UM COMPILADOR BASIC INCREMENTAL PARA O TERMINAL INTELIGENTE

Eliane Martins

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:

Guilherme Chagas Rodrigues
Orientador

Suéli Mendes dos Santos

José Lucas Mourão Rangel Netto

Doris Ferraz Aragon

RIO DE JANEIRO, RJ - BRASIL
MARÇO DE 1982

MARTINS, ELIANE

Um Compilador Basic Incremental para o Terminal Inteligente - (Rio de Janeiro) 1982

VIII, 155 p. 29.7 cm (COPPE-UFRJ, M. Sc., Engenharia de Sistemas e Computação.

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

1. Compilador

I. COPPE/URRJ

II. Título (Série)

À minha mãe

RESUMO

Em um compilador incremental, a análise sintática é feita de modo que o programa não precise ser totalmente recompilado após cada modificação.

Neste trabalho é apresentado um compilador incremental para a linguagem BASIC, a ser implementado no Terminal Inteligente desenvolvido no NCE da UFRJ. O objetivo principal é permitir o uso do referido equipamento por usuários que estejam aprendendo computação.

ABSTRACT

An incremental compiler is a device which is able to perform syntax analysis in an incremental way, avoiding complete reparsing of a program after each modification.

This work presents an incremental compiler for the BASIC language, to be implemented in the intelligent terminal developped by the Nucleus of Electronic Computing of the UFRJ. The project is oriented for users that are learning about computing.

ÍNDICE

Pág.

I. INTRODUÇÃO

I.1 - O Compilador Incremental e suas Vantagens

Um compilador incremental é interativo. A comunicação com o usuário é feita tanto na fase de compilação quanto na de execução.

O principal aspecto do trabalho é o uso de incrementalismo. Com esta técnica, o programa fonte é considerado como um conjunto de linhas independentes entre si. Desse modo, se o usuário fizer qualquer correção ou alteração em uma linha, o programa não precisará ser recompilado: somente a linha alterada é re-analisada.

As vantagens na utilização desse método estão resumidas a seguir:

1. Edição interativa

O processo de desenvolvimento de um programa é constituído por várias alternâncias entre execuções e edições para corrigir os erros encontrados. No presente caso, o usuário poderá interromper a execução se o programa não estiver funcionando direito, corrigi-lo e em seguida continuar a execução.

2. Erros são detetados imediatamente

Quando uma linha errada é fornecida, o usuário é notificado, podendo corrigir o erro imediatamente.

3. Comandos imediatos

Os comandos imediatos são executados no momento em que são fornecidos. São um subconjunto dos comandos do BASIC, e a diferença entre uma linha de um programa e uma linha contendo comando imediato é que a primeira é numerada e a segunda não.

A grande vantagem desses comandos é que eles permitem depuração interativa. Após (ou durante) uma execução do programa em que haja erros, o usuário poderá obter ou alterar o conteúdo de variáveis entre outras.

4. Facilidade de uso

Devido às vantagens já citadas, o uso do computador é facilitado para usuário sem conhecimentos de computação.

Este projeto está baseado no trabalho de M. Berthaud e M. Griffiths, cuja descrição pode ser encontrada nos itens (1) e (2) da bibliografia.

I.2 - Recursos Disponíveis

A implementação se dará no Terminal Inteligente (T.I) desenvolvido pelo Núcleo de Computação Eletrônica da UFRJ, cuja configuração atual é a seguinte: teclado, vídeo, UCP com até 64 K bytes de memória, uma unidade de disco, leitora de cartões e impressora.

O compilador rodará sob o SOCO, o atual sistema operacional em disco do T.I. que contem, entre outras facilidades:

- o método de acesso sequencial indexado;

- um compilador PLTI, que é uma linguagem de alto nível baseada no PL/I, desenhada, entre outros motivos, para a implantação do "software" do T.I.;

- um interpretador, que analisa as instruções de uma linguagem de u'a máquina virtual, desenvolvida para que haja a maior independência possível entre os sistemas desenvolvidos e a máquina. Desse modo, caso haja mudanças no processador, basta alterar esse interpretador, permanecendo o restante dos sistemas inalterados;

- o uso de certas chaves no teclado do terminal para a interrupção de programas para a depuração interativa.

I.3 - Porque o BASIC Incremental

Foi escolhida essa linguagem pois pretende-se que o T.I., seja utilizado para fins educacionais, ou seja, serão atendidos usuários com pouca ou nenhuma experiência em computação.

O BASIC é uma linguagem simples, fácil de aprender e voltada para sistemas interativos.

O método incremental, além das vantagens citadas em (I.1), vai evitar que haja dois níveis de interpretação. A saída do compilador incremental será constituída pelas micro-rotinas do PLTI, acrescidas de algumas rotinas que terão, entre outras finalidades, a tarefa de tornar conversacional a fase de interpretação.

I.4 - Características Gerais da Linguagem

O BASIC a ser implementado tem como principais características:

- permitir a utilização de 3 tipos de dados: inteiros, reais e alfanuméricos;

- contem comandos e funções que manipulem os 3 tipos de dados descritos acima;

- os identificadores poderão ter até 6 caracteres;

- permite a utilização de variáveis subscritas de 1 ou 2 dimensões, sendo que se os limites forem 10 ou 10 x 10, a variável não precisará ser declarada;

- as variáveis e constantes inteiras podem ser usadas como operandos lógicos;

- disponibilidade de comandos de E/S, formatada ou não, permitindo acesso ao vídeo, teclado e disco do sistema;

- permite que o usuário defina funções com mais de uma linha;

- permite ao usuário definir rotinas especiais que admitam a passagem de parâmetros e sejam chamadas por CALL. As subrotinas comuns do BASIC também estarão disponíveis;

- dispõe de comandos IF-THEN-ELSE e comandos iterativos FOR-TO, FOR-WHILE e FOR-UNTIL;

- os espaços não serão transparentes;

- os números de linha são obrigatórios, pois darão a ordem de execução do programa;

- pode ser fornecido mais de um comando por linha, admitindo até 3 linhas de continuação;

- permite a inserção de comentários no programa, o que ajuda ao usuário na documentação do programa.

A descrição completa da linguagem está no diagrama sintático, no capítulo III.3 da Parte II.

I.5 - O Sistema, Visto pelo Usuário

Um programa será constituído por segmentos. Um segmento poderá conter uma função, subrotina ou o programa principal.

O formato de um segmento é mostrado a seguir:

```
segmento <nome do segmento>
    :
    : conteúdo
    :
  fim
```

O <nome do segmento> é dado por um identificador. O seu conteúdo é um conjunto de linhas numeradas contendo um ou mais incrementos. Um incremento, do mesmo modo que no trabalho desenvolvido por Berthaud e Griffiths, pode ser um comando ou declaração do BASIC ou parte de um comando. Como um incremento pode ser fornecido em qualquer ordem, o número da linha vai indicar a sequência de execução dentro do segmento.

A análise de cada linha é feita no momento em que a mesma é fornecida ao sistema. Caso haja algum erro, a mensagem correspondente é emitida e aguardar-se-á a correção do usuário.

O código gerado para cada segmento vai sendo guardado em disco. No momento da execução, todo o programa é carregado na memória. Quando ocorrer uma interrupção, os segmentos serão copiados de volta para o disco.

Não será permitida a definição de funções ou subrotinas embutidas, e nem haverá recursividade. Todos os identificadores serão considerados locais ao segmento que os contém; as ligações entre as variáveis dos diversos segmentos se dará através da passagem de parâmetros.

A interação entre o usuário e o sistema se dará através dos seguintes tipos de comandos:

Comandos de controle

São comandos que indicam ao sistema que devem ser executadas tarefas do tipo: salvar um programa, apagar um programa que havia sido salvo anteriormente, condensar a área objeto, de modo a obter melhor aproveitamento do espaço, etc.

Comandos de edição

Permitem ao usuário manipular com os segmentos que foram fornecidos, ou seja, apagar uma ou várias linhas, listar trechos ou todo um segmento, renumerar as linhas do segmento.

Comandos imediatos

São um subconjunto dos comandos do BASIC que facilitarão ao usuário na depuração de seus programas, podendo também ser chamados de comandos de depuração. Permitem listar ou alterar o conteúdo da área de dados, efetuar testes com esses valores, etc.

A execução de um segmento pode ser suspensa nos seguintes casos:

a) foi encontrado um comando para leitura conversacional;

b) a tela está cheia;

c) foi executado um STOP;

d) foi pressionada a chave de interrupção no teclado;

e) foi encontrado um erro.

Em caso de leitura conversacional, a execução é suspensa até que o usuário forneça os dados necessários para continuar a execução.

No caso de tela cheia, a execução é interrompida pois nenhuma linha mais poderá ser visualizada. Para continuar, basta que o usuário aperte a tecla VALIDADE.

Nos casos c), d) e e) o sistema fica aguardando o próximo passo do usuário. Este pode fornecer novos comandos da linguagem, ou comandos de edição ou ainda comandos imediatos. Após executar cada comando de um dos tipos descritos acima, o sistema continua aguardando até que o usuário resolva continuar a execução do ponto em que ela foi interrompida ou de outro ponto qualquer dentro do segmento. Em caso de erro grave, como por exemplo, desvio para uma instrução inexistente, o programa é cancelado.

Modos de Operação

O sistema poderá estar sempre em um dos seguintes modos de operação:

Modo de Controle

É o modo básico de operação; a partir dele o usuário poderá se utilizar dos comandos de controle, podendo ainda passar para o modo de entrada ou de execução.

Modo de entrada

O sistema estará nesse modo quando for dado o comando SEGMENTO. O usuário poderá então fornecer um novo segmento, inserir novos incrementos, editar um segmento ou ainda fornecer comandos imediatos. Seu término se dará com o comando FIMSEG, fazendo com que o sistema volte ao modo de controle.

Modo de execução

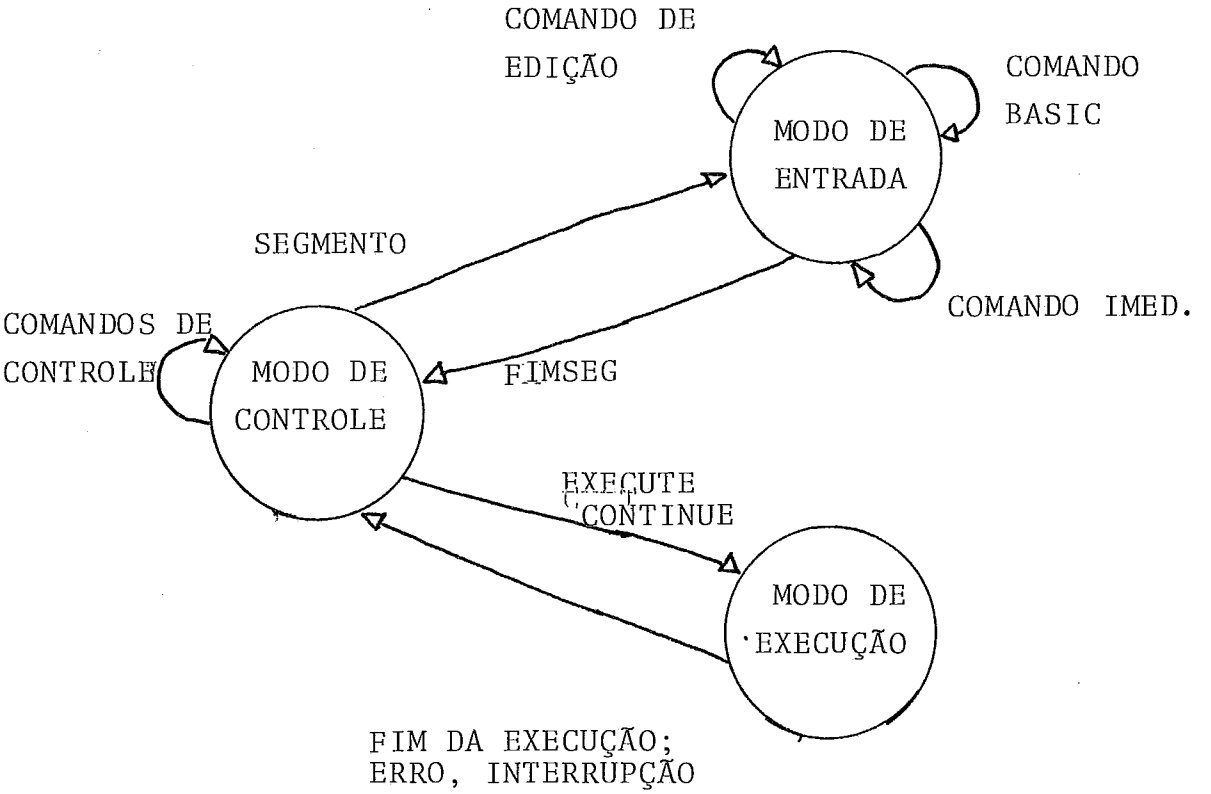
Iniciado através do comando EXECUTE ou CONT. Este último indica que havia algum segmento sendo executado. Uma vez em execução, o sistema ficará nesse modo até que:

1. haja alguma interrupção por chave do teclado;

2. seja executado um STOP;

3. seja encontrado erro ou acabe o programa.

O diagrama seguinte mostra como é feita a ligação entre os 3 modos.


COMANDO DE
EDIÇÃO
COMANDO
BASIC
MODO DE
ENTRADA
SEGMENTO
COMANDO IMED.
COMANDOS DE
CONTROLE
MODO DE
CONTROLE
FIMSEG
EXECUTE
CONTINUE
MODO DE
EXECUÇÃO
FIM DA EXECUÇÃO;
ERRO, INTERRUPÇÃO

PARTE II: GERAÇÃO DO SISTEMA

I. ESTRUTURA DO COMPILADOR INCREMENTAL

I.1 - Características de um Compilador Incremental

- Análise linha por linha

Dentre as principais vantagens da compilação interativa temos a deteção imediata de erros e a edição interativa. Em outras palavras, se o usuário fornecer uma linha com erro de sintaxe, o compilador lhe comunicará imediatamente o erro, podendo este ser logo corrigido.

Devido ao fato acima, a análise de um programa em um compilador incremental será feita linha por linha. Uma linha é analisada, os erros são detetados e para as linhas corretas é feita a geração do código intermediário. Tanto a linha fonte quanto o código gerado são acrescentados ao restante do programa.

A unidade mínima de comunicação entre o usuário e o compilador será portanto uma linha. Uma linha fonte pode conter vários comandos BASIC, bem como um comando pode ocupar várias linhas. Neste caso, o usuário deve teclar uma chave especial, indicando que há continuação. Será considerado o fim quando for teclada a chave indicando o fim da linha.

- Alterações não obrigam a recompilação do programa

Em um compilador incremental as linhas do programa fonte são consideradas independentes umas das outras. Neste

caso, quando o usuário fizer qualquer alteração no programa, este não precisará ser inteiramente recompilado. Somente a(s) linha(s) alterada(s) é(são) re-analisada(s).

- Gera micro-rotinas do PLTI

Não haverá nenhuma espécie de código intermediário (ou linguagem interna). Para cada linha correta serão geradas as micro-rotinas do PLTI que corresponderão ao(s) comando(s) do BASIC que a linha contem.

- Depuração interativa

O usuário contará com comandos imediatos para ajudar a depurar seus programas. Quase todos os comandos do BASIC poderão ser usados em modo imediato (alguns não têm sentido nesse modo, e portanto não serão permitidos, como por exemplo, as declarações).

Os comandos imediatos podem ser considerados como um programa de uma linha, seguido de um STOP. São executados assim que sua análise é completada.

- Edição interativa

A facilidade de edição vai permitir ao usuário: apagar uma ou mais linhas de um segmento; listar trechos de segmentos; re-sequenciar as linhas de um segmento.

A linha será também a unidade mínima de programa fonte a ser manuseada pelo editor. Quando uma linha for apagada, ela será re-analisada, de modo a atualizar certas informações do compilador.

- Tipos de arquivos: programas e dados do usuário

Os dispositivos de entrada e saída considerados no presente trabalho serão o teclado e vídeo do terminal e uma unidade de disco.

Através do teclado o usuário poderá fornecer as linhas de um programa BASIC, comandos do compilador ou comandos imediatos, bem como os dados a serem lidos conversacionalmente pelo seu programa.

No vídeo serão mostradas: as linhas correntes, as mensagens de erro, ou linhas criadas pelo programa do usuário.

Em disco estarão armazenados:

- os programas do usuário (o fonte e o código correspondente);

- informações sobre o programa do usuário, obtidas e utilizadas pelo compilador, em forma de tabelas;

- rotinas que não estejam em uso: o compilador será desenvolvido utilizando-se técnicas de "overlay", devido à restrição de me-

mória: as rotinas menos utilizadas ficarão em disco, sendo trazidas para a memória quando forem necessárias;

- áreas temporárias e de trabalho: sua utilização será descrita no decorrer do texto;

- arquivos criados pelos programas do usuário;

Interrupção

A fase de execução deverá permitir interrupções. O compilador deverá testar continuamente (antes de iniciar a execução de qualquer novo comando) se ocorreu alguma das interrupções citadas em (I.5), na Parte I.

I.2 - Componentes do Sistema

O compilador será programado de forma modular. Cada módulo será constituído por rotinas responsáveis por tarefas específicas. A comunicação entre os diversos módulos se dará através de chamadas de rotinas.

Além de facilitar a implementação e futuras alterações, a divisão em módulos vai permitir o uso da técnica de "overlay", mencionado no item anterior.

Os componentes que constituem o compilador são os seguintes:

- analisador de comandos

- editor

- depurador

- compilador BASIC

- módulo de controle da execução

- módulo de controle principal

- carga de programas

- tratamento de erros

Em seguida será vista uma descrição resumida dos diversos componentes.

I.2.1 - Analisador de Comandos

É o componente responsável pela análise e interpretação dos comandos de controle. É constituído por uma rotina principal que aciona as rotinas correspondentes a cada comando. Será ativada sempre que o sistema não se encontrar em modo de entrada ou execução.

I.2.2 - Editor

Formado por um conjunto de rotinas que analisam e interpretam os comandos de edição. Haverá uma rotina para cada comando, que é acionada quando, em modo de entrada, é reconhecido um comando de edição.

I.2.3 - Compilador BASIC

Conjunto de rotinas responsáveis pela análise dos comandos do BASIC, bem como pela geração do pseudo-código.

I.2.4 - Carga de Programas

É chamado quando o usuário teclou o comando EXEC. Verifica os erros que não puderam ser detetados pela compilação e carrega na memória o programa a ser executado.

I.2.5 - Módulo de Controle da Execução

Esse módulo é constituído de várias rotinas que serão utilizadas em tempo de execução, quais sejam:

- Rotina de Atualização do Contador de Programas

O Contador de Programas (ou PC, da abreviatura do termo em inglês "programm counter") é um registro que deve conter sempre o endereço da próxima instrução a ser executada.

Essa rotina, no nosso caso, atualiza uma variável PC com o endereço do próximo incremento.

- Rotina de Entrada e Saída

Aciona as rotinas necessárias para a execução das operações de entrada e saída. Verifica a cadeia de formato, lê ou grava os dados de acordo com esse formato, verifica se a operação é válida para o arquivo, entre outras.

Testa se a operação foi terminada com sucesso.

- Rotina de Tratamento de Interrupção

Responsável pelas decisões a serem tomadas conforme o tipo de interrupção que ocorra durante a execução de um programa, como por exemplo:

i) se houve algum erro grave, a execução deve ser abortada;

ii) caso seja necessário uma leitura de dados, o controle é passado à Rotina de Entrada e Saída;

iii) caso o controle deva passar ao usuário devido a um STOP ou porque foi acionada alguma chave do teclado, o conteúdo atual da memória deve ser salvo para permitir que a execução possa continuar do ponto em que foi interrompida. O sistema passa então ao modo de controle.

- Rotina de Teste de Chave

Acionada após a execução de cada incremento.

Verifica se foi pressionada a chave do teclado que interrompe a execução. Em caso afirmativo, passa o controle à Rotina de Tratamento de Interrupção.

- Rotinas do FOR

Rotinas de controle de comando iterativo for. São responsáveis dentre outras tarefas, pelo incremento e teste da variável de controle.

- Rotina de Entrada de Blocos

Sempre que for chamada uma subrotina ou função de múltiplas linhas, bem como a entrada em um comando interativo, deve-se verificar se o segmento ou variável de controle correspondente ainda se encontra em uso, para evitar recursividade ou a atualização erronea da variável de controle de um for.

- Rotina de Inicialização de Variáveis

Antes de começar a execução de um segmento, as variáveis locais que são parâmetros formais devem ser inicializadas com os valores dos parâmetros reais correspondentes.

I.2.6 - Módulo de Controle Principal

Esse módulo tem como tarefas principais:

- Gerenciamento da memória

Devido às restrições de memória, ficarão em disco as rotinas referentes a comandos não muito utilizados, como por exemplo o REORGANIZE, que só serão carregadas quando forem ser executadas.

Quando for iniciar a execução de um programa, ficará na memória: o Módulo de Controle da Execução, o pseudo-código que será executado bem como as estruturas necessárias nesta fase.

