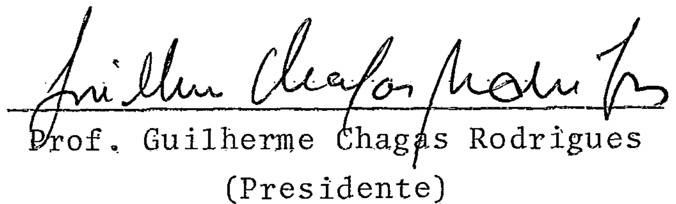
TERMINAL INGELIGENTE : ANÁLISE LÉXICA E GERAÇÃO DE CÓDIGO PARA UM COMPILADOR DA LINGUAGEM PL/STI

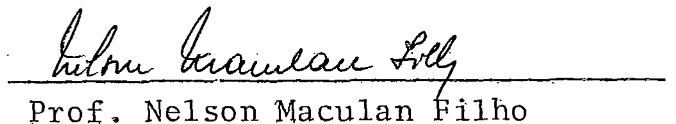
Miriam Aparecida Marques

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.).

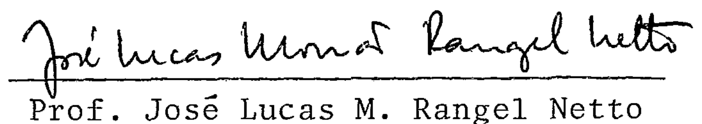
Aprovada por:



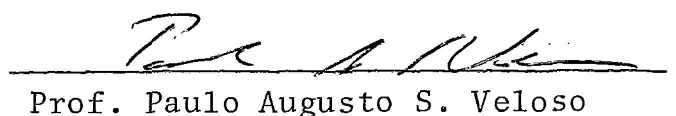
Prof. Guilherme Chagas Rodrigues
(Presidente)



Prof. Nelson Maculan Filho



Prof. José Lucas M. Rangel Netto



Prof. Paulo Augusto S. Veloso

RIO DE JANEIRO, RJ - BRASIL
JANEIRO DE 1978

MARQUES, MIRIAM APARECIDA

Terminal Inteligente: Análise Léxica e Geração de Código para um Compilador da Linguagem PL/STI |Rio de Janeiro| 1978.

X, 157. 29,7cm(COPPE-UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1978)

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.

Aos meus pais

AGRADECIMENTOS

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

À amiga Regina Célia de Souza Pereira pela colaboração na determinação da estrutura da Análise Léxica e Geração de Códigos.

Ao José Antonio dos Santos Borges (NCE) pelas valiosas sugestões apresentadas durante a depuração do código objeto.

RESUMO

Este trabalho constitui-se da Análise Léxica e Geração de Código para o compilador PL/STI, cujo objetivo é 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 a qual o compilador foi projetado.

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

No capítulo III, descrevemos a Análise Léxica e o tratamento dado aos erros encontrados durante esta fase. Focalizamos também a implementação de macros incluida na fase de Análise Léxica.

No capítulo IV, descrevemos a Geração de Código.

Finalmente no capítulo V, fazemos algumas considerações sobre o trabalho.

ABSTRACT

This work consists of the Lexical Analysis and Code Generation for the PL/STI Compiler. The aim of the Compiler is to facilitate the development of basic software for the Intelligent Terminal which is being constructed at the Núcleo de Computação Eletrônica da Universidade Federal do Rio de Janeiro.

Chapter I describes the target language for the Compiler, namely, PL/STI.

Chapter II gives the reader a general ideia of the Compiler and the way it works.

The Lexical Analysis, and the handling of errors encountered during that phase are explained in Chapter III. A view of the implementation of macros is also given.

The Code Generation phase is described in Chapter IV.

Finally, Chapter V provides some comments on the entire work.

ÍNDICE

Páginas

INTRODUÇÃO 1

CAPÍTULO I : A Linguagem PL/STI 3

1.1. Generalidades sôbre a linguagem 3

1.2. Constituintes básicos de um programa PL/STI . . . 4

1.3. Elementos de dados em PL/STI 5

1.4. Declarações de tipo para variáveis 6

1.5. Expressões e atribuições em PL/STI 6

1.5.1. Expressões 7

1.5.1.1. Operadores aritméticos 7

1.5.1.2. Operadores lógicos (ou booleanos) 8

1.5.1.3. Operadores relacionais (ou de comparação) 9

1.5.1.4. Precedência dos operadores PL/STI 10

1.5.2. Atribuições 11

1.6. Comando de declaração 12

1.6.1. Variáveis subscritas 13

1.6.2. O atributo INITIAL 14

1.6.3. A declaração DATA 15

1.6.4. Elementos de declaração 15

1.7. Ponteiros e referências indiretas 16

1.7.1. O operador ponto 17

Páginas

1.8. Comandos rotulados e comandos de desvios 19
1.8.1. Rótulos simbólicos 19
1.8.2. Rótulos numéricos 19
1.8.3. Comandos de desvio 20
1.9. O comando IF 21
1.10. Os comandos compostos 22
1.10.1. O comando composto DO 23
1.10.2. O comando composto DO WHILE 23
1.10.3. O comando composto DO iterativo 24
1.10.4. O comando composto DO CASE 26
1.11. Restrição ao uso de um comando IF 27
1.12. Rotinas 28
1.12.1. Declarações de rotinas 29
1.12.2. O comando RETURN 30
1.12.3. Exemplos de declarações de rotinas . .. 31
1.12.4. Restrições quanto ao uso de rotinas . .. 32
1.12.5. Chamadas de rotinas 32
1.12.6. Exemplos de chamadas de rotinas 33
1.13. Os comandos HALT e EOF 33
1.13.1. O comando HALT 33
1.13.2. O comando EOF 34
1.14. Macros em tempo de compilação 34
1.14.1. A declaração LITERALLY 34
1.14.2. Exemplo de uso de macros 35
1.15. Estruturas de blocos e alcance 36
1.15.1. Como o alcance é definido 36
1.15.2. Alcance de rótulos 37
1.15.3. Declaração de rótulos 37

Páginas

1.15.4. Uso da estrutura de blocos 41
1.16. Funções embutidas 41
1.16.1. Funções LENGTH e LAST 41
1.16.2. As funções LOW, HIGH e DOUBLE 42
1.16.3. Funções BYTE de rotação 43
1.16.4. Funções de rotação do CARRY 44
1.16.5. Funções de Shift-lógico 45
1.16.6. Funções CARRY,ZERO,SIGN e PARITY 46
1.17. Entrada e saída 46
1.17.1. INPUT 47
1.17.2. OUTPUT 48

CAPÍTULO II - O compilador PL/STI 49

2.1. Descrição geral 49
2.2. As fases do compilador PL/STI 50
2.2.1. Análise Léxica 50
2.2.2. Análise Sintática 51
2.2.3. Preparação para Geração de Código 51
2.2.4. Geração de Código 52
2.3. O processo de compilação da linguagem PL/STI . . 52

CAPÍTULO III - A Análise Léxica 54

3.1. Introdução 54
3.2. Visão Geral do procedimento para a Análise Léxica 55
3.2.1. Descrição das possíveis saídas da Análise Léxica 56

Páginas

3.2.2. Descrição da estrutura da Análise Léxica 57
3.2.2.1. Algoritmo para a Análise Léxica de um programa PL/STI 59
3.3. Processamento de Macros 64
3.3.1. Descrição Geral 64
3.3.2. Tratamento para declaração de macros . . . 66
3.3.3. Tratamento para chamadas de macros 68
3.4. Tramento de erros 70

CAPÍTULO IV - A Geração de Código 71

4.1. Introdução 71
4.2. Método usado para a Geração de Código 72
4.3. Visão Geral do procedimento para a Geração de Código 72
4.4. Estrutura do código objeto gerado 74
4.5. Organização do código objeto na memória 77
4.5.1. Alocação de memória para variáveis 80
4.5.2. Inicialização de variáveis 81
4.5.3. Algoritmo de compilação para um comando de declaração 81
4.6. Geração de Código de uma expressão PL/STI 83
4.6.1. Descrição Geral 83
4.6.2. Descrição do Método usado para gerar código de uma expressão PL/STI 89
4.6.3. Visão Geral da Implementação da pilha de expressões 90

Páginas

4.6.4. Alocação de Memória para variáveis temporárias 95
4.6.5. Tratamento para constantes 97
4.6.6. Tratamento para os possíveis tipos de operandos de uma expressão PL/STI 98
4.6.7. Algoritmo de Redução na pilha de expressões 100
4.7. Tratamento de Referências Futuras 107
4.8. Geração de Código para o comando IF 109
4.9. Geração de Código para uma rotina PL/STI 110
4.9.1. Introdução 110
4.9.2. Declaração de Rotinas 112
4.9.3. O corpo de uma rotina 113
4.9.4. Chamada de rotinas 113
4.10. Geração de Código para os Comandos Compostos 116
4.10.1. Geração de Código para o comando composto DO-WHILE 116
4.10.2. Geração de Código para o comando composto DO iterativo 118
4.10.3. Geração de Código para o comando composto DO-CASE 120
4.10.4. Geração de Código para o comando composto DO 123
4.11. Tratamento para rótulos 123
4.12. Geração de Código para comandos de desvio . . . 124
4.13. Geração de Código para comando de atribuição . . 126
CAPÍTULO V : Conclusões 128

Páginas

APÊNDICES

APÊNDICE A - A Gramática da Linguagem PL/STI 130

APÊNDICE B - Lista dos Caracteres Especiais 140

APÊNDICE C - Lista das Palavras Reservadas e Nomes das Funções Internas 143

APÊNDICE D - Tabelas Verdade para os Operadores Booleanos 145

APÊNDICE E - Diagrama de Estados para a Análise Léxica 147

APÊNDICE F - Programa exemplo com erros léxicos ... 150

APÊNDICE G - Programa exemplo compilado 152

REFERÊNCIAS BIBLIOGRÁFICAS 156

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 seria desenvolvido por duas pessoas.

Este trabalho consiste da Análise Léxica e Geração de Código para o compilador PL/STI. A Análise Sintática foi desenvolvida pela professora Regina Célia de S. Pereira em sua tese de mestrado, sob o título "Terminal Inteligente : Análise Sintática para um compilador da linguagem PL/STI".

A descrição da linguagem PL/STI consta dos dois trabalhos por tratar-se de parte comum, necessária para a compreensão de ambos.

No decorrer deste trabalho serão feitas referências à Análise Sintática, à Tabela de Símbolos e ao Tratamento de Erros em um programa PL/STI. Os dois últimos itens, embora sejam utilizados por todas as fases de compilação, encontram-se descritos apenas na tese sobre a Análise Sintática.

Neste trabalho serão utilizadas as seguintes abreviações:

T.S. - Tabela de Símbolos

A.L. - Análise Léxica

A.S. - Análise Sintática

G.C. - Geração de Código

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 obrigatoriamente 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 biná

ria, octal ou hexadecimal é representada respectivamente, pelas letras B,O 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 - 331O

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 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 supondo-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 (∅ - A). Assim -1 por exemplo, é equivalente a ∅ - 1, resultando em um valor BYTE igual à 255. MOD realiza divisão entre seus operandos, devolvendo como resultado de operação o resto da divisão.

As operações de adição e subtração afetam o indicador 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) ∅ 3 H - ∅ 4 H	b) ∅ A E H + 74 H
00000011	10101110
00000100	01110100
1| 11111111	1| 00100010
CARRY = 1	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. <u>Operadores relacionais (ou de comparação)</u>

Os operadores relacionais são usados em comparações e são:

< 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 ope

ração 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 ∅ (zero) a expressão é dita FALSA.

Exemplo:

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

Qualquer expressão aritmética, mesmo as que não contém operadores relacionais, podem ter valor verdadeiro ou falso, visto que somente 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.).

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 precedência são listados na mesma linha:

MENOS UNÁRIO

* / MOD

+ -

< < = <> = >

NOT

AND

OR XOR

Parênteses são usados para sobrepor a precedência 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 armazenado 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. Às 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 váriavel 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 virgulas. 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 implicitamente 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 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 descritos 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 dão 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 simplemente 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 variá

vel (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 mesma 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 do 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 coman

do 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 EXP 3 > 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 anterior, pode ser substituído 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 intenção 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 <u>não retorna um valor</u> , 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 'Ø FFH',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 usadas 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. Estrutura 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, ma-

cros, 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 rótu-

los, será definido ao nível do bloco da declaração. Esta declaração explícita é necessária somente 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, prèviamente 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. <u>As funções LOW, HIGH, DOUBLE</u>

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

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

HIGH(expressão) - retorna o byte de mais alta ordem 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. <u>Funções de Shift-lógico</u>

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

SHL(exp.1,exp.2)

SHL(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 ∅FFH. Se o indicador estiver desligado, então o valor ∅ é retornado.

1.17. Entrada e Saída

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

∅1 RR MMM1

onde RR = ∅∅ , para entrada

e

RR = ∅1, 1∅, 11 , 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 ∅-7, correspondendo ao campo MM 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 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 Código e Geração de Código propriamente dita.

2.2.1. Análise Léxica

A Análise Léxica é a primeira fase do processo de compilação. O módulo que a representa é ativado pela Análise Sintática toda vez que se faz necessário um novo elemento da linguagem PL/STI, ou seja, funciona como uma subrotina da Análise Sintática.

As funções básicas da Análise Léxica são: a leitura do programa fonte, o reconhecimento das cadeias de caracteres que determinam os elementos básicos da linguagem e a associação destes com uma estrutura léxica utilizada pela Análise Sintática.

Durante esta fase são executadas ainda, outras tarefas, entre as quais podemos citar:

- O processamento de macros.
- Conversão de constantes numéricas de sua representação alfanumérica para o valor inteiro correspondente.

- Consulta à Tabela de Símbolos para verificar se o identificador já foi inserido.
- Armazenamento do programa fonte, para posterior listagem no fim do processo de compilação.

2.2.2. Análise Sintática

A Análise Sintática de um programa PL/STI determina a estrutura formal do programa fonte dada a gramática da linguagem. À cada chamada da Análise Léxica, a Análise Sintática tenta enquadrar nas regras gramaticais da linguagem PL/STI, o elemento devolvido. Quando não 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 inserir informações e consultar a Tabela de Símbolos para verificação de erros, ou chamar as rotinas para gerar código.

O método usado para a Análise Sintática foi o Top-Down dos descendentes recursivos, implementado através de uma rotina que contém chamadas para as rotinas que analisam cada comando da linguagem.

2.2.3. Preparação para Geração de Código

Como o compilador PL/STI é de um passo, esta fase constitui-se basicamente da alocação de memória para as variáveis declaradas, que é um procedimento feito durante a compilação de um comando de declaração.

2.2.4. Geração de Código

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 código 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.

