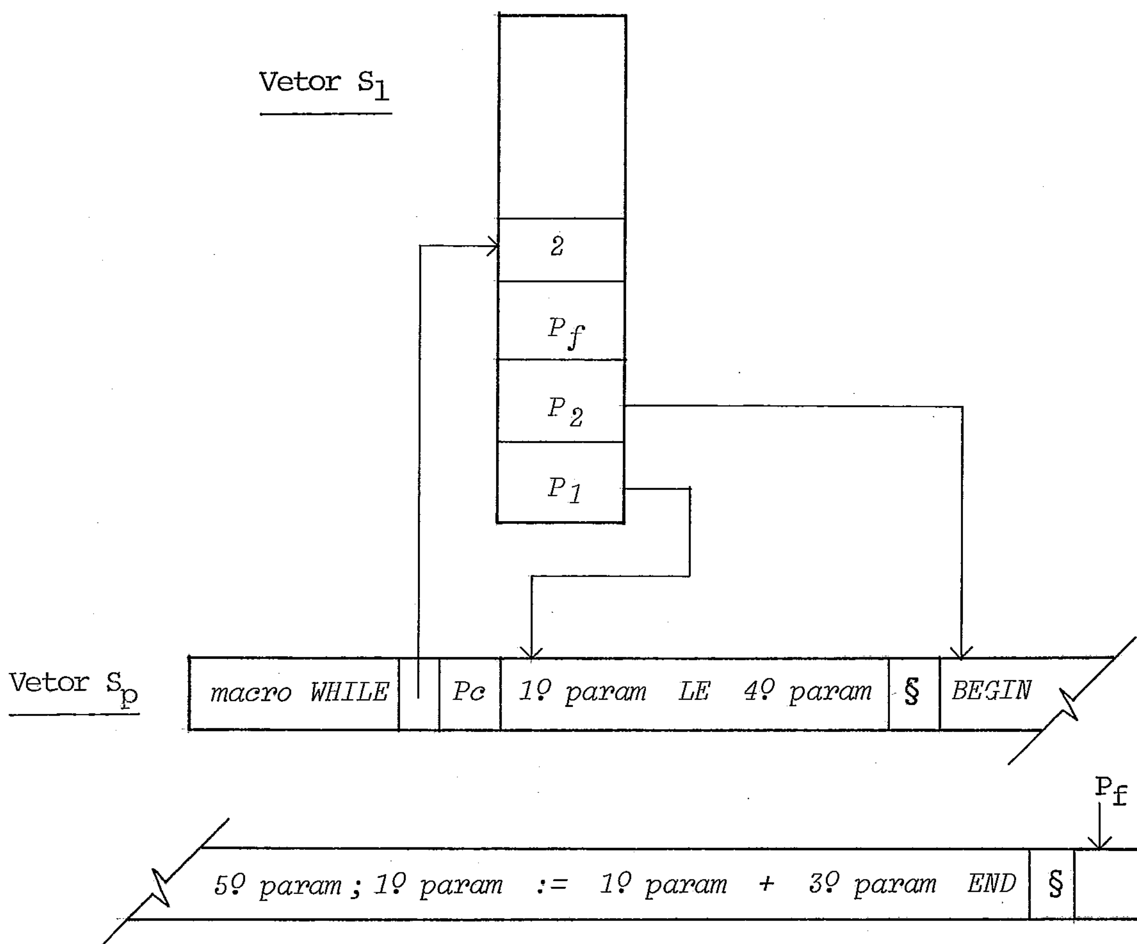


Figura V.7

Neste exemplo temos que considerar que o parâmetro $STAT aparece em ambas as macros, FOR (Figura V.6) e WHILE (Figura V.1). Entretanto, deve ficar claro que a referência a $STAT no corpo da macro FOR é uma referência ao parâmetro da macro FOR; por isto aparece como 5º parâmetro no vetor S_p. O parâmetro $STAT da macro WHILE é aquele apontado por P_2 na Figura V.7 .

Ao final, teríamos o corpo de macro mostrado na Figura V.8 .

```
BEGIN
    1º parâmetro := 2º parâmetro;
    BEGIN LABEL #L;                    % a partir daqui, macro WHILE
        #L: IF 1º parâmetro LE 4º parâmetro THEN
        BEGIN
            BEGIN
                5º parâmetro;
                1º parâmetro := 1º parâmetro + 3º parâmetro
            END;
            GOTO #L
        END
    END                                % fim da macro WHILE
END
```

Figura V.8

V.5 - Expansão de Macros em Profundidade

A expansão de macros em profundidade acontece quando há uma chamada de macro, e em um dos parâmetros é encontrada nova chamada de macro.

Na primeira etapa da rotina de expansão de macros, descrita no Capítulo V.3, se encontrará a nova chamada de macro, durante a análise de um parâmetro. Neste caso, interrompe-se a análise da primeira macro, e se começa a analisar a segunda. Antes disto, no entanto, deve-se guardar alguns elementos que permitam, após a análise da segunda macro, retornar à análise da primeira. Estes elementos são inseridos no vetor S_2, e são os seguintes:

a) Posição, no vetor S, do token (um parâmetro, é claro) da primeira macro que foi lido por último. (este token faz parte da estrutura da macro)

b) Posição, no vetor S_1, do acesso aos parâmetros da primeira macro

c) Posição, no vetor S_1, do endereço do parâmetro (no vetor S_p) que estava sendo analisado

d) Situação do parâmetro (se havia BEGIN ou FBEGIN sem o END ou FEND correspondentes, e outras situações quanto a elementos lidos do parâmetro real. A situação do parâmetro é dada por uma variável, na rotina de expansão)

e) Número de END e FEND que ainda devem aparecer.

Todos estes elementos são referentes à primeira macro. Para a análise dos novos parâmetros, é construído um novo grupo de apontadores no vetor S_1, e os novos parâmetros são inseridos em S_p a partir da primeira posição desocupada.

Se considerarmos, como ilustração, um caso em que uma macro (M_1) com três parâmetros chama outra (M_2) com dois, seria

alcançada a situação descrita na Figura V.9, após a análise dos parâmetros da segunda macro.