- Gerenciamento de arquivos

Criação e manutenção dos arquivos fonte e de pseudo-código correspondente, permitindo o acesso aos diversos programas armazenados. Para isso deve ser mantida informações sobre cada programa, quais sejam: localização, número de segmentos,

etc. Quando o usuário desejar utilizar um programa, o sistema verifica se o mesmo existe e o trâz para a área intermediária ou emite mensagem informando se o programa não existe.

- Passar o controle ao módulo adequado conforme o modo de operação em que o sistema esteja e a entrada fornecida pelo usuário.

Será ativado sempre que o usuário for utilizar o sistema, e só devolve o controle ao SOCO (Sistema Operacional do TI) quando o usuário fornecer o comando ADEUS.

I.2.7 - Tratamento de Erros

Essa rotina é acionada por todos os módulos (à exceção do Módulo de Controle de Execução) sempre que for encontrado um erro em uma entrada fornecida pelo usuário. Sua principal tarefa é a emissão da mensagem correspondente ao erro ocorrido, e informar ao Módulo de Controle Principal que deve-se aguardar a correção do usuário.

I.3 - Estruturas de Dados

No decorrer da análise de um programa, o compilador vai obtendo informações tais como: nome e tipo dos segmentos que compõem o programa, número das linhas, incrementos que constituem cada linha, variáveis e constantes utilizadas, seu tipo e precisão, entre outras.

Estas informações devem ser armazenadas pois serão amplamente utilizadas, tanto na criação quanto nas futuras alterações do programa. Para esse armazenamento são usadas várias estruturas de dados.

Neste capítulo serão descritas as estruturas que serão criadas e utilizadas pela fase de compilação.

I.3.1 - Dicionário de Incrementos

Como as linhas que compõem um programa não serão fornecidas necessariamente na ordem em que serão executadas, essa estrutura, em forma de lista, conterá todos os incrementos de cada segmento, na ordem em que deverão ser executados. Cada nó dessa lista terá o formato mostrado na figura I.3.1.1.

BYTE	CONTEÚDO
Ø-1	número da linha em que se encontra o incremento. Esse campo dará a ordem em que estão as entradas nesse dicionário
2	tipo da instrução: indica se o incremento se refere a comando FOR, GOTO, etc.
3-4	endereço do código: endereço onde foi armazenado o código gerado para o incremento
5-6	apontador para referências: usado para referências futuras, como será mostrado em (III.4)
7-8	endereço do próximo incremento: apontador para a entrada referente ao incremento seguinte, no dicionário de incrementos

Fig. I.3.1.1 - Formato do Dicionário de Incrementos

É criada uma entrada nessa estrutura sempre que:

i) o usuário fornecer uma nova linha; ou

ii) for feita uma referência a uma linha ainda não existente. Nesse caso a entrada conterá somente o número da linha e o endereço da referência.

As informações restantes sobre o incremento só são preenchidas quando a linha fornecida não tiver erros.

I.3.2 - Tabela de Símbolos

Essa estrutura será muito utilizada durante a fase de compilação. Ela conterá os identificadores e as constantes reais ou alfanuméricas.

A busca e inserção de elementos nessa tabela é feita pela análise léxica, mas o restante dos campos é completado ou testado pelas rotinas semânticas.

I.3.2.1 - Organização

O acesso a essa tabela será direto, utilizando uma função de espalhamento aplicada a uma chave. Esse método é chamado de método de "hashing". A função de espalhamento é aplicada à chave e o seu valor dá o endereço onde deve ser iniciada a busca. Na aplicação dessa função pode ocorrer que duas chaves

diferentes sejam associadas ao mesmo endereço. Quando tal ocorre, diz-se que houve uma colisão. Para utilizar esse método deve-se escolher a função adequada e também um esquema para o tratamento de colisão.

A chave será formada pelos caracteres que constituem o identificador ou constante. A função a ser aplicada à chave obedecerá ao esquema multiplicativo, mostrado em Knuth[6].Resumidamente, o método consiste em multiplicar a chave por um valor inteiro A que seja primo com w (o tamanho da palavra do equipamento, no caso, 2 bytes) e deslocar o resultado m bits para a esquerda, onde M (o tamanho da tabela) é um valor da ordem de 2^m. A função, a qual chamaremos aqui de h(K), onde K é a chave, tem seu valor então nos bits mais alta ordem da metade à direita do produto de A por K.

Uma vantagem desse método é que não utiliza divisão, que é uma operação bastante lenta no T.I. Outra vantagem, conforme cita Knuth[6], é que ele possibilita uma boa randomização quando as chaves não são tão aleatórias (por exemplo, K tem valores do tipo TIPO1, TIPO2, TIPO3), o que diminui bastante o número de colisões. Tal vai depender da escolha de um bom valor para A.

Como o método descrito se aplica a uma palavra (isto é, 2 bytes) e a chave será composta de vários bytes (até 6, no caso de identificadores, até 256 no caso de constantes alfanuméricas ou 6, para as reais), antes de se aplicar a

função, os caracteres que compõem a chave serão combinados através de um "ou exclusivo" até obter um resultado em uma palavra. Para obter valores o mais randômico possível, será dado um deslocamento circular de cada palavra que compõe a chave antes de ser dado esse "ou exclusivo". Tal artifício fará com que diminua a possibilidade de que chaves tais como XY e YX sejam reduzidas ao mesmo valor, pois essa operação, é comutativa.

As colisões serão tratadas também através de espalhamento, que consistirá em descobrir um valor que seja primo como M, no caso m número ímpar. A sugestão, contida na mesma fonte de referência citada acima, é deslocar h(K) obtida mais m bits para a esquerda e somar 1. O valor assim obtido será acrescentado ao valor de H(K), dando o novo endereço. Esse processo deve se repetir até que não haja mais colisão ou caso a tabela esteja cheia.

Com esse método, o tempo de acesso à tabela será relativamente curto, principalmente se comparado com outros métodos, o que é de alguma vantagem de vez que essa estrutura será muito acessada na fase de compilação. No entanto, a deleção de uma entrada não é tão simples, pois pode-se reparar, pelo método de tratamento de colisão, que se uma entrada for retirada da Tabela de Símbolos, perde-se o acesso a todos os elementos aos quais a aplicação da função de espalhamento gerou o mesmo endereço. No presente trabalho pode ocorrer que vez por outra seja necessário retirar alguma entrada, pois sucessivas deleções podem fazer com que algum identificador não seja mais usado no progra-

ma, e o usuário deseje criar um identificador com o mesmo nome, mas com características diferentes, conforme pode ser visto pelo exemplo abaixo:

```
(1) segmento X
      ⋮
    10 P% = P%+2
      ⋮
    30 print P%
      ⋮
    fim
```

supor que o usuário forneça o seguinte comando de edição:

```
apaga 10,30
```

e em seguida faça as seguintes inserções

```
(2) 10 dim P(5,3)
       ⋮
    30 mat read P
       ⋮
```

Se as linhas 10 e 30 em (1) forem as únicas referências a P%, as inserções em (2) serão perfeitamente válidas. Caso a entrada referente a P% não seja retirada da lista de

atributos na Tabela de Símbolos, ocorreria erro em (2), pois P% e P( ) não são consideradas aqui como variáveis diferentes.

Para que a deleção de uma entrada não obrigue a recriar toda a tabela (tal ocorrerá quando a lista de atributos referentes a um determinado identificador ficar vazio), o que seria necessário para que algumas entradas não se tornem inatingíveis, haverá um campo indicando se a entrada foi desativada. Desse modo, a busca a um elemento na Tabela de Símbolos seguirá os seguintes passos:

i) aplicar a função de espalhamento ao símbolo;

ii) se a posição estiver vazia, indicar que não houve sucesso na busca;

iii) se a posição estiver ocupada com um símbolo diferente do que é esperado, aplicar os procedimentos para o tratamento de colisão; senão, pare a busca e acuse sucesso;

iv) se a posição foi desativada, presseguir na busca.

Caso a busca não tenha tido sucesso e seja necessário inserir o novo símbolo, essa inserção se dará na primeira posição vazia ou desativada que for encontrada.

I.3.2.2 - Conteúdo

A tabela será acessada pelo método descrito em (I.3.2.1), onde a chave poderá ser formada por:

- nome e identificação do segmento, no caso de variáveis

- nome da função ou subrotina

- valor da constante real ou alfanumérica.

Cada entrada será dividida em 3 partes:

i) espécie: indica o tipo de símbolo (identificador, constante, etc.) ou se está vazia ou desativada

ii) nome ou valor

iii) atributos.

O campo de nome terá 6 bytes, que é o número máximo de caracteres para um identificador. Para as constantes alfanuméricas a busca será um tanto mais demorada, pois como podem conter até 256 caracteres, não caberá todo nesse campo, devendo a busca se dar na área de constantes alfanuméricas.

A terceira parte contém as informações que o compilador irá necessitar sobre o símbolo, o que cons

tituem os atributos. Estes variam de acordo com a espécie de símbolo, como pode ser notado a seguir:

a) Variáveis

- tipo:inteira, real ou alfanumérica
- precisão: simples ou dupla
- endereço de alocação
- se foi definida ou não
- número de dimensões
- contador de referências
- segmento em que está sendo usada
- se é parâmetro formal ou não

b) Funções e subrotinas

- tipo da função: inteira, real ou alfanumérica
- espécie: se é declaração ou função de múltiplas linhas
- se já foi definida ou não
- se foi atribuído valor à função dentro do segmento ou não
- segmento de definição

c) Arquivos

- periférico associado

- tamanho do registro

d) Constantes

- tipo

- precisão

- endereço de alocação

- segmento

O formato da entrada da tabela é mostrado na figura (I.3.2.2). Por questão de economia de espaço as informações mutuamente excludentes ocuparão as mesmas posições de memória.

BYTE	BYTE	CONTEÚDO
0	0-7	espécie:=0: entrada vazia; =1: entrada deletada; =2: identificador; =3: constante real; =4: cte. alfanumérica
1-6	0-47	símbolo: nome do identificador|valor da constante| 6 primeiros caracteres da cadeia alfanumérica
7	0-7	identificador do segmento de definição
8	0-3	uso:=0000: var. simples; =0001: var. subscrita; = 0010: função; =0011: função declaração; =0100: subrotina; = 0101: arquivo; =0110: constante
	4-7	tipo: =0000: inteiro; =0001: real; =0010: alfan.

continua...

Continuação...

BYTE	BITS	CONTEÚDO
9	0	indicador de parâmetro formal
	1	indica se já houve definição
	2	se foi atribuído valor no segmento
	3	precisão (=0: simples; =1: dupla)
	4-7	número de dimensões|número de paramêtros|periférico associado ao arquivo
10	0-7	contador de referências (*)
11-12	0-15	- para arquivos: tamanho do registro - para fç. e subr.: ender. da lista de parâmetro

Fig. I.3.2.2

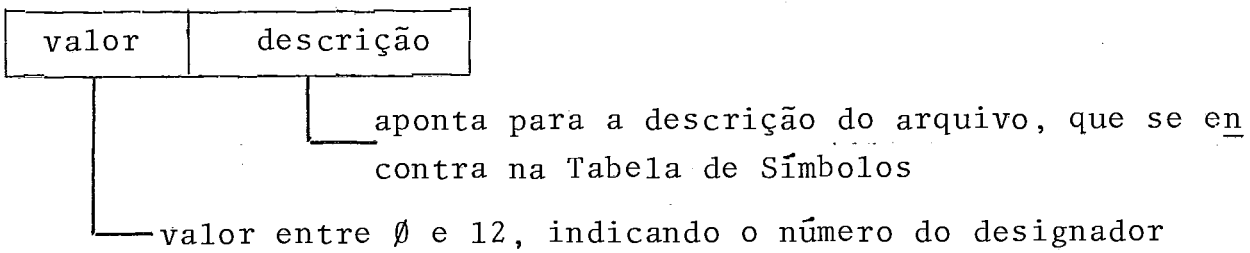
Observações:

(*) é incrementado de 1 a cada aplicação do identificador e decrementado do mesmo modo quando uma linha contendo esse identificador é deletada. Quando chegar a zero, significa que o símbolo não está mais sendo utilizado no segmento, e a entrada correspondente na tabela é marcada como deletada. Uma entrada na Tabela de Símbolos referente a uma função ou subrotina só é retirada da tabela (isto é, marcada como deletada), quando o segmento de definição for deletado. Esse controle será feito pelo Analisador Léxico.

I.3.3 - Tabela de Designadores de Arquivos

Esta tabela será utilizada em fase de compilação para conter os designadores de arquivos que aparecem em cada declaração FILE que é fornecida pelo usuário. Haverá sempre uma entrada correspondente ao terminal, que é dispositivo básico de entrada e saída e está associado ao designador Ø.

Cada entrada dessa tabela terá o formato:



A utilização dessa tabela é para fins semânticos; pode-se assim determinar se o usuário definiu arquivos diferentes associado ao mesmo designador.

I.3.4 - Tabela de Segmentos

A Tabela de Segmentos constitui o primeiro nível de endereçamento para o arquivo objeto. Juntamente com o Dicionário de Incrementos, que seria assim o segundo nível de endereçamento, pode-se acessar o código gerado para qualquer incremento de qualquer segmento.

Esta estrutura será utilizada em fase de compilação e cada entrada terá as seguintes informações:

- nome do segmento: até 6 bytes, contendo o identificador que aparece a cada comando SEGMENTO

- tipo: indica se é o programa principal, subrotina ou função

- endereço da área de dados: indica onde começa a área de dados do segmento

- endereço de alocação: contem a posição de memória onde começará a alocação do segmento

- apontador para o Dicionário de Incrementos: indica onde começa a lista correspondente aos incrementos do segmento

- tamanho: número de bytes indicando a área necessária para alocar o código do segmento

- chave de erro: será ligada sempre que no Passo 2 da análise for detetado algum erro do tipo: número de linha referenciado não foi definido, etc. Esta chave só será desligada quando for dado o comando SEGMENTO, seguido do nome do segmento errado

- total de linha|incrementos: contem o número total de linhas e de incrementos que contem o segmento. É incrementado de 1 a ca

da nova linha que é fornecida sem erro, e diminuído de 1 quando o usuário apagar a linha.

I.3.5 - Tabela de Palavras Reservadas

Conterá todos os comandos que o usuário poderá utilizar no sistema, quais sejam, os comandos de controle, edição, entrada e da linguagem BASIC.

Além do verbo que indica o comando, essa tabela deverá conter o modo em que este pode ser utilizado. Assim, por exemplo, se o sistema se encontrar em modo de entrada e o usuário fornece o comando REORG, é acusado erro. Essa verificação é feita pelo analisador léxico ("scanner").

I.3.5.1 - Estrutura da Tabela

Para maior facilidade no fornecimento de comandos, será permitido o uso de abreviaturas, ou seja, com somente algumas das letras iniciais pode ser possível o reconhecimento de um comando. Assim, por exemplo, o comando APAGA pode ser fornecido de qualquer dos seguintes modos: AP|APA|APAG|APAGA.

Olhando-se a lista de palavras reservadas, no Apêndice E, vemos que somente a letra A não identifica o comando pois temos ainda ADEUS e AND.

Para representar esta tabela, foi utilizada uma variação da "trie structure", mostrada em Knuth[6].

Aqui, a busca de um determinado elemento é feita através de comparações entre caracteres, e não entre chaves, como é feito, por exemplo, no acesso à Tabela de Símbolos. É como se usássemos um dicionário, ou seja, a primeira letra da palavra indica aonde encontrar todas as palavras reservadas que comecem com essa letra.

A "trie structure" descrita na referência acima citada, consiste em um árvore "M-ária", na qual cada nó são vetores de M-elementos, cada qual contendo dígitos ou caracteres. Cada nó em um nível ℓ dessa árvore representa um conjunto de todas as palavras que comecem com a sequência de ℓ caracteres; o nó apresentaria então M possibilidades de desvios, dependendo do $(\ell+1)$-ésimo caracter.

Uma das variações a esse método sugere que se use uma estrutura tipo floresta, ao invés de nós contendo M elementos. Tal economizaria memória, pois a maioria das entradas desse vetor estará vazia. A figura (I.3.5.1) dá um exemplo de como seria a representação da Tabela de Palavras Reservadas usando-se a "trie" (I.3.5.1.a) e usando a variação em forma de floresta (I.3.5.1.b). No caso do exemplo, é mostrado somente um subconjunto das palavras que comecem com R, S e W. O caracter —| representa o fim da palavra.

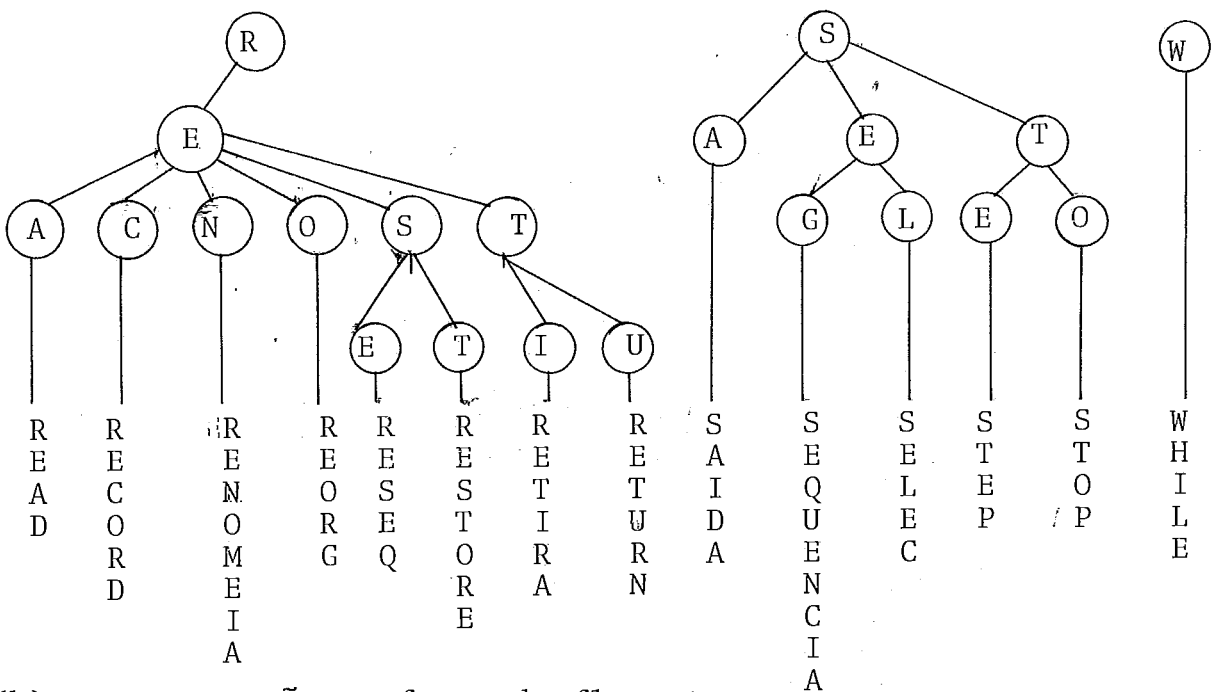
	1	2	3	4	5	6	7	8
⊣								
A				Read				
B								
C				Record				
D								
E		(4)	(5)			Step	Reseq	
F								
G					Segmento			
H								
I								Retira
J								
K								
L					Selec			
M								
N				Renomeia				
O				Reorg		Stop		
P								
Q								
R	(2)							
S	(3)			(7)				
T			(6)	(8)			Restore	
U								Return
V								
W	While							
X								

continua...

Continuação ...

	1	2	3	4	5	6	7	8
Y								
Z								

(a) representação em forma de trie das palavras reservadas começadas por R, T ou W



(b) representação em forma de floresta

Fig. I.3.5.1

O método utilizado neste trabalho englobará as duas estruturas citadas anteriormente, do seguinte modo:

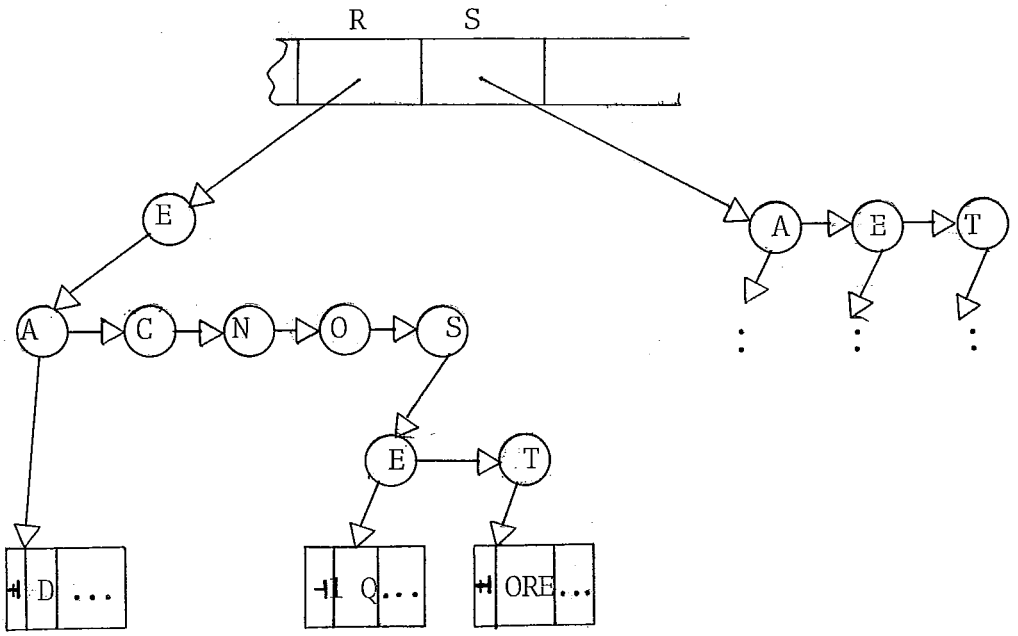
(1) a raiz de cada árvore da floresta em (I.3.5.1.b) estará em um vetor de 26 posições, correspondentes a cada letra do al-

fabeto. O acesso a um elemento se dará diretamente, portanto. O conteúdo de um elemento desse vetor será o endereço da árvore contendo todos os caracteres das palavras que comecem com uma determinada letra.