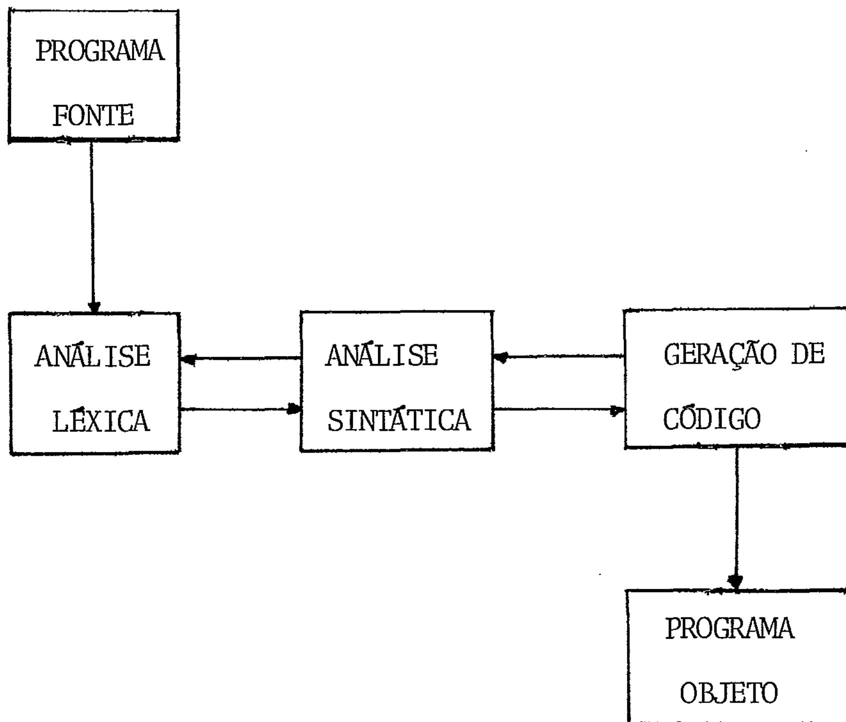


Figura 1

CAPÍTULO III

A ANÁLISE LÉXICA

3.1. Introdução

A Análise Léxica de um programa tem como objetivo determinar a estrutura formal dos elementos do programa fonte, de acôrdo com a gramática da linguagem. Uma vez reconhecida uma construção gramatical, as ações semânticas correspondentes são executadas pela Análise Léxica como visto na seção 2.2.

Neste capítulo apresentamos a idéia geral da fase de Análise Léxica, cujos detalhes podem ser vistos no algoritmo correspondente.

Focalizamos também a implementação de Macros, incluída na fase de Análise Léxica, e o tratamento dado aos erros encontrados durante essa fase.

No algoritmo usamos variáveis inteiras, alfanuméricas e lógicas que não foram especificadas como tal. O tipo dessas variáveis se torna claro no procedimento onde elas ocorrem.

3.2. Visão geral do procedimento para a Análise Léxica

A Análise Léxica consiste da leitura e reconhecimento dos caracteres utilizados no texto fonte para subsequente determinação de representação de elementos básicos, que vão constituir os terminais para a Análise Sintática. Também nessa fase são retirados elementos do texto, tais como caracteres brancos irrelevantes e comentários.

A informação de entrada para o compilador e portanto para a Análise Léxica é uma cadeia de caracteres, lida de cartões, que constitui um programa escrito em linguagem PL/STI.

O módulo relativo à Análise Léxica funciona como uma subrotina da Análise Sintática, sendo chamado sempre que a Análise Sintática necessita de um novo elemento do texto fonte.

Para cada elemento básico reconhecido, a Análise Léxica associa uma estrutura que contém informações que serão necessárias na Análise Sintática e Geração de Código. Essas informações constituem a saída da Análise Léxica, contêm a classe sintática (o tipo) do elemento e um atributo que completa sua descrição, como será visto na seção seguinte.

classe sintática do elemento	o próprio elemento	atributo do elemento

estrutura da saída da Análise Léxica

Figura 2

3.2.1. Descrição das possíveis saídas da Análise Léxica

A Tabela abaixo mostra como interpretar a saída da Análise Léxica.

CLASSE SINTÁTICA	ELEMENTO BÁSICO	ATRIBUTO
0	Identificador	Posição Na Tabela de Símbolos Caso Esteja Inserido
1	Constante Numérica	Valor Numérico
2	Operador de Divisão	Código Numérico
3	Operador de Atribuição Ou:	Código Numérico
4	Operador de Comparação	Código Numérico
5	Cadeia de Caracteres	Número de Caracteres
6	Outros Delimitadores	Código Numérico

Figura 3

3.2.2. Descrição da estrutura da Análise Léxica (A.L.)

A Análise Léxica foi estruturada de maneira a facilitar certas tarefas para as fases de Análise Sintática e Geração de Código, garantindo-se, no entanto, a eficiência desta fase, em que boa parte do tempo de compilação é gasto.

Com base na gramática da linguagem PL/STI, construímos um diagrama de estados (APÊNDICE E), a fim de facilitar a programação da rotina responsável pela A.L. .

Uma rotina que fornece caracteres é usada durante toda a fase de A.L. . Sua função é fornecer o próximo caracter do texto fonte a ser analisado juntamente com a classe a ele associada. Uma tabela onde a representação EBCDIC de um caracter é usada como índice contém as classes dos caracteres.

Além disso, essa rotina se encarrega de chamar outras rotinas que executam as seguintes tarefas:

1 - lê um cartão.

2 - grava esse cartão em um arquivo para posterior listagem do programa fonte no fim da compilação.

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

A cada chamada feita pela A.S. a A.L. tenta reconhecer um elemento da linguagem PL/STI a partir das produções da gramática, devolvendo-o para a A.S. . Caso isso não seja possível, uma mensagem de erro adequada é emitida.

Além das tarefas já citadas a A.L. executa as seguintes:

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

2 - Suprime comentários.

3 - Padroniza a entrada da Análise Sintática em triplas, como mostrado na figura 2 e descrito na seção 3.2.1.

4 - Processa as macros, como será descrito na seção 3.3.

5 - Consulta a Tabela de Símbolos, verificando se o identificador já está inserido, devolvendo a posição do mesmo na tabela em caso afirmativo ou uma mensagem que indica o caso oposto.

6 - Identifica erros léxicos e emite as mensagens correspondentes.

7 - Converte as constantes numéricas, de sua representação alfanumérica para o valor numérico inteiro correspondente.

8 - Converte as cadeias de caracteres de sua representação EBCDIC para ASCII, uma vez que este é o código usado para representação interna de caracteres alfanuméricos.

Para efeito de programação da rotina responsável pela A.L., associamos aos caracteres que podem começar um elemento da linguagem (ou constituir o próprio elemento, como é o caso dos delimitadores de um caracter) as seguintes classes numéricas:

0 - letras

1 - dígitos

2 - /

3 - :

4 - < , >

5 - '

6 - *,+,-, ; , , , = , (,) , .

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

3.2.2.1. Algoritmo para a Análise Léxica de um programa PL/STI

Lembramos que as abreviações T.S. e A.L, são u-

sadas para Tabela de Símbolos e Análise Léxica respectivamente.

```
procedure Análise Léxica;
begin
    chave:=true ;
    while chave do
    begin
        chave:=false ;
        comment procedimento que pesquisa o texto fonte até
                encontrar um caracter não branco;
        while caracter = "  "
        do Pega Caracter ; comment rotina que devolve o pró-
                          ximo caracter e respectiva classe;
        case classe of
        begin
            0:begin comment corresponde à um identificador;
                while classe ≤ 1  do
                begin
                    comment o caracter pode ser letra
                            ou dígito ;
                    i : = i+1 ;
                    simb|i|:=caracter ;
                    Pega Caracter ;
                    comment o identificador é armazenado
                    em simb um caracter por posição;
                end ;
```

```
        if identificador = nome de macro
        then begin
                  conjunto de procedimentos semân
                  ticos para expansão de macro;
                  chave := true ;
                  comment variável booleana liga-
                  da para indicar que a A.L. deve
                  pegar o próximo elemento da lin
                  guagem ;
             end ;
    end ;
  1:begin comment constante numérica;
        conjunto de procedimentos semânticos para
        reconhecer uma constante binária, octal,
        decimal ou hexadecimal e transformar essa
        constante de sua representação alfanumérica
        para o valor inteiro correspondente ;
    end ;
  2:begin comment o elemento é / ou um comentário
          se o próximo caracter for * ;
         Pega caracter ;
         if classe=6
         then begin
                  conjunto de procedimentos para
                  processar um comentário ;
                  chave:=true ;
                  comment variável booleana ligada
                  para indicar que a A.L. deve pe-
```

```
                  o próximo elemento da linguagem;
              end
         else codigo:= código numérico do
                           operador  / ;
    end ;
 3:begin comment o elemento pode
                  ser:  : ou :=  ;
         conjunto de procedimentos semânticos para
         reconhecer um delimitador :
         ou um operador de atribuição :=
         e devolver o código numérico correspondente;
    end ;
 4:begin comment corresponde ao caracter
          <  ou  ao  >   ;
         conjunto de procedimentos semânticos para
         reconhecer um dos seguintes operadores de
         comparação:
         < , > , <= , >=  ou  <> e devolver
         o código numérico associado ao operador;
    end ;
 5:begin comment corresponde ao caracter ' indi-
         cando o início de uma cadeia de caracte-
         res ;
      begin
       conjunto de procedimentos semânticos que
       reconhece uma cadeia de caracteres e guarda
       no vetor simb ;
```

```
    end ;
    if tamanho da cadeia > 255
    then erro
    else if é um texto de macro declarada
         then begin
                   conjunto de procedimentos se-
                   mânticos para armazenar o tex
                   to da macro;
              end ;
  end ;
6:begin comment corresponde aos outros delimita-
          dores de um caracter;
    if caracter = "#"
    then if está ocorrendo uma expansão de macro
         then begin
                conjunto de procedimentos semân-
                ticos para o término do processa
                mento de uma expansão de macro;
                chave:=true ;
                comment variável booleana ligada
                indicar  que a A.L. deve pegar a
                próximo elemento da linguagem;
              end
         else erro ; comment caracter inválido,
                     pois # é uma representação
                     interna de fim de macro, não
                     pertence a linguagem;
    else código:= código numérico do delimitador;
```

```
            comment os delimitadores podem ser
            + - * . ; , ( ) =     ;
         end ;
      end ; comment fim do comando  case classe;
   end ; comment  fim do comando while chave;
end ; comment fim da Análise Léxica;
```

3.3. Processamento de Macros

3.3.1. Descrição Geral

A idéia básica de macro na linguagem PL/STI é uma substituição de texto. Um identificador no programa fonte é substituído por uma cadeia de caracteres com a qual tenha sido associado em um comando de declaração de macro.

A esse identificador chamamos nome de macro e à cadeia de caracteres corpo ou texto. Quando a Análise Léxica reconhece um identificador deste tipo, dizemos que ocorreu uma chamada de macro. Neste caso é feita a substituição de texto, ao que chamamos expansão da macro.

Para exemplificar o procedimento, suponhamos a seguinte declaração de macro:

(a) DECLARE MAC LITERALLY ' B*C ' ;

e que no programa fonte encontramos

(b) A + MAC - R

esta expressão seria substituida por

(c) A + B*C - R

Assim o programador pode escrever o programa como em (b) e o texto a ser efetivamente compilado é o de (c).

De acôrdo com as regras gramaticais da linguagem PL/STI , uma chamada ou mesmo uma definição de macro pode aparecer dentro do texto de uma outra, sendo permitido assim o aninhamento de chamadas e definições de macros.

O alcance para identificadores declarados como nome de macro é semelhante ao definido para os outros identificadores da linguagem.

Uma vez terminada a rotina ou fechado o bloco onde foram declarados, os identificadores são retirados da T. S. juntamente com as respectivas informações. Este procedimento é feito para todo identificador em situação semelhante.

A função relativa ao processamento de macros, executada durante a A.L. consiste basicamente do armazenamento do texto e sua subsequente expansão na ocorrência de uma chamada.

Considerando o exemplo de uso de macros da seção 1.14.2 podemos verificar um aninhamento de chamadas de ma-

cros onde a macro TRUE é chamada dentro da macro FOREVER.

Para exemplificar um aninhamento de declarações de macros e o procedimento feito para este caso, suponhamos a seguinte declaração de macro:

DECLARE A LITERALLY ' DECLARE MAC LITERALLY '' XY '' ' ;

e que no programa fonte encontramos

A ;

este comando seria substituido por

DECLARE MAC LITERALLY 'XY' ;

3.3.2. Tratamento para declarações de macros

A forma geral de um comando de declaração de macro é a seguinte:

DECLARE <identificador> LITERALLY 'cadeia de caracteres ';

como foi visto na seção 1.14.1.

Na ocorrência de uma declaração deste tipo, são executados os seguintes procedimentos, durante a Análise Léxica:

Os textos de todas as macros são armazenados em um único vetor, um caracter por posição, sendo acrescentado após o texto de cada macro o caracter # . Esse caracter é usado apenas internamente, não estando acessível ao programador.

Na T.S. são inseridos o identificador declarado, a informação de que é um nome de macro e o índice que fornece a posição do início do texto no vetor onde são armazenados os textos das macros.

A última informação é utilizada sempre que a A.L. reconhece uma chamada de macro, quando se faz necessária a substituição do nome pelo texto correspondente, como será descrito na seção 3.3.3.

Exemplo do procedimento feito para as macros, LIT, DCL, TRUE, FALSE e FOREVER declaradas no trecho de programa da seção 1.14.2.

. . . .	
LIT	1
DCL	11
TRUE	19
FALSE	24
FOREVER	26
. . . .	

(a) Tabela de Símbolos

LITERALLY#DECLARE#ØFFH#Ø#WHILE TRUE#

(b) Vetor de textos de macros

(um caracter por posição)

Figura 4

3.3.3. Tratamento para chamadas de macros

Uma rotina que fornece caracteres é usada durante toda a fase de A.L., como foi descrito na seção 3.2.2.. Sua função é fornecer o próximo caracter que pode vir do texto fonte ou de uma macro.

Quando a A.L. reconhece um identificador, é feita uma consulta à T.S. . Se esse identificador é um nome de macro, o índice que fornece a posição do início do texto dessa macro no vetor de macros, como foi visto na seção 3.3.2., e a informação de que o texto de entrada corresponde a um texto de macro são passadas para a rotina que fornece caracteres.

Os caracteres da macro que está sendo expandida passam a ser analisados lexicamente até que o caracter # seja reconhecido, indicando que o texto dessa macro já foi completamente analisado.

A A.L. executa então um procedimento que restaura informações para a rotina que fornece caracteres, de forma que o texto de entrada, passe a ser novamente o que estava sendo analisado no momento em que ocorreu uma chamada de macro.

É utilizada uma pilha para guardar informações sôbre a macro e seu estado de processamento, de forma a permitir aninhamento de chamadas de macros, ou seja, a ocorrência de uma chamada em meio a uma expansão.

Para cada chamada, inserimos na pilha a posição do último caracter analisado, fornecido do texto fonte ou de uma macro.

Quando o texto fonte está sendo analisado, a pilha se encontra vazia. Neste caso se o caracter # for detetado será dada uma mensagem de erro para caracter inválido. Caso contrário está ocorrendo uma expansão de macro.

O nível de aninhamento de chamadas de macros é dado pelo apontador para o topo da pilha.

No exemplo da seção 1.14.2, a chamada da macro FOREVER tem nível 1 enquanto que a chamada da macro TRUE tem nível 2.

3.4. Tratamento de erros

A Análise Léxica deteta erro quando:

- um caracter não definido no alfabeto da linguagem PL/STI é encontrado.
- em uma constante numérica os caracteres não pertencem ao respectivo código.
- uma cadeia de caracteres ultrapassa o limite máximo permitido, 255 caracteres.
- o nome de uma macro é reconhecido e o respectivo texto não está gravado.