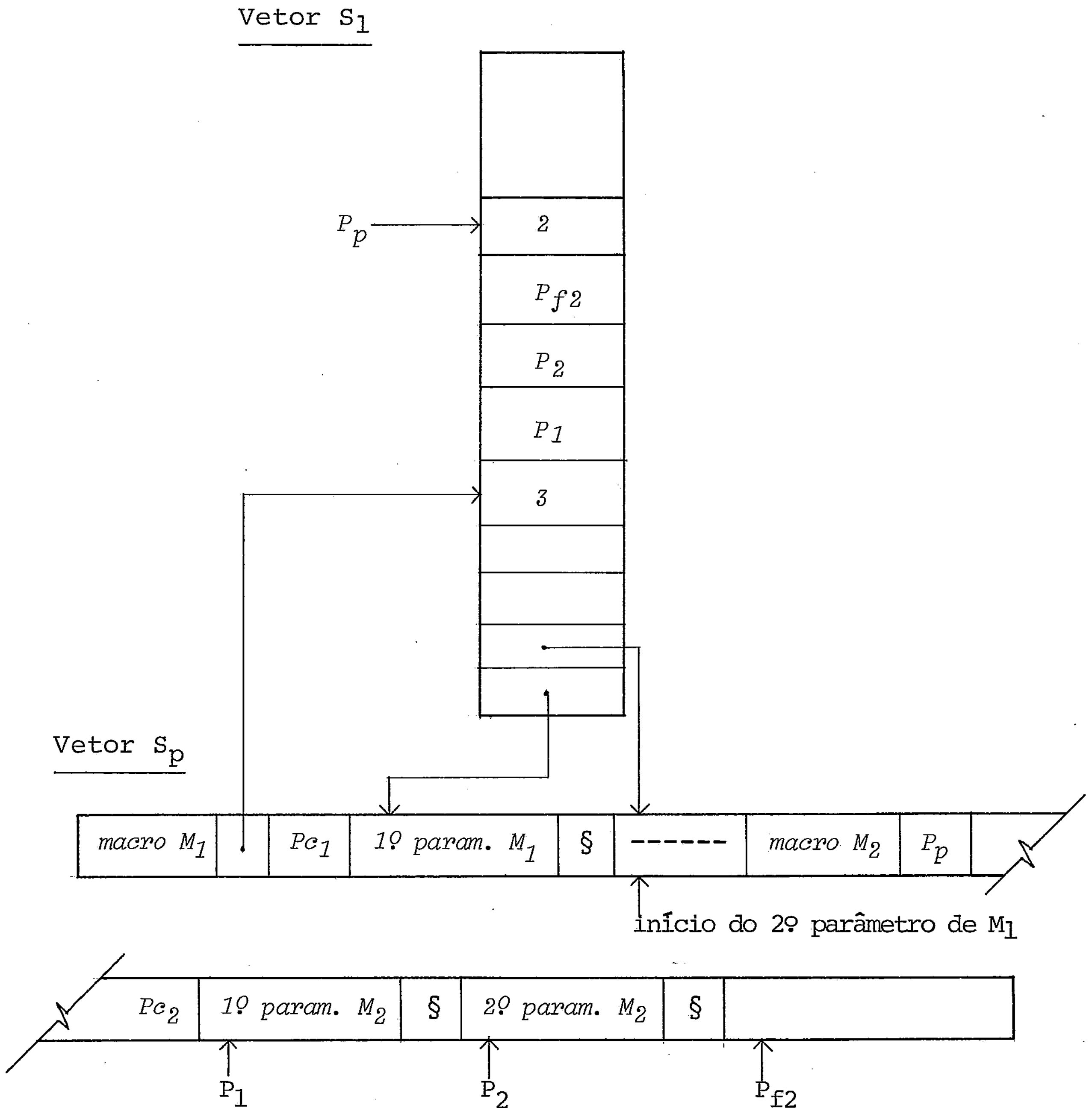


Figura V.9

Neste exemplo, a macro M_2 está sendo chamada no segundo parâmetro da macro M_1 .

Após a análise de M_2, retoma-se a de M_1, continuando-se a inserir os elementos pertencentes ao segundo parâmetro a partir da primeira posição desocupada de S_p, apontada por P_{f2} na Figura V.9 .

Após a análise de M_1, teríamos a situação descrita na Figura V.10 .

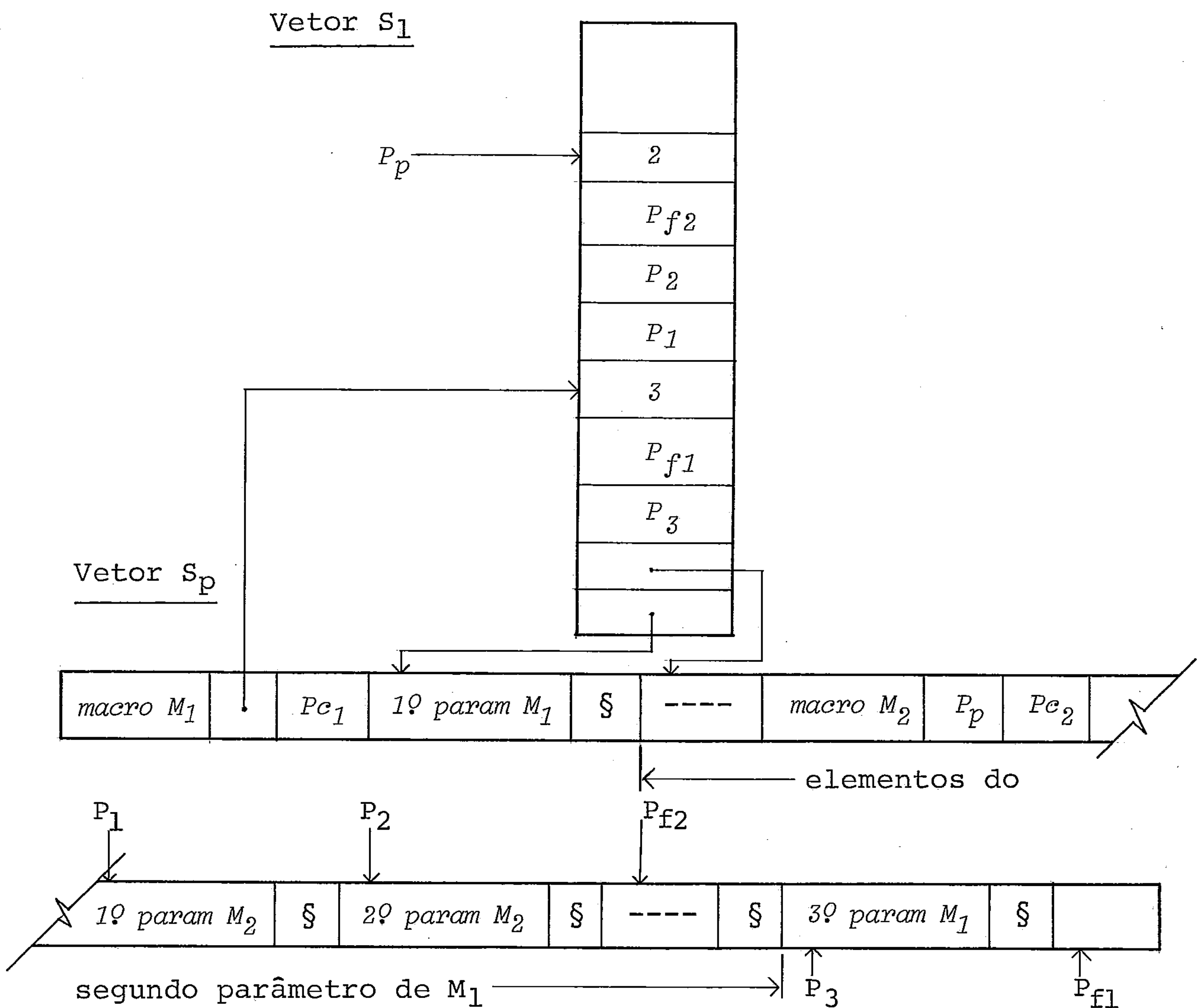


Figura V.10

O segundo parâmetro de M_1 ainda não está totalmente constituído; só o será após o desenvolvimento da macro M_2.