(2) os caracteres que compõem as palavras reservadas estarão em uma árvore binária. A busca será feita comparando-se o caracter da árvore, presseguindo pelos nós à direita até encontrar o caracter procurado. A partir de então toma-se o nó à esquerda e procede-se na busca dos caracteres seguintes do mesmo modo.

(3) as folhas das árvores conterão o restante dos caracteres das palavras, a partir de onde não haja mais dúvida possível sobre qual a palavra procurada. A partir daí, faz-se uma busca sequencial até atingir o fim da entrada.

A figura I.3.5.2 mostra como ficará a representação do exemplo anterior.



Apesar de ser mais lento que a estrutura "trie", esse método tem as seguintes vantagens:

- é mais econômico em termos de espaço;

- o acesso ao 1º caracter da palavra é direto, o que torna a busca um tanto mais rápida que na floresta;

- cada nó não falha tem tamanho fixo, o que os torna mais fáceis de manipular do que a forma de floresta.

I.3.5.2 - Conteúdo dos Componentes

Cada entrada do vetor conterá um endereço (2 bytes) de onde se encontra a árvore contendo as palavras que iniciem com determinada letra.

A árvore conterá dois tipos de nós: as folhas, que contêm o restante dos caracteres da palavra, e os outros, que são raízes de sub-árvores que iniciam com uma dada sequência de caracteres. Estes últimos nós serão formados pelos seguintes campos:

- tipo: =∅, indicando que é um nó não-falha

- apontador para a sub-árvore esquerda

- apontador para a sub-árvore direita

- caracter

O conteúdo de uma folha será:

- tipo: =1, indicando que é folha

- código interno associado à palavra reservada

- modo de operação onde a palavra pode ser usada

- número de caracteres restantes

- resto dos caracteres

A verificação sobre a validade do uso de uma determinada palavra será feita pelo analisador léxico.

I.4 - Organização da Memória

Em um compilador incremental, as tabelas utilizadas pela fase de compilação devem ser preservadas, mesmo que o programa já tenha sido executado. De modo análogo, a área de dados do programa deve ser preservada quando houver uma interrupção. O exemplo a seguir ilustra isto:

```
10 dim A(20)
20 let A(3) = 57
30 end
exec
print A(3)
```

Vê-se que a área contendo os valores de A deve ser preservada mesmo após o término da execução, e a Tabela de Símbolos não deve ser destruída após a compilação, senão o comando imediato: print A(3) jamais acharia o elemento A(3).

Como a memória disponível é pequena, não será possível manter todas as rotinas do compilador, suas estruturas, o programa do usuário com a respectiva área de dados o tempo todo na memória. A fase de compilação e a de execução serão portanto consideradas distintas, de modo que as rotinas e dados necessários em uma delas serão carregados depois que os da outra tenham sido retirados.

Na fase de compilação a memória deverá conter:

a) rotinas do compilador

- Módulo de Controle Principal: controlará a execução do compilador;

- Tratamento de Erros: recebe o controle sempre que for encontrada uma linha com erro de síntaxe;

- Análise de Comandos: analisa os comandos de controle que o usuário forneça;

- Editor: analisa os comandos de edição;

- Compilador BASIC: analisa uma linha de programa fonte e gera o código correspondente.

b) estruturas

As tabelas, dicionários e listas criadas e mantidas pelo compilador;

c) "buffers"

São áreas que conterão a linha fonte e o código gerado para essa linha.

d) Área comum

Conterá informações que são compartilhadas tanto pela fase de compilação quanto pela de execução, como por exemplo: identificação do segmento corrente, ponto em que sua execução foi interrompida, entre outras.

Na fase de execução, o código do programa, que se encontra em disco, deve ser trazido para a memória, bem como sua área de dados. O conteúdo da memória será então:

a) Rotinas

- Módulo de Controle Principal: supervisionará a execução do programa;

- Tratamento de Erros: emitirá mensagens referentes aos erros que ocorram na fase de execução;

- Módulo de Controle da Execução: conterá rotinas necessárias a essa fase;

b) Programa do Usuário

Tem tamanho fixo nesta fase, pois não é permitida a recursividade e nem o aninhamento de funções ou subrotinas.

c) Área de Dados do Programa

Cresce no sentido inverso ao programa, e estará dividida em duas partes:

- área estática: conterá variáveis simples, constantes, áreas de E/S, a Tabela-data, variáveis subscritas e temporárias numéricas. Seu tamanho é fixo no decorrer da execução. Só serão referenciadas as variáveis que estejam na área de dados de cada segmento, e não será permitida a alocação dinâmica de variáveis subscritas.

- área dinâmica: conterá as constantes e variáveis temporárias alfanuméricas. Pode variar de tamanho na fase de execução, tomando e liberando continuamente espaço da área livre.

d) Área Comum.

I.5 - Organização dos Arquivos

Será tratado aqui somente a organização dos programas em disco. Os arquivos de dados do usuário serão manipulados através do SOCO.

Os programas do usuário poderão ser armazenados de forma temporária ou permanente. Neste último caso, o usuário deve fornecer o comando GUARDE. Será necessária portanto uma área temporária, para conter o programa que está sendo criado ou edi-

tado. O conteúdo dessa área é perdido sempre que:

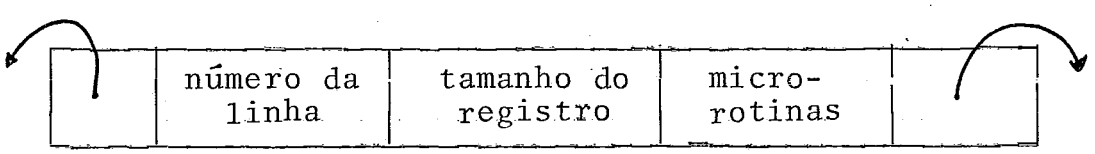
a) o usuário fornecer o comando de controle PEGUE;

b) o usuário fornecer o comando ADEUS, abandonando o sistema.

Os arquivos de programa, quer sejam temporários ou não, estão divididos em duas partes:

- área fonte: contém as linhas do programa fonte, acessadas através do método sequencial indexado do SOCO, organizado por ordem do número das linhas. Tal permitirá tanto o acesso direto, quando se desejar alguma linha específica, quanto o sequencial, por exemplo, quando o programa for listado.

- área de código: organizado em forma de lista de apontadores, por ordem do número da linha dentro de cada segmento. Tal evita que o arquivo tenha que ser re-arrumado sempre que for feita uma edição. Para obter a próxima instrução a ser executada, bastará seguir os apontadores. A cada linha corresponderá um nó nessa lista, com o seguinte formato:



onde:

- os campos das extremidades são apontadores para o registro anterior e para o próximo, necessários para a edição;

- o 2º campo é o número da linha fornecida pelo usuário;

- o tamanho determina onde acaba um determinado registro, de vez que seu formato é variável;

- o 4º campo conterás as micro-rotinas geradas para cada incremento da linha.

Além do programa, a área de código conterá ainda os dados e as tabelas criadas pelo compilador. Precedendo essa área haverá um registro de identificação contendo:

- o nome do programa

- o número total de segmentos

- endereço da Tabela de segementos

- endereço da área livre (área constituida por registros que ainda não tenham sido utilizados pelo programa).

Devido a facilidade de edição, essa área de código é dinâmica, ou seja, seu tamanho varia conforme registros sejam inseridos ou retirados do arquivo. Por questões de simplificação e pouco uso de memória, não serão reaproveitados os espaços referentes a registros que foram apagados. Caso não haja mais espaço disponível, o usuário deverá fornecer o comando REORG, para

compactação do arquivo de código. Apesar do custo envolvido neste tipo de operação, obtem-se maior economia de tempo do que se fosse feita uma rearrumação a cada inserção ou deleção.

Além dos programas, o compilador deve dispor também de uma área de trabalho onde será armazenada, por exemplo, a imagem da memória de um programa que estava sendo executado. Essa imagem é necessária pois, como já foi mencionado, a área de dados do programa que foi executada deve ser preservada na fase de compilação.

I.6 - Resumo do Funcionamento

Será mostrado resumidamente aqui a relação entre os diversos componentes e como eles são acionados, sob a forma de um algoritmo.

1. Início

Quando o usuário teclar BASIC, o SOCO passa o controle ao módulo de controle principal, que traz para a memória as rotinas do analisador de comandos, do editor, do compilador BASIC e de tratamento de erros.

2. Usuário Fornece um Novo Programa

O compilador BASIC analisa cada linha, reporta os erros e gera o código; são criadas as estruturas de dados refe-

rentes ao programa. A linha fonte e o código correspondente vão sendo guardadas nas respectivas áreas temporárias.

3. Em vez de criar um programa, o usuário utilizou o comando PEGUE

Nesse caso deve ser verificado se o programa desejado realmente existe. O fonte e o código são trazidos para a área temporária. As tabelas são carregadas na memória.

4. Usuário fornece o comando EXEC

O módulo de controle principal chama a rotina de carga de programas, que carrega o código a ser executado na memória. Em seguida são carregadas as rotinas do módulo de controle da execução (MCE).

5. Durante a execução

Caso haja algum erro, o módulo de controle de execução (MCE) aciona a rotina de tratamento de erros, e em seguida a execução é terminada.

6. Interrupção da execução

A rotina de tratamento de interrupção, que faz parte do MCE, deve salvar na área de trabalho a imagem da memória e o controle passa ao módulo de controle principal, e volta-se à

fase de compilação.

7. Usuário fornece comando de edição

O módulo de controle principal chama o editor. É acionada uma chave na tabela de segmentos indicando que houve edição.

8. Usuário fornece comando imediato

O compilador BASIC verifica se não há erro de sintaxe e gera o código. Em seguida são carregadas as rotinas de módulo de controle da execução, bem como a área de dados do segmento corrente.

9. Usuário fornece o comando CONT

O módulo de controle principal deve verificar:

a) se havia uma execução de programa do usuário em andamento;

b) se nenhum segmento foi editado. Neste caso, o usuário deverá fornecer o comando EXEC.

Em seguida, são carregadas as rotinas do módulo de controle da execução, e a imagem da memória é reconstituída a partir da área de trabalho.

10. Fim da execução

Procedimento análogo a (6).

11. Usuário fornece ADEUS

O controle é devolvido ao SOCO.

II.O PROCESSADOR DO CÓDIGO A SER GERADO

Para tornar o desenvolvimento de sistemas o mais independente possível da máquina real e do processador ora existente, foi criado um processador hipotético, especialmente voltado para as necessidades do PLTI, que é a linguagem na qual são desenvolvidos a maioria dos sistema do T.I., inclusive o presente projeto. Esse processador foi chamado de hipotético porque na verdade ele não existe fisicamente (i.e., não há hardware que lhe corresponda), constituindo então uma máquina virtual.

Essa máquina vai lidar com dois tipos de endereçamento: de instruções do programa e dos dados. As instruções do programa são carregadas na memória quando da sua execução e permanecem inalterados durante a fase de execução. Os dados são manuseados por uma pilha, e todas as operações aritméticas operam com os dados que estão no topo da pilha. O elemento no topo da pilha é apontado por S, e as posições seguintes são referenciadas como SØ, S1, ..., S|N|. O contador de programas (ou programm counter - PC) contem o endereço da instruções seguinte à que está sendo executada.

As intruções do processador PLTI, chamadas de micro-rotinas do PLTI são interpretadas, e manipulam com dois tipos de dados: byte e address, sendo que este último ocupa 2 bytes. Dispõe-se de instruções que:

- carreguem na pilha os dados a serem manuseados (instruções do tipo LOAD), por exemplo:

  LIB - carrega um dado do tipo byte

- armazenem o topo da pilha em uma posição de memória cujo endereço (byte ou address) se encontra na posição seguinte na pilha. Exemplos:

  SB - armazena dado byte
  SA - armazena dado address

- efetuem operações aritméticas, lógicas e comparações com operando(s) no topo da pilha; como por exemplo:

  ADD - adiciona dois valores no topo da pilha
  MINU - troca sinal do elemento no topo da pilha

- executem deslocamentos ('shift') de bits para a esquerda ou para a direita com o elemento no topo da pilha:

  SLR - deslocamento para a direita

- executem comandos do PLTI:

  CASE - desvio computado
  IF - desvio condicional
  DOTA - comando iterativo com variável de controle addr.

- executem operações de entrada e saída:

  READ, WRITE

- ativem subrotinas quando houver alguma chamada e retornem ao ponto de partida:

  ENT - entrada de rotina
  RET - retorno ao ponto de chamada

O formato de cada instrução vai depender do tipo de parâmetros necessários. Em geral, o tamanho de cada instrução varia de 1 a 5 bytes.

III. O COMPILADOR BASIC

III.1 - Descrição Geral

Para cada linha fonte, o compilador BASIC deve executar os seguintes passos:

(i) análise sintática: verifica a sintaxe da linha fonte, reportando os erros ao usuário;

(ii) análise semântica e geração de código: traduz a linha fonte para as micro-rotinas do PLTI. O código gerado vai sendo guardado em um "buffer". Se a linha não contiver erros, o conteúdo do "buffer" é transferido para a área temporária de código.

Como cada linha é analisada independentemente das outras, não é possível, nessa fase da análise, verificar, por exemplo, o correto fechamento de blocos, ou se todas as variáveis subscritas utilizadas foram declaradas. Para essa verificação, será necessário conhecer a estrutura completa do segmento.

A compilação de um segmento terá então dois passos: no primeiro, são efetuadas as análises descritas em (i) e (ii). Quando o usuário fornecer o comando fim, toda a estrutura do segmento estará descrita pelas estruturas: dicionário de incrementos, tabelas de símbolos e outras. Nesse ponto então inicia o segundo passo da compilação.

Nos capítulos seguintes serão descritos os componentes do compilador BASIC.

III.2 - Análise Léxica

Nem todos os elementos presentes em uma cadeia de entrada constituem-se de um único caracter. Há símbolos que são formados por uma sequência de caracteres, como por exemplo: 3.5, VAR$, etc. A análise léxica é efetuada por uma rotina que tem como principais funções:

- identificar e ignorar os comentários;

- reconhecer as palavras reservadas tais como: DATA, PROG, etc.;

- reconhecer os identificadores;

- reconhecer uma sequência de dígitos como uma constante numérica;

- reconhecer delimitadores: aspas, (,),:, etc.;

- reconhecer pares de caracters: **,==, etc.

- descobrir o uso inválido de comandos; ex.: dim usado em modo imediato.

Uma vez reconhecido um símbolo, o analisador léxico fornece à rotina que o chamou o tipo (se é um delimitador, palavra reservada, e outros) e a sua representação interna.

Para percorrer a cadeia de entrada, o analisador léxico usa uma rotina interna para obter o próximo caracter.Além disso, essa rotina:

- reconhece o caracter de controle que indica o fim da linha;

- informa ao analisador léxico sobre a presença de um caracter inválido (não reconhecido pela linguagem);

- suprime os espaços em branco que separam os diversos símbolos da linha.

Os símbolos que representem constantes reais ou alfanuméricos e os identificadores serão armazenados na Tabela de Símbolos. Desse modo, o uso de um identificador pode ser comparado com suas aparições anteriores, o que permite verificar, por exemplo, se uma variável está sendo usado ora como simples, ora como subscrita. Às constantes será associado um nome interno,que é inserido nessa tabela, a menos das constantes inteiras. Tal se deve ao fato de o código gerado permitir somente a manipulação de constantes inteiras. A descrição detalhada dessa tabela se encontra no item relativo às estruturas utilizadas pelo compilador.

Além da Tabela de Símbolos, o analisador léxico irá manusear com tabela para o reconhecimento de palavras reservadas. Conforme foi visto na PARTE I, o sistema manipula com 4 tipos de comandos, que variam de acordo com o modo em que se encontre. Portanto essa tabela irá conter todos os comandos do sistema e o modo em que eles poderão ser utilizados. Caso o modo corrente não coincida com o da tabela, é acusado erro.

É também tarefa do analisador léxico verificar se o valor de uma constante numérica, ou o número de caracteres de uma cadeia alfanumérica não ultrapassa os limites permitidos. Também verifica se o tamanho de um identificador não tem mais que o número máximo de caracteres estipulado, e se a Tabela de Símbolos está cheia. Nesses casos a análise é interrompida e a mensagem correspondente é enviada ao usuário.

III.2.1 - Autômato de Reconhecimento Léxico

Do mesmo modo que para a análise sintática, pode-se utilizar um grafo para representar as rotinas de um analisador léxico. Só que neste caso, como geralmente as produções que definem os símbolos da linguagem formam uma gramática regular, esse grafo é chamado de autômato finito. A gramática é dita regular pois todas as suas produções são do tipo:

U::=N ou U::=WN

onde N representa o conjunto de símbolos terminais
W representa os símbolos não-terminais

O autômato léxico se encontra no Apêndice A.

III.2.2 - A Saída do Analisador Léxico

Cada símbolo reconhecido pelo analisador léxico tem uma representação interna conforme a categoria a que pertença(delimitador, identificador, constante, etc.). Essa representação é dada por um ou mais valores internos, que são passados às outras fases de análise, sendo processados mais eficientemente do que os símbolos que são reconhecidos.

A representação interna se constitui então no tipo de símbolo encontrado e um valor que o identifique, conforme mostra a tabela (III.2.2.1).

SAÍDA DO ANALISADOR LÉXICO	
CLASSE	ATRIBUTOS
palavra reservada	código interno
constante inteira	valor
delimitador	código interno
separador	código interno
identificador	posição na tabela de símbolos
constante alfanumérica	posição na Tabela de Símbolos
constante real	posição na Tabela de Símbolos
operador	código interno

Fig. III.2.2.1

A posição dos identificadores e constantes na Tabela de Símbolos é necessária pois os seus atributos, tais como tipo, precisão, etc., serão utilizados pelas rotinas semânticas.

III.2.3 - Porque Separar o Analisador Léxico do Analisador Sintático

O analisador léxico pode ser embutido nas rotinas de análise sintática, isto é, cada comando teria a função de descobrir, na cadeia de entrada, os símbolos subsequentes. No presente trabalho essas etapas encontram-se separadas, pelos motivos seguintes:

- o analisador léxico não será feito utilizando-se do método de análise preditiva;

- a análise léxica será compartilhada entre os diversos analisadores que comporão o sistema;

- no caso de alguma mudança no 'hardware' que implicasse, por exemplo, na modificação da forma de representação de caracteres, não será necessário alterar muitas rotinas. Precisamente, somente a rotina que lê cada caracter seria alterada.

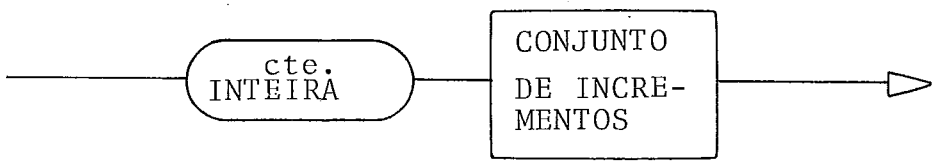
III.3 - Análise Sintática

Para a análise sintática será utilizado o método chamado de descida recursiva. É um método simples e amplamente utilizado. As rotinas do analisador são escritas diretamente a partir da gramática da linguagem.

A gramática do BASIC será representada aqui sob a forma de um diagrama ou grafo sintático. Esse grafo representará o fluxo que seguirá o processo de reconhecimento de uma sentença. Os detalhes sobre a construção desse diagrama e do analisador a partir do mesmo podem ser encontrados em Wirth[6]. A sintaxe da linguagem é mostrada através do diagrama na figura (III.3.1)a seguir.