Nenhum tratamento de erros é feito para os casos citados acima, exceto para o último. Neste caso a A.L. procura a próxima ocorrência do caracter ";" , sendo então continuada a A.S. a partir desse ponto.

O modo como são armazenadas as informações sôbre os erros detetados durante a compilação de um programa PL/STI, é detalhado no trabalho sobre a Análise Sintática.

CAPÍTULO IV

A GERAÇÃO DE CÓDIGO

4.1. Introdução

A Geração de Código transforma cada construção gramatical reconhecida pela Análise Sintática em linguagem de máquina. Deste modo, quando a fase de Análise de um programa fonte em PL/STI termina, o programa objeto correspondente já foi gerado.

Se nenhum erro foi encontrado no programa fonte, o programa objeto gerado é perfurado em cartões, terminando assim a fase de compilação desse programa.

Em seguida, esse conjunto de cartões perfurados que constitui o programa objeto, é carregado no T.I., em posições fixas de memória, (como visto na seção 2.2.), para ser executado.

Neste capítulo damos a idéia geral do método escolhido para a Geração de Código e o modo como foi implementado.

Descrevemos a Geração de Código para expressões e para cada comando da linguagem.

Detalhamos com algoritmos os procedimentos mais interessantes. Nesses algoritmos usamos variáveis inteiras e lógicas que não foram especificadas como tal. O tipo dessas variáveis se torna claro no procedimento onde elas ocorrem.

4.2. Método usado para a Geração de Código

Usamos o método de tradução dirigida por sintaxe, no qual cada produção da gramática tem associada uma rotina encarregada de processar a geração de código relativa à estrutura sintática reconhecida no texto fonte.

Esta técnica que associa uma rotina a cada produção da gramática, simplifica o processo de geração de código, uma vez que é dividido em partes que podem ser programadas em separado. Dessa maneira, pequenas modificações na semântica e sintaxe requerem pequenas alterações nas rotinas.

4.3. Visão Geral do procedimento para a Geração de Código

O método descrito anteriormente é controlado pelas rotinas que fazem a A.S. de cada comando PL/STI. Cada uma dessas rotinas contém chamadas para as rotinas que processam a Geração de Código correspondente ao comando que está sendo analisado.

A cada comando PL/STI é então associado um con-

junto de instruções em código de máquina, que consiste da estrutura lógica correspondente ao comando.

Levando-se em consideração a necessidade de economizar memória, (a memória do T.I. tem 16K), fizemos uma seleção dos comandos usados com maior freqüência em um programa PL/STI. Essa seleção foi baseada em uma estatística de utilização de microrotinas feita para o Sistema Operacional em Disco (S.O.C.O.), atualmente em desenvolvimento no N.C.E. .

Assim os conjuntos de instruções associados aos comandos mais utilizados foram transformados em rotinas. A cada ocorrência de um comando desse tipo no programa fonte é gerado um código que consiste basicamente de uma chamada de rotina. Este procedimento foi feito para as operações lógicas e de comparação. Para os comandos utilizados com menor freqüência, tais como DO-CASE, DO-WHILE e funções internas (ou embutidas, ver seção 1.16) é feita a expansão do código correspondente.

As informações necessárias para a Geração de Código e verificações semânticas relativas a esta fase, são obtidas de:

- vetores auxiliares onde são armazenados endereços de memória. Esses endereços são utilizados na montagem da estrutura lógica de determinados comandos, como será visto na seção 4.7.

- Tabela de Símbolos.
- uma pilha usada para expressões como será descrito na seção 4.6. .

Durante a fase de Geração de Código são mantidas certas informações sobre o que acontecerá em tempo de execução, quando o código objeto gerado é executado. É mantida a informação de onde os operandos serão encontrados e os resultados armazenados, quando o código estiver sendo executado. Deste modo instruções de LOAD e STORE podem ser geradas de maneira apropriada à situação em tempo de execução.

4.4. Estrutura do Código Objeto Gerado

A linguagem de máquina do microprocessador INTEL-8008 |12|, não dispõe das operações de divisão e multiplicação. As operações de subtração e adição são executadas através de instruções próprias, mas que operam em apenas 1 byte. Como as variáveis e constantes de um programa PL/STI podem ser tipo BYTE ou ADDRESS (2 bytes), foi necessário a utilização de rotinas para essas operações. Foram feitas também rotinas para as operações lógicas, de comparação e complementação.

A única dificuldade do ponto de vista da geração de código é que os operandos devem ser carregados em posições fixas para a rotina. No caso, foi convencionado o uso

dos registros duplos BC e DE para receberem o 1º e 2º operandos, respectivamente.

Assim a expressão Y*Z + W poderia resultar no código:

LD1 Y	- o operando Y é carregado no registro BC
LD2 Z	- o operando Z é carregado no registro DE
CALL MUL	- chama rotina que executa a multiplicação devolvendo o resultado em BC
LD2 W	- o operando W é carregado no registro DE
CALL SOMA	- chama rotina que executa a soma deixando o resultado em BC

onde os registros BC e DE são chamados de registros 1 e 2, respectivamente.

As rotinas de operação dupla, citadas acima, sempre devolvem o resultado no registro BC, ao qual chamaremos de acumulador. A rotina que faz a complementação devolve o resultado no registro duplo DE.

Quando uma expressão faz parte de um comando IF, DO-WHILE ou qualquer outro comando onde se faça necessário testar o resultado da expressão, para se obter uma condição FALSA ou VERDADEIRA, a rotina que compila expressões é chamada

sendo gerado o código para avaliar a expressão.

O código que testa o valor da expressão, que está no acumulador ou num endereço de memória caso a expressão seja constituída de apenas uma variável, é gerado no local onde está sendo analisado o comando.

Exemplificando o procedimento, suponhamos o comando IF usado da seguinte forma:

```
IF   A = B  THEN comando 1 ; comando 2 ;
```

Na rotina que compila expressões, seria gerado o seguinte código:

```
LD1  A
LD2  B
CALL COMPIGUAL - chama a rotina que faz a comparação devolvendo o resultado no registro duplo BC
```

e na rotina que compila o comando IF, o código gerado seria:

```
LDAC           - carrega no registro A o conteúdo do registro C
RAR            - desloca o bit mais baixo do registro A para o bit de condição CARRY.
```

```
JFC L1        - desvia para o comando 2 se for
                zero
    - - - -)
    - - - - }   código gerado para o comando 1
    - - - -)
L1: - - - -)
    - - - - }   código gerado para o comando 2
    - - - -)
```

4.5. Organização do Código Objeto na Memória

Um dos primeiros passos na definição da geração de código, foi determinar a organização do código objeto na memória do computador (T.I.).

Optamos por duas formas diferentes de organização, como mostram as figuras 5a e 5b.

A forma escolhida pelo programador é indicada através de um cartão de controle .

O tamanho total de memória que o programador pretende utilizar, assim como o total de área ROM (se optou por esse tipo de divisão), também são indicados através de cartões de controle. Neste caso, a área RAM será o complemento da área ROM.

Se o tamanho de memória não for especificado, o compilador assumirá 16K. O mesmo ocorre com a área ROM, para a qual serão reservados 10K.

No caso mostrado na figura 5a, a memória é dividida em duas áreas: ROM e RAM. Na área ROM são armazenadas as rotinas básicas seguidas do código objeto gerado para o programa fonte.

Se durante a compilação de um programa PL/STI for encontrado um comando do tipo GO TO número ou um rótulo numérico, o compilador faz testes que verificam se o número que aparece no comando GOTO ou representa o rótulo, é um endereço fora da área ROM ou da área reservada para o programa fonte na ROM. Em caso afirmativo, emite uma mensagem de erro e o programa não será executado.

As variáveis declaradas com tipo DATA são alocadas na área ROM como mostra a figura 5a.

Na área RAM, são armazenados os dados e alocadas as variáveis comuns.

É feito ainda um controle, durante a compilação, sôbre o espaço disponível em cada área de memória. Em caso de estourar uma dessas áreas, o compilador emite uma mensagem apropriada e a compilação é interrompida.

A escolha deste tipo de divisão de memória foi baseada no fato de que certos terminais de programação fixa tem a memória assim dividida, além de oferecer facilidades ao programador nas tarefas de programar e depurar um programa em PL/STI.

No segundo caso, como mostra a figura 5b, as

variáveis tipo DATA são alocadas junto com os dados e as variáveis comuns. Esta área cresce em sentido contrário à do programa fonte.

Nesse caso também é feito um controle sobre o espaço de memória disponível. Se esse espaço se esgotar, um procedimento semelhante ao já descrito é executado.

Em ambos os casos, é dada uma mensagem de erro durante a compilação, caso uma variável tipo DATA apareça do lado esquerdo em um comando de atribuição.

Além disso uma área de tamanho fixo, localizada nas últimas posições de memória, é reservada para variáveis temporárias.

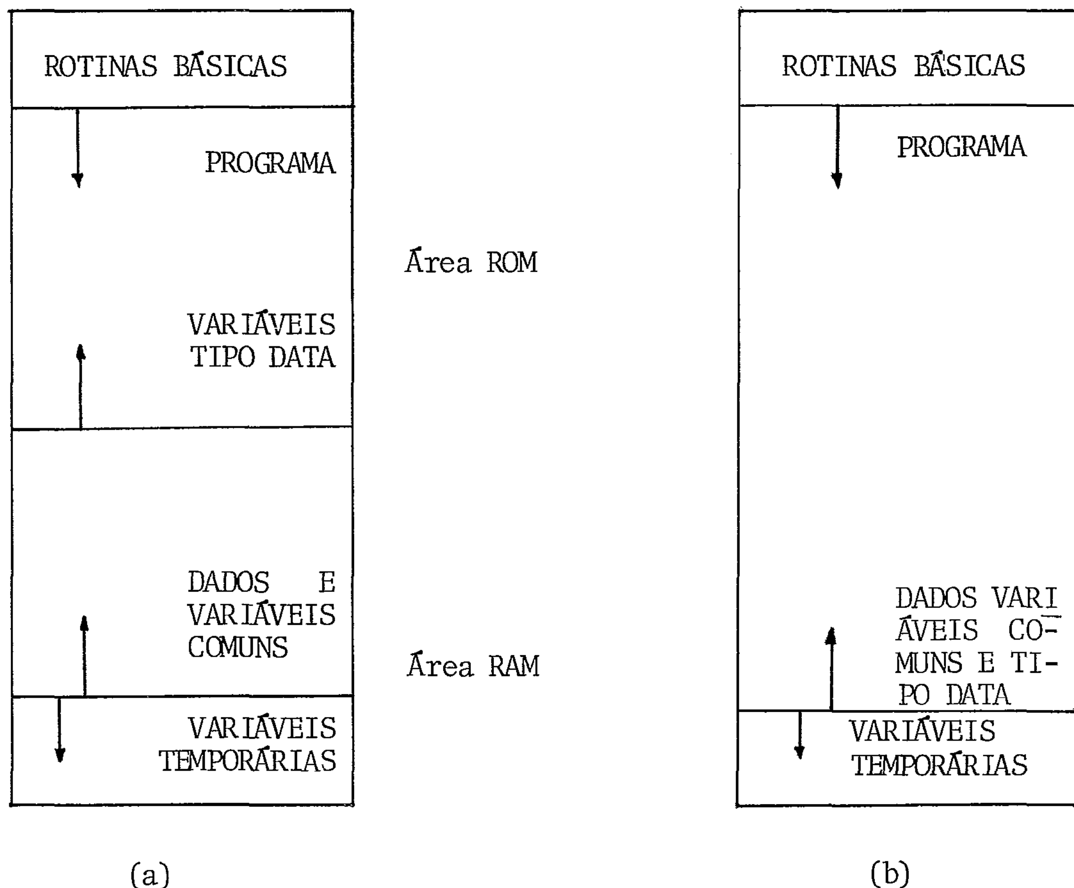


Figura 5

4.5.1. Alocação de memória para variáveis

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,B,C,D)BYTE, M(20)ADDRESS, C(2)INITIAL(0,1);

como visto na seção 1.6.4.

Durante a compilação dos <elementos da declaração> são feitos procedimentos semânticos tais como inserir um identificador na T.S. juntamente com seus atributos, consultar a T.S. para verificação de erros, alocar memória para as variáveis e gerar código que as inicializa.

As posições na T.S. onde foram inseridas as variáveis declaradas, são armazenadas em uma pilha. Essa informação é utilizada na alocação de memória, quando se faz necessário consultar a T.S., para se obter informações sôbre cada variável declarada, para a qual vai ser alocada memória.

No final da rotina que compila um comando de declaração PL/STI, é feito um procedimento que aloca memória para as variáveis, de acôrdo com as atributos associados (tipo BYTE ou ADDRESS e o número de palavras).

Na T.S. é inserido o endereço de memória alocado para cada variável, para ser utilizado posteriormente na geração de código que envolva essas variáveis.

Uma variável declarada como LABEL ou BASED não tem memória alocada.

Variáveis de mesmo nome, declaradas em blocos diferentes, tem posições diferentes de memória alocadas.

4.5.2. Inicialização de variáveis

Quando à rotina que compila um comando de declaração PL/STI reconhece um atributo INITIAL ou uma declaração DATA, seguidos de uma lista de constantes, é chamada a rotina que gera código de inicialização.

4.5.3. Algoritmo de compilação para um comando de declaração

Descrição das variáveis utilizadas no algoritmo:

PILHA - contém as posições na T.S. das variáveis declaradas.

PONTEIRO - variável que fornece o próximo endereço de memória disponível na

área de dados.

K - ordem em que uma variável apareceu na declaração.

```
procedure declare ;
begin
    begin
        compilação dos
        <elementos da declaração>
    end ;
    comment procedimento que aloca memória para variáveis;
    for K:=1  step 1 until número de variáveis declaradas do
    begin
        ptab := pilha |K|; comment posição na T.S. da K-ésima
                                   variável declarada;
        if a variável não é based
        and não é rótulo declarado
        then begin
            ponteiro: = ponteiro - total de posições ;
            T.S.|ptab,22|: = ponteiro;
            comment guarda o endereço alocado na T.S. .  O
            número de posições alocadas é igual ao número de
            posições declaradas vezes o tipo que pode   ser
            byte = 1 palavra ou address = 2 palavras ;
            end ;
    end ;
```

```
if foi declarado atributo INITIAL
  or foi declarado o atributo DATA
then begin
          geração de código
          que inicializa variáveis;
     end ;
if elemento = "," then declare
                  else if elemento ≠ ";"
                       then erro ;
end da rotina declare;
```

4.6. Geração de Código de uma expressão PL/STI

4.6.1. Descrição Geral

A rotina que processa a geração de código de uma operação, é chamada pela rotina que analisa sintáticamente uma expressão. Essa chamada é feita toda vez que uma operação deve ser reduzida. As condições que determinam quando uma redução deve ser feita, são definidas em função da prioridade dos operadores.

As ações semânticas relativas a geração de código de uma expressão, utilizam uma descrição na T.S. de cada variável. São utilizadas também informações fornecidas pela Análise Léxica, tais como o valor de uma constante, o com-

primento de uma cadeia de caracteres e o código de um operador.