Na segunda etapa da rotina de expansão é construido o corpo da macro expandida, lendo-se seus elementos dos vetores S e S_p em paralelo, como foi descrito na seção V.3 . Quando for lido o segundo parâmetro de M_1, em S_p, será encontrada a menção à macro M_2. Neste momento é começada a expansão de M_2, com a obtenção do seu corpo em S e de seu conjunto de parâmetro em S_p e S_1. Para auxiliar a continuação da expansão de M_1, após a de M_2, os seguintes elementos são guardados em S_2 :

a) Posição, no vetor S, do token (um parâmetro) de M_1 que foi lido por último (êste token faz parte do corpo da macro).

b) Posição, em S_1, do acesso aos parâmetros de M_1.

À medida que são obtidos os elementos do corpo de M_2, estes são inseridos no corpo de M_1.

Após a expansão da segunda macro, retoma-se a situação da primeira, como estava antes do encontro da segunda. Um dos aspectos desta situação é que estavam sendo lidos os elementos de um parâmetro da primeira macro, em S_p; devem ser lidos agora os elementos restantes deste parâmetro. A função do apontador P_f da segunda macro (P_{f2} na Figura V.10) é indicar o próximo elemento do parâmetro da primeira macro a ser lido de S_p.

A rotina de expansão de macros pode ser descrita pelas Figuras V.11_a, V.11_b e V.11_c .

```
rotina MACROCALL;
BEGIN
     ETAPA1;
     ETAPA2
END
```

Figura V.11_a

```
rotina ETAPA1;
BEGIN
    novamac1: INSERE-EM-Sp(nomemacro);
    POSICIONA-EM-S(nomemacro,estrutura);
    INICIALIZA-CONJUNTO-PARÂMETROS(apontS1,apontSp);
    LER-S(simb);
    voltaparam1: WHILE simb NE marca DO
    BEGIN
        IF simb EQ paramformal THEN
        BEGIN
            INSERE-EM-S1(apontSp);
            LER(token);
            WHILE FAZ-PARTE(token,simb) DO
            BEGIN
                IF token EQ nomemacro THEN
                  BEGIN
                      SALVA-INFO(simb,apontS1,apontSp,profund);
                      GO TO novamac1
                  END;
               INSERE-EM-Sp(token);
               LER(token)
            END;
            INSERE-EM-Sp(marca)
        END
        ELSE COMPARA(simb,token);
        LER-S(simb)
    END;
    IF profund GT 0 THEN
    BEGIN
        RECUPERA-INFO(simb,apontS1,apontSp,profund);
        GO TO voltaparam1
    END
END
```

Figura $V.11_b$

```
rotina ETAPA2;
BEGIN
    novamac2: POSICIONA-EM-S(nomemacro, corpo);
    OBTEM-CONJUNTO-PARÂMETROS(apontS1);
    LER-S(simb);
    voltaparam2: WHILE simb NE marca DO
    BEGIN
        IF simb EQ paramformal THEN
        BEGIN
            OBTEM-DE-S1(apontSp);
            LER-Sp(tokparam);
            WHILE tokparam NE marca DO
            BEGIN
                IF tokparam EQ nomemacro THEN
                   BEGIN
                       SALVA-INFO(simb, apontS1, profund);
                       GO TO novamac2
                   END;
                CORPO-EXPANDIDO(tokparam);
                LER-Sp(tokparam)
            END
        END
        ELSE CORPO-EXPANDIDO(simb)
    END;
    IF profund GT 0 THEN
    BEGIN
        RECUPERA-INFO(simb, apontS1, profund);
        GO TO voltaparam2
    END
END
```

Figura V.11c

V.6 - Limites de Expansão em Profundidade

Existem duas formas de limitação quanto ao número de macros que a rotina de expansão pode desenvolver ao mesmo tempo. Ela não pode desenvolver macros em profundidade maior que 26, ou seja, trabalhar com mais de 26 macros em que a primeira faça referência à segunda, a segunda à terceira, e assim até a última. Igualmente, a rotina não pode desenvolver macros em que a soma total dos parâmetros mais duas vezes o número de macros seja maior que 256. Note-se que neste último caso as macros não necessitam estar todas em profundidade. Por exemplo, uma macro pode fazer referência a outras duas; neste caso, a profundidade aumenta de 1.

V.7 - Exemplo de Expansão de Macros em Profundidade

Vejamos uma chamada da macro WHILE, definida anteriormente na Figura V.1, e que faça referência à macro IFF, definida, a seguir,na Figura V.12 .

```
MACRO IFF $COND THEN $STAT1 ELSE $STAT2
DEFINE BEGIN LABEL #L;
             IF $COND THEN
             BEGIN
                  $STAT1;
                  GOTO #L
             END;
             $STAT2;
   #L: END
```

Figura V.12

Seja a chamada:

WHILE I+J LT 100 DO IFF I+J LT 50 THEN I:=I+1 ELSE J:=J+2

Na primeira etapa da rotina de expansão, após ser encontrada a chamada à macro IFF, seria alcançada a situação da Figura V.13, com o vetor S_2 salvando informações para a posterior retomada da análise da macro WHILE.