DIAGRAMA SINTÁTICO

Linha BASIC



Conjunto de Incrementos

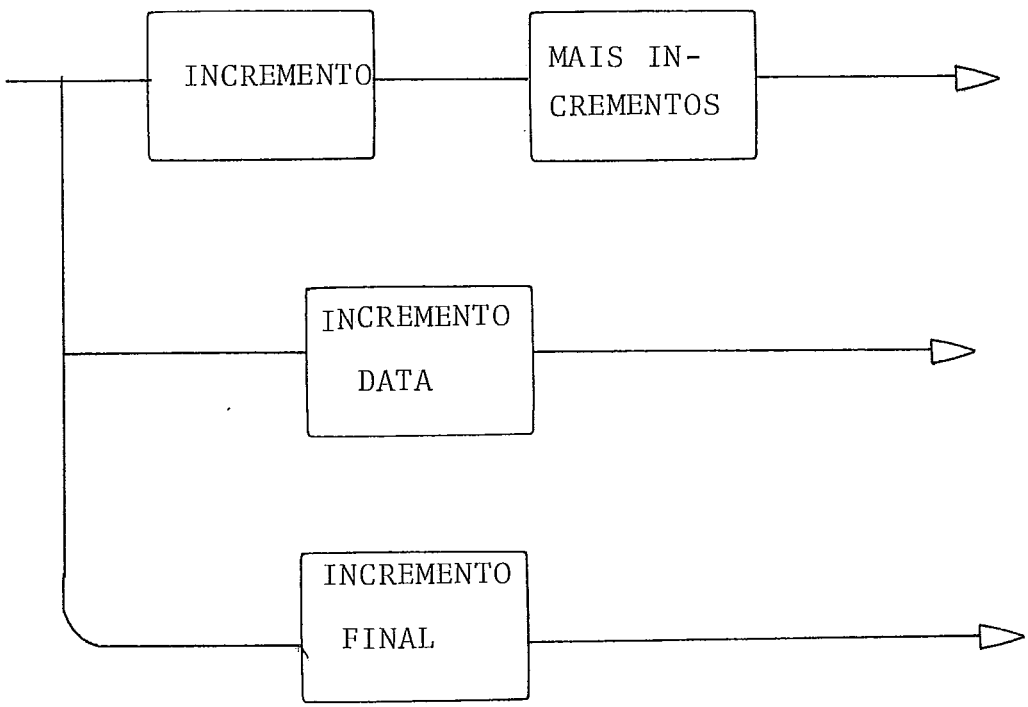
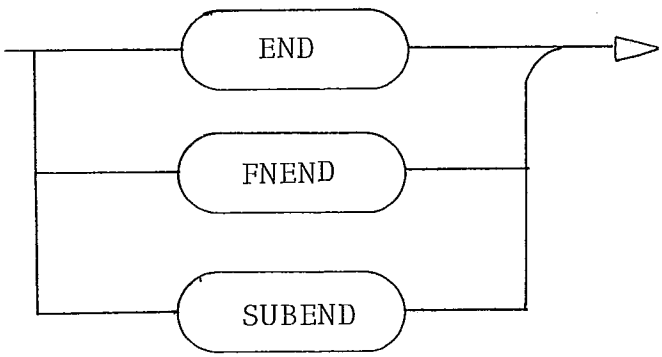
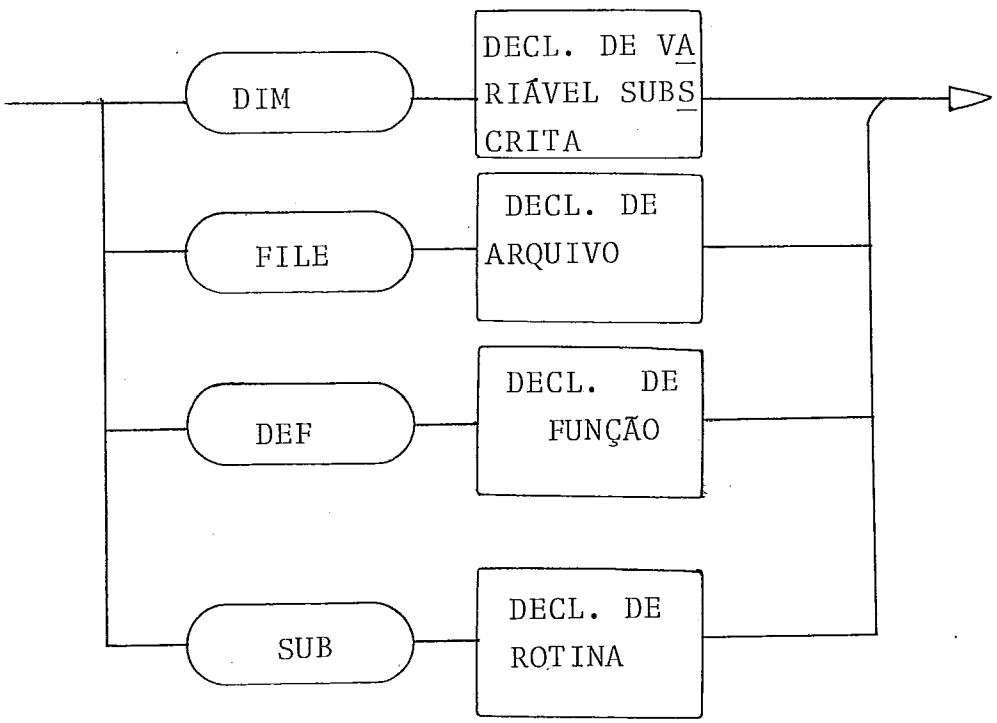