A rotina que processa a geração de código de uma operação utiliza uma descrição em uma pilha, usada para expressões, de cada operando e da operação para a qual vai ser gerado o código.

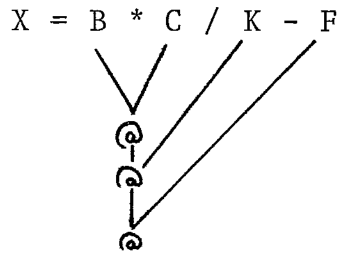
Essa pilha de expressões contém os operadores e respectivas prioridades, operandos e informações sôbre eles.

A geração de código para uma expressão é feita de modo que as operações que aparecem na expressão, sejam executadas segundo a ordem de precedência dos operadores PL/STI. Essa ordem é alterada apenas quando são utilizados parênteses.

Foram necessárias variáveis temporárias para armazenar resultados parciais das expressões. A forma como é feita a alocação e como essas variáveis são utilizadas serão descritas na seção 4.6.4. .

O resultado de toda operação é deixado num registro duplo, que é utilizado pelas rotinas de operações aritméticas, lógicas e de comparação. Esse resultado só é armazenado em uma variável temporária quando se faz necessário utilizar o registro para outra operação.

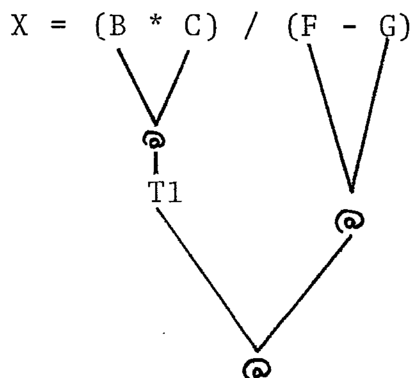
1) Exemplo de uma expressão para a qual não são necessárias variáveis temporárias (@ representa o acumulador) :



o código gerado para essa expressão é:

```
LD1   B
LD2   C
CALL  MULT
LD2   K
CALL  DIV
LD2   F
CALL  SUB
STO   X
```

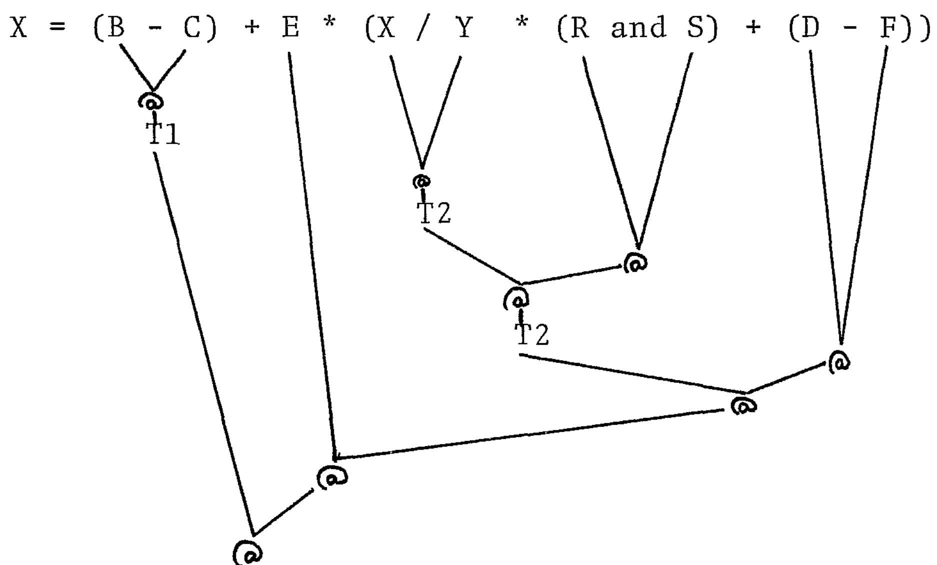
2) Exemplo de uma expressão para a qual é necessário alocar uma variável temporária:



Nesse caso, o resultado da operação de multipli-cação é armazenado em BC (registro duplo), que em seguida se faz necessário. O código gerado para essa expressão seria:

```
LD1  B
LD2  C
CALL MUL
STO  T1
LD1  F
LD2  G
CALL SUB
LD2  T1
CALL DIV.REVERSA
```

3) O exemplo abaixo mostra a utilização de temporárias de modo otimizado.



Toda temporária é liberada assim que a operação da qual ela faz parte é executada.

Podemos observar nesse exemplo que toda vez que se faz necessário utilizar o acumulador, o conteúdo deste registro é armazenado em uma temporária (Ti) e que toda temporária é liberada assim que a operação da qual ela faz parte é executada.

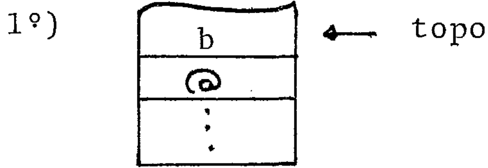
A utilização de uma pilha onde são inseridos os operadores juntamente com as respectivas prioridades dinâmicas, permite o controle da geração de código para as operações de acôrdo com a ordem na qual elas devem ser executadas.

É feito um controle sôbre o estado atual do acumulador (registro BC), onde são feitas as operações e armazenados os resultados durante a execução. Assim evitamos operações de LOAD e STORE desnecessárias.

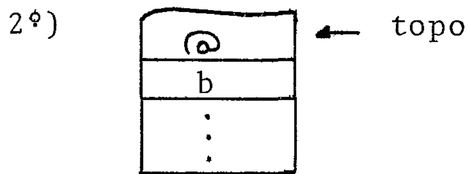
Usamos uma variável durante a compilação que nos indica o estado do acumulador em tempo de execução.

O acumulador só estará disponível quando não estiver na pilha de operandos.

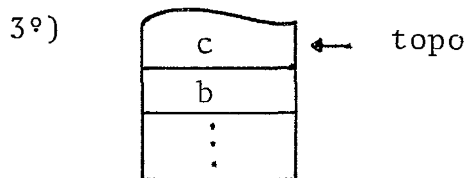
Podemos ter então os seguintes, casos no momento de gerar código para uma operação:



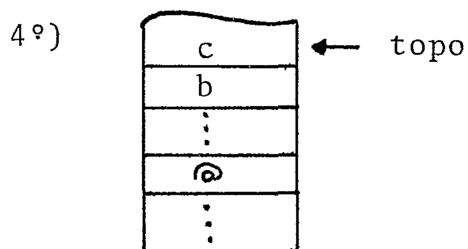
- é gerado um código para operação normal, onde apenas o 2º operando deve ser carregado no registro 2 (DE). O acumulador é colocado no topo da pilha.



- o primeiro operando é carregado no registro 2(DE) e a operação vai ser reversa. O acumulador é colocado no topo da pilha.



- o acumulador não está na pilha. O código gerando carrega os operandos nos registros 1 e 2 e chama a rotina que executa a operação. O acumulador é colocado no topo da pilha.

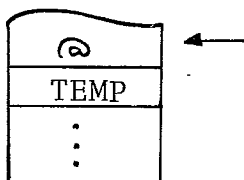


- nenhum dos dois operandos está no acumulador, porém o acumulador está na pilha.

Aloca-se uma temporária, gerando código para armazenar o acumulador nessa temporária. Troca-se na pilha

pela variável temporária recaindo no 3º caso.

```
STO   TEMP
LD1    b
LD2    c
CALL  OP
```

5º) 

- o primeiro operando está em uma variável temporária. Recai no 2º caso, com a diferença de que será liberada uma variável temporária.

4.6.2. Descrição do Método usado para gerar código de uma expressão PL/STI

Foram atribuidas as seguintes prioridades aos operadores PL/STI, de acordo com a precedência:

MENOS UNÁRIO	7
* / MOD	6
+ -	5
< <= > >= <> := =	4
NOT	3
AND	2
OR XOR	1

O operando é inserido direto na pilha, o operador só é inserido se a prioridade dinâmica for maior que a prioridade do operador que está no topo da pilha. Caso contrário é feita uma redução, ou seja, é chamada a rotina que gera código para a operação indicada no topo da pilha. Neste caso são retirados da pilha o operador, a prioridade, e os operandos.

A prioridade dinâmica é o valor da prioridade estática do operador mais a base no momento da inserção.

O valor inicial da base é zero, sendo incrementada de 8 (que é um valor maior que a maior prioridade atribuída) quando aparece um abre-parênteses, e decrementado de 8 quando aparece um fecha-parênteses.

4.6.3. Visão Geral da Implementação da pilha de expressões

A pilha utilizada para avaliação de expressões tem 5 campos, onde são inseridas as seguintes informações:

1 - o operador.

2 - a prioridade dinâmica do operador.

3 - o operando

4 - uma identificação indicando se o operando é uma constante, uma variável do programa,

uma variável temporária ou está no acumulador.

5 - o tipo do operando, que pode ser BYTE ou ADDRESS

Inicialmente é inserida na pilha, no campo correspondente à prioridade, o valor -3 delimitando o início de uma expressão.

Para cada operando reconhecido pela A.S. são feitos procedimentos semânticos que inserem esse operando na pilha juntamente com informações relativas à ele.

O operador e a respectiva prioridade dinâmica são tratados como foi descrito na seção anterior.

No decorrer da análise de uma expressão, quando as condições para uma redução ocorrem, é chamada a rotina que processa a geração de código correspondente a operação indicada no topo da pilha.

Os procedimentos semânticos executados por essa rotina consistem em:

1) Geração de código para carregar um operando no registro duplo BC se a operação é unária. Em caso contrário carrega o primeiro e o segundo operando nos registros duplos BC e DE respectivamente.

2) Alocação de variável temporária e geração de código para copiar nessa variável o conteúdo do acumulador quando este deve ser liberado.

3) Liberação de uma variável temporária, a partir do momento em que não esteja sendo mais necessária.

4) Geração de código para chamar a rotina adequada, que vai executar a operação indicada no topo da pilha.

5) Retirada de um ou dois operandos da pilha (dependendo da operação ser unária ou não), juntamente com as informações associadas. Esse procedimento só é feito quando o código relativo à operação já foi gerado.

6) Retirada do operador e da sua prioridade.

7) Inserção na pilha do acumulador e informações associadas.

Quando a rotina que compila uma expressão, reconhece um delimitador de fim de expressão, o valor -2 é atribuido à prioridade dinamica. Esse valor é menor que qualquer prioridade dos operadores PL/STI e maior que a prioridade que está no fundo da pilha. É simulada a tentativa de inserção de um operador com essa prioridade. Como não consegue, todas as operações que permanecem na pilha são reduzidas, sendo completada a geração de código para uma expressão.

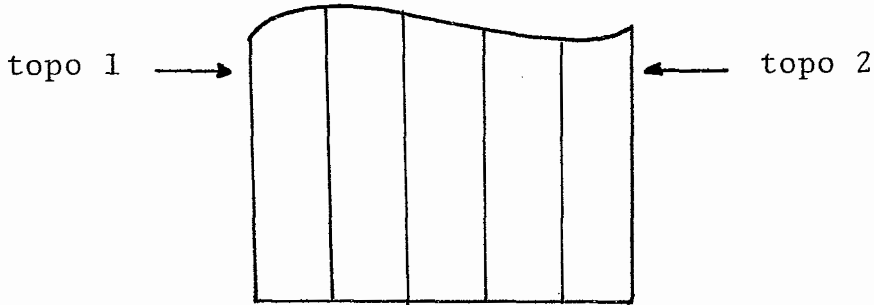
O endereço onde estará o resultado da expressão durante a execução é colocado na pilha, para ser utilizado pela rotina que chamou a rotina que compila expressões (como o exemplo visto na seção 4.4).

Durante a compilação de uma expressão, se um (

é reconhecido, a variável base é incrementada de 8 como foi descrito na seção 4.6.2.. Em seguida, a rotina que compila uma expressão é chamada recursivamente para compilar a expressão que está entre parênteses. Quando um) é encontrado, retorna para o ponto de chamada. Neste ponto então, a variável base é decrementada de 8.

A Análise da expressão é continuada até que um delimitador de fim de expressão seja encontrado.

A seguir, mostramos a organização das informações na pilha de expressões aritméticas.



pilha |topo1,1| - prioridade dinâmica do operador

pilha |topo1,2| - o operador

pilha |topo2,3| - um endereço de memória ou o valor de uma constante

pilha |topo2,4| - informações sobre o conteúdo da pilha |topo2,3|

0 - é uma constante

1 - um endereço de memória

2 - um endereço de memória correspondente à uma variável temporária.

pilha |topo2,5| - tipo do operando

1 - BYTE

2 - ADDRESS

A variável topo1 é incrementada quando vão ser

inseridos um operador e sua prioridade dinâmica, e decrementada em caso contrário. A variável topo2 é incrementada na inserção de um operando e respectivas informações, e decrementada em caso contrário.

4.6.4. Alocação de Memória para Variáveis Temporárias

Durante a geração de código para expressões aritméticas são necessárias posições temporárias de memória para armazenar os resultados parciais de uma expressão, quando o código for executado.

Levando em consideração a limitação de memória, resolvemos alocar uma área fixa na memória, reservando posições sempre que necessário e liberando assim que possível. Assim podemos visualizar a utilização dessa área como de uma pilha.

O controle do ponteiro para o topo dessa pilha é feito durante a compilação.

A cada posição dessa área correspondem 2 palavras do T.I. (2 bytes). As variáveis temporárias são tipo ADDRESS.

Em referências posteriores, chamaremos esta área de pilha de temporárias.

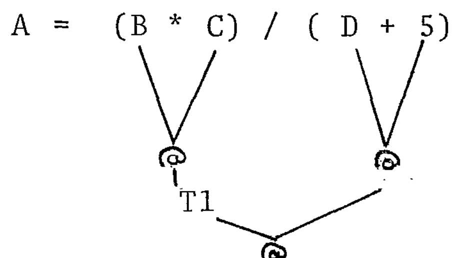
Os endereços temporários alocados para expressões que aparecem em uma rotina com tipo declarado não são liberadas, o que implica num deslocamento da base da pilha de temporárias.

Esse tratamento foi necessário porque um nome de rotina declarada com tipo pode aparecer dentro de uma expressão, o que poderia acarretar na utilização indevida de uma posição de memória, e consequente erro na execução do programa objeto.

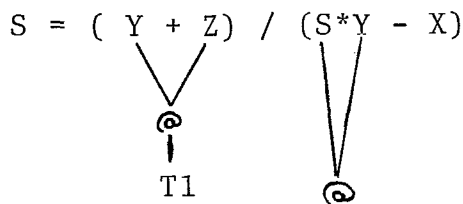
A seguir damos um trecho de programa e um exemplo de como esse tipo de erro poderia ocorrer:

```
                - - - - - - - - - - - - - - -
                - - - - - - - - - - - - - - -
                DECLARE (S,Y,Z) ADDRESS ;
                X:PROCEDURE  BYTE ;
                   DECLARE (A,B,C,D)  BYTE ;
(1)                A = (B*C) / (D+5) ;
                   RETURN A ;
                  END ;
                - - - - - - - - - - - - - - -
                - - - - - - - - - - - - - - -
(2)             S = (Y + Z) / (S*Y - X)
                - - - - - - - - - - - - - - -
                - - - - - - - - - - - - - - -
```

Na geração de código para a expressão (1) que aparece na rotina seriam utilizadas as seguintes variáveis temporárias:



Se essas variáveis fossem liberadas no fim da geração de código para a rotina, quando fosse encontrada a expressão (2), as variáveis temporárias usadas seriam:



quando a chamada de X fosse reconhecida e executada, a posição T1 seria utilizada, sendo destruído o valor de (Y + Z) que lá estava armazenado.

