TERMINAL INTELIGENTE:
ANÁLISE SINTÁTICA PARA UM
COMPILADOR DA LINGUAGEM PL/STI

Regina Celia de Souza Pereira

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. Guilherme Chagas Rodrigues
(Presidente)

Prof. Nelson Maculan Filho

Prof. José Lucas M. Rangel Netto

Prof. Paulo Augusto Silva Veloso

RIO DE JANEIRO, BRASIL
MARÇO DE 1977

DE SOUZA PEREIRA, REGINA CELIA

Terminal Inteligente: Análise Sintática Para Um Compilador da Linguagem PL/STI |Rio de Janeiro| 1977

X , 119p. 29,7cm (COPPE-UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1977)

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

1. Compiladores I.COPPE/UFRJ II. Terminal Inteligente: Análise Sintática Para um Compilador da Linguagem PL/STI

Ao Paulo

AGRADECIMENTOS

Ao professor Guilherme Chagas Rodrigues pela orientação e constante apoio proporcionado.

À amiga Miriam Aparecida Marques pela colaboração na determinação da estrutura da Análise Sintática.

Ao amigo Maurício F. M. de Aguiar pelas críticas construtivas e valiosas sugestões apresentadas na parte de programação deste trabalho.

RESUMO

Este trabalho constitui-se da Análise Sintática para o compilador PL/STI, cujo objetivo e facilitar o desenvolvimento de Software básico para o Terminal Inteligente, projeto que está sendo realizado no Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro.

O capítulo I, consta da descrição da linguagem PL/STI, para o qual o compilador foi projetado.

No capítulo II, damos uma idéia geral do compilador e do seu funcionamento.

No capítulo III, descrevemos a Análise Sintática com detalhes e focalizamos também a Tabela de Símbolos e o tratamento de erros para os programas PL/STI.

Finalmente no capítulo IV, fazemos algumas considerações sôbre o trabalho.

ABSTRACT

This work consists of the syntactic analysis for the PL/STI Compiler. It aims at simplifying the development of software for the Intelligent Terminal Project which has been undertaken by the Núcleo de Computação Eletrônica at the UFRJ.

Chapter one describes the PL/STI language for which the Compiler has been designed.

Chapter two provides the reader with an outline of the Compiler and its working Features.

Chapter three describes the syntactic analysis, in detail, as well as the symbol table, and the handling of errors.

Finally, chapter four presents additional comments on the thesis as a whole.

ÍNDICE

Introdução

Está sendo desenvolvido no Núcleo de Computação Eletrônica da U.F.R.J., o projeto de construção do Hardware e Software necessários ao funcionamento de um Terminal Inteligente (T.I.), no sentido de torná-lo operacional.

Como o desenvolvimento de Software para um terminal desse tipo não é cômodo no próprio terminal, foi desenvolvido no Burroughs/6700, o Sistema Operacional de Simulação (S.O.S.) para o T.I., constituído basicamente de um Simulador, um Montador e um Depurador, no qual está sendo desenvolvido todo o software básico para o mesmo. No entanto, este software básico tem que ser todo programado em linguagem simbólica que pelas suas próprias características, acarreta uma grande perda de tempo na tarefa de programação. Daí então surgiu a necessidade de se ter uma linguagem de alto nível que permita ao programador se concentrar mais no seu problema e menos na tarefa de programar, do que é possível com linguagem simbólica.

Foi escolhido para o T.I., uma CPU INTEL/8008. Verificamos que existe uma linguagem tipo PL/1, própria para microcomputadores com CPU desse tipo. Resolvemos então desenvolver um compilador para uma linguagem com base nessa já existente, a qual chamamos PL/STI, cujo objetivo é facilitar a tarefa de desenvolvimento de software para o T.I.

Por se tratar de um projeto muito extenso, ficou estabelecido que o compilador PL/STI será desenvolvido por duas pessoas.

Este trabalho consiste da Análise Sintática para o mesmo. No outro trabalho, que está sendo desenvolvido pela colega Miriam Aparecida Marques, encontram-se a Análise Léxica e a Geração de códigos para o compilador PL/STI.

CAPÍTULO I

A LINGUAGEM PL/STI

1.1. Generalidades sobre a linguagem:

A linguagem PL/STI tem como base a linguagem PL/M da INTEL, com restrições que se fizeram necessárias. É uma linguagem estruturada, semelhante ao PL/1, orientada para os microcomputadores com CPU tipo 8008 ou 8080, onde uma palavra de memória é representada em 8 bits e uma palavra dupla em 16 bits. Desse modo PL/STI manuseia dois tipos básicos de dados: BYTE e ADDRESS. Uma variável ou constante BYTE é representada em 8 bits; uma variável ou constante ADDRESS é representada em 16 bits. PL/STI tem acesso aos indicadores de condição (bits que representam o estado da máquina depois que uma operação é efetuada), através de funções internas ao compilador e que o programador pode referenciar pelo nome (ver seção 1.16.6). A linguagem apresenta ainda outras facilidades como macros em tempo de compilação, que consiste na substituição automática de textos (ver seção 1.14) e funções internas que fazem instruções de máquina como por exemplo deslocamentos de bits ou rotação de bits através do acumulador (ver seção 1.16).

1.2. Constituintes básicos de um programa PL/STI

Programas em PL/STI são escritos em formato livre, isto é, espaços podem ser inseridos livremente onde um caracter branco é permitido.

O conjunto dos caracteres reconhecidos em PL/STI é constituido de:

a) Caracteres alfabéticos:

ABCDEFGHIJKLMNOPQRSTUVWXYZ

b) Caracteres numéricos:

0 1 2 3 4 5 6 7 8 9

c) Caracteres especiais:

$ = . / ( ) , + - * : ;

Caracteres especiais e suas combinações tem significado especial em um programa PL/STI, como mostrado no Apêndice B. O conjunto de caracteres alfabéticos e numéricos pode ser chamado de conjunto de caracteres alfanuméricos.

Os identificadores em PL/STI, podem ter até 30 caracteres alfanuméricos, onde o primeiro dos quais é obrigatóriamente alfabético. Os nomes de funções internas e as palavras reservadas da linguagem não podem ser declaradas como nomes de variáveis ou de rotinas pelo programador e se encontram no Apêndice C.

Sinais de cifrão podem ser usados em qualquer lugar do programa, pois o compilador os ignora. Por exemplo: INPUT$COUNT e INPUTCOUNT são para o compilador o mesmo identificador e neste caso o sinal de cifrão é usado para facilitar

a leitura.

Cadeias de caracteres em PL/STI são representadas entre apóstrofos. Para incluir um apóstrofo dentro de uma cadeia, basta escrevê-lo como apóstrofos duplos. Por exemplo: A cadeia ''' XY' contém os caracteres 'XY.

O compilador representa cadeias de caracteres pela representação ASCII.

A linguagem PL/STI permite o uso de comentários, que são sequências de caracteres delimitados à esquerda por /* e a direita por */. São totalmente ignorados pelo compilador e podem aparecer em qualquer lugar que um caracter branco é permitido, com exceção de que não podem aparecer dentro de uma cadeia de caracteres.

1.3. Elementos de dados em PL/STI

Elementos de dados em PL/STI são variáveis ou constantes. Uma constante é um número ou uma cadeia de caracteres. Constantes numéricas podem ser números binários, octais, decimais ou hexadecimais. A base de uma constante binária, octal ou hexadecimal é representada respectivamente, pelas letras B,Q ou H seguindo-a. O primeiro caracter de um número hexadecimal deve ser um dígito numérico para evitar confusão com um identificador, um zero no início é suficiente. Constantes numéricas são representadas em 16 bits. As constantes decimais não necessitam de letra alguma seguindo-as. Exemplo:

Constante binária - 11011001B

Constante octal - 331Q

Constante hexadecimal - ØD9H

Constante decimal - 217

Todas as constantes acima são equivalentes.

1.4. Declarações de tipo para variáveis .

Em um programa em PL/STI, toda variável deve ser declarada antes do seu aparecimento em qualquer comando. Elas são ou variáveis simples tipo BYTE ou ADDRESS, ou variáveis unidimensionadas (vetores). A declaração de uma variável define o seu tipo, dimensão se for o caso e dá ainda outras informações sobre ela. Mais adiante isto será visto com detalhes.

1.5. Expressões e atribuições em PL/STI .

1.5.1. Expressões :

Uma expressão PL/STI, consiste de elementos básicos de dados, combinados por meio de operadores lógicos, aritméticos ou relacionais, de acordo com a notação algébrica simples. Todos os operadores, exceto menos unário e o NOT, tomam 2 operandos do tipo BYTE ou ADDRESS e a operação é feita supon

< menor que

> maior que

<= menor ou igual

>= maior ou igual

<> não igual

= igual

Uma operação é dita verdadeira, se a relação especificada entre os seus operandos se verifica e neste caso fornece como resultado um valor de ØFFH e a operação de comparação é dita verdadeira. Se, no entanto, a comparação não é verificada, é fornecido como resultado um valor de ØØH e a operação é dita FALSA. Porém quando o resultado de uma expressão com operadores relacionais é avaliada para verificação das condições VERDADEIRA OU FALSA, apenas o último bit do resultado é testado, se for 1 (um) a expressão é dita VERDADEIRA, se for Ø (zero) a expressão é dita FALSA.

Exemplo:

6 > 4 resulta 00000000B; 5 > 3 resulta 11111111B

Qualquer expressão aritmética, mesmo as que não contêm operadores relacionais, podem ter valor verdadeiro ou falso, visto que sòmente o último bit do resultado é testado para verificação dessas condições. Isso é usado, por exemplo, no caso de um comando IF (veja seção 1.9.).

do-se que os operandos são inteiros binários, sem sinal.

Se um dos operandos é do tipo ADDRESS e o outro do tipo BYTE, este será extendido para ADDRESS, completando-se os 8 bits de mais alta ordem com zeros e a operação será realizada em 16 bits, fornecendo como resultado um valor ADDRESS, exceto no caso de operadores relacionais, que mesmo fazendo operações em 16 bits, fornecem como resultado um valor BYTE.

1.5.1.1. Operadores aritméticos .

Os operadores aritméticos são: +,-,*,/,MOD. Os operadores + e - realizam respectivamente adição e subtração entre os seus operandos. Os operadores * e / fazem respectivamente multiplicação e divisão entre dois operandos e o resultado é sempre tipo ADDRESS. No caso de ocorrer estouro na operação de multiplicação, o resultado é indefinido. O operador de divisão sempre trunca o resultado para um valor inteiro e no caso de divisão por zero, o resultado é indefinido. (A situação de hardware do 8008 para o indicador CARRY, neste caso, é indefinido).

O operador menos unário também é definido em PL/STI. Seu efeito é tal que (- A) é equivalente a (Ø - A). Assim -1 por exemplo, é equivalente a Ø - 1, resultando em um valor BYTE igual à 225. MOD realiza divisão entre seus operandos, devolvendo como resultado de operação o resto da divisão.

As operações de adição e subtração afetam o in-

dicador CARRY, que será ligado, (CARRY = 1), no caso dos operandos ocuparem ambos 1 BYTE ou 2 BYTES e o resultado não se ajustar em 1 ou 2 BYTES, respectivamente.

Por exemplo:

a) Ø 3 H - Ø 4 H

```
  00000011
  00000100
1 11111111
  CARRY = 1
```

b) Ø A E H + 74 H

```
  10101110
  01110100
1 00100010
  CARRY = 1
```

1.5.1.2. Operadores lógicos (ou booleanos) .

Há 4 operadores booleanos em PL/STI que são:NOT, AND, OR e XOR, correspondendo a negação, e lógico, ou lógico e ou exclusivo, respectivamente. NOT é um operador unário, tomado apenas um operando. O restante dos operadores realizam operações bit a bit entre os seus operandos, segundo a definição de Álgebra booleana para cada um deles. No Apêndice D, está a tabela verdade de todos os operadores lógicos.

1.5.1.3. Operadores relacionais (ou de comparação) .

Os operadores relacionais são usados em comparações e são:

1.5.1.4. Precedência dos operadores PL/STI .

Os operadores PL/STI têm uma precedência que determina a maneira como operadores e operandos estão agrupados. Os operandos são ligados aos operadores adjacentes de maior precedencia, ou da esquerda para a direita em caso de empate. Os operadores válidos em PL/STI são listados a seguir da mais alta precedência para a mais baixa, entendendo-se que os de mesma precedencia são listados na mesma linha:

MENOS UNÁRIO

* / MOD

+ -

< < = <> = >

NOT

AND

OR XOR

Parênteses são usados para sobrepor a precedenci a normal. Assim a expressão (A+B)*C fará a soma de A e B ser multiplicada por C.

1.5.2. Atribuições

Comandos de atribuições em PL/STI tem a forma:

VARIÁVEL = EXPRESSÃO;

A expressão é avaliada e o resultado é armazena-

do na variável a esquerda do sinal de igual. A precisão declarada para a VARIÁVEL afeta a operação de armazenamento: se a variável é declarada BYTE e o resultado da expressão é ADDRESS, o byte de mais alta ordem é omitido no armazenamento. Neste caso é dada uma advertência, para que o programador verifique se isso altera os resultados do seu programa. Da mesma forma, se a variável é declarada tipo ADDRESS e o resultado da expressão é BYTE, o byte de mais alta ordem é preenchido com zeros. As vezes, é necessário guardar o resultado de uma expressão em diversas variáveis, isto é possível em PL/STI, listando-se todas elas separadas por vírgulas.

Por exemplo:

```
A,B,C=X+Y;
```

Uma forma especial de atribuição é usada dentro de expressões. Essa atribuição embutida, tem a forma:

```
( VARIÁVEL: = EXPRESSÃO )
```

e pode aparecer em qualquer lugar que uma expressão é permitida.

Por exemplo:

```
A+(B:=C+D) - (E:=F/G)
```

é o mesmo que:

```
A+(C+D) - (F/G).
```

A única diferença é o armazenamento de C+D em B e F/G em E . Esses resultados parciais podem ser usados mais tarde no programa, sem calculá-los novamente.

É desaconselhável o uso de uma atribuição em uma variável que apareça em outra parte da expressão.

1.6. Comando de declaração

O objetivo de um comando de declaração é introduzir alguma entidade computacional (isto é, rótulos, rotinas ou elementos de dados), dar a ela um nome e descrever alguns de seus atributos. Declaração de rotinas será vista na seção 1.12.1. A forma mais simples de um comando de declaração é:

DECLARE identificador atributo 1, atributo 2, ...; onde os atributos são por exemplo: tipo, dimensão, valores iniciais, etc..

Seja por exemplo a declaração de um vetor:

DECLARE XY (100) BYTE;

onde XY é o nome, (100) a dimensão atribuida e BYTE o tipo de variável. Existem regras sintáticas que governam a ordem dos atributos. Todas essas regras se encontram no Apêndice A.

1.6.1. Variáveis subscritas

Às vezes é necessário referenciar cada elemento de um vetor pelo seu nome. No exemplo acima são declarados 100 elementos de dados do tipo BYTE, com nomes XY(Ø), XY(1), XY(2), até XY(99). Daí então, o seguinte comando de atribuição é válido: XY(1)=XY(2)+XY(3); onde os índices ou subscritos podem ser qualquer expressão válida em PL/STI.

Uma variável subscrita pode aparecer em qualquer

lugar onde uma variável é permitida.

1.6.2. O atributo INITIAL

Dentro de um comando de declaração, as variáveis podem ser inicializadas, usando-se o atributo INITIAL, que tem a forma:

INITIAL (lista de constantes)

onde a lista de constantes é uma sequência de constantes, separadas por vírgulas. A alocação de memória é feita como se ele não estivesse presente na declaração.

Exemplos de declarações válidas, usando o atributo INITIAL.

```
DECLARE X BYTE INITIAL (10);
DECLARE Y(10) BYTE INITIAL (1,2,3,4,5,6,7,8,9,10);
DECLARE Z(100) BYTE INITIAL ('SHORT', 'ST', ØFH);
DECLARE (Q,R,S) (10) BYTE INITIAL (Ø,1,2);
```

À última declaração cabe um comentário: foram declarados 3 vetores cada um dos quais com 10 posições, onde à Q(Ø) é atribuido o valor inicial Ø, à R(Ø) o valor 1 e à S(Ø)o valor 2.

1.6.3. A declaração DATA

Às vezes é necessário se ter um vetor com valores

iniciais que não trocam durante a execução do programa. O compilador PL/STI armazena esse vetor particular junto com o código do programa, ao invés de fazê-lo na parte da memória reservada para guardar variáveis. A linguagem PL/STI dá esse tipo de controle de alocação de memória, através da declaração DATA.

Sua forma geral é:

DECLARE identificador DATA (lista de constantes);

o efeito da declaração DATA é semelhante ao de um vetor declarado com atributo INITIAL, com algumas diferenças na forma. Nenhuma declaração de tipo de dado deve aparecer na declaração. O tipo BYTE está implícito. Também não deve aparecer nenhuma dimensão, especificando o tamanho do vetor; isto é dado implícitamente pelo comprimento da lista de constantes.

Vetores declarados com o DATA são usados como qualquer vetor tipo BYTE, com a exceção de que eles não podem nunca aparecer do lado esquerdo de um operador de atribuição.