Fig. III.3.1 - Diagrama Sintático

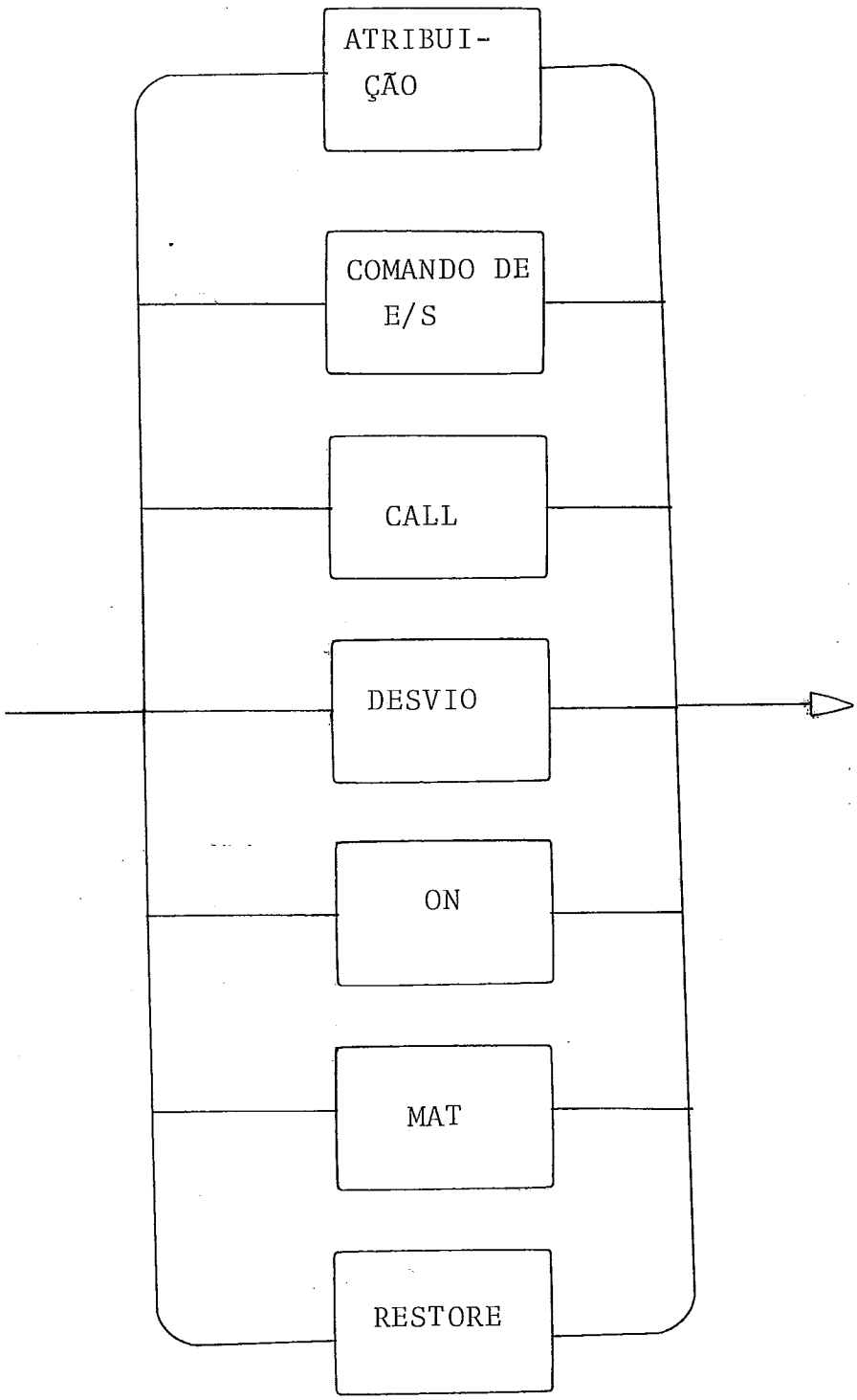
Incremento Final



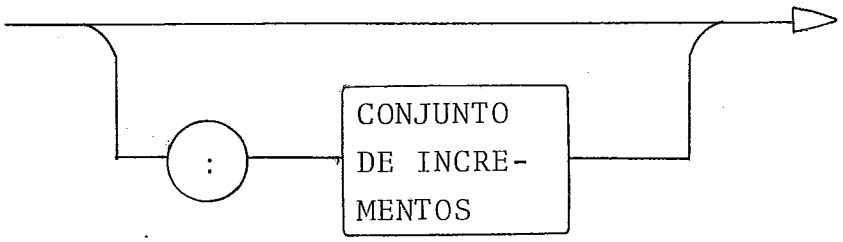
Declaração



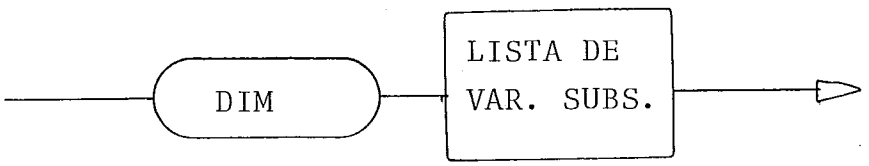
Comando Elementar



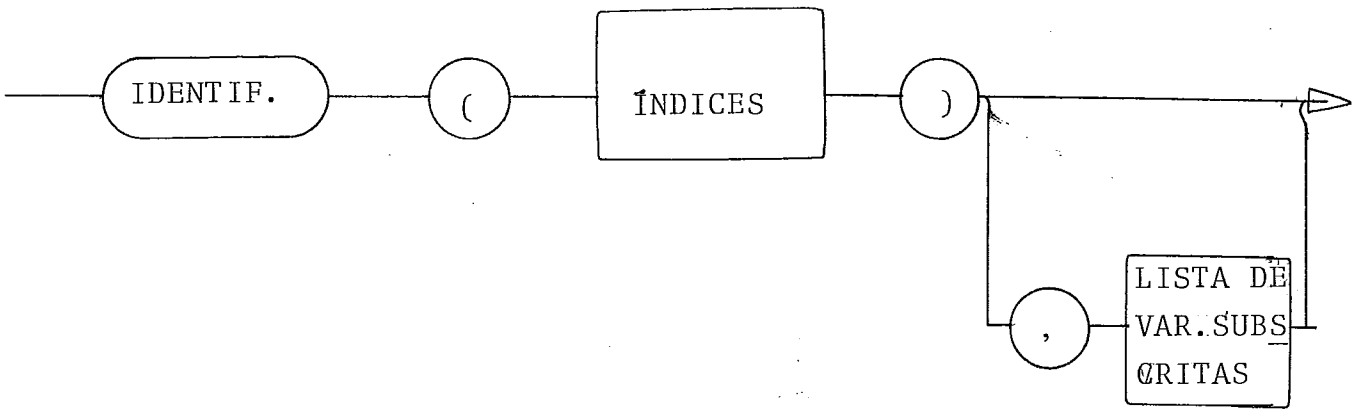
Mais Incrementos



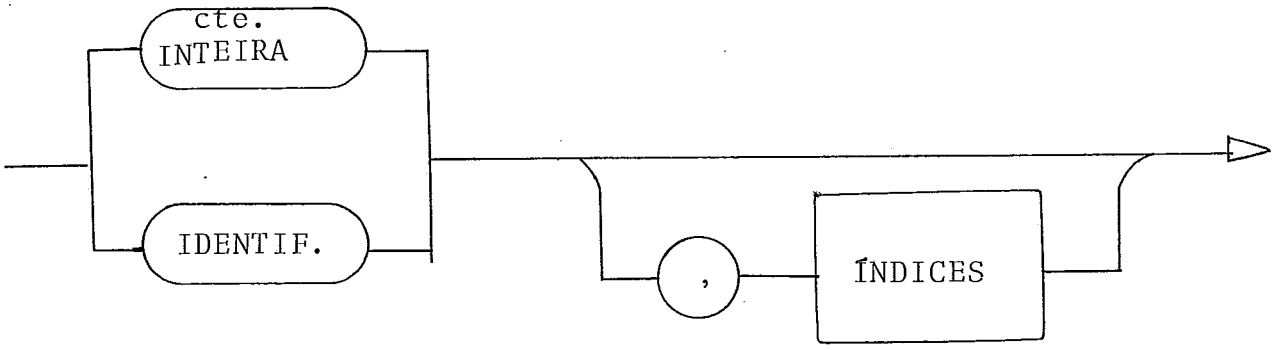
Dim



Lista de Variáveis Subscritas

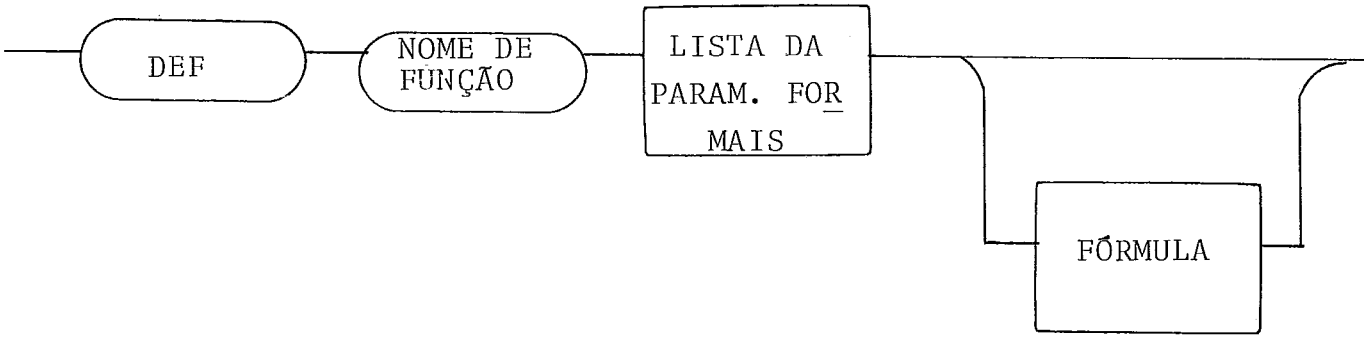


Índices

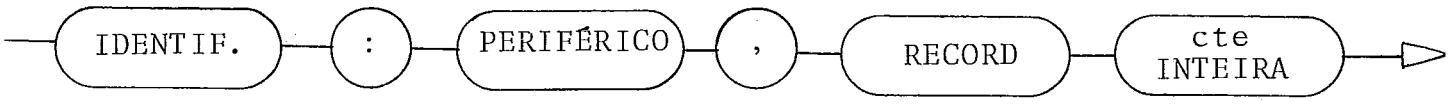


File

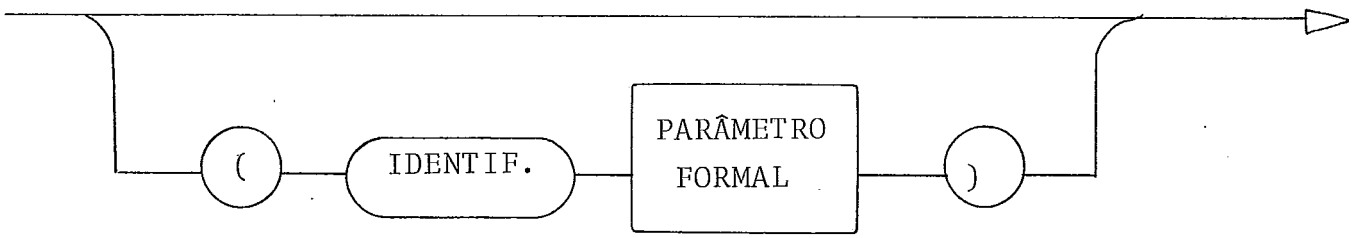
Def



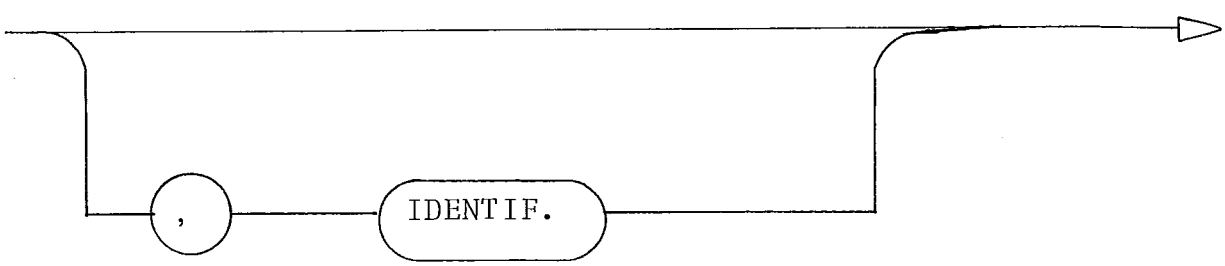
Definição de Arquivo



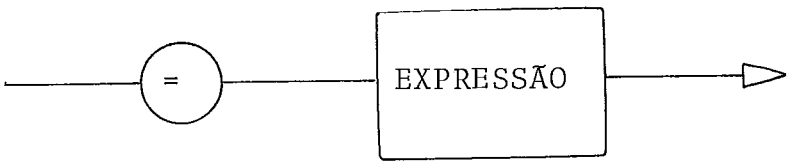
Lista de Parâmetros Formais



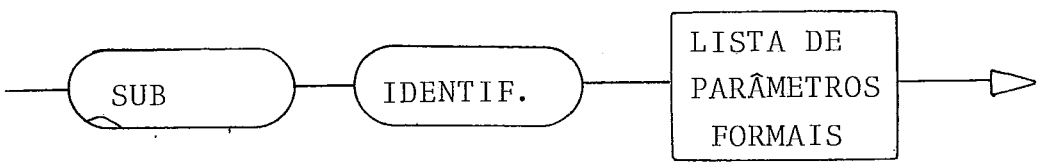
Parâmetro Formal



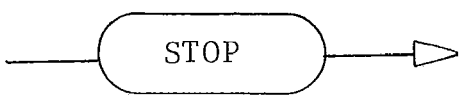
Fórmula



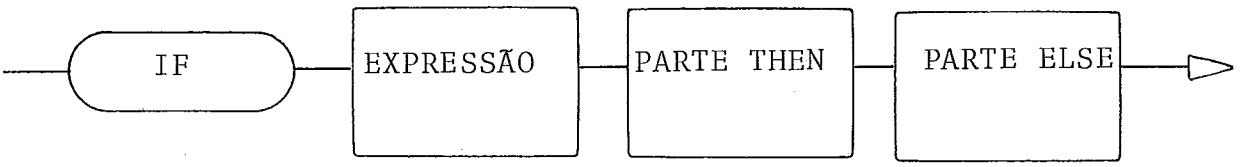
Sub



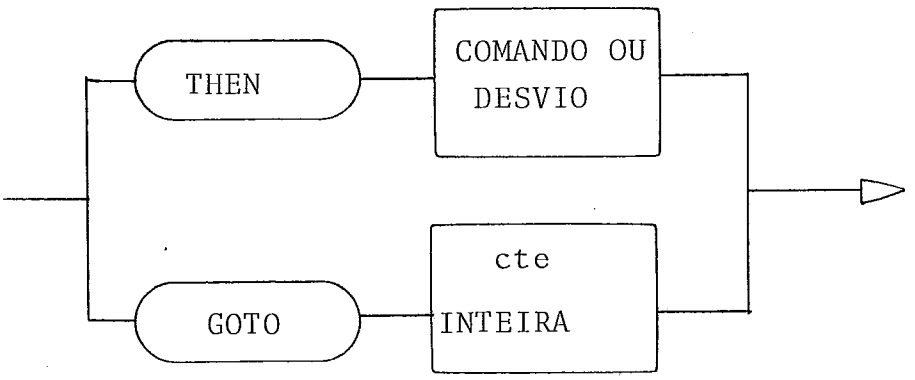
Stop



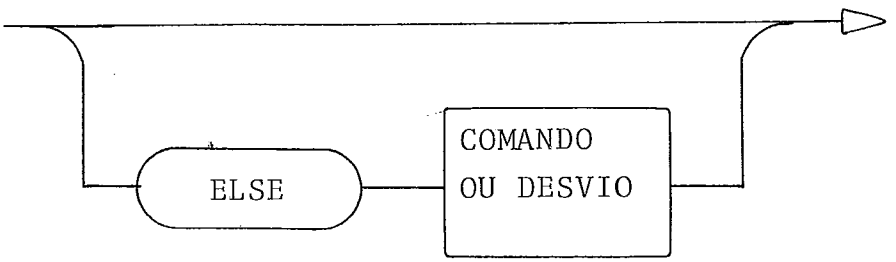
Comando IF



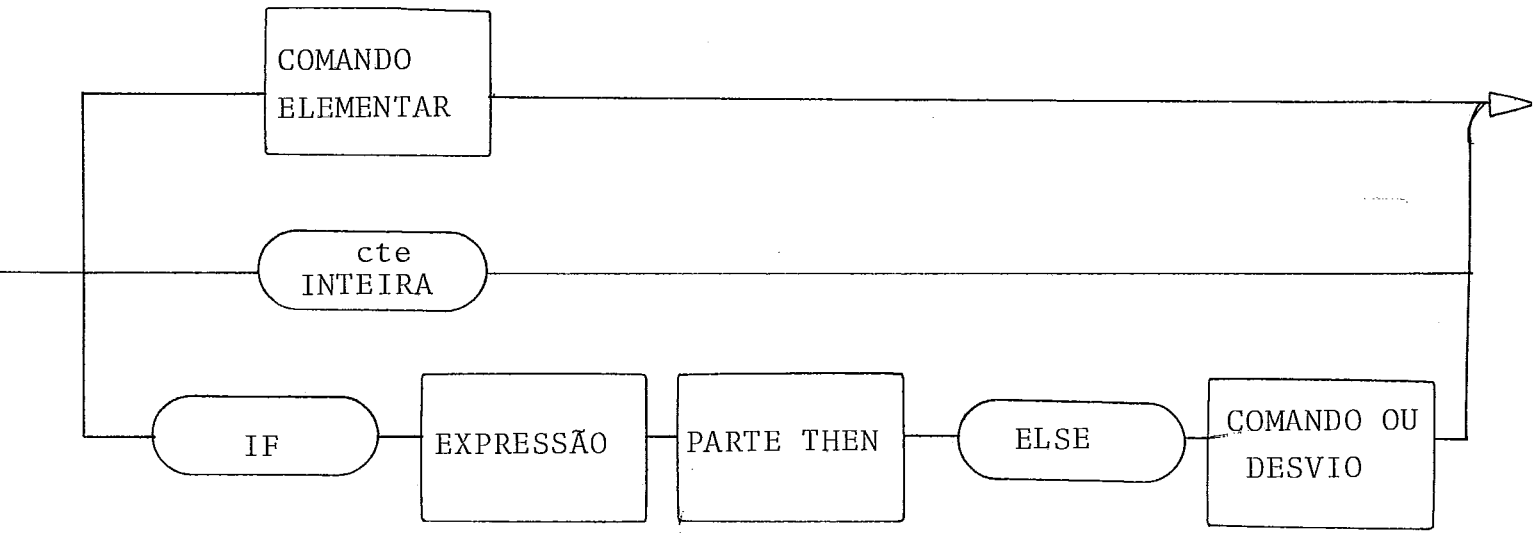
Parte-Then



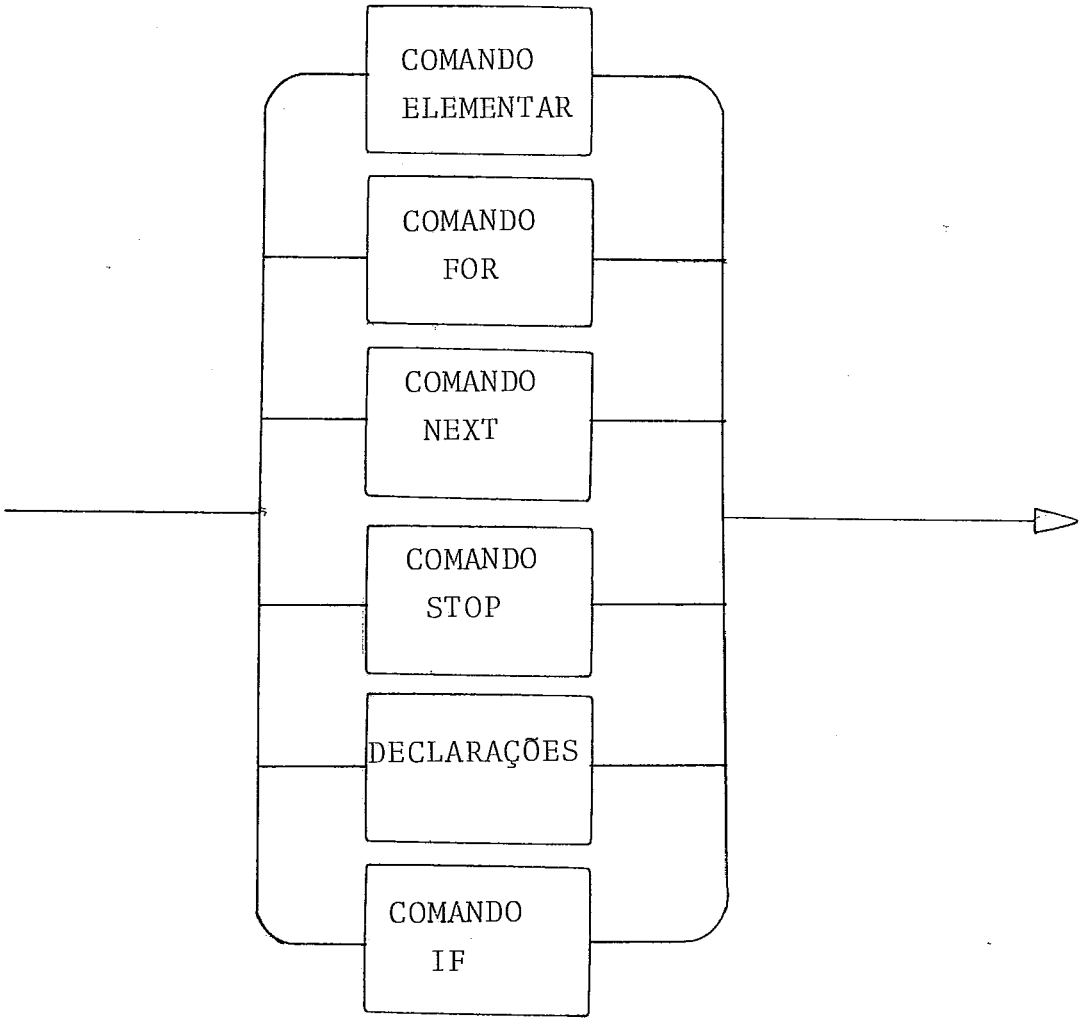
Parte-Else



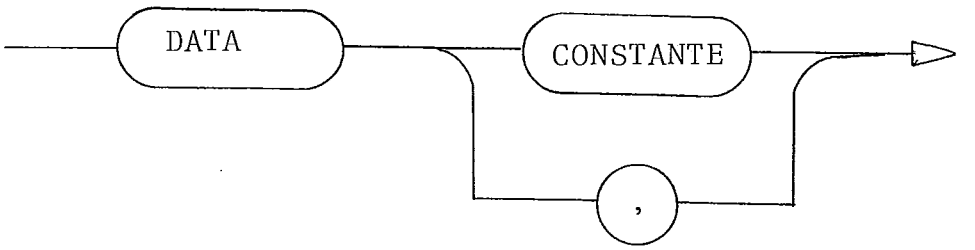
Comando ou Desvio



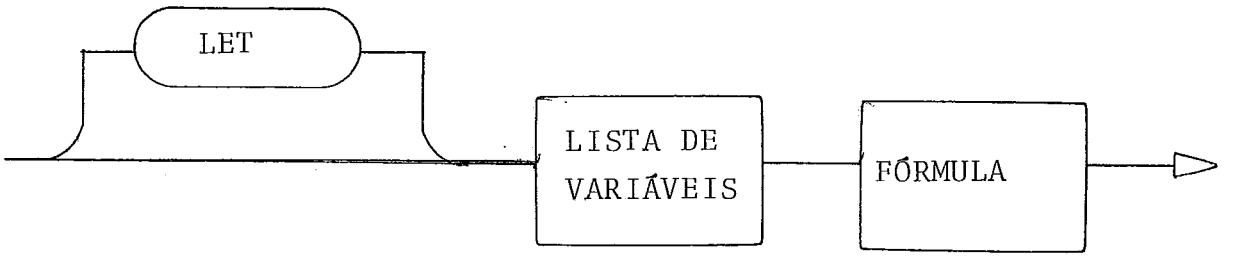
Incremento



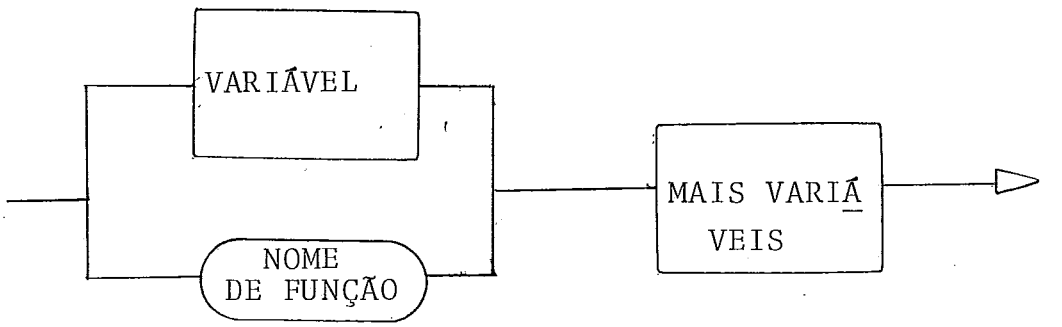
Incremento DATA



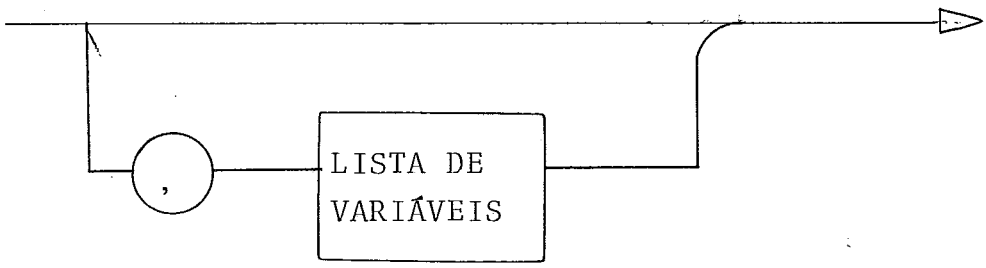
Atribuição



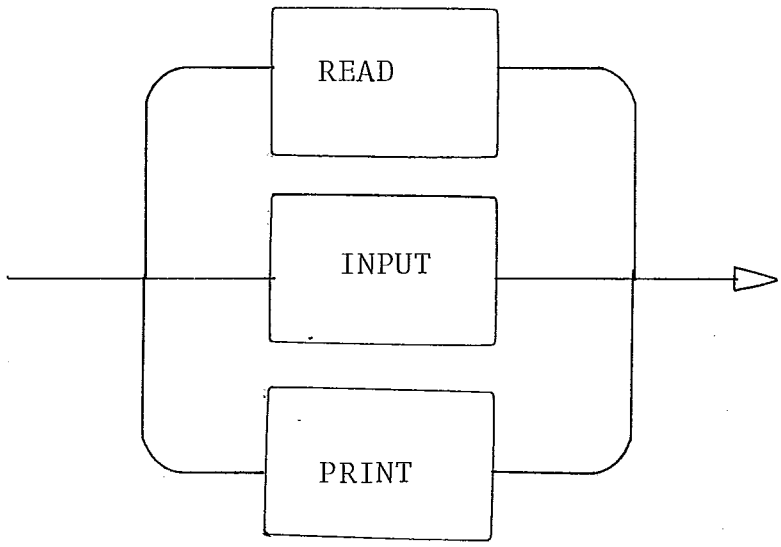
Lista de Variáveis



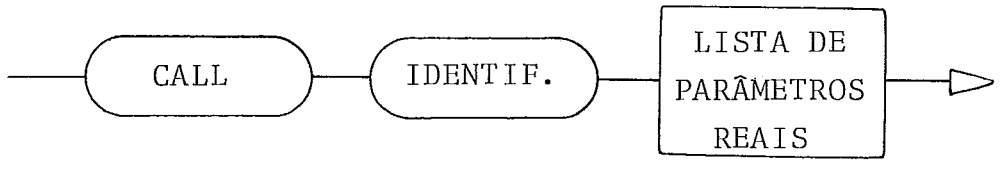
Mais Variáveis



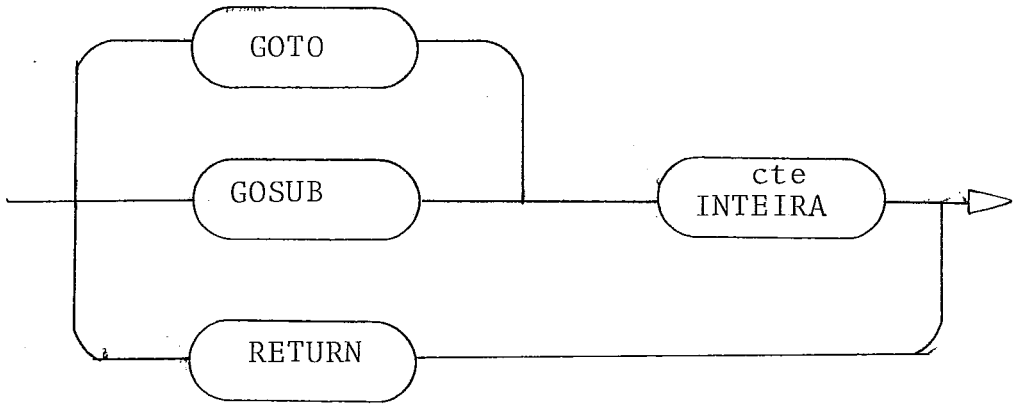
Comando de E/S



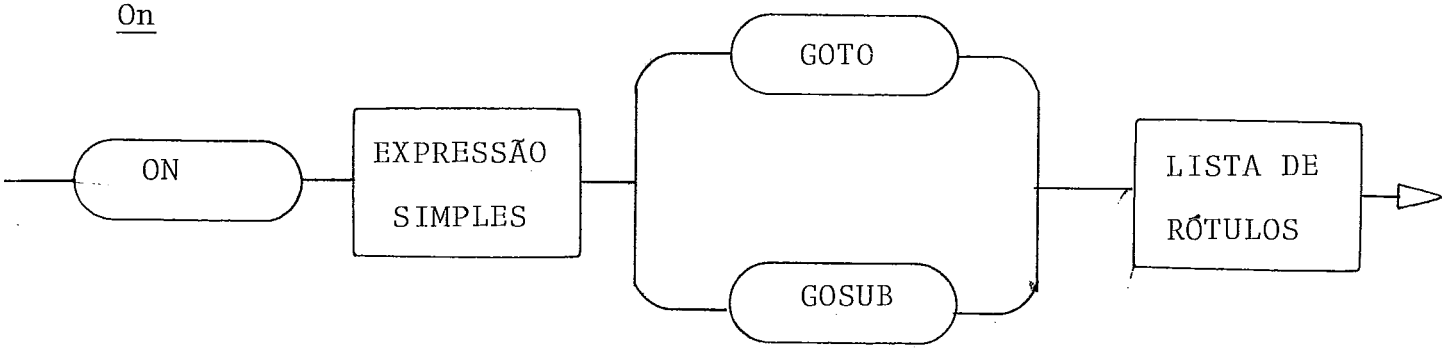
Call



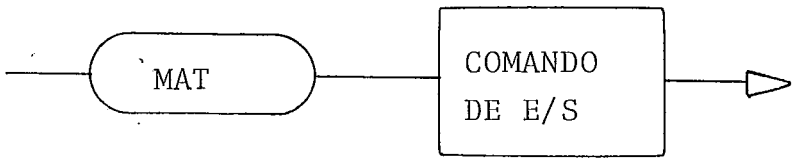
Desvio



On

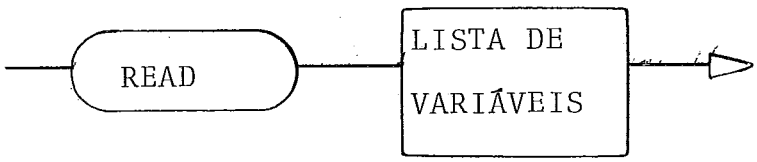


Mat

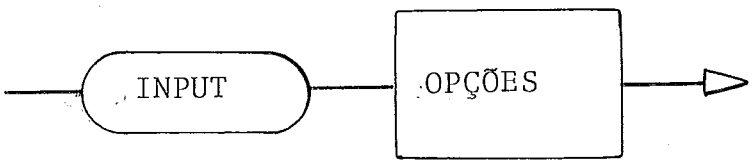


Restore

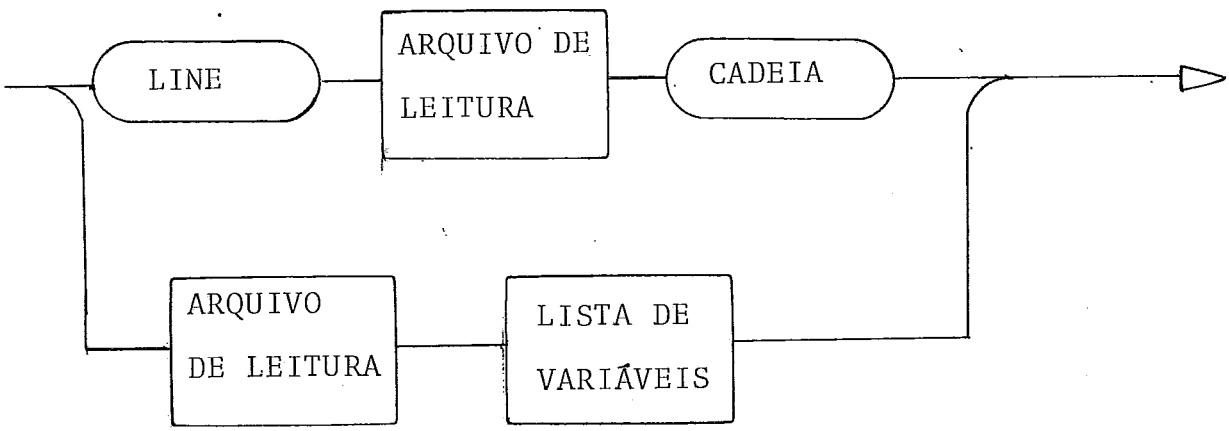
Read



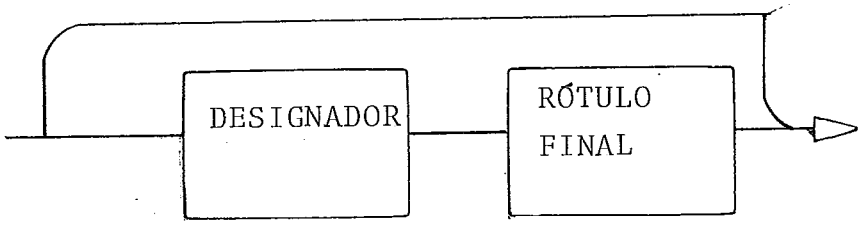
Input



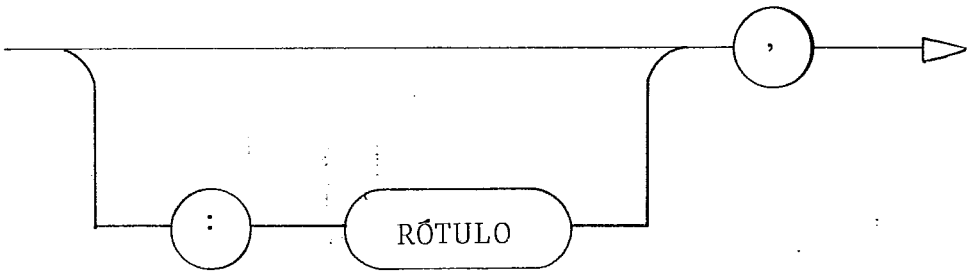
Opções



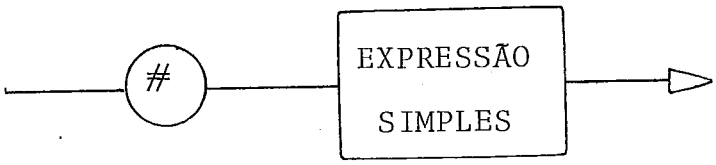
Arquivo de Leitura



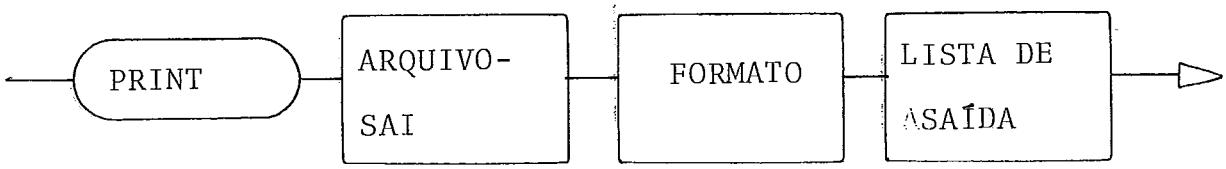
Rótulo Final



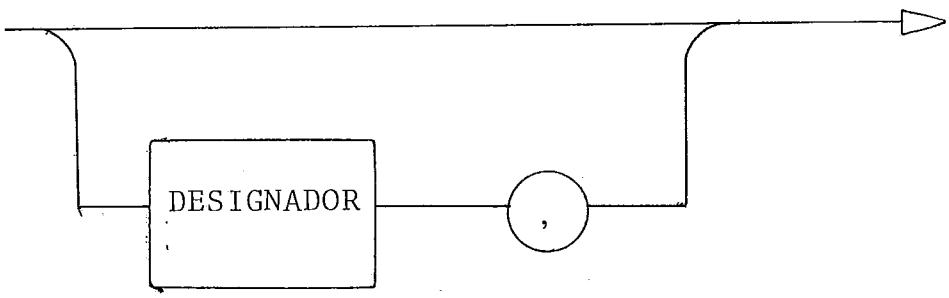
Designador



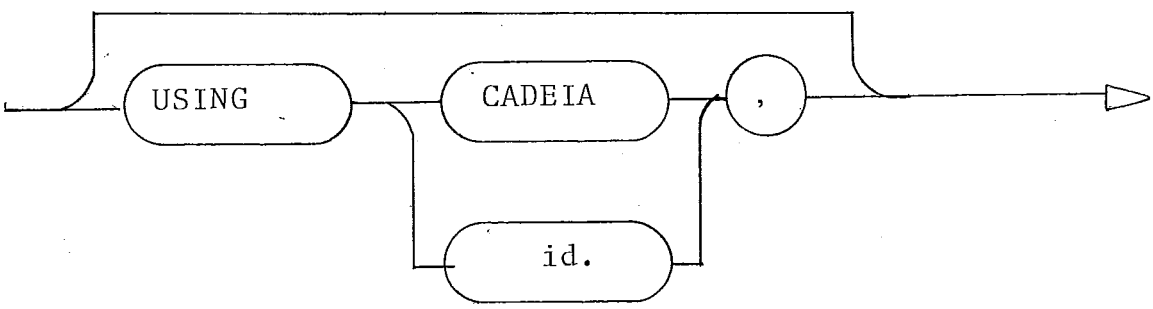
Print



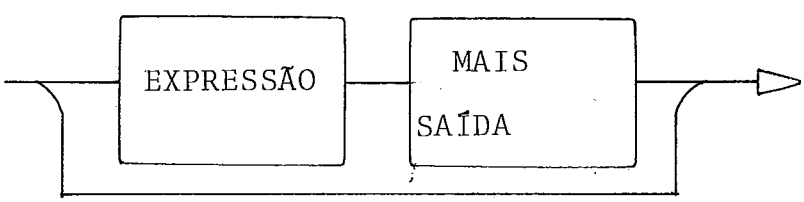
Arquivo-Sai



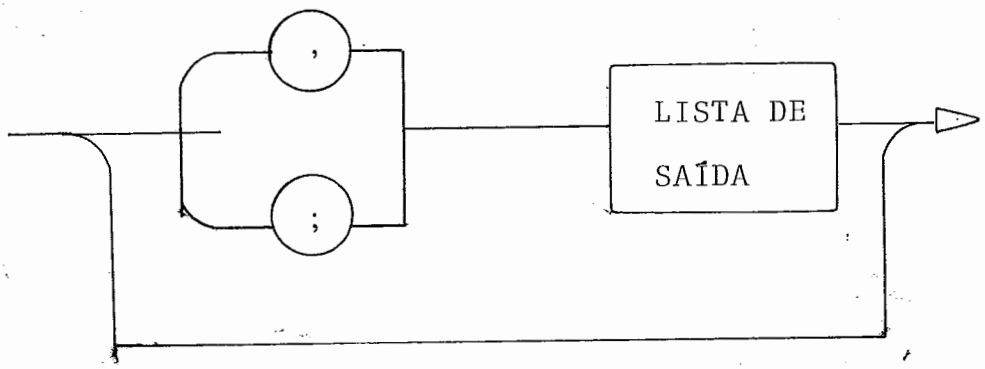
Formato



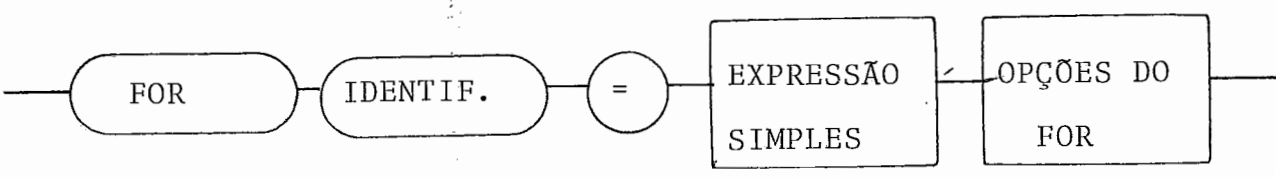
Lista de Saída



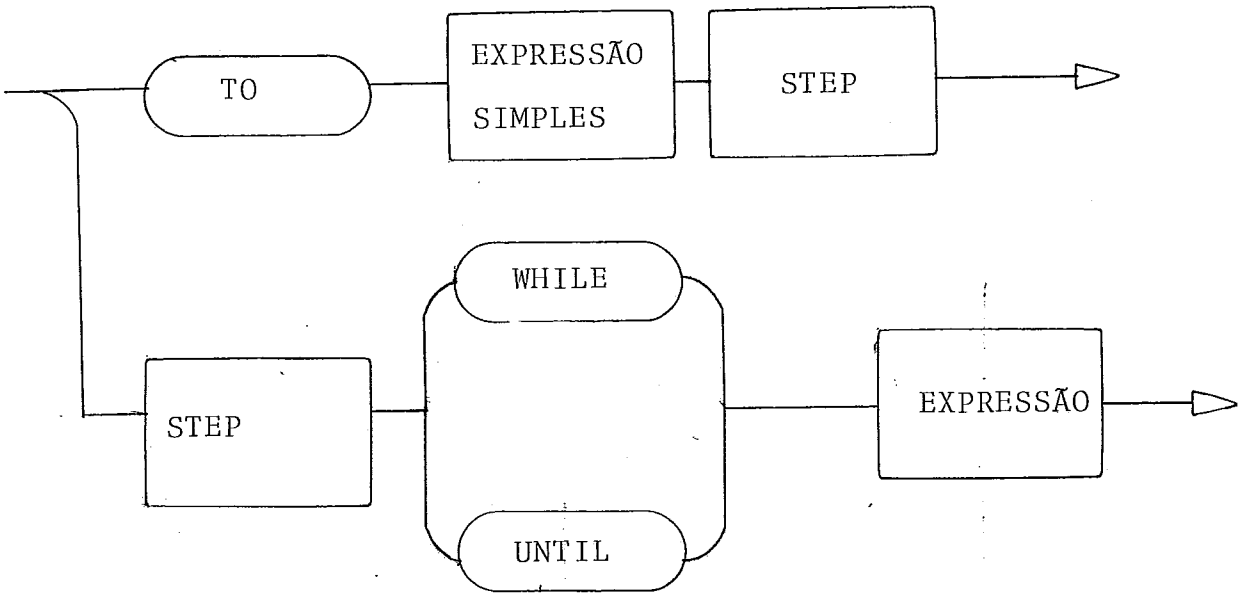
Mais Saída



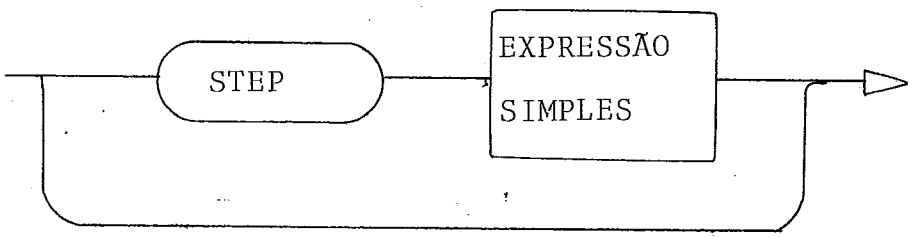
For



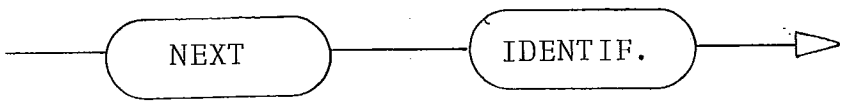
Opção do For



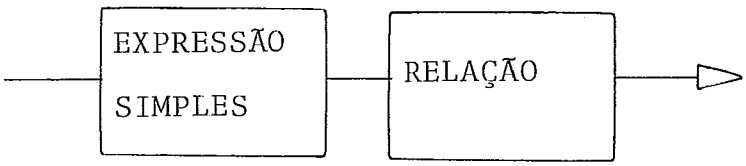
Step



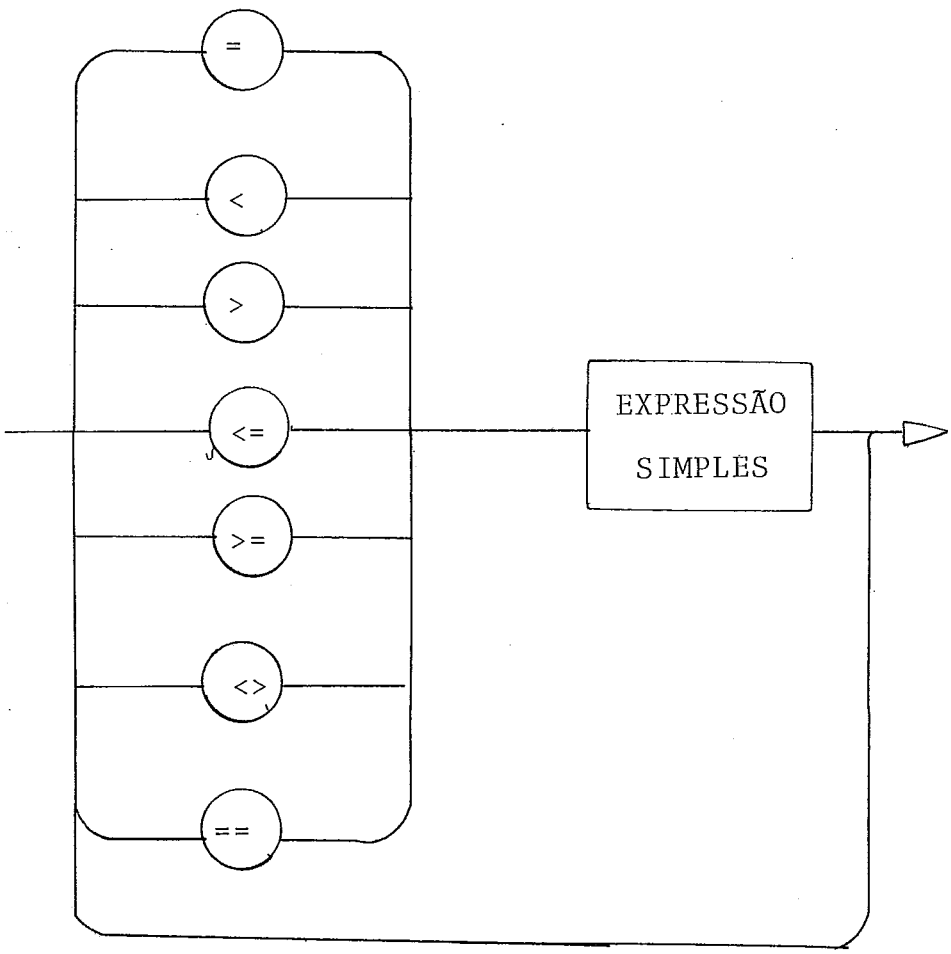
Next



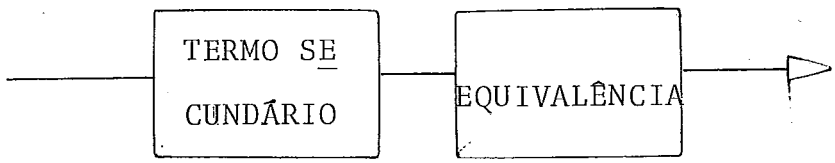
Expressão



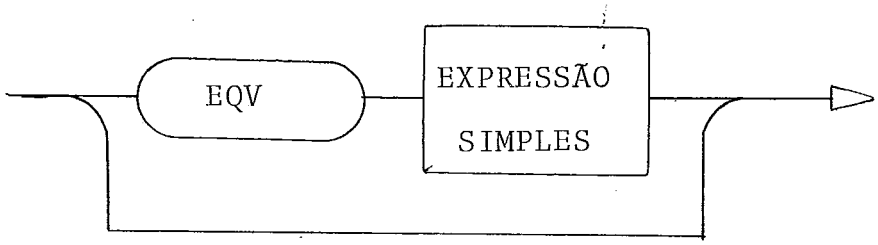
Relação



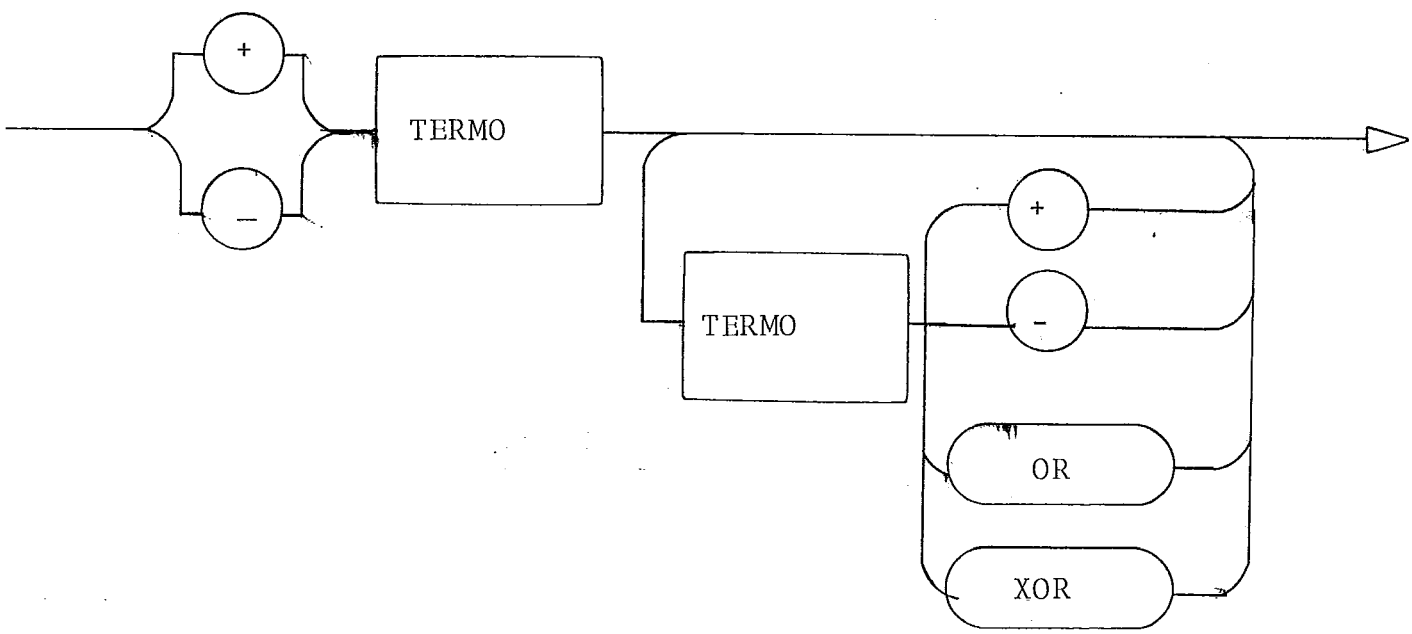
Expressão Simples



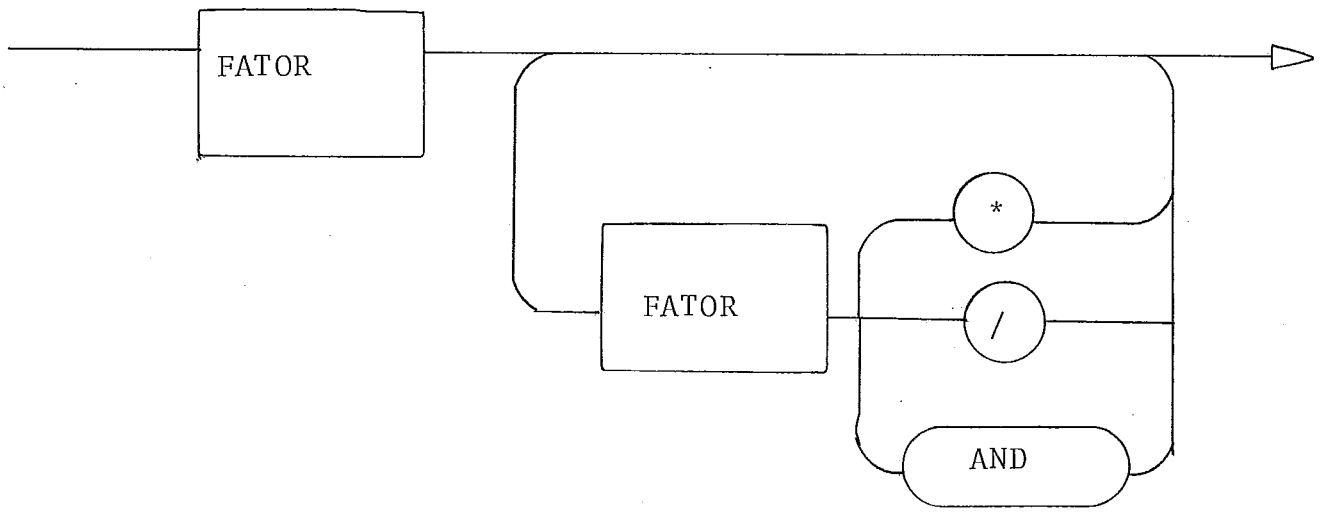
Equivalência



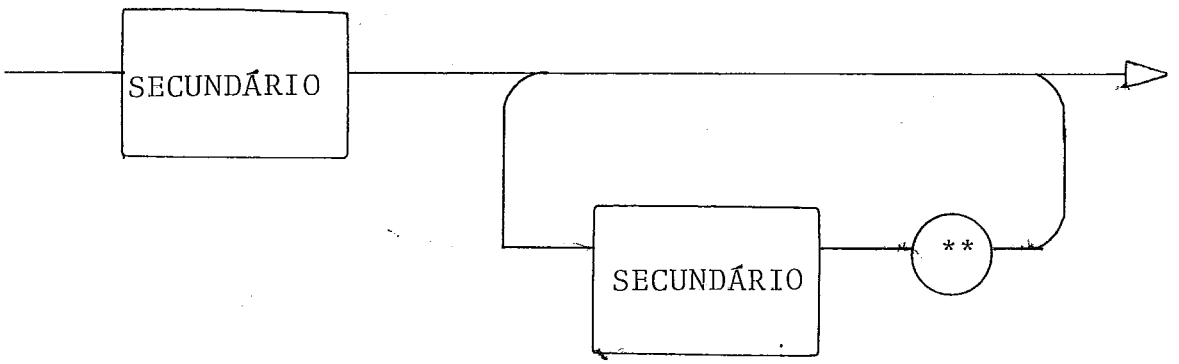
Termo Secundário



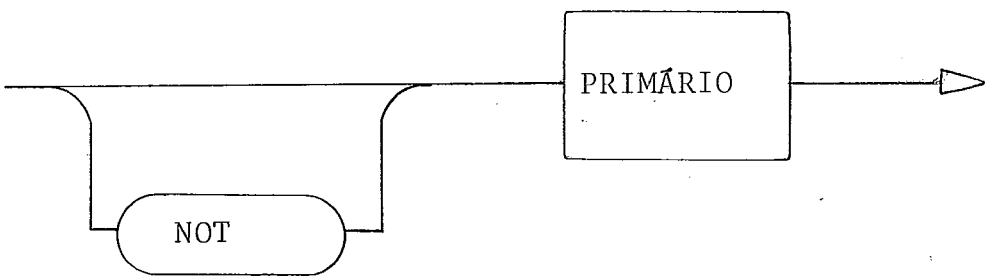
Termo



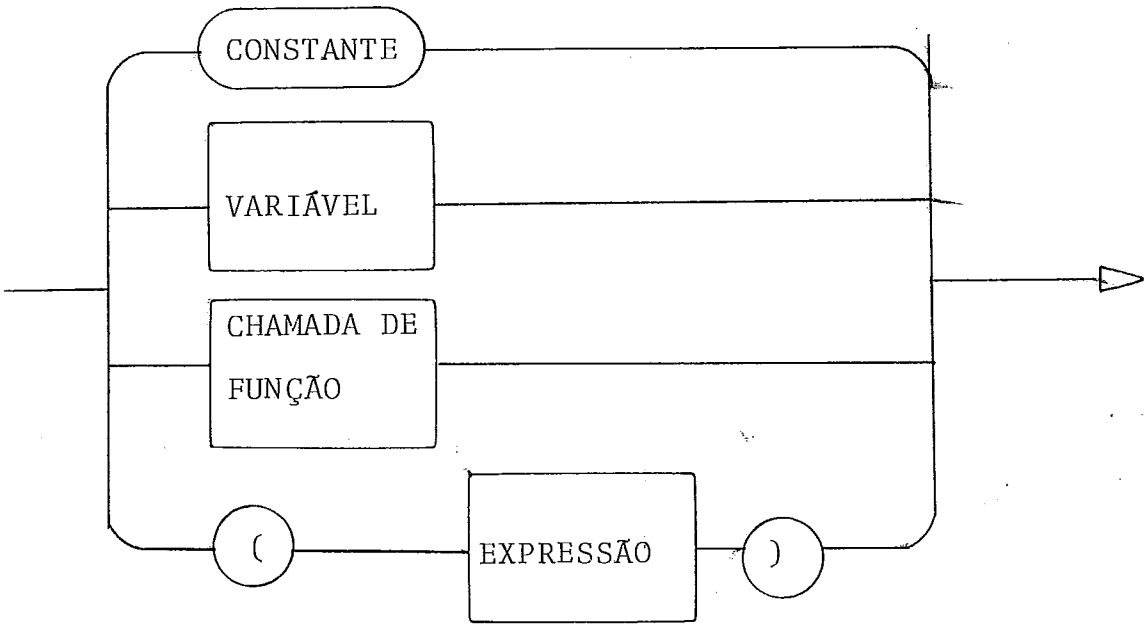
Fator



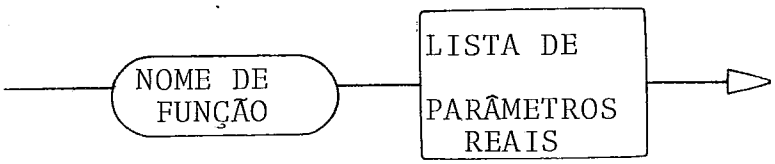
Secundário



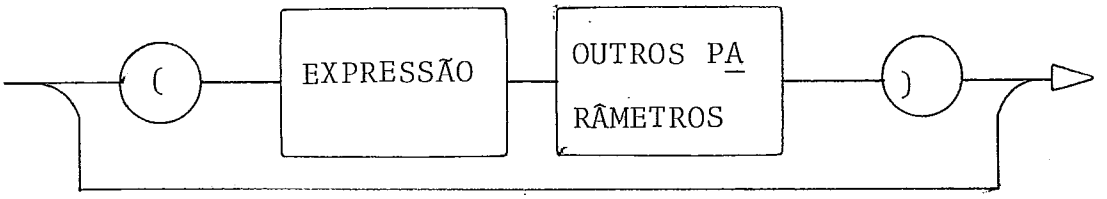
Primário



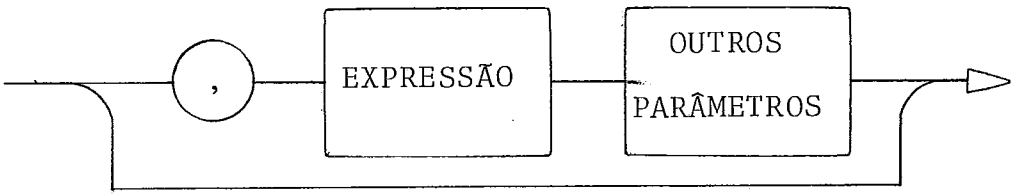
Chamada de Função



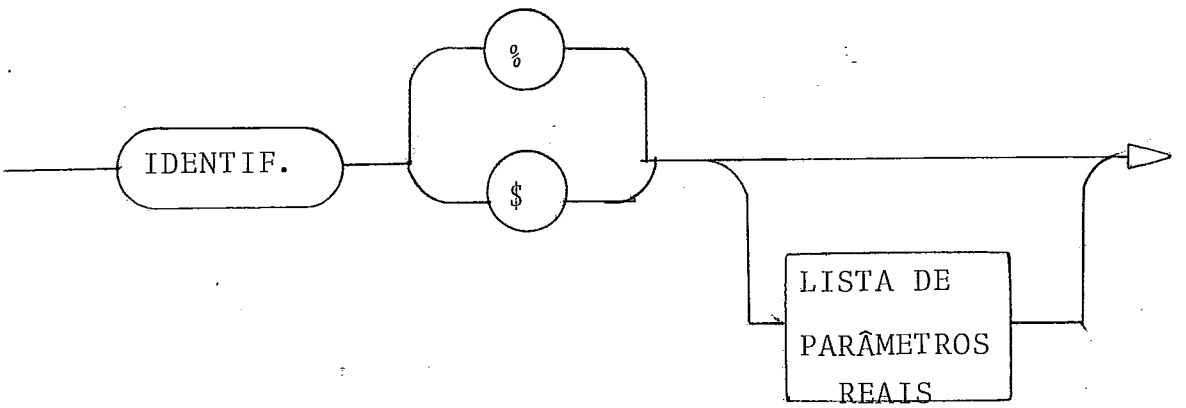
Lista de Parâmetros Reais



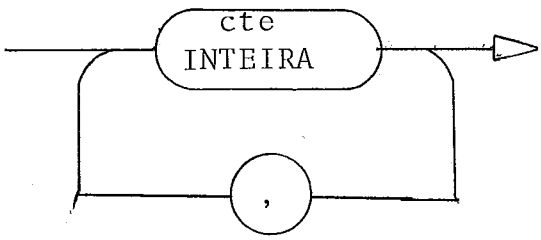
Outros Parâmetros



Variável



Lista de Rótulos



Para acelerar o processo de análise, a gramática está na forma LL(1), ou seja, o diagrama deve ser construído de tal modo que:

- cada ramo que saia de uma bifurcação aponte para símbolos iniciais distintos;

- todos os símbolos que vêm imediatamente após um comando devem ser diferentes dos símbolos iniciais desse comando.

Será necessário então conhecer os conjuntos de símbolos iniciais (first) e os que vêm imediatamente após (follow) para cada grafo. Esses conjuntos estão na tabela da figura (III.3.2).

TABELA FIRST x FOLLOW

NÃO-TERMINAIS	FIRST	FOLLOW
Linha BASIC	int	&
Conjunto de Incrementos	data|end|fnend|subend|dim|file def|sub|stop|for|next|let|id| fnid|read|print|input|call| goto|gosub|return|on|mat| restore|if	:|&
Incremento	dim|file|def|sub|stop|for| next|let|id|fnid|read|print| input|call|goto|gosub|restore| on|mat|return|if	:|&
Mais incremento	:|&	&
Incremento-Data	Data	&
Incremento-Final	end|fnend|suben	:|&
Com. Elem.	let|id|fnid|read|print|input| call|goto|gosub|return|on| mat|restore|	:|else|&
Comando FOR	for	:|&
Comando NEXT	next	:|&
Comando STOP	stop	:|&
Declarações	def|dim|file|sub	:|&
Atribuição	let|id|fnir	:|else|&
Comando de E/S	read|input|print	:|else|&
Desvio	goto|gosub|return	:|else|&
Call	Call	:|else|&
On	On	:|else|&

Fig. III.3.2

continua...

Continuação...

NÃO-TERMINAIS	FIRST	FOLLOW
Mat	Mat	:|else|&
Restore	restore	:|else|&
If	if	:|&
Dim	dim	:|&
Lista de V. Subs.	id	:|&
Índices	int|id	)
File	file	:|&
Def.de Arquivo	id	:|&
DEF	def	:|&
Lista de P. Form.	(|&	=|:|&
Param. Formal	,|&	)
Fórmula	=|&	:|&
Sub	Sub	:|&
Parte-Then	goto|then	:|else|&
Parte-Else	else|&	:|&
Com. ou Desvio	let|id|fnid|read|print|input call|goto|gosub|return|on|mat restore|int|if	:|else &
Lista de Var.	id fnid	=:else &
Mais Variáveis	, &	=:else &
Read	read	:else &
Input	input	:else &
Print	print	:else &
Lista de Rót.	int	:else &
Opções	line id fnid	:else &

Continuação...

NÃO-TERMINAIS	FIRST	FOLLOW
Arq. de Leit.	&	id fnid
Rótulo final	:,	id fnid
Designador		,:
Arquivo-Sai	&	using +- not cte id fnid (:else &
Formato	using &	:+- not cte id fnid (else &
Lista de saída	+- not cte if find (&	: else &
Mais Saída	,;&	: else &
For	for	: &
Opções do FOR	to step	: &
Step	step	while until:&
Next	next	:&
Expressão	+-not id fnid cte (	:else,;then goto)&
Relação	= = = == &	:,;)then goto else &
Expr.Simples	+- not cte id fnid (	=|<|>|<=|>=|<>| ==|:|, then goto else while until to step gosub