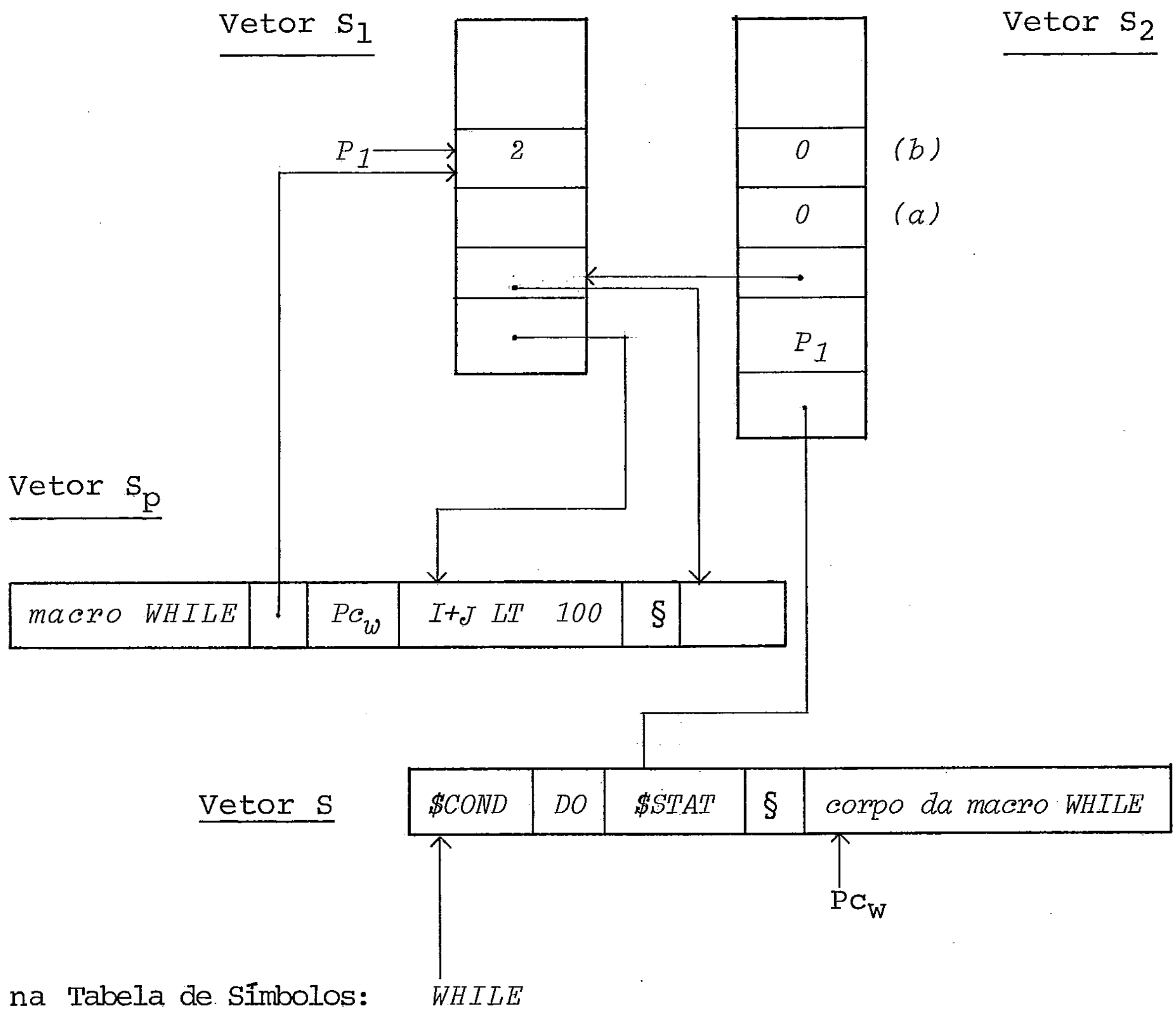


Figura V.13

O primeiro zero no Vetor S_2, na Figura V.13, ao lado de (a), indica o estado do segundo parâmetro da macro WHILE (parâmetro formal $STAT): ainda não foi inserido nenhum elemento no parâmetro. O segundo zero, ao lado de (b), é o número de END e FEND que devem ser lidos.

Após o fim da primeira etapa os vetores S_p e S_1 teriam a situação descrita na Figura V.14 .

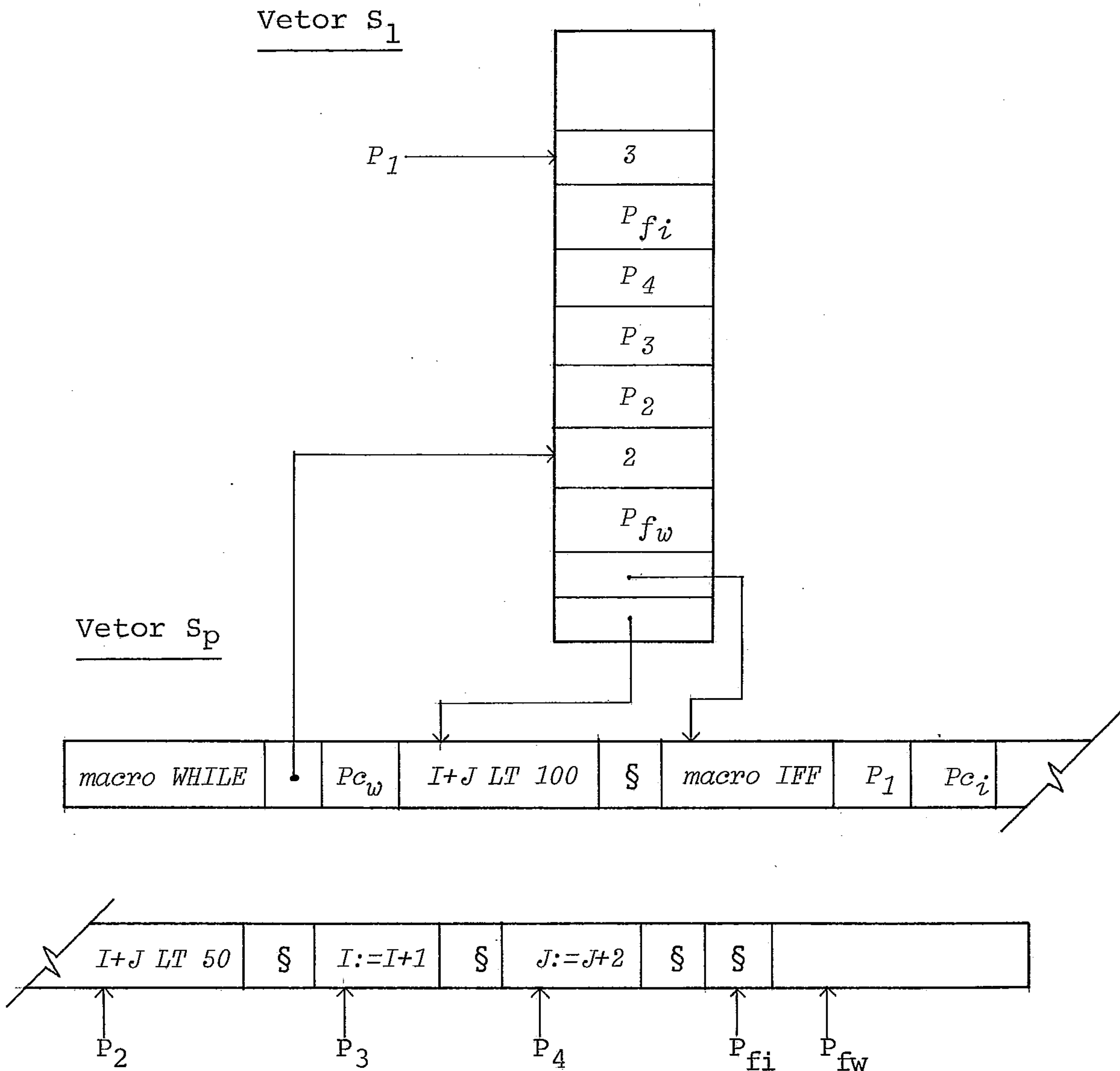


Figura V.14

Durante a segunda etapa, ao ser encontrada a menção à macro IFF, seriam colocados no Vetor S_2 as informações descritas na Figura V.15 .