1.6.4. Elementos de declaração

Não é necessário se ter um comando separado para cada declaração.

Por exemplo: ao invés de se escrever os comandos:

DECLARE CHR BYTE INITIAL ('A');

DECLARE X ADDRESS;

Poderíamos escrever ambas as declarações como um

simples comando, da forma:

```
DECLARE CHR BYTE INITIAL ('A'), X ADDRESS;
```

e este comando contém as duas declarações separadas por vírgulas, que são tratados como dois comandos de declaração diferentes, onde apenas a palavra reservada DECLARE não precisa ser repetida. Parênteses são usados num comando de declaração para agrupar várias variáveis que vão ser declaradas com os mesmos atributos, por exemplo:

```
DECLARE ( A,B,C ) (20) BYTE;
```

1.7. Ponteiros e Referências Indiretas

Às vezes uma referência direta a um elemento de dado PL/STI é ou impossível ou incoveniente. Isto acontece, por exemplo, quando o endereço da memória do dado permanece desconhecido até que seja computado durante o processamento. Em tais casos é necessário manipular os endereços dos dados ao invés dos próprios dados, considerando que os endereços " apontam " para os dados. Tais apontadores podem ser chamados endereços indiretos, referências ou ponteiros. Em PL/STI há facilidades para o manuseio desses ponteiros computacionais, que serão descritas a seguir.

Uma variável apontada é aquela cujo endereço de memória é dado por uma outra variável chamada a sua base. O compilador não aloca memória para variáveis apontadas, seu valor é calculado durante o processamento através de sua base.

A variável apontada é declarada primeiro declarando-se a sua base que é sempre do tipo ADDRESS e então declarando-se a própria variável apontada.

O atributo BASED indica que a variável é apontada e deve seguí-la no comando de declaração.

Exemplo:

1) DECLARE A ADDRESS, X BASED A BYTE;

2) DECLARE (ZA, YA) ADDRESS;
   DECLARE (Z BASED ZA,Y BASED YA) ADDRESS;

1.7.1. O operador Ponto

Variáveis apontadas nos dá uma maneira de se ter acesso a uma variável,dado o seu ponteiro: precisamos agora de uma maneira de construir um ponteiro dada a variável. Isto é possível por meio do operador ponto. O endereço de memória de uma variável é referenciado precedendo-se o seu nome com o caracter ponto. Assim as expressões .XY e .CARD produzem os endereços de XY e CARD respectivamente. Podemos também usar o operador ponto em uma variável com base e o resultado é simplesmente o valor da base.

O operador ponto pode preceder as seguintes construções:

a) .variável

b) .constante

c) .(constante)

d) .(lista de constantes)

Exemplos:

1).'MENSAGEM' retorna um ponteiro para o primeiro caracter,M, da cadeia de caracteres M-E-N-S-A-G-E-M

2).('CUSTO','MES',1Ø,24H) retorna um ponteiro para o primeiro caracter, C, da lista de constantes.

Uma referência a um endereço feita com o operador ponto é válida em qualquer lugar onde é permitida uma expressão PL/STI.

No caso de um operador ponto preceder uma variável (com exceção de variáveis apontadas) o endereço da variável é calculado em tempo de compilação.

1.8. Comandos rotulados e comandos de desvio

1.8.1. Rótulos simbólicos

Um comando ou um grupo de comandos podem ser rotulados para identificação e referência. A forma geral para um comando rotulado é:

RÓTULO 1 : RÓTULO 2 : ..... : RÓTULO N : comando;

onde todos os rótulos são identificadores PL/STI.

Qualquer número de rótulos pode preceder um comando. O objetivo de um rótulo simbólico é servir de alvo para um comando de desvio (que será visto mais adiante).

Rótulos podem ser declarados da mesmo forma que variáveis, em comandos de declarações. No entanto, tais declarações de rótulos nem sempre são necessárias. Isto será visto na seção 1.15.3.

1.8.2. Rótulos numéricos

Um rótulo numérico pode preceder qualquer comando PL/STI, indicando a posição de memória onde vai começar o código objeto para tal comando. Por exemplo:

```
30:Y=X+5;
```

especifica que o código objeto para este comando vai começar na posição 30 da memória.

Um comando não pode ser precedido por mais de um rótulo numérico e quando rótulos simbólicos são usados junto com um rótulo numérico no mesmo comando, o rótulo numérico deve aparecer em primeiro lugar.

1.8.3. Comandos de desvio

Um comando de desvio interrompe a sequência normal da execução do programa, transferindo o controle diretamente para o seu alvo. A execução recomeça então a partir do comando para o qual o controle do programa foi desviado. Há tres formas distintas para comandos de desvio de PL/TSI:

1) GO TO rótulo simbólico;

O rótulo simbólico é um identificador que aparece como um rótulo em um comando rotulado. O efeito desse GO TO é transferir diretamente o controle do programa para esse comando.

2) GO TO número;

Onde o número é um endereço absoluto de memória e o controle do programa é transferido diretamente para esse endereço.

3) GO TO nome de variável;

Neste caso a variável contém um endereço de memória pré-computado e o controle passa diretamente para esse endereço absoluto de memória.

A palavra reservada GO TO pode também ser escrita como GOTO ou simplesmente GO.

1.9. O comando IF

Forma geral

IF EXPRESSÃO THEN comando 1; ELSE comando 2; comando 3;

Este comando tem o seguinte efeito: primeiro a expressão seguinte ao IF é avaliada. Se o resultado é VERDADEIRO (conforme visto na seção 2.6.1.3.) o comando 1 é executado, em caso contrário, o comando 2 é executado.

Depois da execução de um dos comandos, o controle do programa passa para o comando 3 que segue o IF. O comando IF apresenta uma restrição quanto ao seu uso, que será vista na seção 1.11.

Os comandos que seguem as palavras reservadas THEN e ELSE, respectivamente,não podem ser rotulados. Cabe ainda ressaltar que a parte ELSE de um comando IF é opcional.

1.10. Os comandos compostos

Forma geral

```
< definição do comando composto >
                comando 1
                comando 2
                comando 3
                   ⋮
                comando N
        END;
```

onde, seguindo a definição da Gramática, (ver Apêndice A), <definição do comando composto> especifica qual o comando composto que vai ser usado.

1.10.1. O comando composto DO

Comandos podem ser agrupados entre as palavras reservadas DO; END; para formar um único comando, com a forma:

```
DO;
comando 1
comando 2
   '
   '
   '
comando N
END;
```

onde não há restrições quanto aos comandos que aparecem entre as palavras reservadas DO;END;

1.10.2. O comando composto DO-WHILE

Forma geral

```
DO WHILE EXPRESSÃO;
     comando 1;
     comando 2;
        '
        '
     comando N;
END;
```

O efeito deste comando é: primeiro a expressão seguinte à palavra reservada WHILE é avaliada para verificação das condições VERDADEIRA ou FALSA (ver seção 1.5.1.3). Se o

resultado é VERDADEIRO, então a sequência de comandos até o END é executado. A seguir a expressão é novamente avaliada e se o resultado é VERDADEIRO novamente os comandos são executados. Esse procedimento se repete até que o resultado da expressão seja FALSO, quando então o controle do programa passa ao comando seguinte ao grupo DO-WHILE.

Seja por exemplo o seguinte trecho de programa:

```
A=1;
DO WHILE A<= N;
(1) - - -C=C+A;
(2) - - -A=A+1;
END;
```

Os comandos (1) e (2) serão executados N vezes. O valor de A será igual a N+1, quando o controle do programa deixar o ciclo.

1.10.3. O comando composto DO-ITERATIVO

Forma geral

```
DO variável = expressão 1  TO expressão 2 BY
              expressão 3;
    comando 1;
    comando 2;
       .
       .
       .
    comando N;
END;
```

Consideremos o seguinte trecho de programa:

```
        VAR=EXP1;
TESTE:IF  VAR > EXP2   THEN  GO TO  CONTINUA;
        comando 1;
        comando 2;
           '
           '
           '
        comando N;
        VAR=VAR + EXP3 ;
        GOTO TESTE;
CONTINUA:
```

onde EXP3 > 0 e no caso de EXP3 < 0, o operador relacional da expressão seguinte a palavra reservada IF, muda de sentido,ficando então:

```
VAR < EXP2
```

O trecho de programa acima, pode ser substituido pelo comando composto DO-ITERATIVO que funciona como o exemplo e seria:

```
DO VAR = EXP1  TO EXP2  BY EXP3 ;
          comando 1;
          comando 2;
             '
             '
             '
          comando N;
     END;
```

1.10.4. O comando composto DO-CASE

Forma geral

```
DO CASE EXPRESSÃO;
     comando 1;
     comando 2;
        .
        .
        .
     comando N;
END;
```

O efeito desse comando é primeiro a avaliação da expressão seguinte ao CASE. O resultado deve ser um valor K entre Ø(zero) e N-1. K é usado então para selecionar um dos N comandos do DO-CASE. O primeiro comando corresponde a K=Ø, o segundo a K=1 e assim consecutivamente até o último comando corresponde a K=N-1.

Depois da execução do comando selecionado, o controle do programa passa ao comando seguinte a esse comando composto. Se durante o processamento o valor de K é maior que o número total de comandos, N, então o efeito deste DO CASE é indeterminado , sendo portanto erro de programação. Há uma restrição quanto aos comandos que aparecem no corpo de um DO-CASE: eles não podem ser rotulados.

Exemplo:

```
DO CASE  X-5
 X=X+5;      / * CASO Ø * /
  DO;       /  * CASO 1 */
  X=X+10;
  Y=X-3;
  END;
END;
```

Este exemplo ilustra o uso de blocos DO-END para agrupar vários comandos como um único comando PL/STI.

1.11. Restrição ao uso de comando IF

Vejamos de novo a sua forma geral:

IF expressão THEN comando 1;ELSE comando 2; comando 3.

A restrição se resume no seguinte: o comando ligado a cláusula IF, comando 1, não pode nunca ser um comando IF, a não ser que não exista nenhum ELSE comando 2;

A construção : IF condição 1 THEN IF condição 2 THEN comando 3; ELSE comando 2; é ambígua e ilegal (a qual IF o ELSE pertence ?) e deve ser substituido por uma das seguintes construções, dependendo da inteção de quem programa.

```
1)IF condição 1 THEN
      DO;
      IF condição 2 THEN comando 3;
      END;
  ELSE  comando 2;
```

```
2) IF condição 1 THEN
       DO;
       IF condição 2 THEN comando 3;
                     ELSE comando 2;
       END;
                     conforme o caso.
```

1.12. Rotinas

Uma rotina é uma parte do código PL/STI que é declarada sem ser executada e então chamada de outros pontos do programa.

O uso de rotinas facilita a programação e a documentação, reduzindo a quantidade de código objeto gerado pelo programa.

1.12.1. Declaração de rotinas

Uma rotina deve ser declarada no programa, antes de aparecer qualquer comando executável. Uma declaração de rotina consiste de quatro partes:

a) o nome da rotina - é um identificador PL/STI que é associado com a rotina.

b) a especificação dos parâmetros formais existentes, onde um parâmetro formal é um identificador PL/STI que

toma um valor passado para a rotina do seu ponto de chamada.

c) o tipo do valor retornado (se a rotina retorna algum valor), que deve ser BYTE ou ADDRESS.

d) o corpo da rotina (o próprio código), que é formado por quaisquer comandos PL/STI, inclusive chamadas e declarações aninhadas de rotinas.

e) END NOME; onde NOME é opcional.

Estes elementos tomam a seguinte forma:

```
NOME: PROCEDURE (lista de parâmetros formais)TIPO;
      comando 1;
      comando 2;
         '
         '
         '
      comando N;
   END NOME;
```

onde a lista de parâmetros formais tem a forma:(id1,id2,...,idn) e id1,id2,idn são identificadores PL/STI. Todos os parâmetros formais devem ser declarados dentro da rotina de modo que seus tipos sejam definidos. A lista de parâmetros deve ser omitida se nenhum parâmetro passa para a rotina. Da mesma forma se a rotina não retorna um valor, então o tipo é omitido na declaração da mesma.

1.12.2. O comando RETURN

A execução da rotina termina pela execução do comando RETURN dentro do corpo da mesma. Este comando tem uma das duas formas:

a) RETURN; que é usado se a rotina não retorna um valor.

b) RETURN EXPRESSÃO; que é usado se retorna um valor. E nesse caso o valor da expressão é trazido para o ponto de chamada.

1.12.3. Exemplos de declarações de rotinas

```
1) AVG:PROCEDURE (X,Y) ADDRESS;
       DECLARE (X,Y) ADDRESS;
       RETURN (X+Y)/2;
   END AVG;
```

Como retorna um valor, o tipo ADDRESS foi declarado e o comando RETURN é seguido por uma expressão.

```
2) AOUT:PROCEDURE (ITEM);
       DECLARE ITEM ADDRESS;
       IF ITEM = ØFFH THEN I=I+1;
       ELSE I=I+3;
       RETURN;
   END AOUT;
```

Neste caso a rotina não retorna um valor, logo o tipo foi omitido da declaração e o comando RETURN poderia ser omitido, pois há um RETURN implícito no END de qualquer rotina.

1.12.4. Restrição quanto ao uso de rotinas

Há uma restrição quanto ao uso, que é:

Rotinas não podem ser recursivas, isto é, uma rotina não pode chamar ela mesma, nem chamar uma a outra circularmente.

1.12.5. Chamadas de rotinas

Há duas formas de chamadas de rotinas:

a) se a rotina não retorna um valor , a chamada é feita através do comando CALL, que tem a forma:

CALL nome da rotina (lista de parâmetros atuais);

onde a(lista de parâmetros atuais) contém nome de variáveis, constantes ou qualquer expressão PL/STI, separadas por vírgulas.

No tempo de chamada cada parâmetro atual ou parâmetro de chamada é avaliado e seu valor atribuido ao correspondente parâmetro formal da declaração da rotina. Parâmetros das rotinas PL/STI são do tipo "chamada por valor". Parâmetros de chamada devem ainda corresponder em número e tipo aos parâmetros da declaração da rotina e se houver divergência de tipo entre eles será feita uma conversão para o tipo do parâmetro formal, no ponto de chamada.

b) se a rotina retorna um valor , então a sua forma de chamada é:

nome da rotina (lista de parâmetros atuais) que é um operando primário ou termo a ser usado em uma expressão do mesmo modo que o nome de uma variável é usada.

1.12.6. Exemplos de chamadas de rotinas

Dadas as declarações de rotinas, na seção 1.12.3, para AOUT e AVG, as seguintes chamadas são válidas para essas rotinas:

```
1) X=AVG(X,4);
2) CALL AOUT(X);
3) CALL AOUT(1+AVG(X,Y));
4) DO WHILE AVG(X,Y) < MAX;
      X=X+XDEL;
   END;
```

1.13. Os comandos HALT e EOF

1.13.1. O comando HALT

Forma geral

```
HALT;
```

Este comando indica o final do processamento do

programa objeto.

1.13.2. O comando EOF

Forma geral

EOF

Indica o fim da compilação do programa fonte, deve ser o último comando do programa.

1.14. Macros em tempo de compilação

O programador pode declarar um nome simbólico como sendo equivalente a uma cadeia (ou sequência) de caracteres. Quando uma ocorrência do nome é encontrada pelo compilador, a cadeia de caracteres declarados é substituida. Dessa forma o compilador processa a sequência de caracteres substituidos ao invés do nome simbólico.

1.14.1. A declaração LITERALLY

Define uma macro para expansão em tempo de compilação.

Sua forma geral é:

DECLARE identificador LITERALLY 'CADEIA DE CARACTERES';

onde o identificador é qualquer identificador PL/STI que é associado a cadeia de caracteres com no máximo 255 caracteres arbitrários da linguagem.

1.14.2. Exemplo de uso de macros

Consideremos os seguintes trechos de programa.

```
DECLARE LIT'LITERALLY',DCL LIT'DECLARE';
DCL  TRUE LIT 'ØFFH',FALSE LIT 'Ø';
DCL  FOREVER LIT 'WHILE TRUE';
DCL  (X,Y,Z) BYTE;
X=TRUE;
   '
   '
   '
DO FOREVER;
 Y=Y+1;
 IF Y > 1Ø THEN HALT;
END;
   '
   '
   '
EOF
```

A primeira declaração deste programa define abreviações para as palavras reservadas LITERALLY e DECLARE,que

são então usados através do programa.

A segunda declaração define os valores booleanos TRUE e FALSE do mesmo modo como PL/STI manuseia operadores relacionais. Isto torna o programa mais legível.

1.15. Estruturas de Blocos e Alcance

PL/STI é uma linguagem estruturada em blocos. Um bloco é qualquer comando composto, qualquer rotina ou o programa inteiro. Todas as entidades computacionais declaradas dentro de um bloco, são inacessíveis a comandos ou declarações fora dele. O uso de um mesmo identificador para diferentes objetivos, bem como o uso de um bloco dentro de outro não criam nenhuma dificuldade.