Equivalência	EQV &	=|<|>|<>|<=|>=| ==|:|,|;)|then| goto|else|while| until|to|step| gosub
Termo Secund.	+-(not cte id fnid	EQV|=|<|>|<=|>=| |<>|==|:|,|;|) then|goto|gosub| else|while|until to|step

continua...

Continuação...

NÃO-TERMINAIS	FIRST	FOLLOW
Termo	not cte id fnid (	=|<|<=|>|>=|<>| |==|:|,|;|)|+|-| OR|XOR|EQV|then| goto|gosub|else| while|until|to| step
Fator	not cte id fnid (	=|<|<=|>|>=|==| |:|,|;|)|+|-|*| /|AND|OR|XOR| EQV|then|goto| gosub|else|to| step|while|until
Exponenciação	** &	=|<|<=|>=|<>|== :,;)+-*/AND OR XOR EQV then goto gosub else while until to step
Secundário	not cte id fnid (	=|<|<=|>=|<>|== :,;)+-*/AND OR XOR EQV then to goto gosub else step while until
Primário	(cte id fnid	=|<|<=|>|>=|<>| ==:,;)+-*/AND OR XOR EQV then to goto gosub else step until while
Chamada de Função	fnid	=|<|<=|>|>=|<>| ==:,;)+-*/AND OR XOR EQV then to else goto while until step gosub
Lista de Par.R.	(&	=|<|<=|>|>=|<>| ==:;,)+-*/AND OR XOR then else EQV goto step while until to gosub

continua...

Continuação ...

NÃO-TERMINAIS	FIRST	FOLLOW
Outros parâmetros	,)&	)
Variável	id	=|<|<=|>|>=|<>|==:;,)+-*/AND OR XOR EQV then to else goto step while until gosub

Conforme o próprio nome indica, o método de análise utiliza uma série de rotinas recursivas. Como o PLTI não admite recursividade, o processo será simulado através do uso de duas pilhas:

Pilha de Rotinas

Conterá os símbolos não-terminais da gramática, que correspondem a chamadas de rotinas. Caberá à rotina principal do compilador BASIC a chamada da rotina que se encontra no topo dessa pilha. Quando se tornar vazia, significa que acabou a análise da linha.

Pilha Auxiliar

Conterá o valor de variáveis locais que necessitem ser restaurados quando for feito o retorno à rotina. Será utilizado para fins semânticos, principalmente no que se refere à análise de expressões.

Para maiores detalhes sobre esse método de descida recursiva podem ser consultados Gries[3], Aho[4], Bauer[5] e Wirth[6].

III.4 - Análise Semântica e Geração de Código

A análise semântica está embutida nas rotinas que executam a análise sintática e tem como tarefas:

(a) Verificação e conversão de tipos de dados

Ex.: 1Ø for A$=1 to N$. → uso inválido de variável alfanumérica

15 X = B% * Y → necessária conversão

(b) Tratamento de "defaults"

Ex.: 1ØØ for I% = 1 to N% → o compilador admite que o parâmetro step seja omitido quando o valor do incremento é 1.

(c) Criar procedimentos para resolução de referências

Ex.: 1Ø if A = B goto 5Ø

→ supor que a linha 50 ainda não apareceu

2Ø X=X+Y : goto 5Ø

Em seguida será visto o processo de análise e geração de código para algumas das construções mais importantes da linguagem.

III.4.1 - Análise de Declarações

O aparecimento de uma declaração faz com que a Tabela de Símbolos seja acessada para os seguintes procedimentos:

(i) verificar se o símbolo não foi definido anteriormente;

(ii) completar os atributos do símbolo.

Afora isso, é necessário gerar um desvio para o próximo incremento, pois é permitida a seguinte construção:

```
1Ø dim A(3,3), X$(18)
2Ø let B = B + 1
    ⋮
8Ø goto 1Ø
```

III.4.2 - Comando Condicional

Os procedimentos semânticos relativos a um if são os seguintes:

i) deve ter verificado se a expressão que aparece após a palavra IF é do tipo lógico ou condicional. Se for deste último tipo, o resultado da expressão será o valor Ø (falso) ou -1 (verdadeiro). No primeiro caso, o resultado será um valor inteiro; se esse valor for diferente de zero, equivale a resultado verdadeiro.

ii) em seguida deveria ser verificada a validade dos comandos que aparecem logo após o then ou else, mas tal será feito pela própria análise sintática, pois pode ser obtido pelo diagrama sintático.

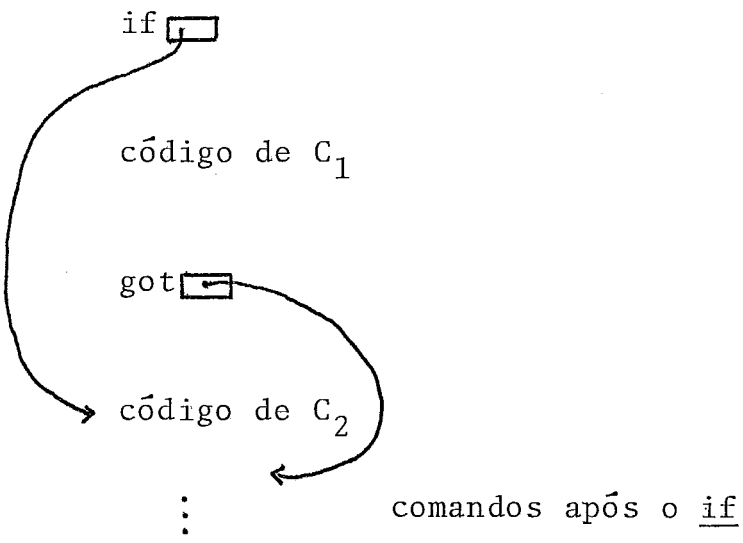
iii) finalmente restam os desvios: se a expressão for verdadeira deve ser executada a parte then e em seguida desviar para o próximo comando. Caso contrário, a parte then deve ser saltada e executam-se os comandos relativos à parte else.

um comando do tipo:

if E_1 then C_1 else C_2

corresponderá ao seguinte código:

avaliação de E_1



onde C_1 e C_2 podem ser outros if-then-else.

Utilizando o fato de que o código objeto pertence a uma lista, o desvio para o comando após o if corresponde a desviar para o endereço contido no campo: endereço do próximo incremento. Basta então carregar esse valor na pilha do sistema e ao

invés de "got" será gerado um "ret", pois conforme pode ser visto na descrição do código no apêndice, esse comando atualiza o contador de programas com o valor que está no topo da pilha.

Para a resolução do desvio quando a expressão é falsa (representado no código por if ⊏→) será utilizada uma outra pilha, que chamaremos de Pilha Auxiliar, já mencionada. Nela serão guardados os endereços onde se encontram esses if.

Cabe aqui lembrar que o código gerado para o incremento fica em uma área de memória, na verdade um vetor, chamado aqui de TAB$COD. O endereço de um if corresponde portanto à posição onde ele foi inserido no TAB$COD.

Quando for encontrado um else, o endereço no topo da pilha auxiliar é retirado e a posição correspondente no vetor é preenchido com o endereço onde começa o código relativo ao else.

Esse último endereço é dado pelo valor de PC, que indica onde será guardada a próxima instrução.

Se ao fim da análise do incremento a pilha auxiliar não está vazia, significa que a parte - else de um if mais externo não existe. Nesse caso, devem ser geradas as instruções que desviam para o próximo incremento e guardar seu endereço na posição de TAB$COD indicada no topo da Pilha Auxiliar.