4.6.5. Tratamento para Constantes

As constantes de um programa PL/STI são convertidas para a forma de representação interna do T.I., binária. As constantes tipo BYTE são consideradas estritamente positivas.

Nenhuma alocação de memória é feita para as constantes, exceto quando elas aparecem na declaração de certos comandos, tais como o DO iterativo e o DO-WHILE e que será visto nas seções que descrevem a geração de código para cada comando.

4.6.6. Tratamento para os possíveis tipos de operandos de uma expressão PL/STI

Durante a A.S. de uma expressão PL/STI são reconhecidos os seguintes tipos de operandos:

- Identificador { nome de função (ver seção 1.16.); nome de rotina; variável }
- Constante numérica.
- Cadeia de caracteres.
- Constantes numéricas, lista de constantes entre parênteses, cadeias de caracteres ou uma variável, precedidos de ponto.

Os procedimentos semânticos e a geração de código efetuados para cada caso são os seguintes:

<u>nome de função (embutida)</u> : certas convenções semânticas da linguagem são testadas, tais como o número de parâmetros e os

respectivos tipos, caso seja uma função com parâmetros. Em seguida, se não foi encontrado nenhum erro, é chamada a rotina que gera código adequado para executar a função. O resultado fica no acumulador. No topo da pilha de expressões é inserida a informação de que o operando está no acumulador juntamente com o tipo que pode ser BYTE ou ADDRESS.

nome de rotina: é feito o procedimento para uma chamada de rotina, descrito na seção 4.9.4. . O endereço do nome da rotina é inserido na pilha juntamente com o tipo que pode ser BYTE ou ADDRESS.

nome da variável: para variável indexada, é gerado o código que calcula o valor do índice e em seguida soma esse valor ao endereço alocado para essa variável, obtendo-se assim o endereço da posição do vetor que está sendo referenciado. Como esse endereço será obtido durante a execução, é gerado o código que copia o conteúdo dessa posição de memória no acumulador, sendo o mesmo inserido na pilha de operandos. Se a variável é simples, apenas o endereço de memória alocado, e que está na T.S. é inserido na pilha de operandos. Em ambos os casos o tipo também é inserido.

Temos ainda o caso da variável declarada como tipo BASED, para a qual não é alocada posição de memória quando declarada. Na ocorrência de uma variável deste tipo é gerado o código que pega o conteúdo da variável declarada como sendo sua base e carrega no acumulador. O valor armazenado no a-

cumulador é o endereço que será utilizado para essa variável. É inserida na pilha de expressões, a informação de que se trata de um endereço de memória que está no acumulador.

constante numérica: a constante é inserida na pilha juntamente com as informações de que é uma constante e o respectivo tipo, BYTE ou ADDRESS. Nenhum código é gerado.

cadeia de caracteres: neste caso, é alocado um número de posições na área de dados, na memória igual ao número de caracteres da cadeia. Em seguida essa cadeia de caracteres é armazenada nessa área reservada, a partir do endereço de início.

O endereço do início dessa área é inserido na pilha de expressões juntamente com a informação de que se trata de um endereço e o respectivo tipo que é ADDRESS.

operandos precedidos de ponto: se o operando é um identificador, o endereço alocado é inserido na pilha. Caso seja uma constante numérica ou uma cadeia de caracteres, é feito o procedimento descrito para operandos deste tipo. O endereço de memória é inserido na pilha de expressões. Além disso são inseridas as informações de que se trata de uma constante e que o tipo é ADDRESS.

4.6.7. Algoritmo de Redução na pilha de expressões

O algoritmo a seguir é chamado dentro da rotina que compila uma expressão PL/STI, no ponto em que foi necessária a redução de um operador da pilha.

```
procedure reduz ;
if not operação de atribuição then
begin if operação unária
      then begin
              if acumulador não está na pilha
              then begin
                      conjunto de procedimentos
                      semânticos para carregar
                      o operando no acumulador ;
                   end ;
              else if acumulador não está no topo
                   then begin
                          comment o operando não está no acu-
                          mulador, mas o mesmo se encontra na
                          pilha;
                          conjunto de procedimentos semânti-
                          cos para alocação de uma variável
                          temporária, geração de código para
                          carregar o conteúdo do acumulador nes
                          sa variável e o operando no acumulador;
                        end
           end
```

```
else if acumulador não está na pilha
     then begin
               conjunto de procedimentos semânticos que
               gera código para carregar os dois operan
               dos nos registros duplos BC e DE respec-
               tivamente ;
          end
     else  comment o acumulador está na pilha ;
         if acumulador está no topo-1
         then begin
                  comment o primeiro operando já está no
                  acumulador;
                  conjunto de procedimentos semânticos
                  que gera código para carregar o segun-
                  do operando no registro duplo DE ;
              end
          else if acumulador está no topo
               then begin
                        comment o segundo operando já es-
                        tá no acumulador;
                        conjunto de procedimentos semanti
                        cos que gera código para carregar
                        o primeiro operando no registro du
                        plo DE e indica que a operação é
                        reversa.
                        if o primeiro operando estava em
                           uma variável temporária
                        then begin
```

```
                            procedimento semântico que
                            libera uma variável tempo-
                            rária ;
                          end ;
                   end
               else begin
                     comment nenhum dos dois operandos
                     está no acumulador, mas o acumula
                     dor se encontra na pilha;
                     conjunto de procedimentos semânti
                     cos para alocação de uma variável
                     temporária, geração de código pa-
                     ra carregar o conteúdo do acumula
                     dor nessa variável e os dois ope-
                     randos nos registros duplos BC  e
                     DE respectivamente. Finalmente
                     insere uma informação, na pilha,
                     de que o operando que estava   no
                     acumulador passou a ser variável
                     temporária;
                   end ;
end ; comment neste ponto já foi gerado o código que carrega
      os operandos nos registros BC e DE, ou só em BC quando
      a operação é unária,exceto para operação de atribuição;
case código da operação of
begin
   0:begin comment operação de divisão.
```

```
        Procedimento que gera código para chamar a rotina
        que executa a divisão, deixando o resultado    no
        acumulador;
        tipo do resultado : = address;
   end ;
 1:begin comment operação de atribuição;
        operador : =  ;
        gera código de atribuição;
        comment chama rotina que faz o procedimento neces
        sário;
        tipo do resultado : = tipo do operando que recebe
        o valor ;
   end ;
 comment operadores >=, >, <=, <>, <
 2:begin comment operador >=.  Procedimento que gera códi-
        go que testa os sinais dos operandos e chama a ro
        tina que executa a comparação e devolve o resulta
        do no acumulador.
        tipo do operando : = byte ;
   end ;
 3:begin
   end ;
    '
    '
    '
    '
 7:begin comment operador +.  Procedimento que gera código
        para chamar a rotina que executa a soma deixando
        o resultado no acumulador;
```

```
        tipo do resultado : = address ;
    end ;
  8:begin comment operação de subtração ou - unário ;
        if prioridade = 7
        then begin
                comment menos unário.
                Procedimento que gera código para executar
                a operação deixando o resultado no acumula
                dor;
                tipo do resultado : = tipo do operando ;
             end
        else begin comment operação de subtração. É gera-
                  do o código que executa a subração dei-
                  xando o resultado no acumulador;
                  tipo do resultado : = if
                  os dois operandos são byte then byte
                                              else address;
             end
  9:begin comment operação de multiplicação.
         O procedimento é semelhante ao feito para divisão;
    end ;
 10:begin comment operação de atribuição ou de comparação o
         perador = ;
         if operação de atribuição then
         begin comment procedimento igual ao feito para  o
               operador : = ;
         end
                                   else
```

```
            begin comment procedimento igual ao feito para os
                  outros operadores de comparação;
            end ;
      end ;
    comment operações lógicas OR, XOR, AND, MOD, NOT .
            O procedimento que gera código para estas opera-
            ções consiste da chamada da rotina que executa  a
            operação deixando o resultado no acumulador;
   11:begin comment operação XOR ;
      end ;
      '
      '
      '
   15:begin  comment operação NOT ;
      end ;
end ; comment fim do case operação ;
 if a operação não era unária
 then topo da pilha : = * -1 ;
 comment procedimento que retira o operador e respectiva prio
         ridade da pilha, retira os operandos e informações a
         dicionais relativas a esses operandos, inserindo  na
         pilha o acumulador e informações sobre esse resulta-
         do;
 PILHA|TOPO,4|:= é o acumulador;
 PILHA|TOPO,5|:= tipo do resultado ;
end ; comment fim da procedure reduz ;
```

4.7. Tratamento de Referências Futuras

Durante a geração do código correspondente aos comandos: IF, DO CASE, DO WHILE, DO ITERATIVO, declaração e chamada de rotinas, deparamos com o problema de referências futuras, que nos leva à geração de uma ou mais instruções de desvio para endereços ainda não conhecidos.

A solução por nós adotada foi a utilização de vetores auxiliares, que durante a compilação guardam os endereços dos operandos dessas instruções de desvio. Esses operandos são substituídos à medida que seus endereços se tornam conhecidos.

Os vetores auxiliares são variáveis do compilador, declaradas nas rotinas que compilam cada tipo de comando. As posições de um vetor auxiliar, estão associadas a instruções de desvio específicas, do código gerado para um comando.

Algumas posições desses vetores são utilizadas para guardar endereços para onde serão feitos desvios posteriormente, durante a geração de código para um comando.

Para exemplificar o procedimento, consideremos a forma geral de um comando composto DO-WHILE:

```
DO  WHILE  EXPRESSÃO ;
    comando1 ;
    comando2 ;
     '
     '
     '
    comandoM ;
END ;
```

A configuração do código gerado para esse comando composto é a seguinte:

```
L1: - - - - - - - - - -)   código gerado para ex-
    - - - - - - - - - -}   pressão lógica
    - - - - - - - - - -)

E1: TESTE, se falso desvia para L2

    - - - - - - - - - -)   código gerado relativo
    - - - - - - - - - -}   aos comandos1,..., co-
    - - - - - - - - - -)   mando n

    ___________________}   desvio para L1.

L2:
```

Como podemos observar, o endereço L1 é utilizado quando termina a geração de código para o comando DO-WHILE, sendo gerada uma instrução de desvio para o endereço L1. O endereço L2 não é conhecido quando é gerada uma instrução de desvio para ele no endereço de memória E1. Assim podemos utilizar o vetor auxiliar para armazenar os endereços L1 e E1 na primeira e segunda posições respectivamente.

O procedimento que gera código para preencher

esses operandos das instruções de desvio, é executado no final da rotina que compila cada comando.

Como os vetores auxiliares são declarados em cada rotina, no caso de recursividade de comandos, por exemplo um DO-WHILE fazendo parte de um dos comandos que aparecem dentro de um DO-WHILE, essas variáveis serão recriadas quando a rotina que analisa o comando for chamada.

4.8. Geração de Código para o comando IF

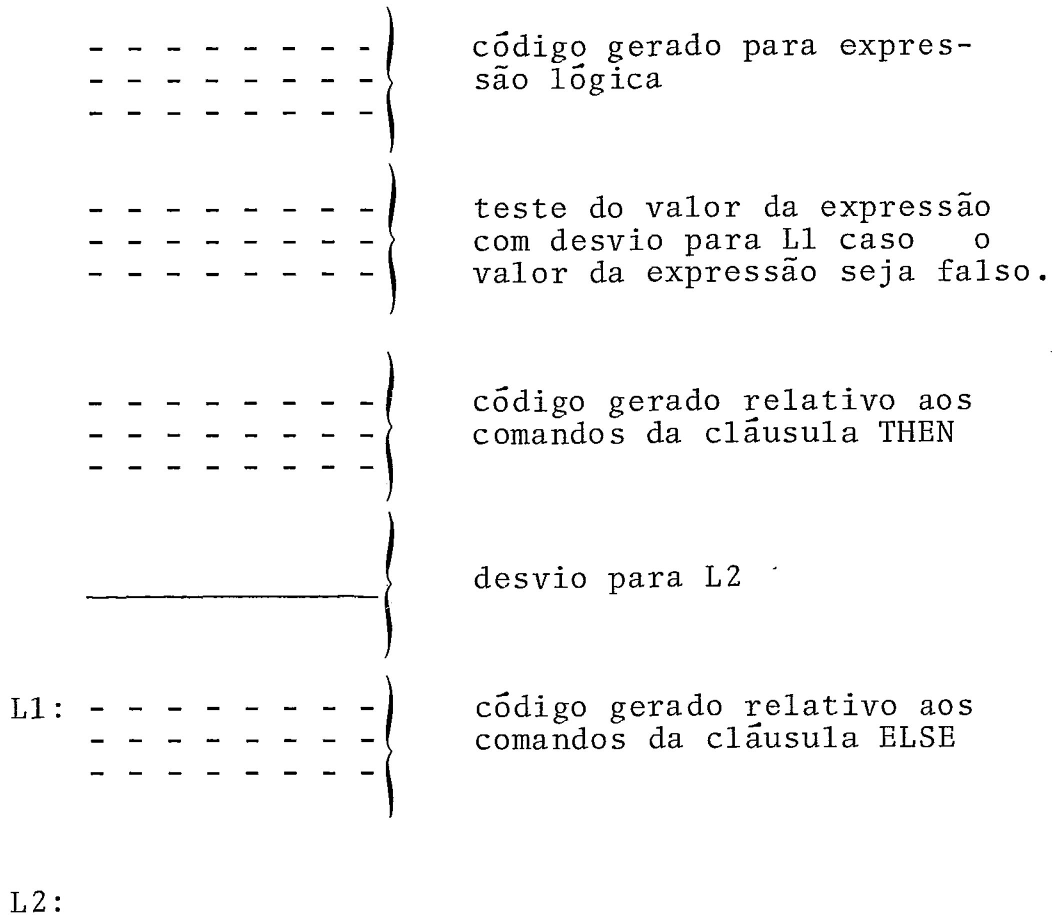
A forma geral de um comando IF é:

IF EXPRESSÃO THEN comando1 ; ELSE comando2; comando3 ;

(ver mais detalhes na seção 1.9)

A rotina que faz a A.S. deste comando, chama a rotina que analisa a expressão e gera o código correspondente, que devolve o resultado no acumulador em tempo de execução. Em seguida é gerado o código que testa o valor lógico do resultado da expressão, seguido de uma instrução de desvio para o comando2 (cláusula ELSE) se o valor é falso, em caso contrário não é feito nenhum desvio, o que significa que em tempo de execução o código a ser executado é o correspondente ao comando1.

A montagem da estrutura lógica do comando IF tem a seguinte configuração:



É feito um tratamento para Referencia Futura relativo aos desvios para os endereços de memória L1 e L2, como descrito em 4.7.

4.9. Geração de Código para uma rotina PL/STI

4.9.1. Introdução

As rotinas de um programa PL/STI são tratadas como blocos para onde são feitos desvios quando ocorre uma chamada.