1.15.1. Como o Alcance é definido

Cada bloco limita o alcance dos identificadores declarados dentro dele; eles são desconhecidos fora do bloco. O alcance de um identificador começa com a sua declaração e termina com o fim do bloco. Nome de variáveis, macros, vetores, dados e rotinas têm alcance cujas regras são as explicadas anteriormente. Há, no entanto, uma restrição a ser feita: comandos de declaração e declaração de rotinas não podem aparecer dentro de um DO WHILE, DO CASE ou DO iterativo.

1.15.2. Alcance de rótulos

Rótulos são também identificadores e como tal, têm alcance. No entanto, normalmente não é necessário declarar o rótulo explícitamente. O primeiro uso de um rótulo não declarado (em um comando rotulado ou comando de desvio) contém uma declaração implícita do rótulo e desse modo essa declaração governa o alcance do rótulo, de acordo com as regras da seção precedente.

1.15.3. Declaração de rótulo

As vezes torna-se conveniente declarar o rótulo para passar por cima do alcance implícito. Esta declaração toma a forma:

```
DECLARE identificador LABEL;
DECLARE (identificador1,.....,identificadorN)LABEL;
```

tais declarações especificam que o rótulo ou coleção de rótulos, será definido ao nível do bloco da declaração. Esta declaração explícita é necessária sòmente se a declaração implícita não satisfaz as intenções do programador.

Consideremos os seguintes trechos de programas como exemplo:

EXEMPLO(1):

```
    X=X+1
      '
      '
     DO;
      '
      '
     GO TO EXIT;
      '
      '
    END;
    EXIT:HALT;
    EOF
```

EXEMPLO(2):

```
     X=X+1;
       '
       '
       '
     DO;
       '
       '
     DECLARE EXIT LABEL;
     GO TO EXIT;
       '
       '
       '
    END;
       '
       '
       '
    DECLARE EXIT LABEL;
     EXIT : HALT;
    EOF
```

Nossa intenção óbvia em (1) é desviar para o comando rotulado EXIT no fim do programa. Mas de acordo com as regras de declaração implícita para rótulos, o que nós escrevemos é equivalente a (2).

A declaração implícita limita o alcance do rótulo ao comando composto. Assim na 2ª. ocorrencia de EXIT nós estaremos fora daquele alcance, EXIT é novamente definido, e uma nova declaração implícita ocorrerá. Assim temos 2 rótulos diferentes, devido às declarações implícitas, um interno ao bloco e outro externo a ele. Desse modo o comando GO TO não tem um alvo a atingir.

Para satisfazer o propósito original, o programa teria que ser escrito do seguinte modo:

```
DECLARE EXIT LABEL;
X=X+1;
   '
   '
 DO;
   '
   '
  GO TO EXIT;
   '
   '
END;
   '
   '
EXIT:HALT;
EOF
```

As declarações implícitas são suprimidas. Elas não são necessárias pois existe um rótulo EXIT, cujo alcance

agora é o programa inteiro sem restrições.

1.15.4. Uso da estrutura de blocos

Transferência de controle de dentro do corpo de uma rotina só é possível através do comando RETURN, do mesmo modo a entrada em uma procedure só é possível através de uma chamada por meio de um comando CALL, ou pelo próprio nome da rotina em uma expressão como foi visto na seção 1.12.5. Não é permitido também ter desvios de um bloco mais externo para um mais interno.

Estrutura de blocos em uma linguagem de programação, dá a oportunidade de definir módulos de programa bastante independentes, deixando para o compilador a tarefa de juntá los.

1.16. Funções internas (ou embutidas)

São funções supridas pelo compilador PL/STI que para serem usadas basta que sejam indicadas pelo seu nome.

Chamadas para todas as funções embutidas podem aparecer em qualquer lugar que uma expressão é permitida.

1.16.1. Funções LENGTH e LAST

São baseadas nos tamanhos declarados para vetores, e tem a forma:

LENGTH (identificador) - dá o comprimento declarado para o identificador.

LAST (identificador) - dá o índice do elemento final do identificador onde identificador é qualquer nome de variável, vetor ou dado, préviamente declarado.

Para um identificador qualquer VAR,LENGTH(VAR)==1+LAST(VAR).

LENGHT é definida para qualquer variável, mas LAST não é definida para variáveis simples, pois estas tem comprimento zero.

1.16.2. As funções LOW,HIGH,DOUBLE

As duas funções internas seguintes, convertem valores ADDRESS para BYTE. Ambas tomam como argumentos valores ADDRESS e tem a forma:

LOW(expressão) - retorna o byte de mais baixa ordem de seu argumento.

HIGH(expressão)- retorna o byte de mais alta ordem de seu argumento.

Um terceiro tipo de rotina de conversão, converte um valor BYTE para um ADDRESS, preenchendo o byte de mais alta ordem com zeros.

Sua forma de chamada é: DOUBLE (expressão).

1.16.3. Função BYTE de rotação

Chamadas para as duas funções ROL e ROR tomam a forma:

ROL (exp.1,exp.2)

ROR (exp.1,exp.2)

onde exp.1 e exp.2 devem resultar em uma quantidade BYTE e exp. 2 deve ser sempre diferente de zero. Ambas as funções retornam um valor BYTE.

ROL faz rotação em exp.1 para a esquerda e exp. 2 dá o número de bits a serem rodados em exp.1.

ROR retorna a correspondente rotação para a direita.

Seja o exemplo:

ROR(10011101B ,1) retorna o valor 11001110B

ROL(10011101B ,2) retorna o valor 01110110B

ROL e ROR tem o efeito secundário de deixar no indicador CARRY o último bit que saiu na rotação. No primeiro exemplo CARRY será ligado (isto é, CARRY=1), no segundo exemplo, CARRY é desligado (isto é, CARRY=0).

1.16.4. Funções de rotação CARRY

Chamadas para essas funções tomam a forma:

SCL (exp.1,exp.2)

SCR (exp.1,exp.2)

onde exp.2 deve resultar em uma quantidade BYTE, diferente de zero e exp.1 pode ser um valor BYTE ou ADDRESS, daí então o valor retornado será BYTE ou ADDRESS respectivamente.

O primeiro parâmetro (exp.1) é rodado para a esquerda (SCL) ou para a direita (SCR) de acordo com o contador dado por exp.2.

O bit que sai fora na rotação entra no bit CARRY e o valor antigo do bit CARRY entra na outra extremidade.

Por exemplo:

Vamos supor que o bit CARRY é zero.

SCL (10011101B,1) retorna o valor:00111010B e CARRY=1

SCR (10011101B,2) retorna o valor:10100111B e CARRY=0

Os mesmos princípios servem para valores de exp. 1 com 16 bits.

1.16.5. Funções de Shift-lógico

Chamadas para essas duas funções SHL e SHR tomam a forma:

SHL(exp.1,exp.2)

SHR(exp.1,exp.2)

onde exp.2 é tipo BYTE e sempre diferente de zero e exp.1 pode ser BYTE ou ADDRESS, retornando então respectivamente um valor BYTE ou ADDRESS.

O primeiro argumento (exp.1) é deslocado para direita (SHR) ou para esquerda (SHL) de acordo com o contador de bits dado pelo segundo argumento (exp.. 2). Os bits deslocados para a direita ou para a esquerda são deslocados para o bit CARRY enquanto zeros ocupam os bits que ficaram vazios. O valor anterior do bit CARRY é perdido.

Seja por exemplo:

SHL (10011101B,1) - retorna o valor 00111010B e CARRY=1;

SHR (10011101B,2) - retorna o valor 00100111B e CARRY=0;

1.16.6. Funções CARRY,ZERO,SIGN,PARITY

São usadas para testar os códigos de condição da CPU 8008.

Suas chamadas são respectivamente:

CARRY

ZERO

SIGN

PARITY

Uma ocorrência desses identificadores em uma expressão, gera um teste do correspondente indicador de condição. Se o indicador estiver ligado (=1), o valor retornado é ØFFH. Se o indicador estiver desligado, então o valor Ø é retornado.

1.17. Entrada e Saída

As instruções de entrada e saída de dados para a CPU 8ØØ8 tem a seguinte forma:

ØlRRMMMl

onde: RR = ØØ , para entrada

e

$$RR = \begin{cases} Ø1 \\ 1Ø \\ 11 \end{cases} \text{, para saída}$$

Ao ser dada esta instrução, o campo MMM definirá uma das 8 possíveis funções de cada grupo RR a ser interpretada pelo Sistema de Entrada e Saída.

Para uma instrução de entrada, é colocado no acumulador, o que se encontra na barra de dados.

Para uma instrução de saída, o conteúdo do acumulador é transferido para a barra de dados.

Entrada e saída de dados em PL/STI é feita através das funções INPUT e OUTPUT, respectivamente.

1.17.1. INPUT

Forma geral : INPUT (número)

É usada em uma expressão, exatamente como uma chamada de qualquer rotina tipo BYTE. Seu valor é uma quantidade BYTE, que será o valor presente na entrada da CPU.

O argumento é uma constante numérica que deve estar entre os limites Ø-7, correspondendo ao campo MMM da instrução de máquina de entrada, gerada por um INPUT.

1.17.2. OUTPUT

A pseudo-variável OUTPUT sempre aparece como a parte esquerda de um comando de atribuição. Em particular, ela nunca pode aparecer como o destino de um comando de atribuição embutido. Sua forma geral é:

OUTPUT (número) = expressão;

onde o argumento é uma constante numérica entre os limites Ø-23, dada pelo valor do campo, RRMMM-8, da instrução de máquina de saída que é gerada.

CAPÍTULO II

O COMPILADOR PL/STI

2.1. Descrição Geral

Um compilador é entendido como um programa que traduz uma linguagem fonte, em sua correspondente linguagem objeto, que pode ser linguagem de máquina ou linguagem simbólica de um determinado computador.

Durante as fases de definição e execução do projeto para o compilador PL/STI, optamos sempre pelas soluções mais simples. Por isso decidimos por um compilador de um passo, gerando código objeto absoluto em linguagem de máquina.

O compilador PL/STI dá como saída, se não houver erros de compilação, um conjunto de cartões perfurados, a ser carregado e executado no Terminal Inteligente. Possui ainda como opções de saída, a listagem do programa fonte com mensagens de erros, se houver, a listagem do programa objeto e também a listagem dos atributos do programa (tamanho, tabela de símbolos e alocação).

O compilador foi totalmente programado em Algol Extendido do Burroughs/6700.

2.2. As fases do Compilador PL/STI

O compilador PL/STI é constituído das seguintes fases: Análise Léxica, Análise Sintática, Preparação para Geração de Códigos e Geração de Códigos propriamente dita.

2.2.1. Análise Léxica

À cada chamada da Análise Sintática, a Análise Léxica devolve um elemento da linguagem PL/STI.

A Análise Léxica tem por finalidades:

1 - Ler o programa fonte.

2 - Gravar o programa fonte em um arquivo para posterior listagem do mesmo no fim da compilação.

3 - Suprimir sinais de cifrão e caracteres inválidos, neste caso dando uma mensagem de erro apropriada.

4 - Reconhecer os elementos da linguagem que são: identificadores, constantes numéricas, cadeias de caracteres, delimitadores de 1 caracter (* , : ; / ( ) . + = ' < - >) e delimitadores de 2 caracteres (/* < = > = < > := */).

5 - Suprimir comentários.

6 - Padronizar a entrada da Análise Sintática em triplas do tipo:

o tipo do elemento	o próprio elemento	atributos

entre os atributos podemos citar: o valor da constante numérica, o número de caracteres de uma cadeia e a posição do identificador na Tabela de Simbolos, se ele estiver inserido.

7- Fazer processamento de macros.

8- Consultar a Tabela de Símbolos, verificando se o identificador está inserido.

9- Identificar erros léxicos.

Como podemos verificar, a Análise Léxica foi estruturada com o objetivo de facilitar certas tarefas para o Analisador Sintático sem, no entanto, se tornar muito complexa.

2.2.2. Análise Sintática

O compilador PL/STI é totalmente controlado pela Análise Sintática. As outras fases da compilação são por ela ativadas.

À cada chamada da Análise Léxica, a Análise Sintática tenta enquadrar nas regras gramaticais da linguagem PL/STI, o elemento devolvido. Quando não o consegue dá uma mensagem de erro adequada. Cada vez que uma construção gramatical

é por ela reconhecida, são feitos os procedimentos semânticos convenientes, entre os quais colocar informações na Tabela de Símbolos (ver seção 3.8 onde são dados detalhes sobre ela) , consultar a Tabela de Símbolos para verificação de erros, ou chamar as rotinas para gerar códigos. A Análise Sintática será vista com detalhes no Capítulo III.

2.2.3. Preparação para Geração de Códigos

Como o compilador PL/STI é de um passo, esta fase contitui-se basicamente da alocação de memória para as variáveis declaradas, que é um procedimento feito dentro da Análise Sintática, mas que faz parte do outro trabalho sobre o compilador.

2.2.4. Geração de Códigos

O compilador PL/STI, gera o programa objeto em código de máquina absoluto, (isto significa que ele vai ser montado em posições fixas da memória), por meio de rotinas que são chamadas pela Análise Sintática, cada vez que uma construção gramatical é reconhecida.

Estas fases são realizadas em paralelo, de modo intercalado, pois o compilador PL/STI é de um só passo.

2.3. O processo de Compilação da linguagem PL/STI

A Análise Sintática chama a Análise Léxica, sempre que necessita um novo elemento para continuar analisando o programa fonte de acordo com a gramática da linguagem PL/STI. Quando uma construção gramatical é reconhecida, são feitos os procedimentos semânticos convenientes, entre os quais, se for o caso, é chamada uma rotina que gera códigos para a construção. A seguir continua a análise do programa fonte como descrito anteriormente, até que seja reconhecido pela Análise Sintática o comando EOF, marcando o fim da compilação do programa PL/STI. No caso de faltar o comando EOF, a Análise Léxica vai tentar ler um novo cartão que não existe, pois todos os cartões do programa já foram lidos e analisados. Então é dada uma mensagem de erro e a compilação do programa fonte termina.

A figura Nº 1 representa a ligação lógica entre as fases do compilador PL/STI.

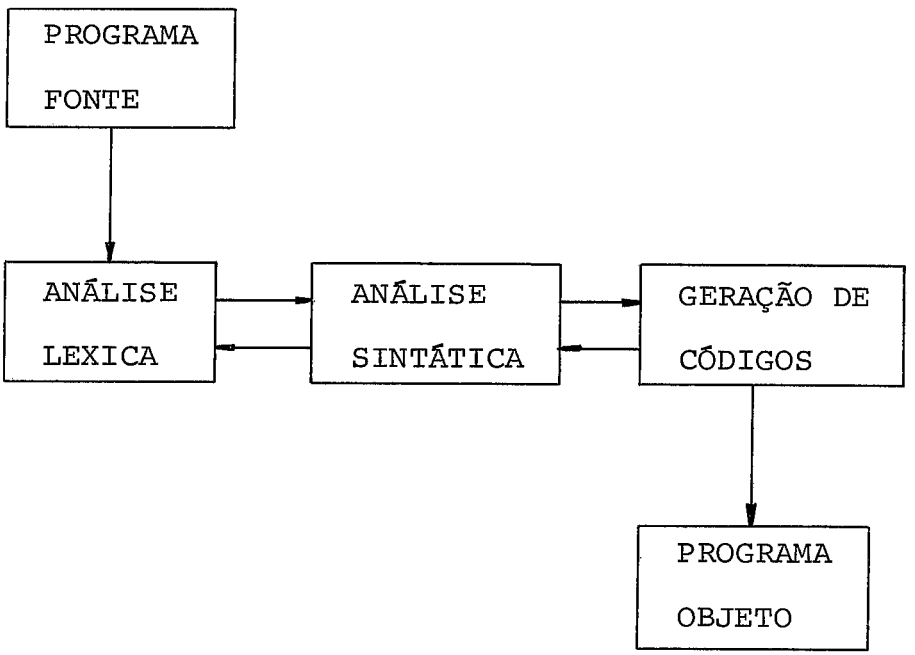

PROGRAMA
FONTE
ANÁLISE
LEXICA
ANÁLISE
SINTÁTICA
GERAÇÃO DE
CÓDIGOS
PROGRAMA
OBJETO

Figura 1

CAPÍTULO III

A ANÁLISE SINTÁTICA

3.1. Introdução

A Análise Sintática de um programa PL/STI, tem como objetivo determinar a estrutura formal do programa fonte, dada a gramática da linguagem. Uma vez que uma contrução gramatical é reconhecida, certas convenções semânticas da linguagem fonte são testadas pela Análise Sintática, como foi visto na seção 2.2.

Neste capítulo damos a idéia geral do método escolhido para fazer a Análise Sintática e o modo como ele foi implementado.

Focalizamos também a Tabela de Símbolos, que é usada durante toda a Análise Sintática, e o tratamento para erros.

Detalhamos com algoritmos os procedimentos mais interessantes. Nesses algoritmos usamos variáveis inteiras ou booleanas, que não foram especificadas como tal, pois isso se torna claro no procedimento onde elas ocorrem.

