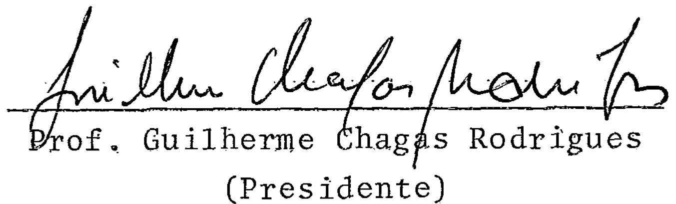
TERMINAL INGELIGENTE : ANÁLISE LÉXICA E GERAÇÃO DE CÓDIGO PARA UM COMPILADOR DA LINGUAGEM PL/STI

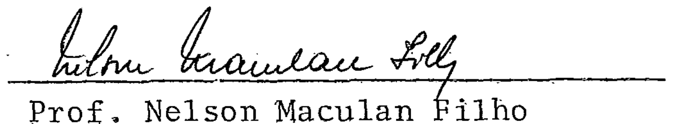
Miriam Aparecida Marques

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.).

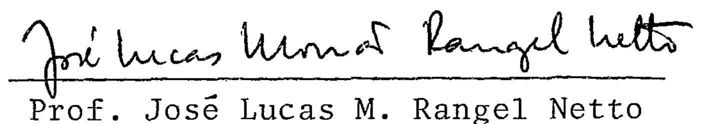
Aprovada por:



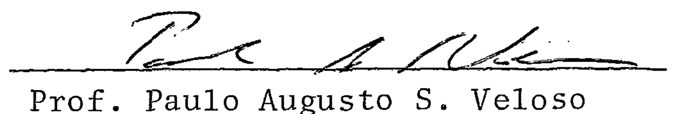
Prof. Guilherme Chagas Rodrigues
(Presidente)



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 (Ø - A). Assim -1 por exemplo, é equivalente a Ø - 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) Ø 3 H - Ø 4 H	b) Ø 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. Operadores relacionais (ou de comparação)

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, fi cando 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 <u>indeterminado</u> , 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 não retorna um valor , a chamada é feita através do comando CALL, que tem a forma:

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

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

b) se a rotina <u>retorna um valor</u> , então a sua forma de chamada é

<u>nome da rotina (lista de parâmetros atuais)</u>

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

<u>Forma geral</u>

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:

<u>EXEMPLO(1)</u>:

```
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. As funções LOW,HIGH,DOUBLE

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

```
         Ø1
RR =     1Ø     , para saída
         11
```

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

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

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

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

1.17.1. INPUT

Forma geral : INPUT (número)

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

O argumento é uma constante numérica que deve estar entre os limites Ø-7, correspondendo ao campo 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.

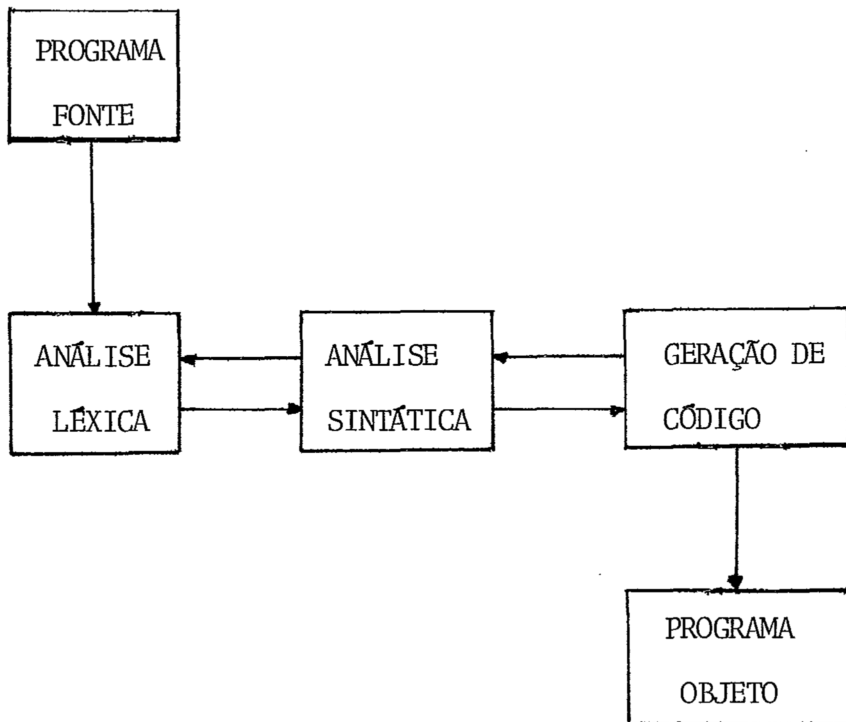


Figura 1

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