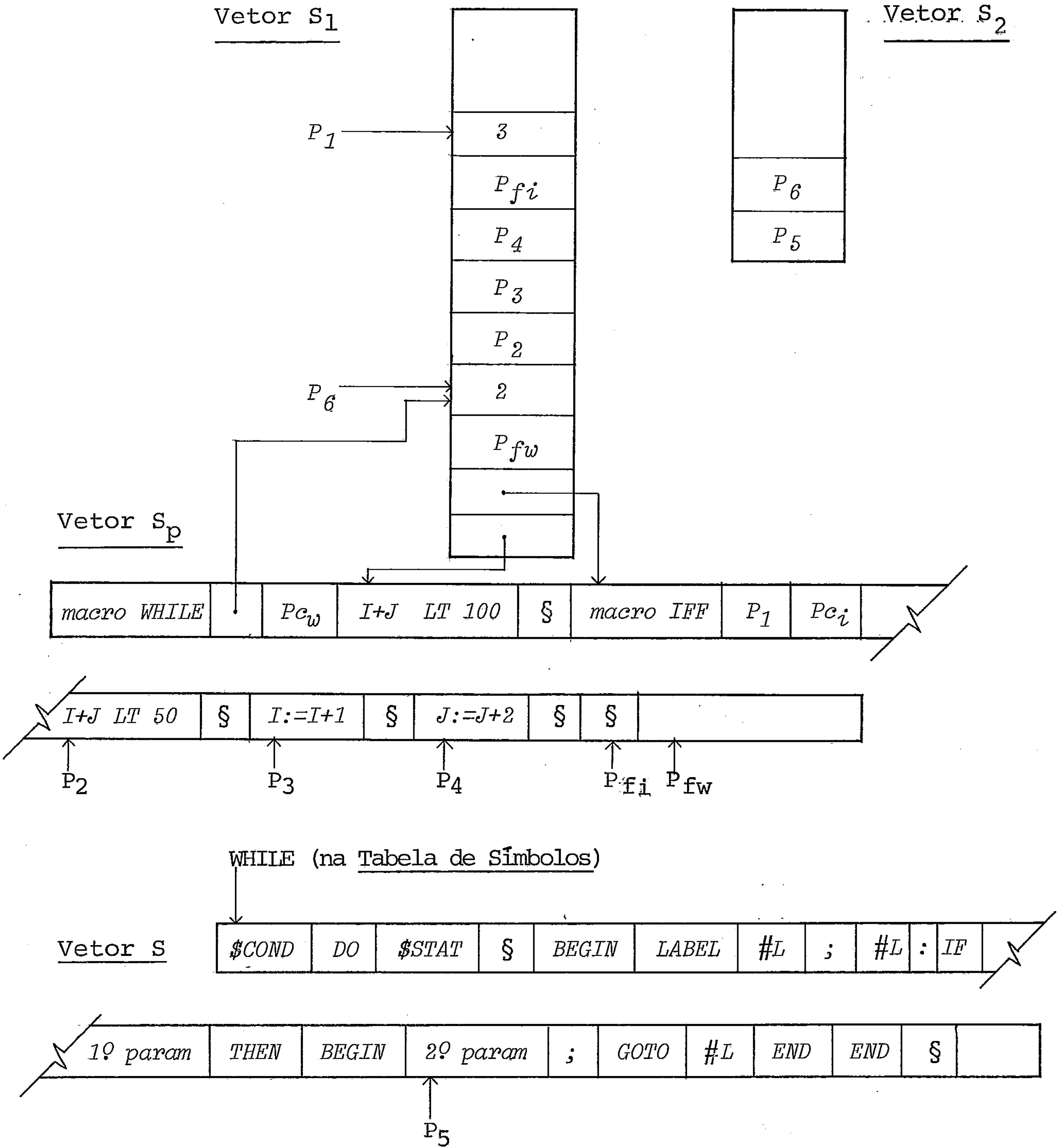


Figura V.15

A expansão alcançada no final da rotina está descrita na Figura V.16 .

```
BEGIN LABEL #L;
      #L: IF I+J LT 100 THEN
      BEGIN
           BEGIN LABEL #L;                    % a partir daqui, macro IFF
                 IF I+J LT 50 THEN
                 BEGIN
                      I:=I+1;
                      GOTO #L
                 END;
                 J:=J+2;
           #L: END;                           % fim da macro IFF (sem ";")
            GOTO #L
      END
END
```

Figura V.16

CAPÍTULO VI

OUTROS ASPECTOS DO COMPILADOR

VI.1 - A Análise Léxica

O analisador léxico tem que levar em conta que há uma parte da linguagem, o formato das entradas e saídas, com regras diferentes das do restante para a formação de símbolos. Por isso, foi considerado vantajoso |Gries[9]| dividir o analisador em duas partes distintas, com um mecanismo de decisão para saber qual parte irá atuar. Cada uma das partes é composta de uma instrução CASE. A variável de contrôle do CASE é a classe do primeiro caracter de cada token (cada caracter, ao ser lido, é enquadrado numa classe).

O resultado da análise léxica é a transformação da cadeia de caracteres que compõem o programa em um conjunto de tokens, cada um possuindo um código de representação interna. Alguns destes tokens e códigos tem significado apenas para o tratamento das macros, como por exemplo as palavras reservadas MACRO e ENDMACRO e seus códigos. Do processo de substituição da chamada de macro pelo corpo da macro expandida resultarão apenas tokens que pertençam à linguagem base, enquanto que as definições de macro não são passadas ao analisador sintático.

Há além disso um pequeno número de situações que exigem o uso de outros códigos desconhecidos pelo analisador sintático, mas estes códigos serão substituídos antes da análise sintática. A coleção de códigos está descrita na Figura VI.1.

Nesta figura os códigos estão representados pela variável T_1. Alguns códigos referem-se a um conjunto de tokens, como o conjunto das palavras reservadas. Neste caso é necessário um sub-código, representado na Figura VI.1 por T_2, para diferenciar cada código do conjunto dos outros, o que é feito na Fig. VI.2 . A variável T_2 tem uso em outras situações, como por exemplo guardar o número de caracteres de um identificador. Fora o código do token, o analisador léxico guarda o token, quando necessário, usando para isto um vetor de nome TOK.