3.2. Método usado para a Análise Sintática

Usamos o método TOP-DOWN dos descendentes recursivos, no qual um não terminal U, tem uma rotina, que é recursiva quando isto se faz necessário, para analisar frases de U. Por exemplo: seja o não terminal < comando de declaração >. Entre muitas outras, ele pode derivar as seguintes frases:

<comando de declaração>::=DECLARE<elementos de declaração>

::=DECLARE<declaração de tipo>

::=DECLARE<especificação do identificador> <tipo>

::=DECLARE<nome da variável> <tipo>

::=DECLARE<identificador> <tipo>

::=DECLARE A <tipo>

::=DECLARE A BYTE;

A análise é feita sem volvtar atrás, isto porque associamos procedimentos semânticos às regras gramaticais. O único contexto necessário para a análise é o símbolo seguindo a frase parcialmente analisada. Uma rotina encontra a frase correspondente, comparando o programa fonte, no ponto indicado, com as partes à diretia das regras gramaticais para U, chamando outras rotinas para reconhecer frases intermediárias, quando isto se torna necessário. Com base na gramática da linguagem PL/STI, construímos diagramas sintáticos (Apêndice E) a fim de facilitar a programação dessas rotinas, que correspondem, em suma, à compilação de cada comando da linguagem.

3.3. Vista geral do procedimento para a Análise Sintática

O método descrito anteriormente é controlado por uma rotina (rotina de controle), que contem as chamadas para as rotinas que analisam cada comando da linguagem PL/STI.

Todo comando, exceto comando de atribuição ou comando rotulado, começa com uma palavra reservada. Durante a Análise Léxica, cada vez que um identificador é reconhecido, a tabela de símbolos é consultada. No caso do identificador estar inserido a posição do mesmo na tabela de símbolos é devolvida pela Análise Léxica. Como as posições ocupadas pelas palavras reservadas e nomes de funções internas são conhecidas,pois a tabela de símbolos é preenchida inicialmente com todas as palavras reservadas e nomes de funções nas primeiras posições, este valor devolvido pela Análise Léxica é usado pela rotina de controle para chamar a rotina que analisa o comando correspondente.

No caso do identificador ser um rótulo ou uma variável à esquerda do sinal de atribuição, a posição da tabela de símbolos correspondente a ele não se encontra nos limites de palavras reservadas e nomes de funções. Então a rotina de controle chama a Análise Léxica para devolver o elemento que segue o identificador. Se o elemento devolvido não é igual a dois pontos (:), é chamada a rotina que analisa comandos de atribuição. Em caso contrárito, é chamada a rotina que faz procedimentos semânticos para rótulos e depois então o programa volta pa-

ra o início da rotina de controle a fim de analisar o comando precedido pelo rótulo.

3.3.1. Algoritmo para a rotina de Controle

No algoritmo abaixo, T.S. é abreviação usada para tabela de símbolos.

```
procedure controle;

begin
while  não encontrou EOF do
 begin
   Análise Léxica;
   if  elemento=identificador then
                              begin
                              if posição do identificador
                              na T.S.> última posição de
                              palavras reservadas e nomes
                              de funções
                               then  índice:= 12
                               else
                               if  posição do identifica
                               dor na T.S.=posição de pa
                               lavra reservada e nome de
                               função
```

```
                                    then  índice:=posição do iden
                                    tificador na T.S. else erro;
                                   end
                              else if elemento=constante numérica
                                   then índice:=12 else erro;
case  índice of begin
 0: chamada de rotina que compila comando CALL;
 1: chamada de rotina que compila comando de declaração;
 2: chamada de rotina que compila comando composto;
 3: compilação do comando EOF;
 4: compilação do comando END;
 5: chamada de rotina que compila comando de desvio;
 6: chamada de rotina que compila comando de desvio;
 7: compilação do comanto HALT;
 8: chamada de rotina que compila comando IF;
 9: chamada de rotina que compila rotinas PL/STI;
10: chamada de rotina que compila comando RETURN;
11: compilação da pseudo-variável OUTPUT;
12: begin
    Análise Léxica;
    if elemento = ":"  then begin
                            chamada de rotina que faz procedi-
                            mentos semânticos para rótulos;
                            end
                       else chamada de rotina  que compila co-
                        mando de atribuição;
    end
   end do case índice;
   end do while;
   end da rotina controle;
```

Como podemos observar o algoritmo anterior contém duas chamadas seguidas para rotina que compila comandos de desvio. Isto se torna necessário porque um comando de desvio começa por uma das palavras reservadas GO ou GOTO, que ocupam posições diferentes na tabela de símbolos.

3.4. Compilação de um comando de declaração

Pela definição da linguagem PL/STI:

<comando de declaração>::=DECLARE<elementos da declaração>
|<comando de declaração>,<elementos da declaração>

Isto significa que podemos ter, por exemplo, um comando de declaração da seguinte forma:

DECLARE A BYTE, B(20)ADDRESS,C(2)INITIAL(0,1);

como foi visto na seção 1.6.4.

A compilação desse comando se resume então em:

1 - A rotina de controle identifica a palavra reservada DECLARE e desvia para o ponto de chamada da rotina DECLARE que compila o comando;

2 - Aí são compilados os <elementos da declaração> que se forem seguidos por uma vírgula, acarretará em uma chamada recursiva da rotina DECLARE.

O algoritmo abaixo descreve esse procedimento.

3.4.1. Algoritmo de compilação para um comando de declaração

```
procedure  declare;

begin
   begin
   compilação dos
   <elementos da declaração>;
   end;
   Análise Léxica;
   if elemento "," then declare;
                   else if elemento ≠ ";" then erro;
   end da rotina declare;
```

Durante a compilação dos <elementos da declaração> são feitos procedimentos semânticos tais como inserir um idendificador na tabela de símbolos juntamente com seus atributos, consultar a tabela de símbolos para verificação de erros e alocar memória para as variáveis.

3.5. Compilação dos blocos de um programa PL/STI

Como foi visto na seção 1.15, uma rotina ou um comando composto formam um bloco da linguagem PL/STI. Vejamos a forma geral das suas respectivas definições, para verificar-

mos o que há de comum entre eles:

```
<definição do comando composto>        <declaração da rotina>
        comando 1;                             comando 1;
        comando 2;                             comando 2;
           '                                      '
           '                                      '
           '                                      '
        comando n;                             comando n;
END;                                   END;
```

Observando-se a forma geral em ambos os casos, verificamos que eles englobam outros comandos na sua definição. Daí a razão pela qual existe um procedimento comum para as rotinas que os compilam e que se resume no seguinte:

1 - A rotina de controle, desvia para o ponto de chamada da rotina que compila o bloco.

2 - Depois de analisada a <definição do comando composto> ou a <declaração da rotina>, a rotina que compila o bloco fica chamando a rotina de controle recursivamente, para que sejam compilados os comandos que aparecem no corpo do bloco, até que o "END;" seja encontrado.

A figura 2 esclarece este procedimento:

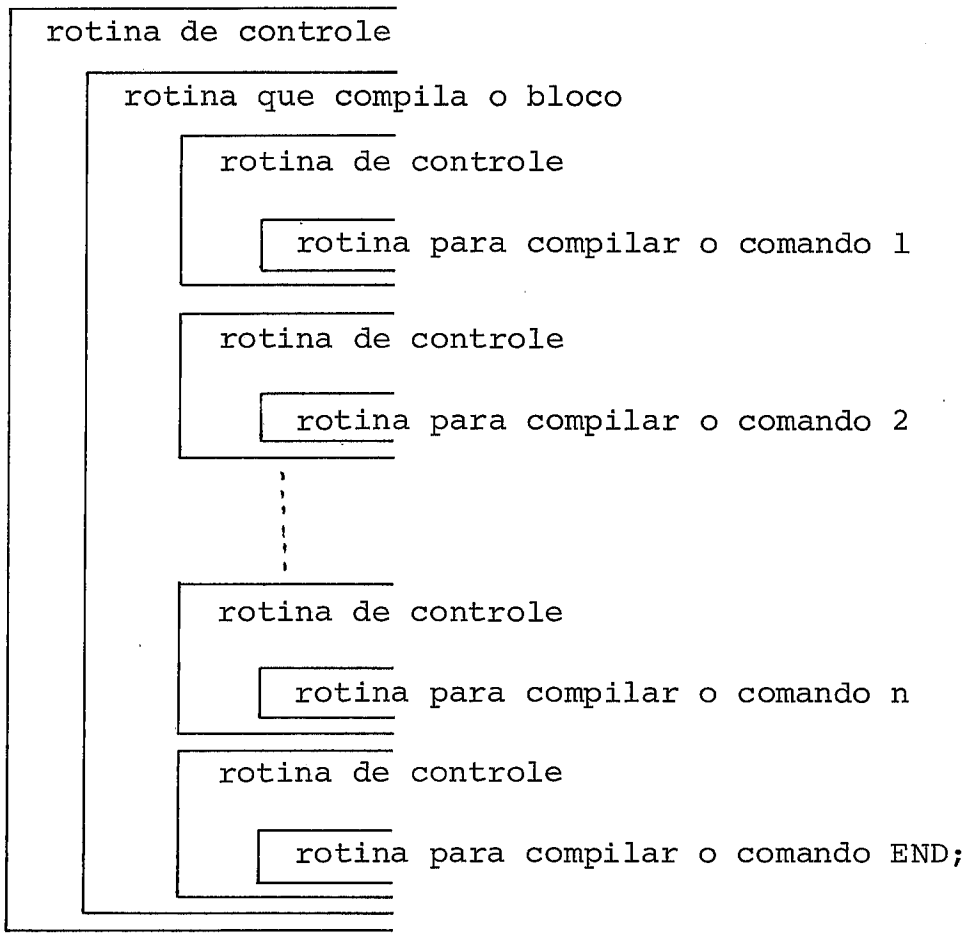


figura 2

3.6. Compilação de uma expressão PL/STI

Observando-se os diagramas sintáticos para expressões, verificamos que depois de um operador aritmético, lógico ou relacional, temos uma contrução gramatical que para ser analisada precisamos voltar ao início do diagrama.

Na rotina que analisa expressões usamos essa característica como idéia básica, isto é, depois de um operador,

voltamos ao início da rotina para continuar a análise.

Para efeito de programação da rotina que compila expressões PL/STI, associamos aos elementos que podem começar uma expressão, os inteiros:

0 - para identificadores;

1 - para constantes numéricas;

2 - para elementos precedidos pelo operador ponto (.cadeia de caracteres, .identificador,.(lista de identificadores)).

3 - para o delimitador abre-parênteses;

4 - para o elemento menos unário;

5 - para cadeia de caracteres;

6 - para o operador "NOT".

O algoritmo seguinte esclarece como foi feita a programação desta rotina.

3.6.1. Algoritmo para compilação de uma expressão PL/STI

```
procedure expression;

begin
 chave:=true ;
 Análise Léxica; Comment devolve sempre um elemento da
 linguagem;
 while chave do
 begin
  if elemento ≠ de elemento que começa uma expressão
```

```
then begin
      erro; Comment a sintaxe da expressão está errada.
      A rotina de erro procura ";";
      chave:= false ; comment liga a variável chave pa-
                      ra sair do laço e a rotina expression
                      volta para o ponto de chamada
      end
 else begin
         expr= número inteiro correspondente ao elemen-
               to que começa a expressão;
         case expr of begin
            o: begin
                 Comment corresponde a um identificador;
                 if posição do identificador na T.S.=po-
                  sição de nome de função  then
                     begin
                 função:=posição do identificador na T.S.
                   - posição do 1º nome de função;
                 case função of begin
                           conjunto de procedimentos para
                           a compilação de funções;
                                    end
                      end
                      else begin
                       conjunto de procedimentos semân-
                       ticos para análise de um identi-
                       ficador dentro de uma expressão;
                            end;
             end;
```

```
1: begin
   comment corresponde a uma constante numérica;
   conjunto de procedimentos semânticos correspondentes
   à constante numérica;
   end ;
2: begin
   comment corresponde a uma construção precedida por
          ponto;
   conjunto de procedimentos para a compilação da cons-
    trução;
   end ;
3: begin
    comment corresponde a um abre-parênteses;
     expression;
    if elemento ≠ ")" then erro;
    end ;
4: begin
   comment corresponde a menos unário;
    conjunto de procedimentos semânticos;
    booleana:= true ;
    comment a variável booleana é ligada, para indicar
     que operador menos unário não pode ser seguido por
     outro operador;
    end ;
```

```
   5: begin
       comment corresponde a cadeia de caracteres;
       conjunto dos procedimentos semânticos corresponden-
       tes à cadeias;
      end ;
   6: begin
      comment corresponde ao operador "NOT";
       conjunto de procedimentos semânticos;
       booleana:=true;
       comment a variável booleana é ligada indicando que
        "NOT" não pode ser seguido por operador;
      end ;
        end do case expr;
Análise Léxica;
if elemento = operador
    then begin
          if booleana = true then begin
                                    erro;chave:=false ;
                                comment rotina de erro
                                procura ";" e a rotina
                                expression volta para o
                                ponto de chamada;
                                end
                           else
                          begin
                           conjunto de procedimentos semân
                           ticos para operadores;
                           end ;
          end
```

```
        else if booleana = false then chave = false;
      end;
  end do while chave;
end da rotina expression;
```

3.7. Pilha usada para procedimentos semânticos

Durante a compilação dos comandos compostos, rotinas PL/STI , comando IF e comando END, é usada uma pilha com o objetivo de guardar uma informação correspondente a cada um deles. No início da rotina que os compila, o topo da pilha é incrementado e aí é inserida a respectiva informação.

Como podemos verificar, esses comandos permitem outros comandos na sua definição. Há necessidade de se ter certos procedimentos semânticos associados a eles, que para serem implementados, exigiriam que a Análise Sintática voltasse atrás, se o método usado para ela permitisse isso. Nesse caso então, usamos a informação que está guardada na pilha, para implementar esse procedimentos.

No final da compilação dos comandos acima citados, o topo da pilha é decrementado, para que a informação respectiva seja desvinculada do comando.

Descrevemos a seguir os comandos que usam a pilha e qual o procedimento semântico associado a ela.

3.7.1. No comando DO-CASE

Pela definição da linguagem PL/STI, nenhum dos comandos que compõem o comando DO-CASE, pode ser rotulado. Depois que é analisada a <definição do comando composto>, o procedimento que analisa o resto do comando DO-CASE é o mesmo para todos os comandos compostos. Para que a restrição seja feita sòmente a êste comando, no fim da compilação da construção <selecionador do CASE>, inserimos no topo da pilha, uma informação associada à essa construção. Sempre que aparecer um comando rotulado, no momento em que o rótulo for analisado, vai ser testado o topo da pilha. Caso a informação aí contida seja aquela associada à construção <selecionador do CASE>, será detectado um erro no programa fonte.

3.7.2. No comando IF

Como foi visto na seção 1.9, em um comando IF, os comandos que seguem as palavras reservadas THEN e ELSE, não podem ser rotulados.

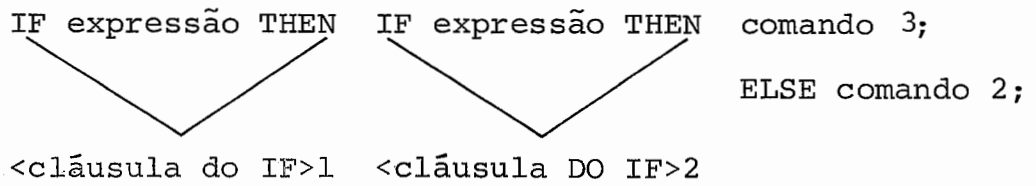
Depois que é analisada a construção a<cláusula do IF>, é inserido no topo da pilha uma informação associada à ela. Se o comando que segue essa construção é rotulado, o erro será detectado quando o rótulo for analisado, da mesma forma que no comando DO-CASE. Caso ocorra um ELSE, é inserido no topo da pilha uma informação associado a ele, que será testada

para dar o erro, da mesma forma que em <cláusula do IF>. Além disso,a construção:

IF condição1 THEN IF condição2 THEN comando3; ELSE comando2;

é errada em PL/STI (como foi visto na seção 1.11). Cada vez que um ELSE ocorre, é feito um teste na posição (TOPO DA PILHA-1), para verificar-mos se a informação aí contida está associada a uma construção <cláusula do IF> e em caso afirmativo é detectado um erro no programa fonte.

Exemplificando o procedimento, temos:



A pilha apresenta então, o aspecto mostrado na figura 3.

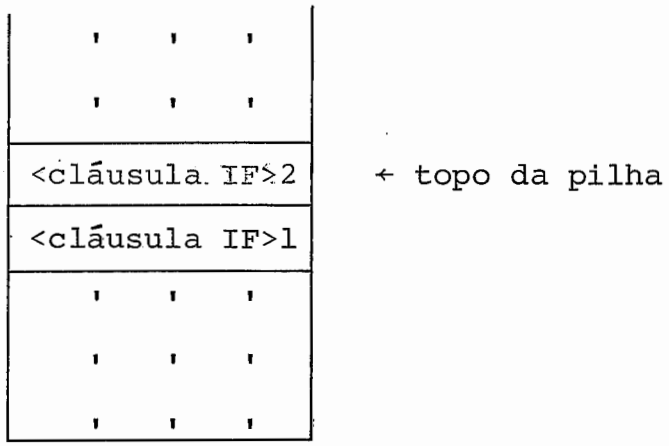


figura 3

3.7.3. Em rotinas PL/STI

Um comando RETURN só pode aparecer no corpo de uma rotina PL/STI. Em qualquer outro lugar do programa ele é ilegal. Entretanto, dentro de uma rotina, podemos ter um comando RETURN associado a um comando IF (IF expressão THEN RETURN ou IF expressão THEN comando 1; ELSE RETURN) ou ainda como parte do comando composto DO-CASE. Por isso, quando a construção <declaração da rotina> é analisada, é inserido no topo da pilha uma informação associada a ela. Daí então precisamos ter uma variável (VAR), para guardar o topo da pilha neste instante. Toda vez que um comando RETURN aparece é feito um teste na posição da pilha correspondente à VAR, que deverá conter uma informação indicando que o mesmo faz parte da rotina PL/STI ou então é detectado um erro no programa fonte.