III.4.3 - Referências Futuras

Na geração de código, quando é feita uma referência a uma função, subrotina ou o desvio para algum incremento, deve-se ter, entre outras informações, o endereço de início da rotina ou função ou da linha para onde é feito o desvio.

Contudo, o usuário poderá fazer uma ou mais das referências descritas acima, sem ter ainda definido o símbolo.

A resolução disso se dará "alinhavando-se" todas as instruções que se refiram a um determinado símbolo não definido, sendo que o começo da lista deve ficar, no caso de funções e subrotinas, no campo "segmento de definição", na Tabela de Símbolos. Este campo só é atualizado quando surgir a declaração DEF ou SUB. Aí então essa lista de alinhavos é percorrida e os endereços vão sendo resolvidos.

No caso de desvio para outra linha, será utilizado o campo "apontador para referências", no dicionário de incrementos.

O alinhavo entre as referências é feito do seguinte modo:

i) o endereço da primeira referência é inserido na Tabela de Símbolos ou no Dicionário de Incrementos, conforme o caso;

ii) caso surjam outras referências, o código será completado com o endereço da referência precedente, e a entrada mencionada em i) é atualizada com o endereço da referência atual.

Uma outra espécie de referência futura, aqui empregado no sentido de ser uma informação que não pode ser completada logo que o símbolo apareça, diz respeito às variáveis subscritas. Tal poderá ser compreendido pelo exemplo seguinte:

ex.: 15Ø mat print A

no aparecimento dessa linha, sabe-se que A deverá ser uma variável subscrita, pois é usada em um comando mat, mas não se tem mais nenhuma informação sobre ela.

Será criada então uma "lista de símbolos indefinidos" contendo ponteiros para a Tabela de Símbolos, indicando as variáveis subscritas e funções/subrotinas indefinidas.

Todas as entradas referentes a matrizes serão resolvidas no Passo 2. As que se refiram a funções/subrotinas só são retiradas da lista quando surgirem as respectivas definições.

Quando for dado um EXEC, a rotina de Carga de Programas verifica essa lista. Se não estiver vazia, os símbolos indefinidos são mostrados ao usuário, e a execução não é inicializada.

III.4.4 - Análise de Expressões

Do mesmo modo que os comandos, as expressões serão analisadas pelo método preditivo. A relação de prioridade entre os operadores é dada pela própria estrutura sintática da linguagem, como pode ser visto no diagrama.

O interpretador das micro-rotinas do PLTI se utiliza de uma pilha, que será chamada aqui de pilha do sistema, para diferençar das outras estruturas utilizadas nesse compilador. As operações aritméticas, lógicas e de comparação são executadas entre os elementos que estão no topo da pilha do sistema. O compilador deve re-sequenciar as expressões de modo a colocá-las na forma polonesa pósfixa. Assim os operadores vêm após os operandos que lhes correspondam, em vez de ficarem embutidos no meio da expressão como na forma usual de escrita, chamada por isso de forma infixa. Esse formato é também conhecido como independente de parênteses (parentheses free), pois estes se tornam supérfluos nesse tipo de notação. Os exemplos abaixo mostram a relação entre a forma infixa e a polonesa pósfixa:

forma infixa	forma pósfixa
(x-y)+z	xy-z+
x-(y+z)	xyz+-
x*(y+z)*w	xyz+*w*

As operações são executadas pelas midro-rotinas neste formato, ou seja, primeiro carregam-se os operandos na pi-

lha do sistema e em seguida a operação a ser executada. Por exemplo, o código gerado para a expressão (supondo que todos os operandos são inteiros): a+b/c*d seria

```
la a    carrega na pilha do sistema o valor de a
la b    carrega na pilha do sistema o valor de b
la c
div     saem b e c, no topo da pilha fica s2=b/c
la d    carrega d na pilha do sistema
mul     saem s2 e d, no topo da pilha fica s1 = s2*s
add     saem s1 e a, no topo da pilha fica so = a+s1
```

Nota-se então que tal código corresponderá à forma pósfixa: abc/d*+.

Para essa transformação, será utilizada a Pilha Auxiliar, que conterá os operadores e informações sobre os operandos que constituem a expressão. Sob o termo "operando", deseja-se designar as variáveis, ctes., funções ou o resultado de alguma operação. Os operandos portanto podem ser de 3 tipos: inteiros, reais ou alfa. O seu aparecimento faz com que seja gerado o código que o carregue na pilha do sistema: caso seja do tipo inteiro, é carregado o seu valor; senão, carrega-se o seu endereço. Na pilha auxiliar é deixada somente uma informação sobre o tipo do operando, para que, no momento em que for gerado o código da operação, sejam feitos os seguintes testes semânticos:

- verificar se o tipo dos operandos é compatível com a operação. Por exemplo, a expressão
  'MINHA CASA' ** 'CAIU'
  é inválida

- verificar se é necessário haver conversão de tipos. Por exemplo, na expressão
  A% + B
  o operando A% deve ser convertido para real antes de ser executada a soma. O código correspondente seria:

```
la A%        carrega na pilha do sistema o valor de A%
call float   chama rotina que substitui o topo da pilha do sistema pelo seu conteúdo, na forma real
lia B        carrega na pilha do sistema o endereço de B
call fadd    soma os dois valores reais.
```

Os operadores podem ser aritméticos, lógicos, relacionais ou especiais, estes últimos criados para fins semânticos, serão explicados mais adiante.