SÍMBOLO	VAR. T_1 (código)	VARIÁVEL T_2	VETOR TOK	OBS
constante alfanumérica	*1*	*1*	*constante alfa*	*(1)*
cte. alfanumérica ou cte. inteira, valor ≤255	*2*	*1*	*cte. alfa, ou valor da cte. inteira*	
cte. inteira, valor >255	*3*	*2*	*vlr. da cte. inteira*	
constante real	*4*	*2*	*valor da constante*	
símbolo especial	*5*	*Ver Tabela 1*	*vlr. no caso T_2=40*	
palavra reservada	*6*	*Ver Tabela 2*	-	*(4)*
cadeia	*7*	*# caracteres*	*cadeia*	
%eod	*8*	-	-	*(2)*
"MACRO"	*9*	-	-	*(3)*
parâmetro formal macro	*10*	*Ver Tabela 3*	-	*(3)*
nome de macro	*11*	*Apontador para a Tab. de Símbolos*	-	*(3)*
"ENDMACRO"	*12*	-	-	*(3)*
"DEFINE"	*13*	-	-	*(3)*
palavra reservada de macro	*14*	*Apontador para a Tab. de Símbolos*	-	
cte. inteira em formato	*15*	-	*valor da constante*	*(1)*
identificador	*16*	*# caracteres*	*carac. do identif.*	
"NOLIST"	*17*	-	-	*(3)*

Figura VI.1

Tabela 1 (símbolos especiais)

SÍMBOLOS	VARIÁVEL T_2	OBS.
=	0	
;	4	
(	10	
)	11	
,	12	
[	13	
:	14	
]	15	
:=	17	
F	32	(5)
.	33	
E	34	(5)
I	35	(5)
A	36	(5)
X	37	(5)
L	38	(5)
cte. inteira, em formato	39	(5)
+	41	
-	42	
*	43	
/	44	
"	45	

Figura VI.2_a

Tabela 2 (palavras reservadas)

PALAVRA RESERVADA	VARIÁVEL T_2
BEGIN	*2*
END	*3*
INTEGER	*5*
REAL	*6*
LABEL	*7*
VECTOR	*8*
PROCEDURE	*9*
LT	*18*
LE	*19*
EQ	*20*
NE	*21*
GT	*22*
GE	*23*
IF	*24*
THEN	*25*
RESULT	*26*
GO	*27*
TO	*28*
GOTO	*29*
WRITE	*30*
READ	*31*
FBEGIN	*46*
FEND	*47*

Figura $VI.2_b$

Tabela 3 (parâmetros formais)

PARÂMETRO	VARIÁVEL T_2
$VAR	*4*
$EXP	*3*
$COND	*2*
$STAT	*1*

Figura $VI.2_C$

OBSERVAÇÕES:

1) Estes tokens tem, cada um, dois códigos diferentes. O da linha da observação é usado por circunstâncias da implementação. O outro é reconhecido pelo analisador sintático.
2) O %eod não chega a ser analisado pelo analisador léxico. Sua presença é detectada pela rotina de entrada/saída do Mitra.
3) Estes tokens, com seus códigos, não aparecem mais no programa que é passado para o analisador sintático.
4) Refere-se às palavras reservadas da linguagem base. Às palavras MACRO, ENDMACRO, DEFINE e NOLIST também são reservadas, para o compilador.
5) Estas letras e constantes são consideradas símbolos especiais, quando fazem parte de um formato.

Quando são encontradas constantes inteiras ou reais, são chamadas rotinas de transformação, pertencentes à biblioteca do sistema |Mitra[10]|. Estas rotinas, M%DCBN para inteiros e DCVF para reais, transformam as constantes numa forma interna do computador, binária para os inteiros, e de vírgula flutuante para os reais. São sob estas formas que as constantes podem ser inseridas na Tabela de Símbolos. Naturalmente, todas as restrições às constantes que servem de entrada a estas rotinas são extensivas às constantes da linguagem.

O analisador léxico, no caso de uma definição de macro, estando analisando o corpo da macro, insere o nome da macro ao lado dos identificadores começando com "#" (aumentando do seu tamanho). Este detalhe, entretanto, não está representado nos exemplos até aqui fornecidos.

O analisador léxico, no caso de uma definição ou chamada de macro, ou simplesmente ao analisar um identificador, pesquisará na Tabela de Símbolos, para descobrir se o símbolo analisado tem algum significado especial (para o tratamento das macros), ou se é uma palavra reservada, ou ainda para inserir o símbolo na Tabela de Símbolos.

VI.2 - Uso da Tabela de Símbolos

O uso da Tabela de Símbolos obedece a duas necessidades:

a) Pesquisa para descoberta de palavras reservadas;

b) Inserção de identificadores e constantes.

As palavras reservadas estão prèviamente inseridas na Tabela de Símbolos. A pesquisa de um identificador que seja uma palavra reservada encontrará esta palavra reservada na Tabela de Símbolos.

Nem sempre é necessária a inserção de identificadores e constantes na Tabela de Símbolos. O analisador léxico grava os identificadores e constantes em disco, de onde serão lidos para a análise sintática (junto com seus códigos). Entretanto foi decidido pela inserção no caso do tratamento de macros, para uma redução no tamanho das informações contidas nos vetores S e S_p e a padronização no processo de leitura e inserção destas informações. Assim, teremos nestes vetores apenas o código (representado por T_1 na Figura VI.1) e outra informação de 1 byte, que pode ser um subcódigo ou um endereço para a Tabela de Símbolos. Apenas estas duas informações serão manipuladas no tratamento das macros, até o momento da gravação em disco. Existe uma rotina para gravação em disco e é esta própria rotina que recupera os identificadores e constantes, nos casos apropriados, para gravá-los.

A Tabela de Símbolos é, na realidade, uma tabela de apontadores (TAB) e um vetor de informações (VETNOMES). Dado um identificador ou uma constante obtém-se uma entrada da tabela de apontadores, e o apontador assim obtido aponta para o começo das informações sobre o identificador ou a constante, no vetor de informações.

VI.2.1. Acesso à tabela de apontadores

O acesso à tabela de apontadores é feito usando-se "hashing". A função de "hashing" é obtida usando-se o método de dobramento seguido do meio quadrado |Souza[11]|. No caso de colisões procurar-se-à um lugar vago na Tabela de Símbolos usando hashing duplo |Souza[11]|, considerando-se a tabela como sendo circular |Knuth[12]|.

Assim, obtêm-se da função um endereço inicial (home-address) $h_o(K)$ para cada identificador ou constante K, e um incremento i(K) para o tratamento das colisões.