Consideremos como exemplo:

```
N:  PROCEDURE BYTE;
    '
    '
    '
    '
    IF expressão THEN RETURN;
    '
    '
    '
    '
    END;
```

Ao ser compilado o comando RETURN, a pilha terá o aspecto mostrado na figura 4:

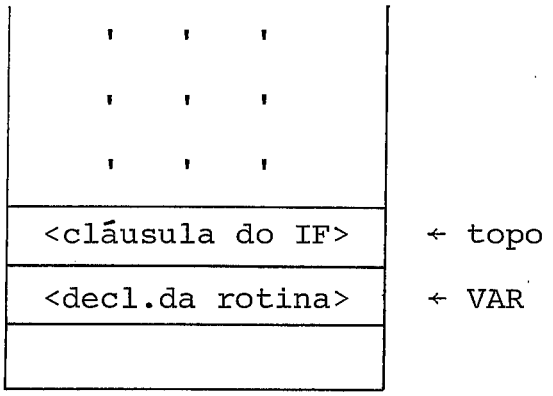


figura 4

No final da compilação da rotina, o topo da pilha é decrementado e a variável (VAR) é associada ao novo topo, pois a linguagem PL/STI, permite declarações de rotinas, dentro de rotinas.

3.7.4. No comando END

Um comando END marca o fim da compilação de um comando composto, ou de uma rotina PL/STI. Quando ele é encontrado, uma informação correspondente é colocada no topo da pilha, indicando à rotina que compila o bloco, o fim do mesmo. O algoritmo abaixo, mostra como esse procedimento se realiza dentro da rotina que compila o bloco:

Algoritmo:

```
 begin
while topo da pilha ≠ informação de fim de bloco
 do   controle;
if  símbolo = "EOF"  then  begin
                           erro;
                           comment faltou comando "END"
                            no bloco;
                           go fim de controle;
                          end;
                    else  topo da pilha:= topo da pilha-2;
end
```

A rotina de controle, como foi visto na seção 3.3.1., é chamada recursivamente, durante a compilação de um bloco, até que o"END"marcando o final do bloco, seja encontrado.

Como podemos verificar pelo algoritmo o topo da pilha é decrementado de 2. Isto se fez necessário para tomar o algoritmo genérico. Sendo um bloco uma rotina PL/STI, ou um comando composto, mesmo os comandos DO-END, e DO-WHILE que não apresentam restrições quanto aos comandos que os compõem, associam uma informação ao topo da pilha, que é retirada, juntamente com a informação do "END" no fim da compilação do bloco.

3.8. A Tabela de Símbolos (T.S.)

3.8.1. Descrição Geral

Durante a compilação do programa PL/STI, várias tabelas são criadas. Dentre elas , a mais importante é a Tabela de Símbolos que guarda informações a partir de declarações de variáveis, rótulos e macros.

A Tabela de Símbolos é formada por um conjunto de listas duplamente ligadas. São ao todo 509 listas com seus inícios em uma Tabela "Hash". Os nós dessas listas, na T.S., são alocados em paralelo, cada um deles tem 2 campos de ligação e mais 12 campos, onde são inseridas as seguintes informações, obtidas durante a Análise Sintática:

1 - o próprio identificador;

2 - o tipo do identificador que pode ser variável BYTE ou ADDRESS, rótulo declarado explícitamente, nome de rotina e nome de macro;

3 - a dimensão da variável;

4 - o número do bloco;

5 - o número de parâmetros se a rotina foi declarada com parâmetros;

6 - o apontador para uma tabela de parâmetros, apontando para o 1º parâmetro que apareceu na declaração da rotina;

7 - uma identificação para identificador declarado como DATA;

8 - uma identificação para identificador declarado como variável apontada;

9 - o tipo BYTE ou ADDRESS do valor retornado pela rotina;

10 - o endereço de memória da variável;

11 - uma identificação para rótulo referenciado;

12 - uma identificação para rótulo definido.

O compilador tem acesso a essas informações, em todas as fases da compilação.

Quando o compilador começa a traduzir o programa fonte, a T.S. já contém o nome das funções internas e as palavras reservadas da linguagem que são inseridas diretamente pelo programa principal da Análise Sintática, em posições pré determinadas.

No final de cada bloco de um programa PL/STI, todos os identificadores aí declarados, são apagados da T.S. Isto é feito comparando-se o número do bloco nesse momento, com o campo da T.S. onde é guardado o número do bloco do identificador. Enquanto eles são iguais, são apagados os identificadores e seus atributos. Quando se trata de uma rotina PL/STI, que também é um bloco, o seu nome permanece na T.S., juntamente com seus atributos, e no caso, os parâmetros formais são apagados. Porém eles já se encontram salvos em uma tabela de parâmetros que é apontada pela T.S. Isto se faz necessário para que, ao ser chamada uma rotina PL/STI sejam feitos os procedimentos semânticos relativos ao seu nome e aos seus parâmetros formais.

Seja por exemplo:

```
N: PROCEDURE (A,B);
```

a declaração de uma rotina. O nó correspondente ao seu nome

na T.S., contém um campo que aponta para o 1º parâmetro formal da sua declaração, na tabela de parâmetros, como é mostrado na figura 5.

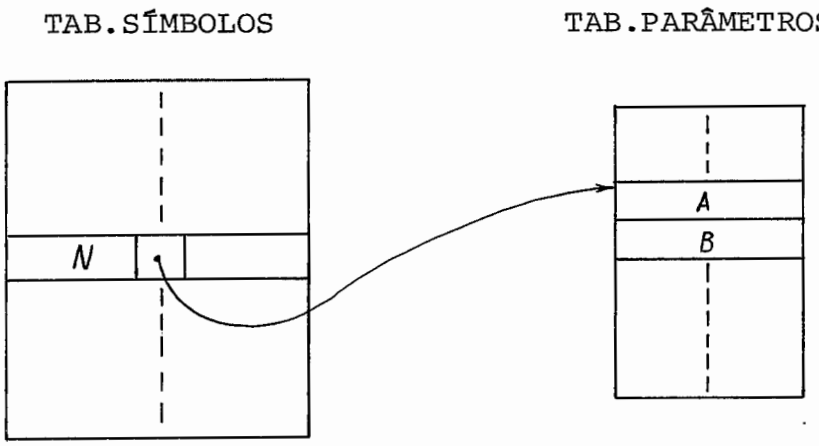


figura 5

Na seção 3.8.3 está o algoritmo que mostra como é feito, no compilador PL/STI, o procedimento para apagar da T.S., os identificadores declarados em um bloco.

3.8.2. Método de Endereçamento da T.S.

Durante a compilação, uma entrada na T.S. é adicionada sómente uma vez para cada novo identificador declarado, mas a tabela é pesquisada a cada ocorrência de um identificador no programa fonte. Como muito tempo é gasto nesse procedimento, procuramos um método de endereçamento para a T.S., que facilitasse a inserção e a busca. Optamos, então, pela técnica de Tabela "Hash" Encadeada, que usa uma Tabela "Hash" com

os campos inicialmente zerados, a Tabela de Símbolos que está inicialmente vazia e um Ponteiro, apontando sempre para a próxima entrada vazia na T.S.

Se quisermos procurar ou inserir um determinado identificador na T.S., em primeiro lugar aplicamos a ele um algoritmo (seção 3.8.3.1), que nos dá um número aleatório entre zero e 508, que é o índice de entrada na Tabela "Hash". O campo relativo a esse índice aponta então então a T.S., na posição dada pelo Ponteiro, (figura 6).

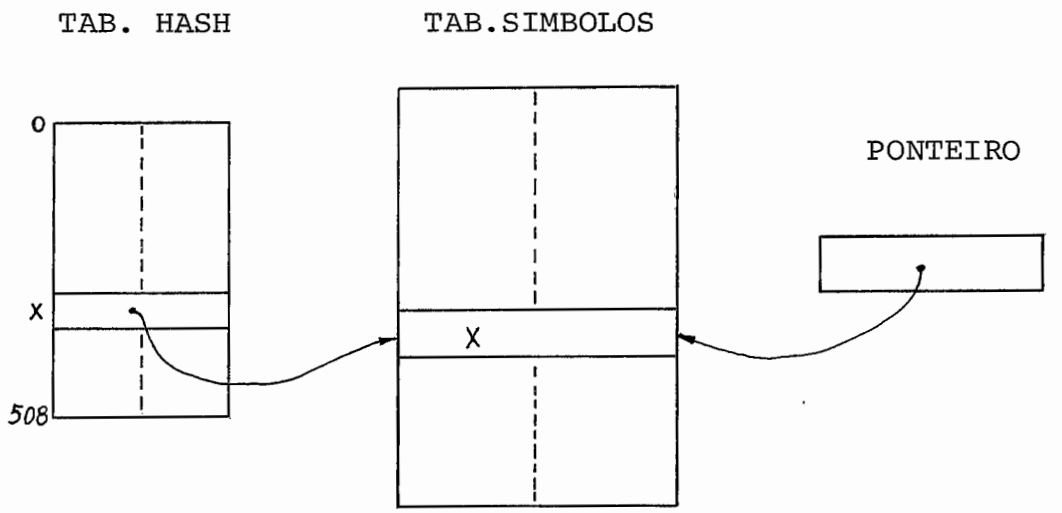


figura 6

Para inserir um identificador na T.S., se o campo relativo a ele na tabela "Hash" é zero, nenhum problema ocorre e tudo se passa como foi dito anteriormente. Se, no entanto, esse campo é diferente de zero, isto significa que houve uma colisão, ou seja, dois ou mais identificadores tiveram como resultado da aplicação do algoritmo, o mesmo índice de entrada na tabela "Hash". Aí então é formada uma lista dupla-

mente ligada na T.S. Aos campos de ligação, chamaremos de COLISÃO E RETORNO, figura 7.

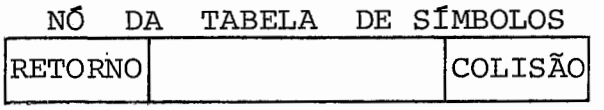


figura 7

O campo COLISÃO do nó onde está o 1º identificador que foi inserido na T.S., aponta para a próxima entrada vazia dada pelo Ponteiro, onde é inserido o identificador que colidiu.

O campo RETORNO do nó onde é inserido o identificador que colidiu, aponta para a posição da T.S., onde se encontra o identificador com o qual ele colidiu. Este campo é necessário para desfazer a ligação do campo COLISÃO com um identificador que é apagado da tabela, quando termina um bloco.

Sejam por exemplo 3 idenfiticadores ID1, ID2 e ID3, que colidiram. Ao serem inseridos na T.S., ela apresenta o aspecto mostrado na figura 8.

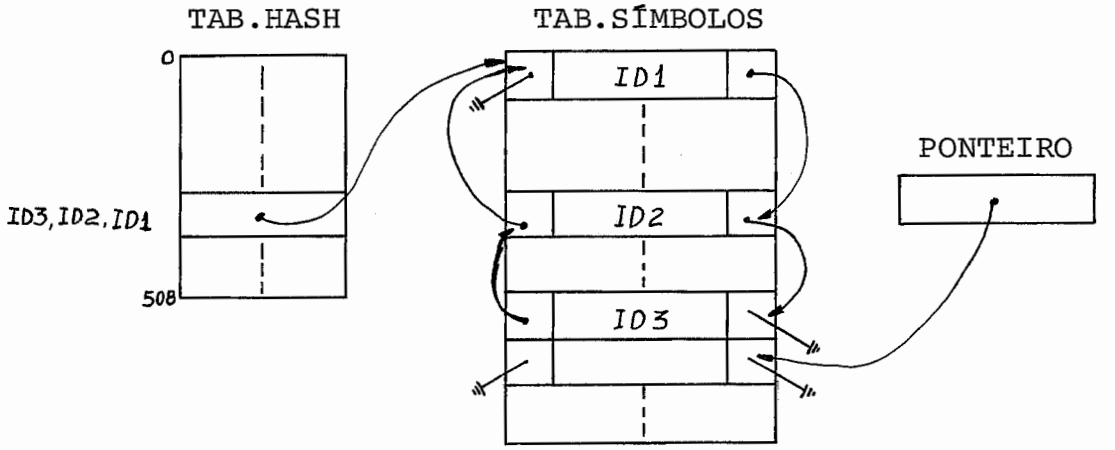


figura 8

Assim sendo, para inserirmos ou procurarmos um identificador na T.S., quando há colisão, é feita uma busca sequencial através de uma lista duplamente ligada.

3.8.2.1. Algoritmo de Inserção e Busca

O algoritmo aplicado a um identificador para se obter um índice de entrada em uma tabela "Hash" é denominado função "Hash".

A função "Hash" constitui-se do seguinte: se o identificador para o qual estamos procurando um índice de entrada na tabela "Hash", ocupa mais do que uma palavra do computador, o 1º passo da função é criar uma única palavra, resultante das palavras que ele ocupa. A seguir é aplicada uma operação aritmética ou lógica à palavra resultante para obtermos o índice de entrada na tabela "Hash".

No nosso caso, cada identificador PL/STI, pode ter até 30 caracteres, ocupando 5 palavras de memória do Burroughs/6700. Aplicamos então a elas, uma função interna do Algol, que faz equivalência (Ø EQV 1 = Ø , 1 EQV Ø = Ø , 1 EQV 1 = 1, Ø EQV Ø = 1) bit a bit entre dois argumentos. Escolhemos equivalência porque, pela sua própria definição, há igual possibilidade na ocorrência de Ø's e 1's na palavra resultante. A função equivalência é aplicada, então, às duas primeiras palavras ocupadas pelo identificador, a seguir é novamente aplicada à palavra resultante com a 3ª. e assim consecuti

vamente, até que ela seja aplicada às 5 palavras do identificador. Finalmente, para obtermos o índice de entrada na tabela "Hash", optamos pela operação MOD, que dá como resultado o resto da divisão da palavra resultante por 509 (o tamanho da tabela "Hash"). Desse modo o índice procurado é um número aleatório entre zero e 508.

Como foi dito anteriormente, a T.S. do compilador PL/STI é inicializada com as palavras reservadas e os nomes da funções embutidas da linguagem. Verificamos que a aplicação do algoritmo acima não acarretou nenhuma colisão entre elas.

3.8.3. Algoritmo para apagar a T.S.

Como foi visto na seção 3.8.1, no final de cada bloco do programa PL/STI, são apagados da T.S., todos os identificadores declarados nesse bloco, bem como os seus atributos e os índices de entrada na T.S., correspondente a eles. Se o bloco é uma rotina PL/STI, o seu nome permanece na T.S., com o campo onde é inserido o número do bloco, correspondendo ao bloco onde ela foi declarada.

```
Algoritmo
    procedure  apagatabela;
begin
    ponteiro = ponteiro - 1;
    while nº do bloco atual = nº do bloco do iden-
    tificador apontado por ponteiro na T.S. and
    identificador apontado por ponteiro ≠ de nome
    de rotina do
      begin
      if não houve colisão then begin
                                  apaga da TS o identi
                                  ficador,seus atribu-
                                  tos e o índice corres
                                  pondente na tabela
                                  "hash";
                                  end ;
                          else  begin
                                  apaga da TS o campo
                                  COLISÃO do idenfiticador
                                  apontado pelo campo
                                  RETORNO;
                                  apaga o identificador,
                                  seus atributos e o cam-
                                  po RETORNO;
                                end
       ponteiro:= ponteiro - 1;
      end ;
nº do bloco atual = nº do bloco atual - 1;
```

if identificador apontado por ponteiro = nome de rotina
then número do bloco do identificador apontado por ponteiro = número do bloco atual;
end da rotina apagatabela;

3.9. Erros de um Programa PL/STI

O compilador tem oportunidade de detectar erros em um programa PL/STI, durante todas as fases da compilação e, principalmente durante a Análise Sintática.

Qualquer erro encontrado causa a impressão de uma mensagem de erro, que dá detalhes sobre o seu tipo e a posição do texto onde ele ocorreu.

Além disso são dadas também mensagens de advertência, que chamam a atenção para partes do texto PL/STI, que embora válidas, estão provàvelmente com erros.

Na listagem do programa fonte, a posição onde um erro foi encontrado é indicado por:

a) Um asterisco impresso na linha logo abaixo e o mais próximo possível do erro.
b) Na linha seguinte, o número total de erros acumulados até aquele ponto, seguido de uma mensagem com detalhes sôbre o mesmo.