O endereço do início do código correspondente ao corpo de uma rotina é inserido na T.S., associado ao seu nome, quando ocorre a declaração a declaração dessa rotina. Assim o código gerado para uma chamada de rotina consiste de uma instrução de desvio para esse endereço.

Para cada comando RETURN encontrado no corpo de uma rotina, é gerada uma instrução de desvio para o fim dessa rotina. Neste endereço se encontra uma instrução de desvio para o ponto seguinte ao da chamada. O operando dessa instrução de desvio é preenchido em tempo de execução, através de instruções de máquina. Essas instruções são geradas para cada chamada da rotina encontrada no programa fonte.

O endereço do operando da instrução de desvio que faz o retorno, é obtido da T.S. e está associado ao nome da rotina.

Assim reservamos o uso da pilha de chamadas de rotinas |1,2| apenas para as rotinas básicas (soma, divisão, etc...), em código objeto, que são chamadas através de intruções de máquina CALL e RETURN.

Levando em consideração que as subrotinas de um programa PL/STI serão sempre compiladas juntamente com o programa principal (dispensando a necessidade de uma link-edição posterior), os endereços dos parâmetros de uma rotina já estão disponíveis durante a fase de geração de código. Isto nos levou a optar por uma solução simples, não convencional, para transmissão de parâmetros em chamadas de rotina.

A solução convencional seria reservar para cada rotina uma área, de acordo com o número de parâmetros declarados, onde seriam armazenados os resultados dos parâmetros atuais avaliados. O código que transfere os valores armazenados nessa área para os formais correspondentes faria parte do início do código gerado para o corpo da rotina.

No nosso caso, o procedimento adotado consiste na geração de código que avalia cada parâmetro atual e em seguida armazena o resultado no parâmetro formal correspondente, cujo endereço de memória é obtido na T.S.

4.9.2. Declaração de Rotinas

Quando a declaração de uma rotina contém uma declaração do tipo, BYTE ou ADDRESS indicando que a rotina retorna um valor, o procedimento correspondente aloca uma posição de memória para o nome da rotina de acôrdo com o tipo. O endereço alocado é inserido na Tabela de Símbolos associado ao nome da rotina. Esse endereço é utilizado na geração do código da instrução RETURN EXPRESSÃO, onde se incluem instruções que armazenam o valor da expressão no endereço alocado para o nome da rotina e que preparam o retorno para o ponto seguinte ao da chamada.

O endereço do início de montagem do código da rotina é inserido na T.S., sendo usado quando ocorre uma chamada.

A forma como deve ser declarada uma rotina em PL/STI está detalhada na seção 1.12.1. .

4.9.3. O corpo de uma rotina

O corpo de uma rotina é formado por comandos PL/STI. O código é gerado portanto da mesma forma que no programa principal.

No final da montagem do código correspondente ao corpo da rotina é incluída uma instrução de desvio para o ponto seguinte ao da chamada. O endereço do operando dessa instrução é inserido na T.S. para ser preenchido em tempo de execução, quando ocorrer uma chamada. Assim todo comando RETURN reconhecido durante a compilação da rotina implica na geração de uma instrução de desvio para essa instrução que faz o retorno.

4.9.4. Chamada de Rotinas

Quando a chamada de uma rotina declarada com parâmetros é reconhecida, são geradas instruções que realizam a transferência do valor dos parâmetros atuais para os correspondentes formais, fazendo conversão quando necessário. A transmissão de parâmetros quando da chamada de uma rotina PL/STI é feita por valor (ver seção 1.12), o que significa que nenhuma

alteração é feita nos parâmetros atuais durante a execução da rotina.

Em seguida é gerado o código que preenche o operando da instrução de desvio que faz o retorno, com o endereço do ponto seguinte ao da chamada (como foi descrito na seção 4.9.3). É gerado então o código que faz um desvio para a rotina. O endereço do início da rotina é encontrado na T.S. . Esse endereço foi inserido quando foi encontrada a declaração e gerado o código correspondente ao corpo da rotina, como descrito na seção 4.9.2. .

Podemos visualizar o código gerado para uma rotina como o gerado para um bloco, seguido de um código que armazena o valor a ser retornado no nome da rotina, quando for o caso, e finalmente uma instrução de desvio para o ponto seguinte ao da chamada.

O procedimento adotado para rotinas foi possível porque não é permitida a recursividade de rotinas. Esse tratamento, onde não são utilizadas as instruções de máquina específicas para chamada e retorno de rotinas, economiza espaço na pilha interna de memória que serve para guardar endereços de retorno, no máximo até 8.

Como a rotina é montada na ordem em que ela aparece, antes do início da montagem é gerada uma instrução de desvio para o ponto seguinte ao do fim da rotina.

A montagem da estrutura lógica de uma rotina tem a seguinte configuração:

Na declaração:

- - - - - - - - - - - - } código gerado para o programa fonte que precede a declaração da rotina.

____________ } desvio para L2

L1: - - - - - - - - - - - - } código gerado para o corpo da rotina

- - - - - - - - - - - - } código gerado para a expressão aritmética que aparece num comando RETURN, se for o caso (1)

- - - - - - - - - - - - } código gerado para armazenar o valor da expressão no nome da rotina (2)

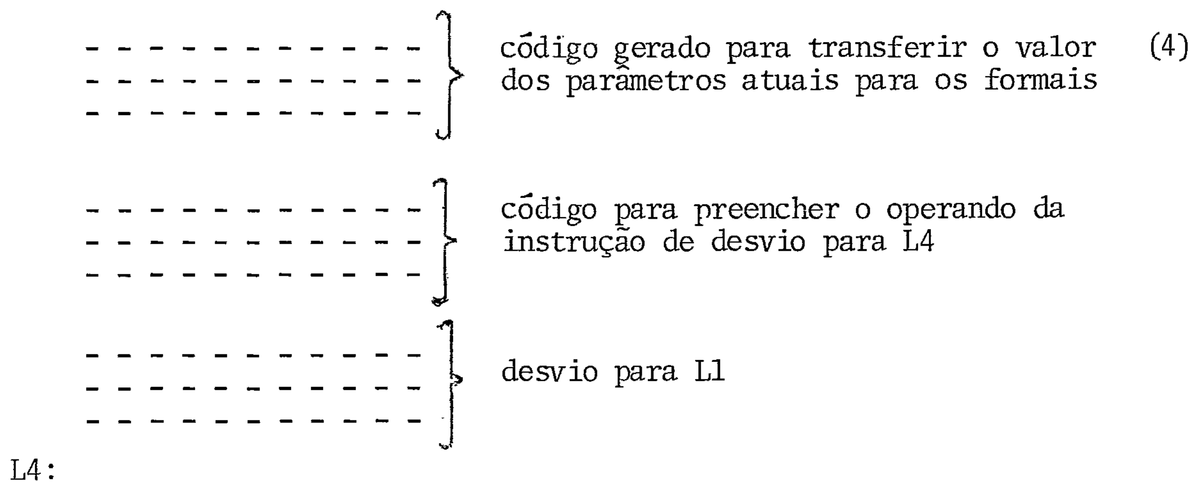
(1), (2) } código gerado para um comando RETURN

- - - - - - - - - - - - } desvio para L3 (3)

L3: ____________ } desvio para o ponto seguinte da chamada, L4

L2:

Na chamada:



- (1) e (2) só são gerados para rotinas com tipo declarado.
- (4) só é gerado para rotina com parametros declarados.
- (3) é gerado para cada comando RETURN que aparece no corpo da rotina.

Os endereços L2 e L3 só são conhecidos no final da compilação da rotina. Para estes casos é feito então o tratamento de Referencias Futuras, como descrito na seção 4.7. .

Os endereços L1 e L3 são inseridos na T.S. e usados quando ocorre uma chamada, na forma descrita na seção 4.9.4. .

O endereço L4 só será conhecido na ocorrência de uma chamada, assim o endereço do operando da instrução de desvio para L4 fica na T.S., sendo preenchido no momento propício, através de instruções de máquina.

4.10. Geração de Código para os Comandos Compostos

4.10.1. Geração de código para o comando composto DO-WHILE

Recordando a forma geral temos:

```
DO WHILE EXPRESSÃO ;
   comando 1 ;
   comando 2 ;
      ,
      ,
      ,
   comando n ;
END ;
```

A configuração do código gerado para este comando é a seguinte:

```
L1: - - - - - - -  \
    - - - - - - -   } código gerado para a expres-
    - - - - - - -   | são lógica
    - - - - - - -  /

    TESTE, se falso desvia para L2

    - - - - - - -  \
    - - - - - - -   } código gerado relativo aos
    - - - - - - -   | comandos 1,..., comando n
    - - - - - - -  /

    ___________     } desvio para L1
L2:
```

Na rotina que faz a A.S. do comando DO-WHILE é

ativado o procedimento que trata Referência Futura para as instruções de desvio para os endereços de memória L1 e L2. Em seguida é chamada a rotina que analisa a expressão gerando código para executá-la, devolvendo o resultado lógico no acumulador em tempo de execução, seguido de uma instrução de desvio para L2 caso o valor lógico da expressão seja falso.

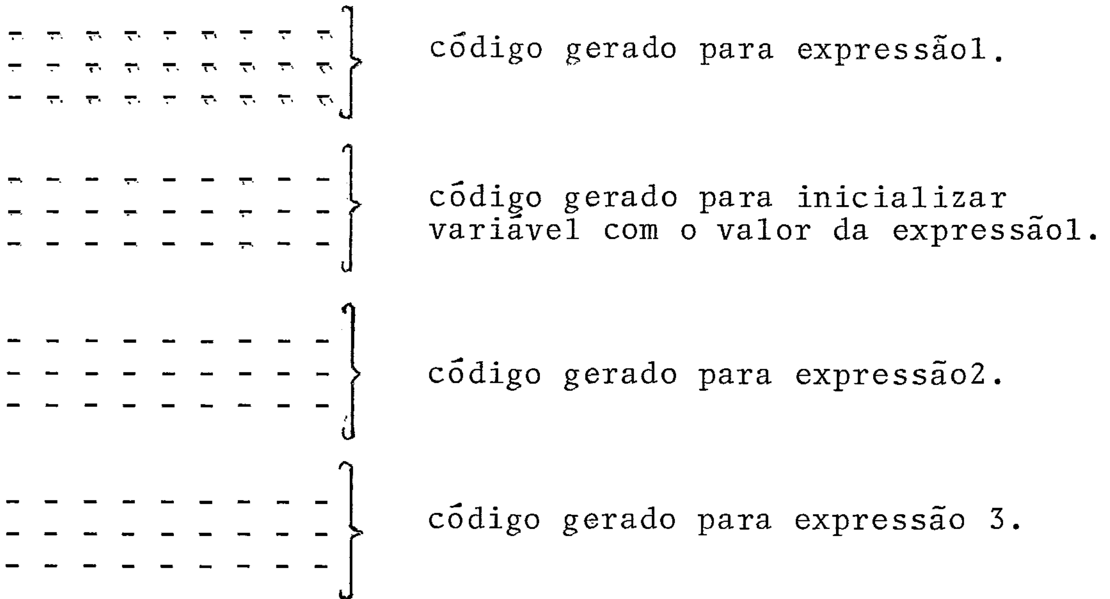
Finalmente é gerado o código relativo aos comandos de nº 1 até N seguido de uma instrução de desvio para o endereço L1.

4.10.2. Geração de Código para o comando composto DO iterativo

Forma geral:

```
DO variável = expressão1 TO expressão2  BY expressão3
   comando 1 ;
      '
      '
      '
   comando N ;
END ;
```

A configuração do código gerado para este comando é a seguinte:



L1: TESTE, se resultado da expressão3 (incremento) é negativo desvia para L2.

- - - - - - - - - - código gerado para testar se a variável controle é maior que o limite expressão2. Caso seja verdade desvia para L3, senão desvia para L4.

L2: - - - - - - - - - - código gerado para testar se a variável controle é menor que o limite expressão2. Caso seja verdade desvia para L3.

L4: - - - - - - - - - - código gerado relativo aos comandos1,..., N

- - - - - - - - - - código gerado para incrementar a variável controle do valor da expressão 3

__________ desvio para L1

L3:

Quando expressão1, expressão2 e expressão3 são expressões aritméticas ou constantes, ou seja, não são variáveis, alocamos variáveis temporárias para armazenar os resultados. Essas variáveis temporárias só são liberadas no final da compilação do comando DO iterativo. De qualquer modo os endereços dessas variáveis são inseridos na pilha de operandos para serem usados durante a geração de códigos correspondente ao comando.

Para gerar código relativo a essas expressões é chamada a rotina que analisa expressões.

O tratamento de Referências Futuras descrito na seção 4.7. é feito para os endereços L1, L2, L3 e L4.

O código que testa o sinal e compara a variável controle com o limite (expressão2) e o código que incrementa a variável controle são ambos gerados na rotina que compila um comando DO-iterativo.

O código equivalente aos comandos1,...,N é gerado nos procedimentos que compilam cada comando, e que são chamados pela rotina que controla a A.S.