Consideramos o identificador K como sendo uma sequência de no máximo 64 caracteres de um byte de tamanho.

$$K = a_1a_2 \ldots a_n \quad , \quad 1 \leq n \leq 64$$

Definindo X[m:n] como sendo os n bits de X a partir de m para a direita, são executadas as seguintes operações para a obtenção da função de hashing.

$$T_1 = \begin{cases} a_1a_2 \oplus a_3a_4 \oplus \ldots \oplus a_{n-1}a_n & \text{se n é par} \\ a_1a_2 \oplus a_3a_4 \oplus \ldots \oplus a_na_{n+1} & \text{se n é ímpar, com } a_{n+1} = 0 \end{cases}$$

T_1 é do tipo word

$Dob = T_1 [15:16]$

$T_2 = Dob*Dob$ (T_2 é do tipo long)

$T_3 = T_2 [21:8]$

$T_4 = T_2 [4:5]$

$i(K) = T_4*2$

$MQ = T_3*2$

A função de hashing do identificador K é definido como:

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

Note-se que T_3 tem um byte de tamanho. Multiplicado por 2 resulta num número par, podendo acessar 256 posições de 2 bytes de tamanho (cada apontador da tabela tem 2 bytes de tamanho). Este é o tamanho da tabela (256). As posições não ocupadas da tabela tem como conteúdo o valor zero.

VI.2.2. O vetor de informações

O vetor de informações (VETNOMES) é dividido em nós de tamanhos variados. O vetor começa parcialmente preenchido (o mesmo acontecendo com TAB) com as informações referentes às palavras reservadas. Cada nó tem informações referentes a um símbolo. Os nós são preenchidos sequencialmente, a medida que os símbolos aparecem (as posições mais baixas tem os nós das palavras reservadas). A estrutura dos nós está descrita na Figura VI.3

n	t_1	t_2	*símbolo*	*informação suplementar*

Figura VI.3 : nó do vetor VETNOMES

Nesta figura, n é o tamanho em bytes do símbolo, menos um. Ocupa um byte. t_1 e t_2 são o código e o subcódigo, ou outra informação, conforme definidos em VI.1, e ocupam, cada um, um byte. O próprio símbolo vem depois, ocupando n+1 bytes. A informação suplementar existe únicamente no caso de nomes de macro e parâmetros formais.

Para os nomes de macro, são inseridos no nó dois apontadores para o vetor S, com dois bytes cada um, e ainda o número de parâmetros, ocupando um byte. Um apontador é para a estrutura da macro, o outro para o corpo. No caso dos parâmetros formais, é inserido o número de ordem da aparição do parâmetro.

VI.3 - Tratamento de Erros

Vários tipos de erro podem ser detetados na fase de compilação que corresponde a este trabalho. No caso mais ge-

ral, erros léxicos serão encontrados pelo analisador léxico. Entretanto, alguns erros sintáticos poderão ser encontrados no que se refere às macros, tanto na definição como expansão, e também dentro de formatos de entrada/saída.

Na maior parte dos erros léxicos o token sendo analisado não é enviado para o conjunto que será analisado sintaticamente. Uma mensagem de erro é emitida, e a rotina de erro procura o próximo caracter branco. Neste ponto é retomada a análise léxica. Nos casos de constantes que ultrapassem os limites admissíveis, a constante zero será enviada em seu lugar, para que a análise sintática possa ser executada.

Sendo encontrados erros dentro de formatos, será emitida uma mensagem e o compilador descartará não só o token sendo analisado como também todos os caracteres seguintes, até o fim do formato.

Para erros encontrados na fase de definição de macros, foi usado um procedimento que permite prosseguir com a análise até o fim das definições: a macro errada é considerada inexistente, retirando-se o atributo de nome de macro do nome da macro; procura-se a próxima palavra MACRO ou ENDMACRO, e daí prossegue a análise das macros restantes, até o ENDMACRO. Neste ponto será emitida uma segunda mensagem (a primeira, ao encontrar o erro), informando que a compilação foi suspensa por erro na definição de macro.

Erros durante a expansão de macros e outros difíceis de serem contornados, como fim da memória disponível, causam

a interrupção imediata da compilação.

No segundo passo da compilação, serão detetados erros sintáticos não referentes a macros, e erros semânticos. Outros erros semânticos só podem ser detetados em tempo de execução |Aho[13]|. Neste último caso, os erros serão detetados pelos módulos externos chamados, ou pelo próprio programa gerado. Mais detalhes em |Zakimi[5]| .

CAPÍTULO VII

CONCLUSÕES E CONSIDERAÇÕES FINAIS

O presente trabalho, que inclui a análise léxica e o tratamento de macros da linguagem EXPAND, foi desenvolvido em paralelo com o trabalho de |Zakimi[5]|, que trata da análise sintática e geração de código. Por causa da implementação em paralelo, foi necessário um esforço suplementar de cada um para simular a atuação do outro, e novo esforço para unificar os dois trabalhos. Com a unificação, está disponível o compilador no computador MITRA 15 do Laboratório de Automação e Simulação de Sistemas. O trabalho de |Zakimi[5]| foi completado no início de 1980.

A linguagem EXPAND tem como característica um compilador pequeno para o tipo de linguagem. Um dos objetivos de se implementar um compilador assim é seu uso em minicomputadores, e, como tal, o computador MITRA 15 representou uma boa opção para a implementação.

O uso de macros sintáticas nesta implementação acarreta em muitas das vantagens e desvantagens inerentes a qualquer tipo de macro. A definição de macros através de outras macros pode ser uma ferramenta perigosa, gerando código extremamente ineficiente. Isto levaria ao pensamento de se adotar um possante módulo de otimização de código, o que por sua vez não é compatível com a idéia de um compilador pequeno. Uma

solução de compromisso pode ser alcançada.

A macro sintática apresenta uma série de vantagens sobre a macro comum. A facilidade de compreensão de sua semântica auxilia a compreensão do programa, e a utilização de macros feitas por diferentes usuários. Seu formato livre possibilita a simulação de diferentes instruções, e assim, poderia ser executado no MITRA-15 um programa em uma linguagem inexistente neste computador. Em certos casos, é claro, isto poderia ser oneroso para um minicomputador. Por outro lado, o uso de um formato mais livre requer um cuidado maior na delimitação dos parâmetros, e o usuário terá que se conscientizar disto.

Deve-se notar que |Cheatham[1]|, |Leavenworth[2]| e |Cole[14]| colocam todo o tratamento de macros durante a análise sintática, enquanto que o presente trabalho faz este tratamento, através de uma análise sintática simplificada, durante a análise léxica. Esta última abordagem foi motivada pela necessidade de se dividir o compilador em dois, já que havia trabalho justificando duas teses, e a melhor divisão de tarefas foi a acima referida. |Cheatham[1]| define mesmo a macro sintática como sendo aquela cujo tratamento ocorre durante a análise sintática. Isto colocaria o esquema de macros implementado neste trabalho fora do contexto de macros sintáticas. Preferimos, entretanto, considerar macros sintáticas como aquelas que permitem adicionar novas estruturas sintáticas à linguagem base.