Durante a Análise Sintática são detectados erros sintáticos e erros semânticos. Um erro sintático ocorre quando a Análise Léxica devolve um elemento da linguagem dife-

rente do que é esperado pela rotina que analisa o comando ou a expressão. Como exemplos podemos citar um erro quando é esperado o sinal de igual em um comando de atribuição ou ainda quando é esperado um fecha parênteses em uma expressão e a Análise Léxica devolve outro elemento.

Um erro semântico tem origem em um procedimento semântico, aplicado à uma construção gramatical que pode ser por exemplo, consultar a T.S. ou a pilha de informações semânticas. Entre outros podemos citar os seguintes erros:

- a palavra reservada ELSE ligada a dois ou mais comandos IF'S.
- um identificador usado mas não declarado;
- uma chamada de rotina onde o nº de parâmetros atuais é diferente do nº de parâmetros formais, etc...

3.9.1. Tratamento de Erros

Há dois procedimentos diferentes para o tratamento de erros:

a) erros que não impedem a continuação da compilação do comando onde aparecem. São erros tais que embora impeçam a execução do programa objeto, permitem que o comando seja compilado até o fim. Como exemplo podemos citar um erro onde uma variável simples é usada como variável indexada, ou ainda, um erro onde um identificador é declarado com atributo INITIAL e a lista de constantes que segue o atributo, contém um elemento diferente de uma constante;

b) erros que impedem a continuação da compilação do comando onde ocorrem porque não seria possível gerar código significativo para o mesmo. Neste caso a Análise Sintática chama a rotina da Análise Léxica, até que um ponto e vírgula seja devolvido, denotando o fim daquele comando. Daí então a compilação continua a partir do comando seguinte. Este simples procedimento é descrito pelo algoritmo abaixo:

```
        Algoritmo:
begin
  while elemento ≠ ";" do Análise Léxica;
  go to fim da rotina que compila o comando onde
    ocorreu o erro;
end ;
```

Esses erros impedem a execução do programa objeto. Como exemplos deste caso, podemos citar um erro em que uma palavra reservada é usada como um identificador, ou ainda, um erro onde um identificador é usado e não foi declarado.

As advertências não impedem a execução do programa objeto.

Cada vez que um erro é encontrado, são guardadas em uma matriz as informações sôbre o mesmo (total de erros acumulados, linha e coluna do programa fonte onde ele ocorreu e o apontador para a mensagem correspondente que se encontra em outra matriz). Como a matriz de erros, contém 100 linhas, o limite máximo é de 100 erros em um programa PL/STI, quando então pára a compilação e é impresso uma mensagem dizendo que excedeu o limite máximo de erros permitidos e a compilação foi interrompida.

CAPÍTULO IV

CONCLUSÕES

4.1. Compilador de 1 passo

Considerando-se que do projeto faz parte dois trabalhos diferentes, o compilador deveria ser definido em 2 passos, ao invés de um apenas, o que permitiria dividí-lo em 2 módulos mais independentes. Devido a nossa pouca experiência na construção de um projeto desse tipo, entretanto, optamos por um compilador de 1 passo. A maior dificuldade encontrada, foi determinar a sua divisão em duas partes equivalentes em volume de trabalho. Isto porque em se tratando de um compilador em que todas as fases são interligadas, fica difícil determinar onde termina exatamente uma e começa a outra. Para resolvermos isto, optamos pela maior modularização possível do programa do compilador, o que mostrou ser uma boa filosofia, pois desse modo, dentro de convenções préviamente estabelecidas, as dificuldades foram contornadas.

4.2. Método usado para a Análise Sintática

O método usado, dos descendentes recursivos, é um método que exige muito trabalho de programação e depuração. É também um método rígido, isto porque qualquer alteração nas regras gramaticais da linguagem, exige uma alteração nas rotinas correspondentes ao comando que é compilado pelas regras mudadas. Entretanto, êste método foi escolhido por ser simples e eficiente e também porque uma mudança em qualquer rotina do programa compilador não é uma tarefa difícil pois elas foram programadas de modo bem estruturado.

4.3. Sugestões para expansão da linguagem PL/STI

A linguagem PL/STI não possue comandos que permitem manipulação de cadeias ou processamento com ponto flutuante. Essas facilidades podem ser introduzidas e acarretarão algumas mudanças no programa compilador. A seguir são dados, como sugestões alguns comandos que podem ser inseridos na linguagem com estes objetivos.

4.3.1. Comandos que permitem manipulação de cadeias

1. SCAN variável FOR expressão1 WHILE operador relacional expressão2;

A função do SCAN é examinar a cadeia de caracteres dada em um vetor, que começa na posição de memória indicada pela variável que segue a palavra reservada SCAN. É examinado um caracter por vez, da esquerda para a direta. A cadeia é percorrida até que um caracter fonte falhe o teste ou até que seja atingido uma quantidade de caracteres no máximo igual ao valor da expressão1. Neste caso será setada uma variável booleana, indicando que nenhum caracter falhou no teste.

Exemplo:

```
DECLARE  ALFA(16) BYTE INITIAL ('ABCDEFGHIJKLMNOP'),
A ADDRESS;
    '
    '
    '
A=  ALFA(1);
    '
    '
    '
 SCAN  A   FOR  16  WHILE  < > 'M'
    '
    '
    '
  EOF
```

Neste exemplo, o vetor ALFA ocupa 16 bytes. O comando SCAN percorrerá a cadeia de caracteres a partir da posição dada pela variável A, até encontrar o caracter M.

2. MOVE variável1 TO variável2 FOR expressão;

Este comando permitirá o deslocamento da cadeia que começa na posição dada pela variável1, para a posição dada pela variável2 e a expressão seguindo a palavra reservada FOR, indicará o nº de caracteres da cadeia que serão deslocados para outra posição da memória.

Exemplo:

```
DECLARE  VETOR(6) BYTE  INITIAL ('CADEIA'),
VETOR AUX(10) BYTE;
   '
   '
   '
A=  . VETOR(1);
B=  . VETOR AUX(1);
   '
   '
   '

MOVE  A  TO B  FOR 6;
   '
   '
   '

 EOF
```

Neste exemplo os 6 caracteres da cadeia que começa na posição de memória . VETOR(1), dada por A, serão translados para a posição que começa em . VETORAUX(1), dada por B.

3. TRANSLATETABLE identificador (conjunto1 de caracteres da linguagem) TO (conjunto2 de caracteres da linguagem);

Esta declaração define uma tabela de conversão com nome dado pelo identificador, a ser usada com o comando MOVE. Os conjuntos de caracteres são equivalentes a cadeias contendo todos os caracteres especificados no conjunto.

O exemplo abaixo mostra como este comando deve ser usado junto com o comando MOVE.

```
DECLARE (A1, A2) ADDRESS, TAB(2)BYTE INITIAL('AB'),
TABAUX(2)BYTE;
TRANSLATETABLE    TABELA ('ABCDEF') TO ('12345')
 '
 '
 '
A1 = . TAB(1);
A2 = . TABAUX(1);
 '
 '
 '
MOVE  A1  TO  A2  FOR 2  WITH TABELA;
 '
 '
 '
EOF
```

Este trecho de programa determina a transferência da cadeia '12' para o vetor TABAUX.

A implementação das sugestões anteriores implica em modificações no programa compilador, que de um modo geral, se restrigem em inserir na Tabela de Símbolos as novas palavras reservadas, fazer as rotinas para a compilação de cada um dos novos comandos e introduzir a chamada dessas rotinas na rotina de Controle (seção 3.3.1).

4.3.2. Processamento com Ponto Flutuante

Outra extensão possível na linguagem PL/STI, são números decimais com ponto flutuante.

1. Modificações na linguagem

Para se permitir ponto flutuante, a constante decimal terá que ser redefinida a fim de que sejam aceitas constantes com ponto decimal e no caso, todos os números decimais com ponto flutuante devem ter pelo menos um dígito precedendo o ponto, pois a linguagem PL/STI usa o ponto também como operador de endereço.

Por exemplo:

0.123 - representará uma constante decimal em ponto flutuante.

.123 - representa um endereço de memória, no caso 123.

É necessário ainda a criação de uma palavra reservada, por exemplo, FLOATING, que determinará, em um comando de declaração, as variáveis que assumirão valores decimais em ponto flutuante. Por exemplo:

```
DECLARE (A , K) FLOATING;
```

2. Representação interna de uma constante decimal em ponto flutuante

A constante poderá ser armazenada em 6 bytes, onde em 5 bytes ficará a parte significativa e em 1 byte o expoente. Isso acarretará em se ter constantes de $10^{\pm 38}$. Esta configuração permitirá o uso em aplicações comerciais.

3. Modificações no compilador

Algumas alterações terão que ser feitas no programa do compilador.

A rotina da Análise Léxica, sofrerá algumas modificações onde são analisadas constante numéricas, para que sejam aceitas constantes decimais com ponto flutuante.

Na Análise Sintática serão feitas modificações na rotina que analisa comandos de declaração para que seja reconhecido o atributo FLOATING e na rotina que analisa expressões PL/STI, onde deverá ser feita uma revisão para inserir os procedimentos semânticos relativos à operações com ponto flutuante.

APÊNDICES

APÊNDICE A

A GRAMÁTICA DA LINGUAGEM PL/STI

O VOCABULÁRIO

Símbolos Terminais	Símbolos não Terminais
1.	<programa>
2. ;	<lista de comandos>
3. HALT	<comando>
4. IF	<comando básico>
5. THEN	<comando if>
6. ELSE	<comando de atribuição>
7. DO	<comando composto>
8. CASE	<definição de rotina>
9. <número>	<comando de retorno>
10. PROCEDURE	<comando de chamada de rotina>
11. <identificador>	<comando de desvio>
12.)	<comando de declaração>
13. (	<definição de rótulo>
14. ,	<cláusula do IF>
15. END	<parte verdadeira>
16. :	<expressão>
17. RETURN	<definição do comando composto>
18. CALL	<término>
19. GO	<passo de definição>
20. TO	<cláusula do WHILE>
21. GOTO	<selecionador do CASE>
22. DECLARE	<variável>
23. LITERALLY	<operador de atribuição>
24. <cadeia de caracteres>	<controle da iteração>
25. DATA	<até>

Símbolos Terminais	Símbolos não Terminais
26. BYTE	<por>
27. ADDRESS	<definição de rotina>
28. LABEL	<declaração da rotina>
29. BASED	<nome da rotina>
30. INITIAL	<tipo>
31. =	<lista de parâmetros>
32. :=	<início da lista de parâmetros>
33. OR	<desvio>
34. XOR	<elementos da declaração>
35. AND	<declaração de tipo>
36. NOT	<lista de dados>
37. <	<início da lista de dados>
38. >	<constante>
39. +	<especificação do identificador>
40. -	<início da dimensão>
41. *	<lista de valores iniciais>
42. /	<variável apontada>
43. MOD	<início da lista>
44. .	<parte esquerda>
45. BY	<expressão lógica>
46. WHILE	<fator lógico>
47.	<fator lógico secundário>
48.	<fator lógico primário>
49.	<expressão aritmética>
50.	<operador relacional>
51.	<comparação>
52.	<termo>

Símbolos Terminais	Símbolos não Terminais
53.	<operando primário>
54.	<início da lista de constantes>
55.	<início do subscrito>
56.	<nome de variável>

Os símbolos <identificador>, <número>, <cadeia de caracteres>, são terminais para a Análise Sintática e não terminais para a Análise Léxica. As produções da Gramática para eles são:

<letra>::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|Y|W|U|V|
X|Z

<dígito binário>::= 0|1

<dígito octal>::= <digito binário> |2|3|4|5|6|7

<dígito>::= <dígito octal> |8|9

<código binári >::= <dígito binário>
| <código binário><dígito binário>

<número binário>::=<código binário> B

<código octal>::= <dígito octal>
| <código octal> <dígito octal>

<número octal>::= <código octal> O

<dígito hexadecimal>::= <dígito> |A|B|C|D|E|F

<código hexadecimal>::= <dígito hexadecimal>
| <código hexadecimal><dígito hexadecimal>

<número hexadecimal>::= <código hexadecimal> H

<número decimal>::= <dígito>

| <número decimal><dígito>

<número>::= <número binário>

| <número octal>

| <número decimal>

| <número hexadecimal>

<identificador>::= <letra>

| <identificador><letra>

| <identificador><dígito>

<cadeia de caracteres>::= '<caracteres de linguagem>'

<caracteres especiais>::= .|,|:|=|<|>|/|*|+|-|(|)|;|'|$

<caracteres da linguagem>::= <caracteres especiais>

| <letra>

| <número>

| <caracteres da linguagem><letra>

| <caracteres da linguagem><número>

| <caracteres da linguagem><caracte-

res especiais>

AS PRODUÇÕES

```
 1-<Programa>::=<lista de comandos>EOF
 2-<Lista de comandos>::=<comando>
 3-                     |<lista de comandos><comando>
 4-<comando>::=<comando básico>
              |<comando IF>
 6-<comando básico>::=<comando de atribuição>;
 7-                  |<comando composto>;
 8-                  |<definição de rotina>;
 9-                  |<comando de retôrno>;
10-                  |<comando de chamada de rotina>;
11-                  |<comando de desvio>;
12-                  |<comando de declaração>;
13-                  | HALT;
14-                  |  ;
15-                  |<definição de rótulo><comando básico>
16-<comando IF>::=<cláusula do IF><comando>
17-              |<cláusula do IF><parte verdadeira><comando>
18-              |<definição de rótulo><comando IF>
19-<cláusula do IF>::=IF<expressão>THEN
20-<parte verdadeira>::=<comando básico>ELSE
21-<comando composto>::=<definição do comando composto><término>
22-<definição de comando composto>::=DO;
23-                                  |DO<passo de definição>;
24-                                  |DO<Cláusula do WHILE> ;
25-                                  |DO<Selecionador do CASE> ;
26-                                  |<definição do comando composto>
                                     <comando>
```

27-<passo de definição>::=<variável><operador de atribuição>
<expressão><controle da interação>

28-<controle da interação>::=<até><expressão>

|<até><expressão><por><expressão>

30-<cláusula do WHILE>::=WHILE<expressão>

31-<selecionador do CASE>::=CASE<expressão>

32-<definição de rotina>::<declaração de rotina><lista de comandos><término>

33-<declaração de rotina>::=<nome da rotina>;

34- |<nome da rotina><tipo>;

35- |<nome da rotina><lista de parâmetros>;

36- |<nome da rotina><lista de parâmetros><tipo>;

37-<nome darotina>::=<definição de rótulo>PROCEDURE

38-<lista de parâmetros>::=<início da lista de parâmetros><identificador>

39-<início da lista de parâmetros>::=(

40- |<início da lista de parâmetros><identificador>

41-<término>::=END

42- |END<identificador>

43- |<definição de rótulo><término>

44-<definição de rótulo>::=<identificador>:

45- |<número>:

46-<comando de retorno>::=RETURN

47- |RETURN<expressão>

48-<comando de chamada de Rotina>::=CALL<variável>

49-<comando de desvio>::=<desvio><identificador>

50- |<desvio><número>

51-<desvio>::= GO TO

52- |GOTO

53- |GO

54-<comando de declaração>::=DECLARE<elementos da declaração>

55- |<comando de declaração>,<elementos da declaração>

56-<elementos da declaração>::=<declaração de tipo>

57- |<identificador><LITERALLY ><cadeia de caracteres>

58- |<identificador><lista de dados>

59-<lista de dados>::=<início da lista de dados><constante)

60-<início da lista de dados>::=DATA(

61- |<início da lista de dados><constante>

62-<declaração de tipo>::=<especificação do indentificador><tipo>

63- |<início da dimensão><número>)<tipo>

64- |<declaração de tipo><lista de valores iniciais>

65-<tipo>::=BYTE

66- |ADDRESS

67- |LABEL

68-<início da dimensão>::=<especificação do identificador>(

69-<especificação do identificador>::=<nome da variável>

70 |<lista de identificadores><nome da variável>)

71-<lista de identificadores>::= (

72- |<lista de identificadores><nome da variável>)

73-<nome da variável>::=<identificador>

74- |<variável apontada><identificador>

75-<variável apontada>::=<identificador>BASED

76-<lista de valores iniciais>::<início da lista><constante>)

77-<início da lista>::=INITIAL (

78- |<início da lista><constante>,

79-<comando de atribuição >::=<variável><operador de atribuição>
<expressão>

80- |<parte esquerda><comando de atribuição>

81-<operador de atribuição>::= =

82-<parte esquerda>::=<variável>,

83-<expressão>::=<expressão lógica>

84- |<variável :=<expressão lógica>

85-<expressão lógica>::=<fator lógico>

86- |<expressão lógica>OR<fator lógico>

87- |<expressão lógica>XOR<fator lógico>

88-<fator lógico>::=<fator lógico secundário>

89- |<fator lógico>AND<fator lógico secundário>

90-<fator lógico secundário>::=<fator lógico primário>

91- |NOT<fator lógico primário>

92-<fator lógico primário>::=<expressão aritmética>

93- |<expressão aritmética><operador relacional><expressão aritmética>

94-<operador relacional>:: = =

95- | <

96- |>

97- |<comparação>

```
 98-<comparação>:: = < >
 99-              |   < =
100-              |   > =
101-<expressão aritmética>:: =<termo>
102-                |<expressão aritmética>+<termo>
103-                |<expressão aritmética>-<termo>
104-                |-<termo>
105-<termo>:: =<operando primário>
106-         |<termo>*<operando primário>
107-         |<termo>/<operando primário>
108-         |<termo>MOD<operando primário>
109-<operando primário>::=<constante>
110-                    |.<constante>
111-                    |<início da lista de constantes><constante>)
112-                    |<variável>
113-                    |.<variável>
114-                    |(<expressão>)
115-<início da lista de constante>:: = . (
116-                   |<início da lista de constante><constante>
117-<variável>:: =<identificador>
118-           |<início do subscrito><expressão>
119-<início do subscrito>:: = <identificador> (
120-            |<início do subscrito><expressão>,
121-<constante>::=<cadeia de caracteres>
122-          |<número>
123-<até>:: = TO
124-<por>:: = BY
```