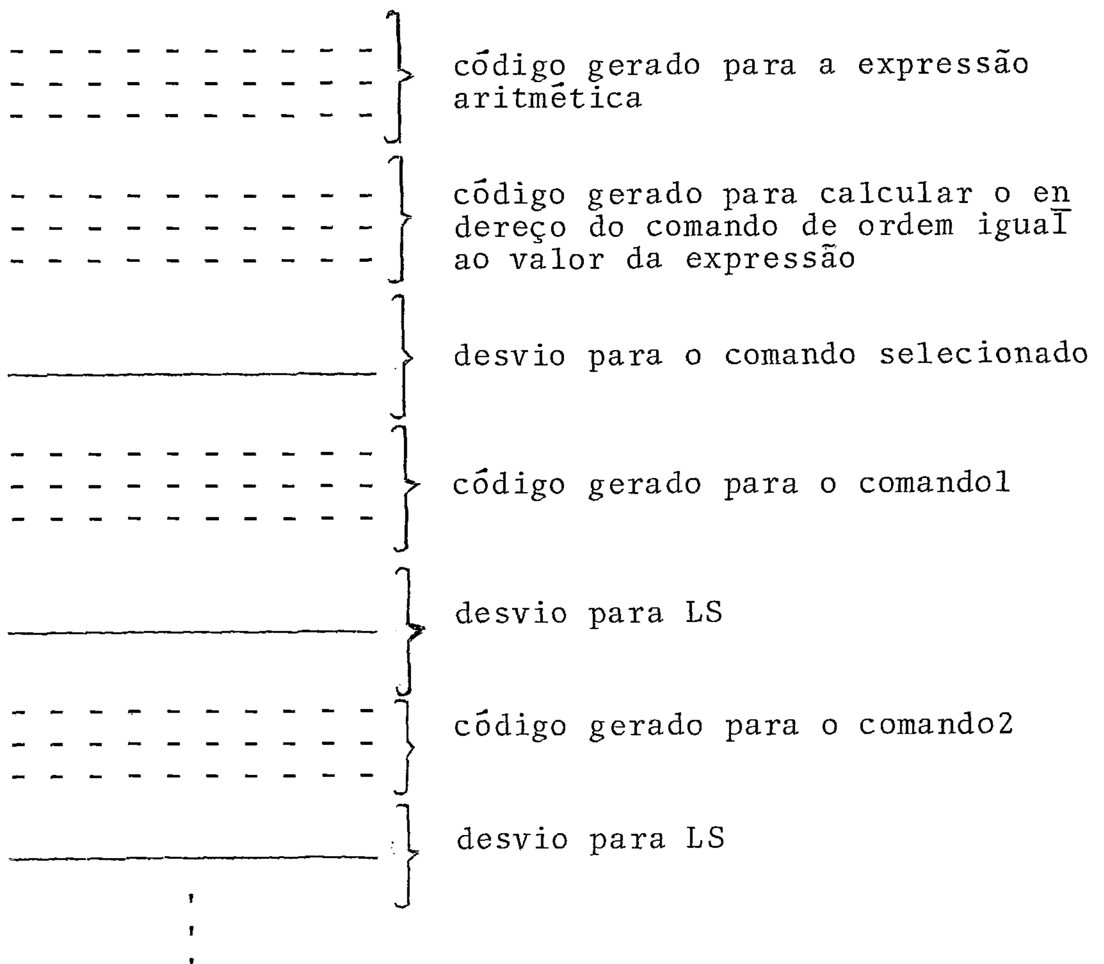
4.10.3. Geração de Código para o comando composto DO-CASE

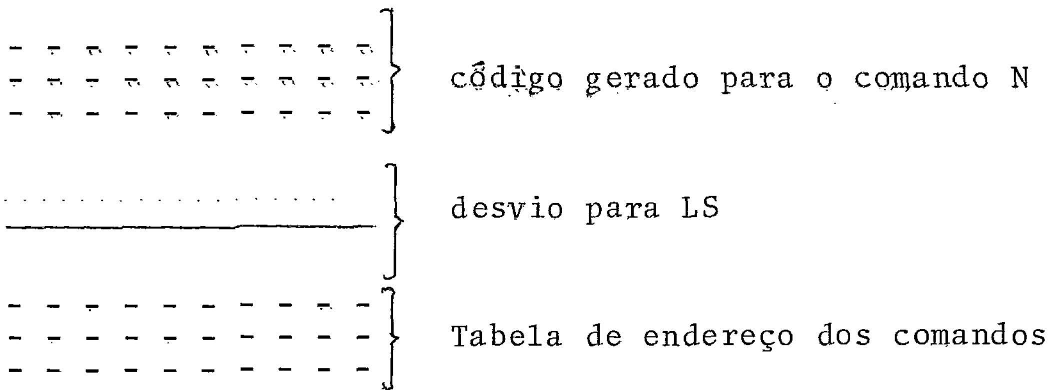
Recordando a forma geral temos:

```
DO CASE EXPRESSÃO ;
    comando1 ;
    comando2 ;
       ,
       ,
       ,
    comandoN ;
END ;
```

(ver mais detalhes na seção 1.10.4)

A montagem da estrutura lógica de um comando DO CASE tem a seguinte configuração:





LS:

A rotina que faz a A.S. de um comando composto DO CASE, chama inicialmente a rotina que compila expressões. É então gerado o código que executa essa expressão e devolve o resultado no acumulador.

Em seguida é gerado o código que usa o valor dessa expressão como um índice para uma tabela que fica armazenada na memória e que contém os endereços dos comandos1,..., comando N, seguido de uma instrução de desvio para o endereço calculado e que corresponde ao comando selecionado.

Após o código gerado para cada comando é gerada uma instrução de desvio para o fim do comando DO CASE. Como o endereço do fim deste comando só é conhecido quando termina a sua compilação com a montagem da tabela de endereços dos comandos, é feito o tratamento de Referências Futuras para essas instruções de desvio.

A tabela na memória, que contém os endereços relativos ao início do código de cada comando, é preenchida no

final da compilação do DO CASE através de instruções de máquina, quando todos os endereços são conhecidos e estão armazenados no mesmo vetor auxiliar usado para Referencias Futuras.

4.10.4. Geração de código para o comando composto DO;

Nenhum código é gerado para este comando.

4.11. Tratamento para rótulos

A linguagem PL/STI permite o uso de dois tipos de rótulos: numérico e simbólico.

Rótulo numérico: Na ocorrência de um rótulo deste tipo é feito um teste que verifica se o respectivo valor numérico é um endereço dentro dos limites da área reservada para o código do programa fonte. Caso esteja, o ponteiro para a memória passa a apontar o endereço de memória dado pelo rótulo, e a montagem do programa continua a partir desse endereço. Em caso contrário é emitida uma mensagem de erro e impedida a execução do programa.

Rótulo simbólico: Neste caso o endereço de memória apontado no momento em que ocorreu um rótulo simbólico é inserido na T.S., sendo usado sempre que ocorre um comando GO TO para esse rótulo.

4.12. Geração de Código para comandos de desvio

Há tres formas distintas para comandos de desvio em PL/STI:

1) GO TO rótulo simbólico ;

O rótulo simbólico é um identificador que aparece como um rótulo em um comando rotulado. (ver seção 1.8.3.), podendo ou não ter sido declarado anteriormente.

Quando um comando desse tipo é reconhecido pela rotina, que compila comandos de desvio, é feita uma consulta na T.S. . Se esse rótulo já foi usado é gerada uma instrução de desvio para o endereço de memória associado a esse rótulo (ver seção 4.11). Em caso contrário é feito o seguinte procedimento:

É gerada uma instrução de desvio cujo operando deve ser preenchido quando o endereço do rótulo for conhecido.

Na T.S. são inseridos o rótulo,caso não esteja inserido,e o endereço do operando da instrução de desvio gerada.

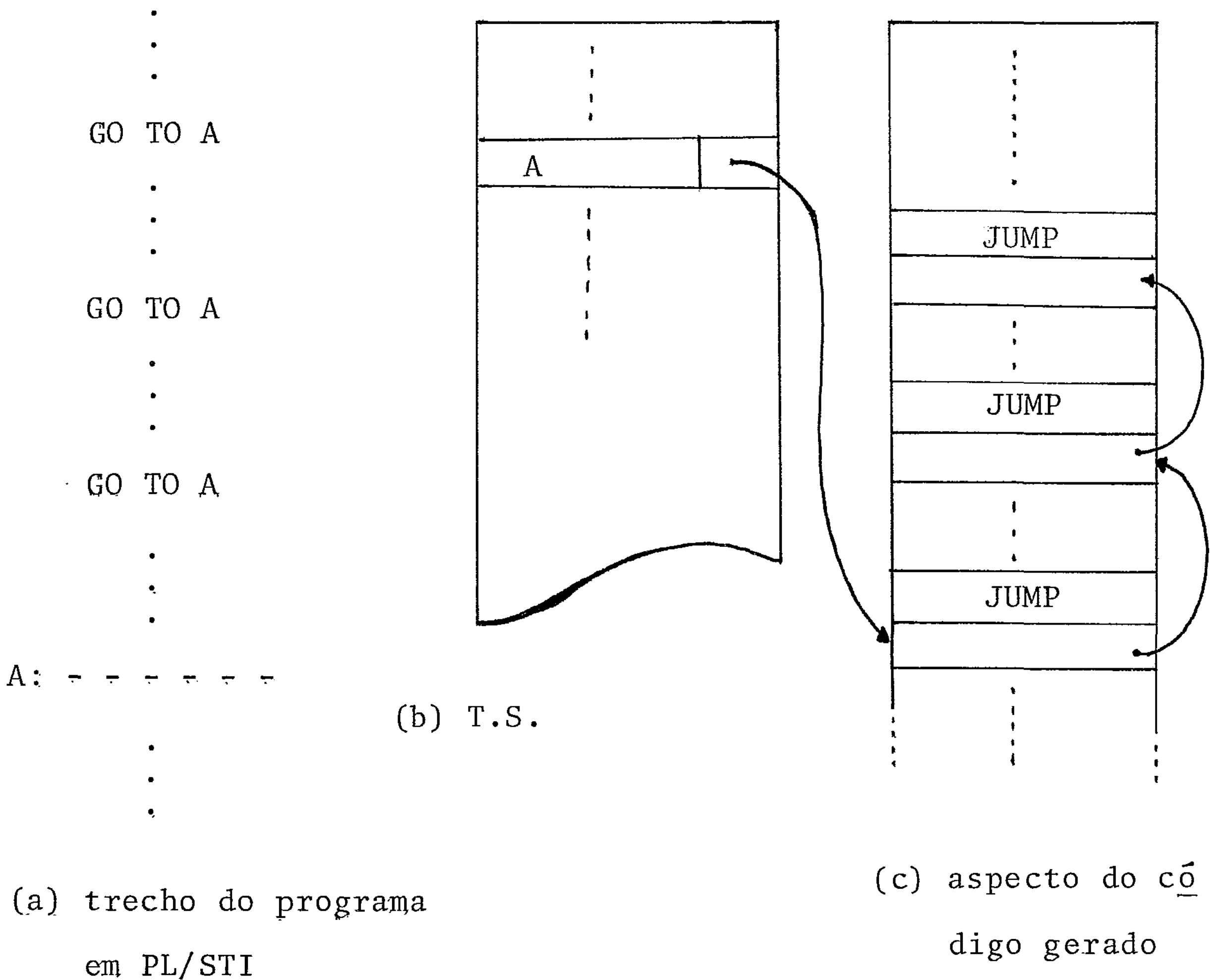
Para solucionar o problema de vários comandos de desvio, para um mesmo rótulo não usado, criamos uma lista, ligada da seguinte forma:

A T.S. contém um campo que aponta o operando da última instrução de desvio para esse rótulo. O operando de

cada instrução de desvio aponta o operando da última instrução de desvio, que ocorreu antes dela.

Assim quando o rótulo é finalmente usado, é gerado o código que preenche os operandos dessas instruções de desvio, utilizando a informação fornecida por essa lista e pela T.S. .

Exemplo do tratamento de comandos de desvio para um rótulo não usado:



(a) trecho do programa em PL/STI

(c) aspecto do código gerado

Figura 7

2) GO TO número ;

Onde o número é um endereço absoluto de memória (ver seção 1.8.3).

Quando um comando desse tipo é reconhecido, é feito um teste em tempo de compilação que verifica se o número está dentro dos limites da área reservada para o código do programa fonte. Caso não esteja é emitida uma mensagem de erro e o código gerado para o programa não é executável. Em caso contrário será gerada uma instrução de desvio para êsse endereço.

3) GO TO nome de variável ; (ver seção 1.8.3)

Neste caso a variável contém o valor do endereço para onde deve ser feito o desvio. É gerado então o código que faz o desvio para o endereço de memória dado por essa variável. Nenhum teste é feito quanto ao valor do endereço, como no caso anterior. Se o endereço estiver fora dos limites descritos, o efeito em tempo de execução fica indefinido.

4.13. Geração de Código para o comando de atribuição

A forma geral de um comando de atribuição é:

VARIÁVEL = EXPRESSÃO

Se a variável é simples, o endereço de memória alocado para a mesma, e que está na T.S. é inserido na pilha. Caso seja uma variável indexada, é gerado o código que calcula o valor do índice e soma que esse valor ao endereço alocado para a variável. O resultado dessa soma é devolvido no acumulador, obtendo-se assim o endereço da posição do vetor que está sendo referenciada.

O acumulador e a informação de que êle contém um endereço são inseridos na pilha, juntamente com o operador de atribuição e a respectiva prioridade. Em seguida a rotina que compila expressões é chamada, se encarregando de gerar o código apropriado para avaliar a expressão e atribuir o resultado à variável. O modo como é feita a conversão do tipo do resultado da expressão para o tipo da variável está descrita em 1.5.2..

CAPÍTULO V

CONCLUSÕES

Este trabalho em conjunto com o referenciado em |[1]|, na bibliografia, completam um compilador para a linguagem PL/STI.

O compilador foi implantado no B.6700, em ALGOL extendido; e encontra-se à disposição dos usuários interessados.

O sistema consiste em um "cross-compiler", cujo objetivo é facilitar o desenvolvimento de software para o T.I.. Deste modo os programas em PL/STI deverão ser compilados poucas vezes, mas muito executados. Neste caso, demos especial atenção à simplicidade e eficiência do código objeto gerado.

Uma facilidade que poderia ser oferecida ao usuário, seria fazer a ligação da saída do compilador também com o S.O.S. . Deste modo o programa objeto poderia ser depurado e executado no próprio simulador, aproveitando as facilidades de depuração do mesmo, quando fosse de interesse do programador.

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>
26. ADDRESS	<definição de rotina>
27. BYTE	<por>
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>
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 <u>terminais</u> para a Análise Sintática e <u>não terminais</u> 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

<digítio octal>::=<digíto binário>|2|3|4|5|6|7

<dígito>::= <dígito octal> |8|9

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

<número binário>::=<código binário> B

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

<número octal>::= <código octal> 0

<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 da 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 comando><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 da rotina>::=<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>::= <identificadores>

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 decla-
ração>

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

57 - |<identificador><LITERALLY><cadeia de carac-
teres>

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 idenfiticador><tipo>

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

64 - |<declaração de tipo><lista de valores ini-
ciais>

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 variá
vel>)

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 relacio-
                      nal><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 DE 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
<=	Menor ou igual	Operador de teste relacional
>=	Maior ou igual	Operador de teste relacional
<>	Diferente	Operador de teste relacional

Símbolo	Nome	Uso
:	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 - <u>Ou exclusivo (XOR)</u> :

Ø XOR Ø = Ø

Ø XOR 1 = 1

1 XOR Ø = 1

1 XOR 1 = Ø

2 - <u>E lógico (AND)</u> :

Ø AND Ø = Ø

Ø AND 1 = Ø

1 AND Ø = Ø

1 AND 1 = 1

3 - <u>Ou lógico (OR)</u> :