O tratamento de macros durante a fase de análise sintática possibilita o uso do analisador sintático nestas oca-

siões. Esta é uma ferramenta poderosa e traz vários benefícios, possibilitando uma extensão mais poderosa e mais maleável que a oferecida nesta implementação. Para começar, os parâmetros reais podem ser checados quanto à sintaxe, e os erros imediatamente enunciados. No caso de EXPAND, na fase da análise sintática a macro já deixou de existir como um conjunto, e os erros sintáticos podem não ser facilmente identificáveis com os parâmetros. Outra vantagem é que os parâmetros podem ser da classe de qualquer elemento sintático da linguagem base, e não apenas o conjunto de metavariáveis escolhidas para EXPAND. |Leavenworth[2]| também define o corpo e a estrutura da macro como pertencendo ou à categoria sintática <comando> ou à categoria <função> , para facilitar o reconhecimento pelo analisador sintático. |Cole[14]| e |Cheatham[1]| vão mais além, generalisando o conceito e permitindo macros de qualquer categoria sintática da linguagem base. |Cole[14]| analisa também dificuldades da implementação das idéias propostas, e lembra que parâmetros e macros podem ter que ser analisados várias vezes, como por exemplo no caso de macros em profundidade, mostrando também um caso em que um parâmetro correto não seria aceito. Para remediar estes casos, propõe a produção de código parcialmente compilado a cada parâmetro verificado, assim como a compilação parcial do corpo da macro, deixando espaços para os parâmetros.

Seria desejável a existência de uma biblioteca de macros, onde estariam macros semelhantes às instruções mais co-

muns das linguagens mais importantes, aumentando muito, com isto, o escopo de EXPAND. Esta biblioteca estaria melhor localizada em disco, e necessitaria de um módulo no começo do compilador, para ler as macros desejadas para transportá-las para a memória. O formato dos programas poderia ter que ser alterado, para dar a opção de utilização de macros desta biblioteca.

BIBLIOGRAFIA

|1| Cheatham, T.E. - The Introduction of Definitional Facilities Into Higher Level Programming Languages, Proc. AFIPS, 1966; Fall Joint Computes Conference, USA, 29: 623-637, 1966.

|2| Leavenworth, B.M. - Syntax Macros and Extended Translation - CACM, USA, 9(11):790-793, 1966.

|3| Naur, Peter - Revised Report on The Algorithmic Language Algol 60 - CACM, USA, 6(1):65-87, 1960.

|4| Aho, Alfred e Ullman, Jeffrey - Theory of Parsing, Translation and Compiling - N.J., Prentice Hall, 1978.

|5| Zakimi, Miyasato, Beatriz - Expand Uma Linguagem Extensível Através de Macros Sintáticas: Compilador da Linguagem Base, Rio de Janeiro, Tese de M.Sc., COPPE-UFRJ, 1980.

|6| 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.

|7| MITRA-15, Editeur Temps Réel MTR - França, CII, 1974

|8| MITRA-15, Langage LP15, LP15E, França, CII, 1974.

|9| Gries, David - Compiler Construction for Digital Computers, John Wiley & Sons, Wiley, 1971.

|10| MITRA-15, Editeurs Chargeur ECH, Editeurs de liens EDL, EDL-E, EDL-EX, EDL-D, EDL-DX, França, CII, 1975.

|11| De Souza, Jano Moreira - Algoritmos de Hashing para Problemas Específicos, Rio de Janeiro, Tese de M.Sc., COPPE UFRJ, 1978.

|12| Knuth, D.E. - The art of computer programming vol.3: Sorting and Searching - Reading, Mass., Addilson-Wesley, 1968.

|13| Aho, Alfred e Ullman, Jeffrey - Principles of Compiler Design, Califórnia, Addison-Wesley, 1978.

|14| Cole, A.J. - Macro Processors, Cambridge, Cambridge University Press, 1976.

APENDICE I

PROGRAMA EXEMPLO

```
& PRØGRAMA EXEMPLØ
&
& DECLARACØES DE MACRØ
&
& MACRØ IFF-THEN-ELSE
&
MACRØ IFF $CØND THEN $EXP1 ELSE $EXP2
DEFINE FBEGIN LABEL #L;
              IF $CØND THEN BEGIN
                                   RESULT $EXP1;
                                   GØ TØ #L
                               END;
              RESULT $EXP2;
              #L:
       FEND
&
& MACRØ FØR
&
MACRØ FØR $VAR := $EXP1 UNTIL $EXP2 DØ $STAT
DEFINE BEGIN LABEL #L1, #L2;
             $VAR := $EXP1;
             #L1: IF $VAR GT $EXP2 THEN GØ TØ #L2;
             $STAT;
             $VAR := $VAR + 1;
             GØ TØ #L1;
             #L2:
       END
ENDMACRØ
&
& FIM DE DECLARACØES DE MACRØ
&
& PRØGRAMA QUE CALCULA Ø N-ESIMØ NUMERØ DE FIBØNACCI
&
BEGIN INTEGER N,F;
      READ (I2,N);
      F:= IFF N EQ 0 THEN 0
          ELSE IFF N EQ 1 THEN 1
          ELSE FBEGIN INTEGER ULT,J,FIB;
                      ULT,FIB := 0,1;
                      FØR J := 2 UNTIL N DØ ULT,FIB := FIB,ULT+FIB;
                      RESULT FIB
               FEND;
      WRITE("NUMERØ"; I2,N; "DA SERIE DE FIBØNACCI"; I6,F);
END

     F Ø N T E   E X P A N D I D Ø :

BEGIN INTEGER N , F ; READ ( I 2 , N ) ; F := FBEGIN LABEL IFF-#L ; IF N
EQ 0 THEN BEGIN RESULT 0 ; GØ TØ IFF-#L END ; RESULT FBEGIN LABEL IFF-CL
; IF N EQ 1 THEN BEGIN RESULT 1 ; GØ TØ IFF-#L END ; RESULT FBEGIN
INTEGER ULT , J , FIB ; ULT , FIB := 0 , 1 ; BEGIN LABEL FØR-#L1 ,
FØR-#L2 ; J := 2 ; FØR-#L1 : IF J GT N THEN GØ TØ FØR-#L2 ; ULT , FIB :=
FIB , ULT + FIB ; J := J + 1 ; GØ TØ FØR-#L1 ; FØR-#L2 : END ; RESULT
FIB FEND ; IFF-#L : FEND ; IFF-#L : FEND ; WRITE ( "NUMERØ" ; I 2 , N ;
"DA SERIE DE FIBØNACCI" ; I 6 , F ) ; END
```