APÊNDICE B

LISTA DOS CARACTERES ESPECIAIS

Símbolo	Nome	Uso
$	sinal de cifrão	Pode ser usado em qualquer lugar do programa o compilador ignora-o.
=	sinal de igual	Operador de teste relacional Operador de atribuição
:=	sinal de atribuição embutida	Operador de atribuição embutida
.	ponto	Operador de endereço
/	barra	Operador de divisão
/*		Delimitador de comentário à esquerda
*/		Delimitador de comentário à direita
(	Abre parênteses	Delimitador à esquerda de listas, subscritos e expressões
)	Fecha parênteses	Delimitador à direita de listas, subscritos e expressões
+	Sinal de mais	Operador de adição
-	Sinal de menos	Operador de subtração
'	Sinal de apóstrofo	Delimitador de cadeias de caracteres
*	Asterisco	Operador de multiplicação
<	Menor que	Operador de teste relacional
>	Maior que	Operador de teste relacional

Símbolo	Nome	Uso
<=	Menor ou igual	Operador de teste relacional
>=	Maior ou igual	Operador de teste relacional
< >	Diferente	Operador de teste relacional
:	Dois pontos	Delimitador de rótulo
;	Ponto e vírgula	Delimitador de comando
,	Vírgula	Delimitador de elemento de lista

APÊNDICE C

LISTA DAS PALAVRAS RESERVADAS E NOMES DAS FUNÇÕES INTERNAS

PALAVRAS RESERVADAS

IF

THEN

ELSE

DO

PROCEDURE

END

DECLARE

BYTE

ADDRESS

LABEL

INITIAL

DATA

LITERALLY

BASED

GO

TO

BY

GOTO

CASE

WHILE

CALL

RETURN

HALT

OR

AND

XOR

NOT

MOD

EOF

NOMES DE FUNÇÕES INTERNAS

CARRY

DOUBLE

HIGH

LAST

LENGTH

LOW

PARITY

ROL

ROR

SCL

SCR

SHL

SHR

SIGN

ZERO

APÊNDICE D

TABELAS-VERDADE PARA OS OPERADORES BOOLEANOS

1- Ou exclusivo (XOR) :

Ø XOR Ø = Ø

Ø XOR 1 = 1

1 XOR Ø = 1

1 XOR 1 = Ø

2- E lógico (AND) :

Ø AND Ø = Ø

Ø AND 1 = Ø

1 AND Ø = Ø

1 AND 1 = 1

3- Ou lógico (OR) :

Ø OR Ø = Ø

Ø OR 1 = 1

1 OR Ø = 1

1 OR 1 = 1

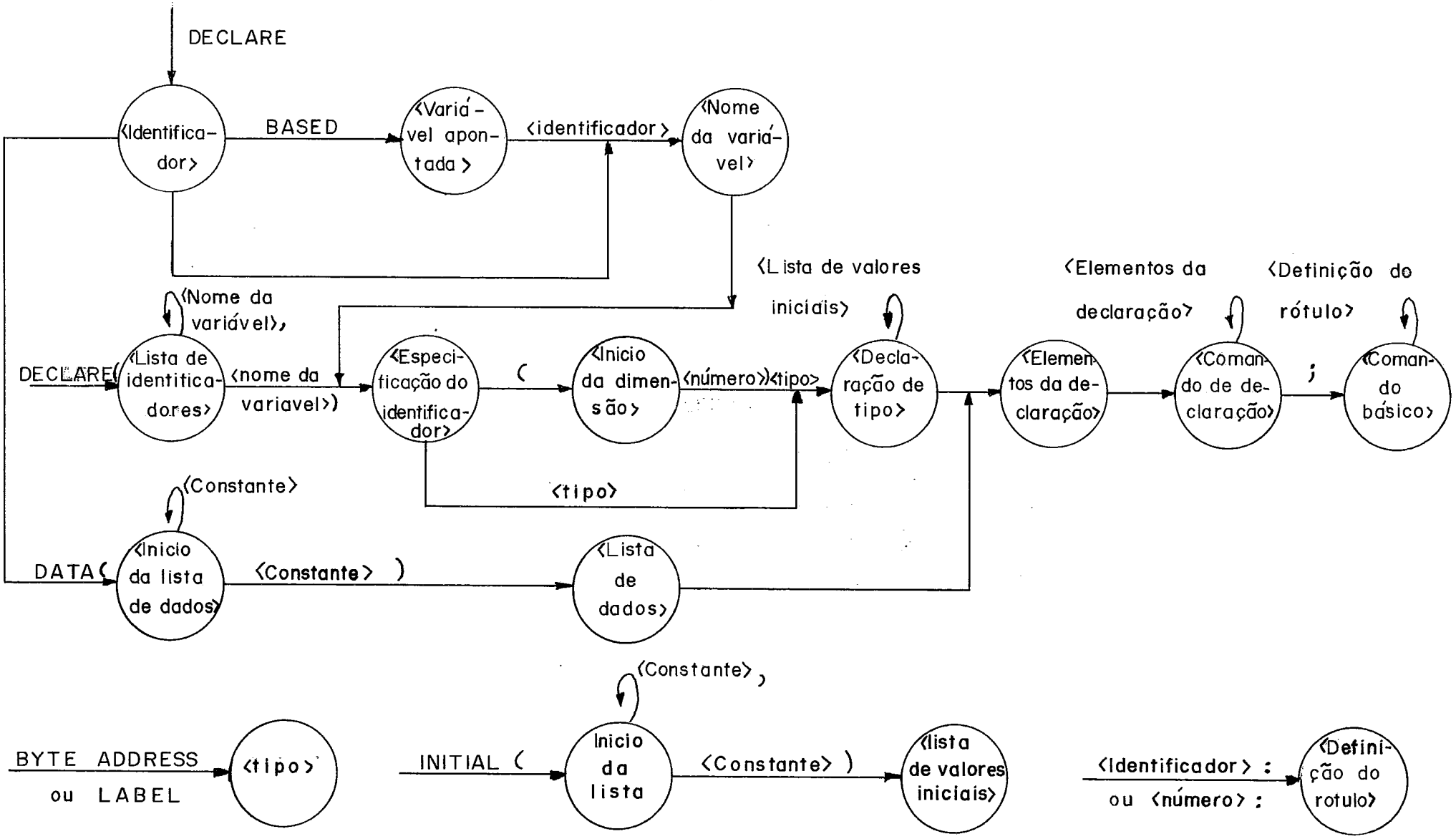
4- Negação (NOT) :

NOT Ø = 1

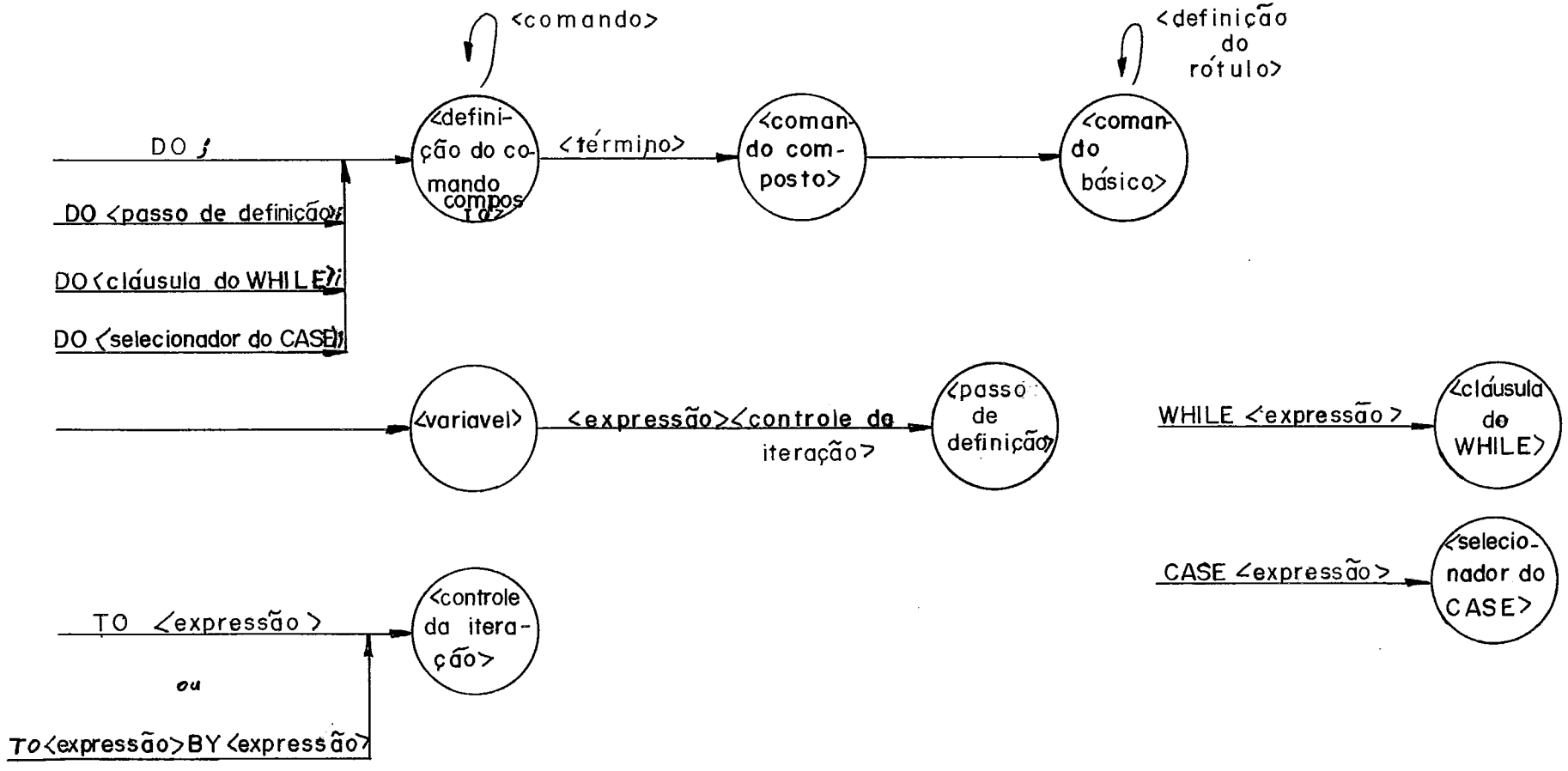
NOT 1 = Ø

APÊNDICE E

DIAGRAMAS SINTÁTICOS



DIAGRAMAS PARA COMPILAÇÃO DE UM COMANDO DE DECLARAÇÃO



DIAGRAMAS PARA COMPILAÇÃO DE UM COMANDO COMPOSTO

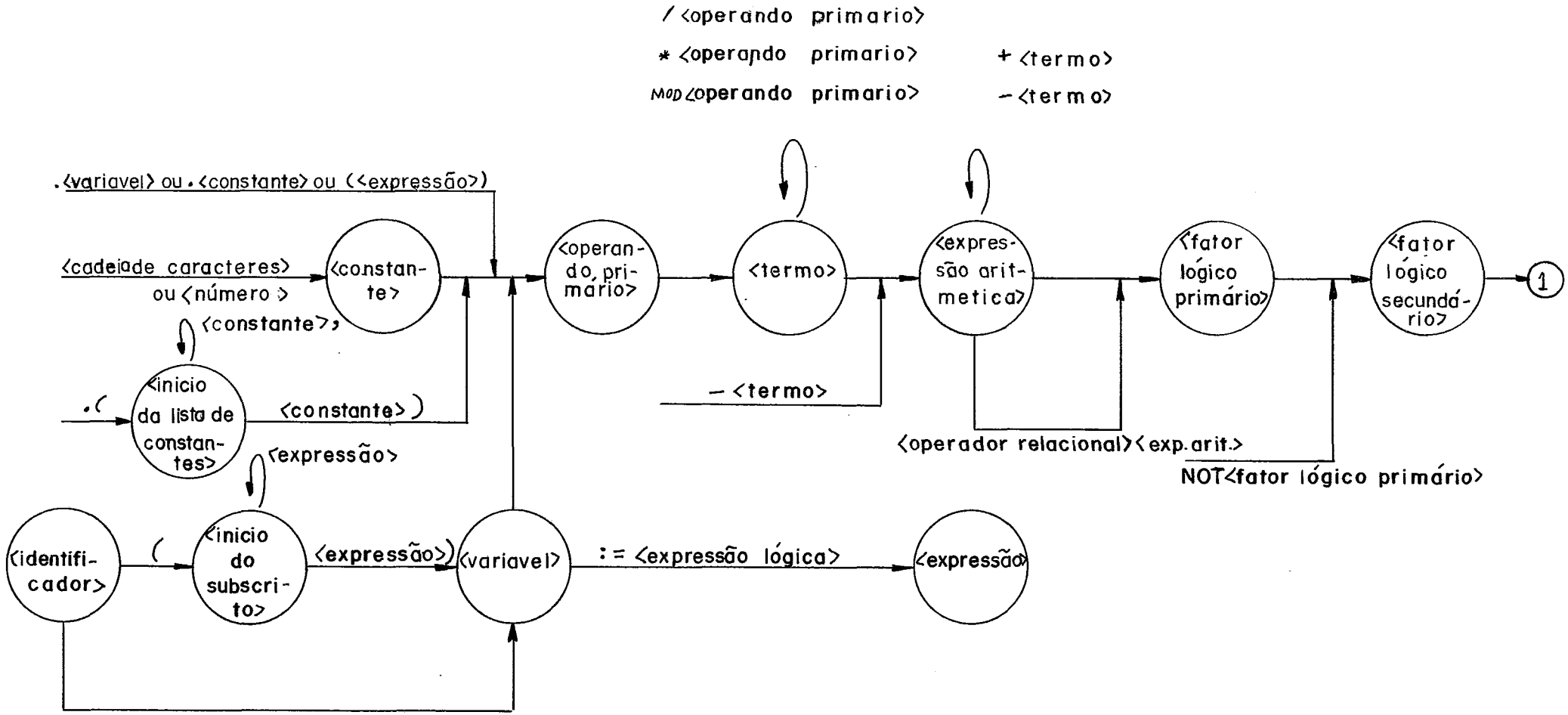
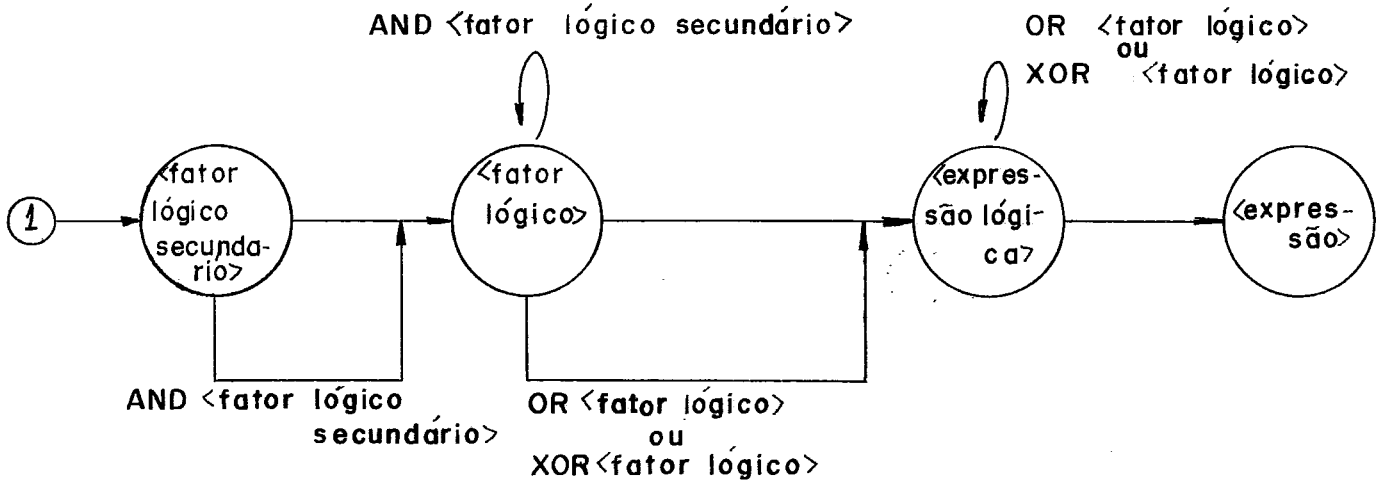
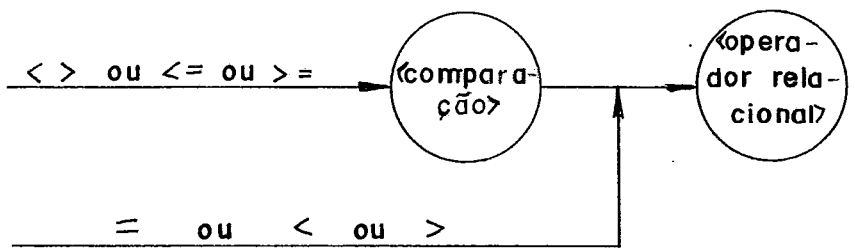