Ø OR Ø = Ø

Ø OR 1 = 1

1 OR Ø = 1

1 OR 1 = 1

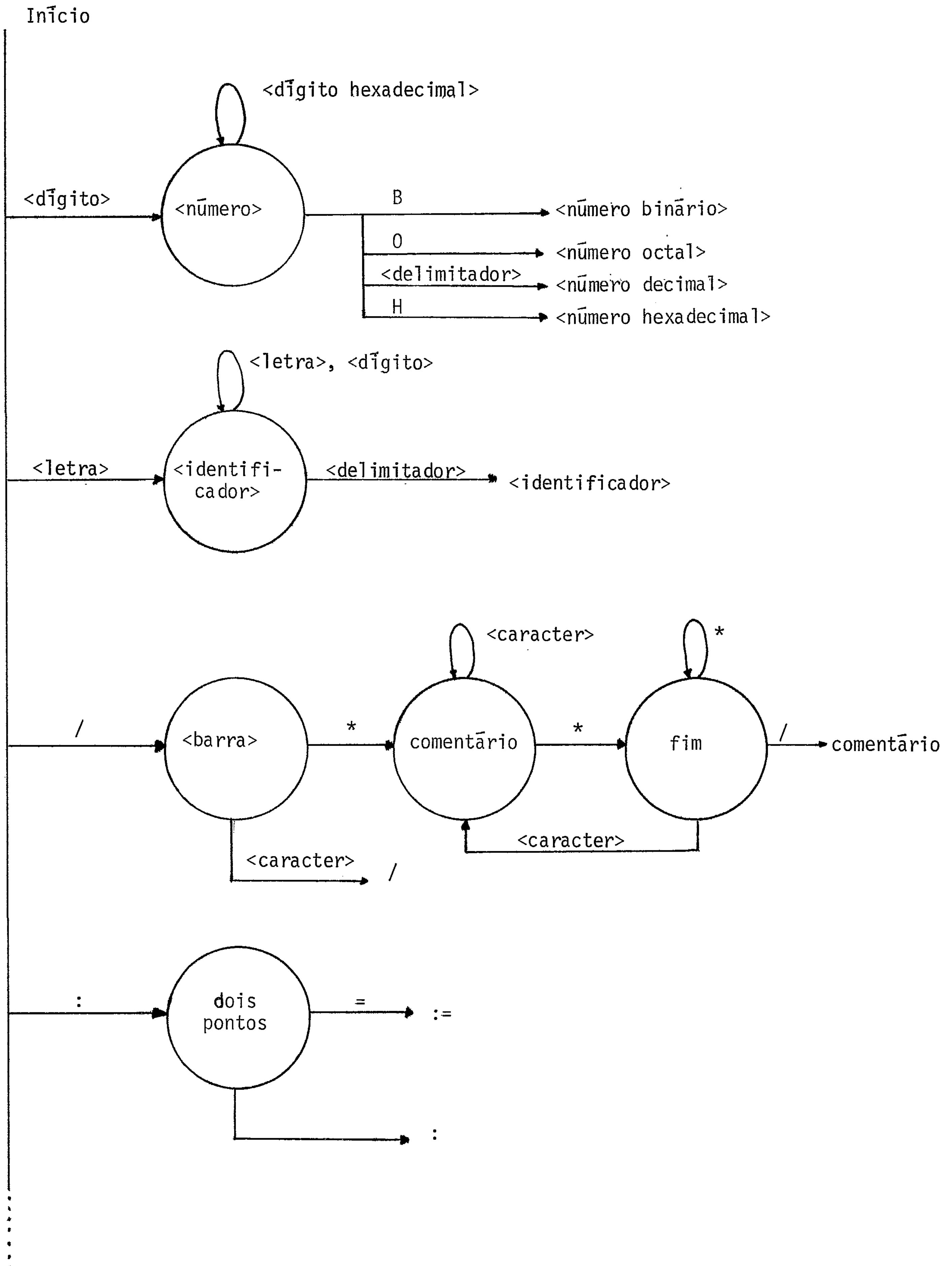
4 - <u>Negação (NOT)</u> :

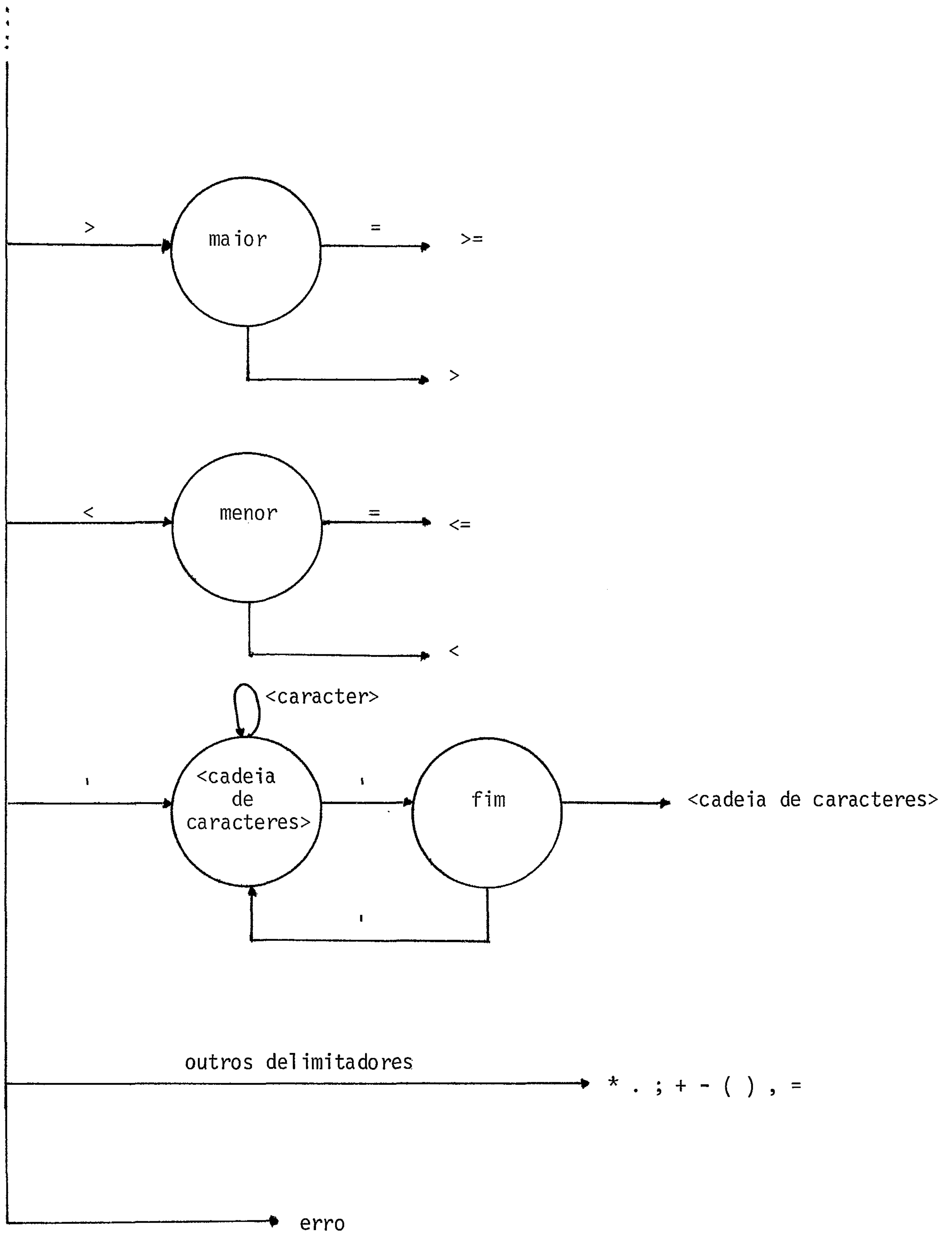
NOT Ø = 1

NOT 1 = Ø

APÊNDICE E

DIAGRAMA DE ESTADOS PARA A ANÁLISE LÉXICA


Início
<dígito hexadecimal>
<dígito>
<número>
B
<número binário>
O
<número octal>
<delimitador>
<número decimal>
H
<número hexadecimal>
<letra>, <dígito>
<letra>
<identificador>
<delimitador>
<identificador>
<caracter>
*
/
<barra>
*
comentário
*
fim
/
comentário
<caracter>
<caracter>
/
:
dois pontos
=
:=
:



Obs : O procedimento que deteta erro em constantes numéricas é feito na rotina de conversão de constantes.

APÊNDICE F

PROGRAMA EXEMPLO COM ERROS LÉXICOS

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


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


15/01/78


   ORDENACAO: PROCEDURE(PONTEIRO,N) ADDRESS;

                 /*
                 N=DIMENSAO DO VETOR A SER
                   ORDENADO .
                 PONTEIRO=ENDERECO DE MEMORIA DO
                          VETOR A SER ORDENADO.
                 NTROCAS=NUMERO DE TROCAS.
                 TROCOU=VARIAVEL LOGICA QUE INDICA
                 TROCOU=VARIAVEL LOGICA QUE
                        INDICA SE UMA TROCA FOI
                        EFETUADA P/ UM PAR DE
                        VALORES DO VETOR.
                 */

                 DECLARE
                        (PONTEIRO,NTROCAS,TEMP) ADDRESS
                       ,(A BASED PONTEIRO)(13) ADDRESS
                       ,(N,I,TROCOU) BYTE
                       ,TRUE LITERALLY 'OFFH'
                       ,FALSE LITERALLY        ;
                                               *
>>>>  1>>>>  ERRO(57): O CORPO DA MACRO DEVERIA ESTAR ENTRE PLIC'S

                 TROCOU=TRUE ;
                 NTROCAS=0X   ;
                         *
>>>>  2>>>>  ERRO( 1): CARACTER INVALIDO EM CONSTANTE NUMERICA
                 DO WHILE TROCOU;
                  TROCOU=FALSE ;
                             *
>>>>  3>>>>  ERRO(56): ESTA MACRO NAO ESTA DEFINIDA
                  DO I=0 TO N-2 ;
                   IF A(I) > A(I+1)
                   THEN DO;
                        /* TROCA O PAR FORA
                           DE ORDEM     */
                        TE"MP=A(I) ;
                          *
>>>>  4>>>>  ERRO( 3): CARACTER INVALIDO SUPRIMIDO
                        A(I)=A(I+1) ;
                        A(I+1)=TEMP ;
                        TROCOU=TRUE ;
                        END ;
                  END ;
                  /* PERCORREU O VETOR */
                 END ;
                 /* O VETOR ESTA ORDENADO */
                 RETURN NTROCAS ;

                 END ORDENACAO ;
EOF
( 4) ERROS DE COMPILACAO
```

APÊNDICE G

PROGRAMA EXEMPLO COMPILADO

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

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


15/01/78


    /* PROGRAMA PARA ORDENAR E IMPRIMIR
    UM CONJUNTO DE 13 VALORES          */

    DECLARE
             N1 ADDRESS
            ,N BYTE INITIAL (13)
            ,B(13) ADDRESS INITIAL (20000,
             -19367 , -30213 , 22345 ,
             -12510 ,  13450 , 17291 ,
             -15345 ,  11215 , 18421 ,
             -11213 ,  15725 , 17200 ) ;

    LUZES: PROCEDURE (A) ;
            DECLARE (A,B) BYTE , L1 LABEL ;

            /* SELECIONA LUZES */
            OUTPUT(8)=0 ;

            /* ESCREVE NAS LUZES */
            OUTPUT(1)=A ;
            IF A>128 THEN RETURN ;

            /* ESPERA VALIDADE  */
            L1: B=INPUT(1) ;
                IF NOT B THEN GO TO L1 ;

            RETURN ;
            END LUZES ;

    SAICARACTER: PROCEDURE (X) ;
                DECLARE(X,STATUS) BYTE ;

                /* SELECIONA IMPRESSORA */
                INICIO: OUTPUT(8)=30H ;
                        OUTPUT(3)= 0  ;

                /* ESPECIFICA NOT BUSY */
                LOOP: STATUS=INPUT(1) ;
                      IF (STATUS AND 40H <>0 )
                      THEN GO TO LOOP ;

               /* TESTA SE NOT READY */
               IF (STATUS AND 20H <> 0 )
               THEN DO ;
                    CALL LUZES ( 0BBH );
                    GO TO INICIO ;
                    END;

            /* ESCREVE CARACTER */
            OUTPUT(1)=X ;

            RETURN ;

            END ; /* FIM DA ROTINA SAICARACTER */
```

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

```
IMPRIME: PROCEDURE ;

         DECLARE I BYTE
                ,JASAIRAM BYTE INITIAL(0)
                ,DIGITO(4) BYTE
                ,LIT LITERALLY 'LITERALLY'
                ,LF LIT '0AH'
                ,CR LIT '0DH'
                ,TAMANHO$DA$LINHA LIT '13'
                ,K BYTE ;

         K=0;
         DO WHILE (JA$SAIRAM<>TAMANHO$DA$LINHA) ;
          JASAIRAM=JASAIRAM+1 ;

          /* IMPRIME SINAL  */
          IF B(K) < 0
          THEN DO;
               CALL SAICARACTER('-') ;
               B(K)=-B(K) ;
               END ;
          ELSE CALL SAICARACTER(' ') ;

          /* CONVERTE NUMERO BINARIO */
          DO I=4 TO 0 BY -1 ;
           DIGITO(I)=B(K) MOD 10 AND 40H ;
           B(K)=B(K)/10 ;
          END ;

          /* IMPRIME O NUMERO */
          DO I=0 TO 4 ;
           CALL SAICARACTER(DIGITO(I)) ;
          END ;

          DO I=0 TO 3 ;
           CALL SAICARACTER(' ') ;
          END ;

          IF K=N
          THEN JA$SAIRAM=TAMANHO$DA$LINHA;
          K=K+1 ;
         END ;    /* DO WHILE */

         DO;
          CALL SAICARACTER( CR );
          CALL SAICARACTER( LF );
         END;

         RETURN ;

         END IMPRIME ;
```

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

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

```
ORDENACAO: PROCEDURE(PONTEIRO,N) ADDRESS;

                /*
                N=DIMENSAO DO VETOR A SER
                  ORDENADO .
                PONTEIRO=ENDERECO DE MEMORIA DO
                        VETOR A SER ORDENADO.
                NTROCAS=NUMERO DE TROCAS.
                TROCOU=VARIAVEL LOGICA QUE INDICA
                TROCOU=VARIAVEL LOGICA QUE
                       INDICA SE UMA TROCA FOI
                       EFETUADA P/ UM PAR DE
                       VALORES DO VETOR.
                */

                DECLARE
                       (PONTEIRO,NTROCAS,TEMP) ADDRESS
                      ,(A BASED PONTEIRO)(13) ADDRESS
                      ,(N,I,TROCOU) BYTE
                      ,TRUE LITERALLY '0FFH'
                      ,FALSE LITERALLY '00H' ;

                TROCOU=TRUE ;
                NTROCAS=0 ;
                DO WHILE TROCOU;
                 TROCOU=FALSE ;
                 DO I=0 TO N-2 ;
                  IF A(I) > A(I+1)
                  THEN DO;
                      /* TROCA O PAR FORA
                         DE ORDEM     */
                      TEMP=A(I) ;
                      A(I)=A(I+1) ;
                      A(I+1)=TEMP ;
                      TROCOU=TRUE ;
                      END ;
                 END ;
                 /* PERCORREU O VETOR */
                END ;
                /* O VETOR ESTA ORDENADO */

                RETURN NTROCAS ;

                END ORDENACAO ;


   /* PROGRAMA PRINCIPAL */
   CALL IMPRIME ; /* IMPRIME VETOR NAO ORDENADO */
   N1=ORDENACAO(.B,LENGTH(B));
   CALL IMPRIME ; /* IMPRIME VETOR ORDENADO */

EOF

( 0) ERROS DE COMPILACAO
```

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

REFERÊNCIAS BIBLIOGRÁFICAS

|1| Pereira, R.C.S., 1977 - "Terminal Inteligente: Análise Sintática para um compilador da linguagem PL/STI" - Tese de Mestrado, COPPE/UFRJ .

|2| Naur P. (e), 1963 - "Revised report on the algorithmic language Algol 60" - Computer Journal, vol.5.

|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 implementantion" - A.P.I.C. Studies in Data Processing.

|10| Ullman, J.D., 1976 - "Fundamental Concepts of Programming Systems.

|11| Nicholls, J.E., 1975 - "The structure and Design of Programming Languages" - The systems programming series, Addison-Wesley publishing company.

|12| Manual do Usuário do Sistema Operacional de Simulação (S.O.S) - NCE/UFRJ.

|13| MCS-8 Assembly - Language Programming Manual (INTEL-8008).

|14| 8008 and 8080 PL/M Programming Manual.