DIAGRAMA PARA COMPILAÇÃO DE EXPRESSÕES PL/STI





CONTINUAÇÃO DA FÔLHA Nº 108

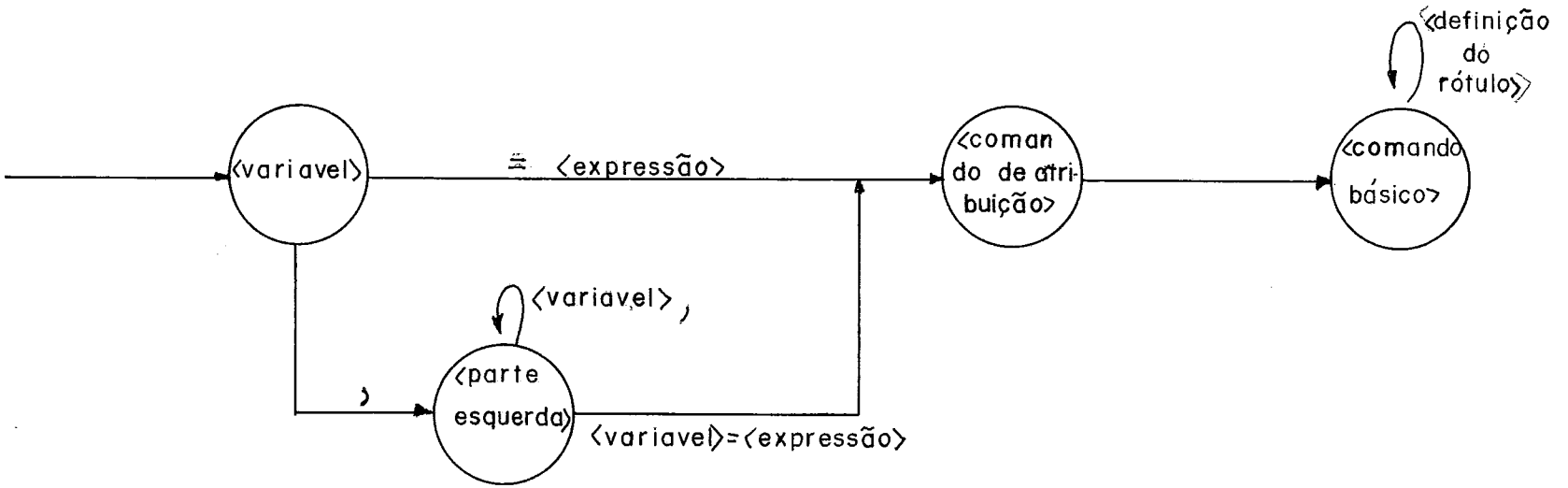
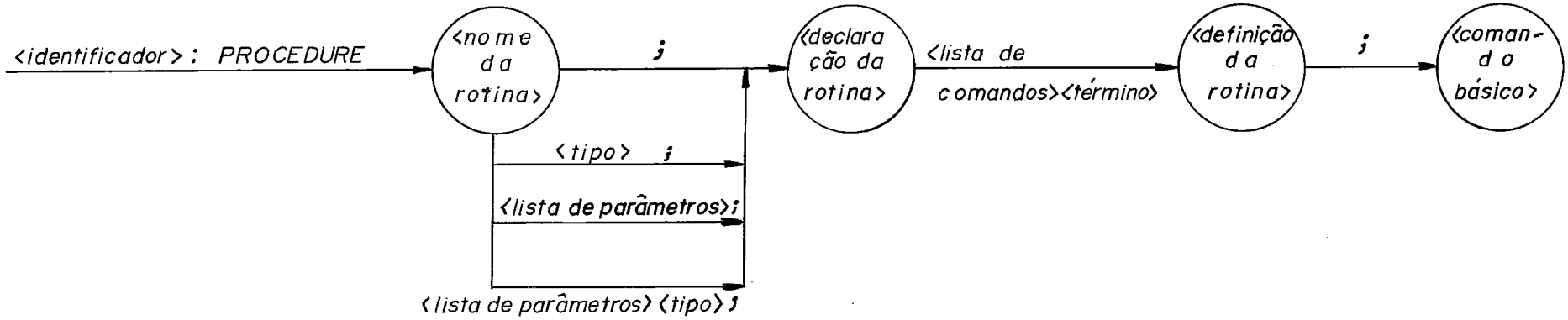
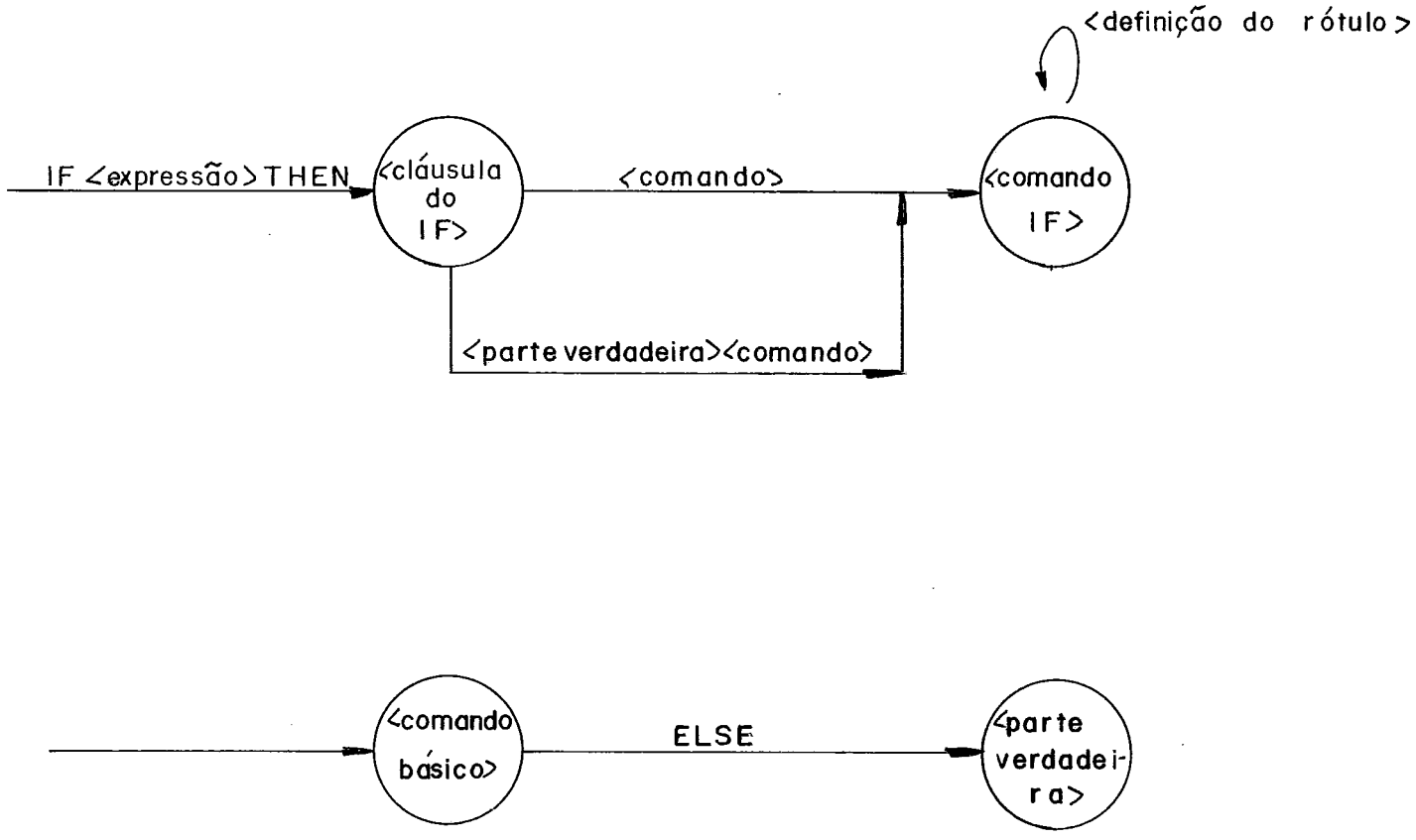


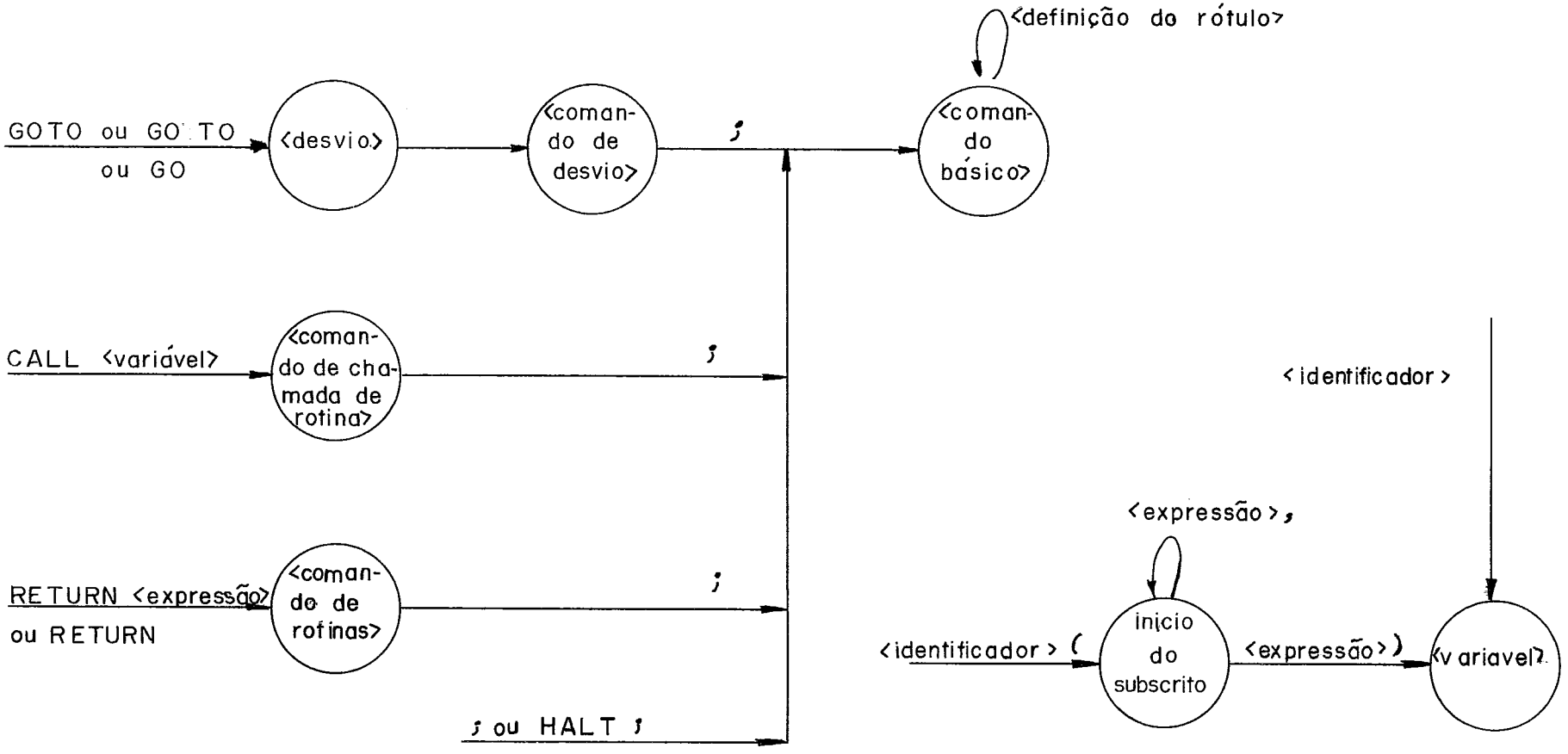
DIAGRAMA PARA COMPILAÇÃO DE UM COMANDO DE ATRIBUIÇÃO



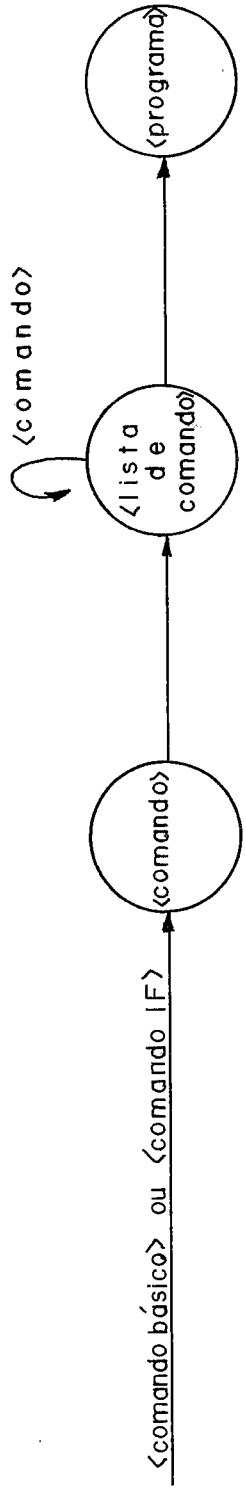
DIAGRAMAS PARA COMPILAÇÃO DE UMA ROTINA PL/STI



DIAGRAMAS PARA COMPILAÇÃO DE UM COMANDO IF



DIAGRAMAS PARA COMPILAÇÃO DOS COMANDOS DE DESVIO, CHAMADA DE ROTINA, RETORNO E HALT.



REDUÇÃO DE COMANDOS BÁSICOS E COMANDO IF.

APÊNDICE F

Programa exemplo com erros

```
                ****** COMPILADOR PL/STI *** VERSAO 1 ******
                ---------------------------------------------


13/04/77


                /* A ORDEM INICIAL DE A E
                     ARBITRARIA */

                DECLARE A(10) ADDRESS INITIAL
                (33,10,5,11,3,3,0,8,9,1);


                SORT:PROCEDURE(N)ADDRESS;


                /* N=COMP. DE A
                COUNT=NR.DE TROCAS FEITAS*/
                /* SWIT=INDICA SE ALGUMA
                TROCA FOI FEITA DURANTE
                CADA SCAN DO PROGRAMA */

                DECLARE(N,I,SWIT)BYTE,
                 (TEMP,COUNT)ADDRESS;
                SWIT=1;
                /*INDICA QUE NADA FOI FEITO
                  AINDA */
                COUNT=0;
                DO WHILE SWIT;
                 SWIT=0;
                 DO I=0 TO N-2;
                  IF A>A(I+1) THEN
                       *
>>>>>>  1>>>>>>  ERRO(48): VARIAVEL INDEXADA USADA COMO VARIAVEL SIMPLES
                   DO; /*ACHOU 1 PAR FORA DE*/
                   COUNT=COUNT+1;     /*ORDEM*/
                   SWIT=1;
                   TEMP=A(I; /*COLOCA-O*/
                            *
>>>>>>  2>>>>>>  ERRO(71): FALTA FECHA-PARENTESES NA EXPRESSAO.
                   A(I)=A(I+1);  /*NA ORDEM*/
                   A(I+1)=TEMP;
                   END;
                 END;
                END;/*WHILE*/
                RETURN COUNT;


                END SORT;

                NSWITCH=SORT(10);
                        *
>>>>>>  3>>>>>>  ERRO(27): IDENTIFICADOR USADO MAS NAO DECLARADO
                EOF
```


UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
NÚCLEO DE COMPUTAÇÃO ELETRÔNICA
NCE UFRJ

REFÊRENCIAS BIBLIOGRÁFICAS

1. Naur P. (ed), 1963 - "Revised report on the algorithmic language Algol 60" - Computer Journal, vol. 5.

2. Wulf W. A., 1971 - "Programming withouth the "goto" IFI PS 71, TA - 3 - 84.

3. Hoare C. AR., 1973 - "Hints on programming language design" - Comp. Sci. Dep. Rep. no CS-403- Stanford.

4. Knuth D. E., 1972 - "The art of Computer programming" - Fundamental Algorithm, vol 1 - Addison-Wesley.

5. Gries, D., 1971 - "Compiles construction for digital Computers" - John Wiley.

6. Hopgood, F. R. A., 1970 - "Compiling Techniques" - MacDonald.

7. Pollack, W. B., 1972 - "Compiler Techniques" - Averbach.

8. Ullman, D. J. - "The theory of parsing,translation and Compiling", vols. 1 e 2 - Prentice-Hall.

9. Russell, R., 1964 - "Algol 60 implementation" - A.P.I.C. Studies in Data Processing

10. 8008/8080 - "PL/M Programming Manual"
1975, STA.CLARA - INTEL CORP.

11. B6700/B7700- "Algol Language Reference Manual" -
Burroughs.