Um operador é inserido na Pilha Auxiliar no momento em que é reconhecido pelo analisador sintático. Ou seja, no momento em que, na geração da árvore sintática, foi expandido um nó que deriva diretamente esse operador, entre outras símbolos. Além disso a rotina que fará os procedimentos semânticos relativos a esse operador é inserida na Pilha de Rotinas. Quando essa rotina for executada, o operador é retirado da Pilha Auxiliar e gera-se o código da operação, efetuando-se os procedimentos semânticos já mencionados.

Os operadores especiais são: dim, def e parm. Sua inserção na Pilha Auxiliar se dará nos seguintes casos:

- dim: foi encontrada uma variável subscrita na sequência de entrada

- def: foi encontrada uma função definida pelo usuário

- parm: foi encontrada a ",", separadora de índices/parâmetros.

A retirada desses operandos ocasionará, em linhas gerais, os seguintes procedimentos:

i) dim: geração do código que carregue na pilha do sistema o número de subscritos e chame a rotina que verificará se o uso da variável está correto.

ii) def: geração do código que carregue na pilha do sistema: endereço de retorno, valor de cada parâmetro, tipo e número de parâmetros e o endereço do descritor da função. Em seguida é chamada a rotina que verifique se o uso é correto e deixa na pilha do sistema o endereço de início da função.

O código gerado para uma chamada do tipo:

$f_n\ XY(p_1, p_2, \ldots, p_n)$

seria:

lia endereço de retorno

lia endereço do descritor

ξ → carrega valor de p_1

⋮

ξ → carrega o valor de p_n

la tipo de p_1

⋮

la tipo de p_n

lib n → número de parâmetros

call PARAM → chama rotina de teste de uso

ret → desvia para o endereço que está no topo da pilha que pode ser: início da função ou da rotina de erro.

iii) parm: soma 1 a um contador de índices/parâmetros. Esse contador é zerado ao final dos procedimentos relativos a um def ou dim.

III.4.5 - Comando FOR

Há duas formas de comando iterativo:

for V = e_1 to e_2 step e_3

for V = e_1 step e_2 {while / until} C

onde: V - indica a variável de controle, que pode ser do tipo inteiro ou real, não subscrita

e_1, e_2, e_3 - expressões artiméticas

C - condição.

O término da malha desse comando é dado por: next V.

Por ser um compilador incremental, no qual o usuário tem a facilidade de fornecer as linhas em qualquer ordem, a ligação entre o for e o next será feita, em tempo de execução, através de um descritor.

Para a execução desse comando, serão necessárias as seguintes rotinas:

FORIN → inicialização do for. Carrega na pilha de blocos o descritor. Deixa na pilha do sistema: o valor inicial, o valor final, o valor do incremento e o endereço da variável de controle.

FORFIM → término do for. É chamada quando for atingida a condição/valor final ou quando é encontrado um desvio para fora da malha. Tira o descritor da pilha de blocos. Tira da pilha do sistema: o valor atual da variável de controle, o valor final e o valor do incremento.

ITEST → no caso da forma for-to, com variável de controle inteira, teste se foi atingido o valor final. Se for esse o caso, chama FORFIM e desvia para o endereço de escape.

RTEST → análoga a ITEST quando a variável de controle é real.

IEND → chamada quando é executado o next. Verifica-se a variável de controle utilizada nesse comando coincide com a do descritor que está no topo da pilha de blocos. A variável de controle, do tipo inteiro, é incrementada e em seguida é feito um desvio para a rotina de teste.

REND → análoga a IEND, quando a variável de controle é real.

O código gerado para cada formato pode ser descrito em linhas gerais do seguinte modo:

i) for V = e_1 to e_2 step e_3

ξe_1 valor atual da variável de controle

ξe_2 valor final

ξe_3 incremento

lia endereço do descritor

call FORIN

TEST call ITEST (ou RTEST)

ξ bloco-for

ii) for V = e_1 step e_3 while C

ξe_1

lib $\emptyset$ → a posição na pilha do sistema correspondente ao valor final é zerada

ξe_3

lia a descritor

call FORIN

TEST

ξ C

<u>if</u> <u>FORFIM</u> → desvia para a rotina de FIM se a condição for falsa

ξ bloco-for

iii) <u>for</u> V = C_1 <u>step</u> e_3 <u>until</u> C

Análoga a (ii), só que a rotina de TESTE fica assim:

TESTE

ξ C

Not

<u>if</u> FORFIM → desvia para a rotina de fim se a condição é verdadeira

iv) <u>next</u> V

lia V

call IEND (ou REND)

O procedimento semântico consistirá somente em verificar se a variável de controle não é subscrita e do tipo numérico (inteiro ou real), bem como as expressões que dão o valor inicial, o final e o tipo.

III.4.6 - Procedimentos de Entrada e Saída

Os comandos de entrada e saída são de dois tipos:

- comandos de acesso à Tabela DATA: READ, RESTORE

- comandos de acesso a dispositivos: INPUT, PRINT

No primeiro caso, os dados a serem lidos se encontram na área de dados do segmento.

Na execução de um comando READ, será verificado:

a) se a lista de variáveis não esgota a tabela;

b) se o tipo de dado lido está de acordo com a variável que consta da lista.

O apontador para a Tabela DATA é atualizado, de modo a indicar a próxima posição a ser lida por outro comando READ.

Para o comando RESTORE, bastará utilizar o apontador com o endereço de início dessa tabela, que é a posição seguinte à sua na área de dados.

Para a execução do segundo tipo de comando, serão utilizadas as rotinas de E/S, descritas a seguir:

Rotina de Acesso à Tabela de Arquivos

A Tabela de Arquivos será criada pela rotina de Carga de Programas e conterá uma entrada para cada arquivo definido pelo usuário. Para uniformizar o tratamento dos comandos de E/S, será colocada aí também uma entrada referente ao terminal, que é o dispositivo básico de E/S.

Cada entrada dessa tabela terá os seguintes dados:

- valor do designador de arquivo: no caso do terminal, esse valor é zero
- código do periférico
- tamanho do registro
- endereço do arquivo no disco
- endereço do próximo registro a ser lido/gravado
- endereço da área que conterá o registro ("buffer" de E/S).

O acesso à tabela é sequencial, utilizando o valor do designador de arquivo. Ela estará organizada ascendentemente em relação a este valor. Imediatamente antes da tabela será alocado o apontador que irá percorrre-la.

Essa rotina terá como tarefas principais:

- verificar se a expressão que aparece no comando de E/S se refere a um designador válido, i.e., que exista nessa tabela;

- deixa na pilha do sistema as informações contidas na tabela.

Rotinas de Formatação

Preparam os dados a serem lidos/gravados pelos comandos de E/S. Há dois tipos de formatos: o padrão e o do usuário.

Caso o usuário não tenha fornecido uma cadeia de formato, é utilizado o padrão.

Rotina de Entrada

Executa as seguintes tarefas:

1) conforme o código do periférico, chama a micro-rotina de leitura apropriada;

2) atualiza o campo de endereço do próximo registro, na Tabela de Arquivos;

3) caso tenha sido encontrado o fim de arquivo, desvia para a rotina com os procedimentos específicos, que pode ter sido definida pelo usuário ou é a rotina de erro;

4) chama a rotina de formatação, para ler os dados que foram guardados na área de E/S desse dispositivo;

5) guarda os dados lidos na variável correspondente.

Rotina de Saída

Executa as seguintes tarefas:

1) chama a rotina de formatação, para colocar os dados na área de E/S associada ao dispositivo;

2) se essa área está cheia é acusado erro, pois significa que o usuário deseja gravar um registro maior do que o tamanho que foi definido;

3) verifica se o arquivo não foi esgotado, i.e., se não está sendo gravado um registro além do fim do arquivo;

4) aciona a micro-rotina responsável pela operação da saída para o periférico em questão.

Antes de acionar estas rotinas, a pilha do sistema é preenchida com as seguintes informações:

- valor do designador de arquivo;

- endereço da rotina de fim de arquivo;

- endereço da Tabela de Arquivos;

- lista de E/S: contem o endereço (ou valor) de cada elemento da lista de variáveis, o tipo desses elementos e a tabulação (posição da área de E/S de ou para onde o elemento será lido ou gravado;

- tamanho da lista de E/S.

Os descritores e a Pilha de Blocos serão vistos em detalhes nos capítulos seguintes.

IV. EXECUÇÃO

Um programa entrará nessa fase quando o usuário fornecer o comando EXEC e for determinado que não foram detetados erros nem na compilação nem na fase de carga do programa.

Essa fase será controlada pelo Módulo de Controle da Execução que terá como tarefas principais:

a) detetar o maior número possível de erros que possam vir a ocorrer nessa fase;

b) informar ao usuário qual o erro encontrado e onde ele ocorreu, de modo a facilitá-lo na depuração do programa.

A tarefa (a) fará com que a execução se torne um tanto lenta, devido ao grande número de testes que deverá ser feito. Mas como essa implementação se destina a usuários que estejam aprendendo computação, o mais importante aqui é a deteção de erros, que serão muito frequentes.

Nos capítulos que se seguem será abordado com detalhes a formação da área de dados do programa e os tratamentos de erros, as interrupções e alguns procedimentos que não foram abordados no capítulo referente à geração de código.

IV.1 - Tratamento de Erros

Os erros a serem detetados nessa fase são de dois tipos:

(i) Semânticos

São os erros que não puderam ser detetados na fase de compilação, devido à estrutura incremental do compilador. Esses erros se referem ao uso incorreto de uma função, subrotina ou variável subscrita. A comparação entre a declaração e o uso desse tipo de dados não é feita na fase de compilação, porque o usuário não precisa fornecer a declaração antes do uso, e também, editando o programa, o usuário poderá alterar algumas declarações.

(ii) De execução

São erros do tipo:

- "overflow" de valor numérico
- índice inválido em variável subscrita
- seleção incorreta em comando on
- cadeia alfanumérica muito longa
- tabela DATA esgotada

- uso de recursividade

- formato inválido

entre outros.

Como a área de dados é sempre inicializada com zeros ou brancos, não ocorrerá erro do tipo: uso de variável para a qual não foi atribuído nenhum valor.

A mensagem do erro que ocorra nessa fase deve conter a identificação do segmento e o número da linha que estava sendo executada quando ocorreu o erro.

Após ser detetado um erro, o programa é abortado, para evitar que venham a ocorrer outros erros em decorrência deste, podendo inclusive destruir algum dado do usuário, que será importante caso ele queira utilizar comandos imediatos para depurar o programa.

Alguns erros de execução são detetados pelo próprio equipamento, como por exemplo: "overflow", acesso a um arquivo inexistente, erro de acesso a um arquivo. Nesses casos, o procedimento relativo ao erro seria tomado pelo SOCO, o que teria duas desvantagens:

(i) a mensagem seria emitida pelo SOCO, e não pelo compilador, o que confundiria o usuário, de vez que essa mensagem não terá certamente o formato daquela que seria emitida pelo compilador;

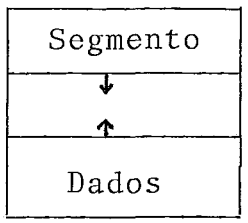
(ii) o compilador perderia o controle da execução, fazendo com que o programa do usuário seja também perdido.

Para evitar esses problemas, o Módulo de Controle da Execução interceptará esses erros, evitando que o SOCO tome conhecimento deles.

IV.2 - Alocação de Variáveis

Neste capítulo o armazenamento dos diferentes tipos de dados, bem como a maneira como eles serão referenciados na codificação do programa serão abordados.

A área de dados de um segmento crescerá em sentido inverso ao código de seus incrementos, como mostra a figura:



O fim do espaço livre é determinado quando as duas áreas se encontrarem. Nesse caso, o usuário deve utilizar o comando REORG, pois provavelmente haverá espaços vazios embutidos na área de programas.

A área de dados por sua vez tem duas partes distintas:

- uma estática, que contem as variáveis e constantes de tipo numérico;

- outra dinâmica, que conterá as constantes alfanuméricas.

A alocação de variáveis simples e constantes se dará no primeiro passo da compilação, bem como os descritores de variáveis subscritas, funções, subrotinas e o FOR. Essa alocação dependerá do tipo de dado, e será vista a seguir.

IV.2.1 - Variáveis Inteiras

Ocuparão uma área do tipo "address", isto é, de 2 bytes. Seu valor será armazenado segundo a forma interna para representação de valores inteiros.

IV.2.2 - Constantes Inteiras

Ocuparão 2 bytes e não precisarão ser alocadas, pois o código gerado permite a manipulação com esse tipo de dados.

IV.2.3 - Variáveis e Constantes Reais

Ocuparão 6 bytes: 4 para a mantissa normalizada e 2 para o expoente. Como o T.I. não dispõe de processador de ponto flutuante, as operações que envolvam valores reais serão tratadas através de subrotinas que receberão como parâmetros o

endereço dos operandos e deixarão no topo da pilha do sistema o endereço onde foi colocado o resultado.

IV.2.4 - Variáveis Subscritas

Os elementos que compõem um vetor ou matriz ocuparão posições contíguas de memória, cada qual com um tamanho fixo, que pode variar entre 2 e 6 bytes, conforme o tipo de variável.

Os componentes de um vetor serão armazenados em ordem ascendente de índice.

As matrizes serão alocadas em ordem crescente da linha, ou seja, a variável A(M, N) estará armazenada da seguinte forma:

A(Ø,Ø) A(Ø,1) ... A(Ø, N) A(1,Ø) ... A(M,Ø) A(M,1) ... A(M,N)

O endereço de A(i, j) será calculado assim:

ender(A(i,j)) = ender(A(Ø,Ø)) +i.c.(M+1) + c.j.

onde c é o número de bytes que ocupará o elemento.

O número de elementos que uma variável subscrita pode conter é conhecido no segmento principal em tempo de compilação, através da declaração DIM ou assume-se 10 ou 10x10, caso a variável não seja declarada. Em tempo de execução, toda vez

que for calculado um índice será verificado se o seu valor não excede o limite da dimensão. Esses limites são conhecidos em tempo de compilação, mas devido às facilidades de edição, para evitar que a deleção de uma declaração DIM obrigue a alterar todos os incrementos onde haja referência à variável, será criado um descritor para conter as informações necessárias sobre a variável e que será alocado em fase de compilação na área de dados do segmento onde a variável está sendo referenciada. Os elementos da matriz ou vetor só serão alocados no Passo 2. A rotina para alocação de variáveis subscritas irá calcular o tamanho de cada vetor ou matriz e associará à essa variável uma área de memória que a contenha, e preenche o descritor com as informações necessárias, quando for o caso. O descritor terá o formato seguinte:

n	N	M	d	E

onde:

n - número de dimensões

N,M - limites das dimensões; serão utilizados em tempo de execução para testar a validade da referência à variável

d - constante; viu-se anteriormente que no cálculo do endereço de um elemento entra o termo c.(M+1).i. O valor c.(M+1) é constante para todos os elementos. Para que não seja necessário calculá-lo a cada acesso a um elemento, quando for completado o descritor esse valor é calculado e guardado. Assim,o endereço de A(i,j) pode ser expresso como:

ender(A(Ø,Ø)) + d.i + c.j

E - endereço do elemento A(Ø,Ø)

Para o cálculo do endereço de um elemento em tempo de execução será utilizada uma rotina (INDEX), que fará os seguintes passos:

- localiza no descritor da variável: o número e limite das dimensões;

- verifica se o número de índices e os limites das dimensões estão de acordo com os valores do descritor;

- caso não haja erro, devolve ao programa o valor ou o endereço do elemento, conforme o caso.

Para a obtenção de um elemento qualquer $A(i_1,i_2)$, o código gerado, em linhas gerais, seria:

ξ i_1 deixa valor de i_1 na pilha do sistema

ξ i_2 calcula o valor de i_2

ℓa número de índices

ℓa tipo do resultado (1=endereço; Ø=valor)

ℓia ender. do descritor de A

call INDEX

A descrição das micro-rotinas do PLTI se encontram no Apêndice D.

IV.2.5 - Variáveis e ctes. Alfanuméricas

As ctes. alfanuméricas serão acessadas através de apontadores. As variáveis do usuário ou as internas criadas para conter as constantes na verdade serão apontadores para a cadeia, que estará na parte dinâmica da área de dados. Essa área tem tamanho fixo e será alocada em fase de compilação ao final da área de dados do segmento.

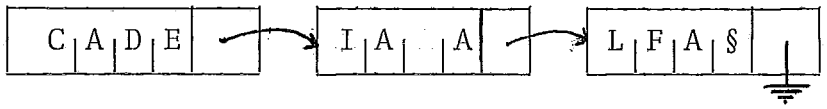
A estrutura utilizada para representá-las será uma lista, em que cada campo tem o seguinte formato:

caracteres	apontador

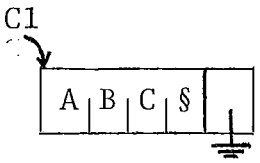
O campo de caracteres tem 4 bytes. Caso a constante necessite de mais espaço, é alocado outro nó até esgotar a cadeia.

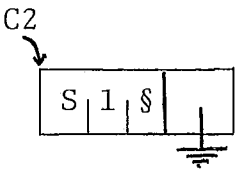
Ao final haverá u'a marca de fim (representado no texto por §).

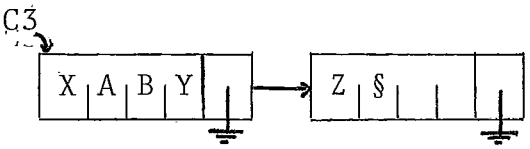
Ex.: 'CADEIA ALFA'



As operações com cadeias serão mostradas com os exemplos abaixo; supor as seguintes cadeias:







i) atribuição: C1 = C2

- C2 é copiada para uma outra área. A lista apontada por C1 é liberada e C1 passa a conter o endereço da cópia de C2.

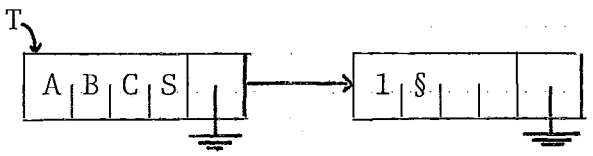
ii) concatenação: C1 + C2

deve obedecer aos seguintes passos:

- copia C1 para uma área temporária (T);

- insere os caracteres de C2 na lista apontada por T;

ao final ter-se-á:



iii) obtenção de sub-cadeias: C1 = right (C3, 2), ou seja, obter os 2 caracteres mais à direita de C3.

Será necessário criar uma lista com os dois últimos caracteres de C3, liberar a cadeia apontada por C1 e colocar em C1 o endereço da nova cadeia.

Pelos exemplos acima pode-se ver que a avaliação de uma expressão alfanumérica faz com que seja gerada uma nova cadeia. Caso seja feita alguma atribuição, o valor antigo deve ser liberado e a variável passa a apontar para a nova cadeia. Se não for feita uma atribuição, a cadeia gerada é liberada após a expressão.

Apesar da manipulação com cadeias ser um tanto lenta, especialmente quanto às operações para obtenção de sub-cadeias, o método escolhido tem a vantagem de permitir que a alocação e a liberação de áreas seja simples, não necessitando sequer de rearrumação da área de cadeias alfanuméricas.

IV.2.6 - Tabela DATA

Essa tabela conterá as constantes que façam parte de declarações DATA que ocorram no segmento. Cada entrada dessa tabela terá o formato:

tipo	conteúdo

- o tipo ocupa 1 byte e indica se a constante é inteira, real ou alfanumérica

- o conteúdo ocupa 2 bytes e pode conter o valor de uma constante inteira ou o endereço de uma constante real ou cadeia.

O apontador para essa tabela estará alocado no endereço imediatamente anterior ao início da tabela. Na fase de execução, sempre que houver um READ, a constante indicada pelo ponteiro é armazenada na variável, desde que os tipos de ambas coincidam. O ponteiro passa a indicar a próxima constante disponível para a leitura. É acusado erro se houver tentativa de leitura e a tabela tenha se esgotado.

Quando for executado um RESTORE, o apontador passa a indicar a primeira posição da tabela, que é o endereço seguinte ao seu. Desse modo não é necessário manter dois ponteiros (um para o início e outro para percorrer a tabela), pois o começo da tabela pode ser conhecido desde a fase de compilação.

IV.2.7 - Descritor do FOR

Em um compilador incremental a associação entre o FOR e o NEXT não pode ser feita na primeira fase da compilação. É criado então um descritor, contendo o número da linha inicial e o final da malha, o endereço de início da rotina de teste e o

endereço de fim (da instrução que vem imediatamente após o NEXT). O uso desse descritor será visto com mais detalhes quando da descrição do uso da pilha de blocos e da análise de comando FOR.

IV.2.8 - Descritor de Função/Subrotina

Conterá informações sobre uma função/subrotina a serem utilizadas em fase de execução. Essas informações compreendem: número e tipo dos parâmetros, endereço de início. Além disso, é necessário também uma informação sobre o tipo de blocos(se é função ou subrotina), pois esse descritor será inserido na pilha de blocos, conforme será visto adiante.

Esse descritor é criado quando o nome da função ou subrotina aparece pela primeira vez no programa e completado quando surgir sua definição.

O uso desse descritor evita que o teste de uso incorreto de função/subrotina seja feito em tempo de compilação, o que faria com que as aparições subsequentes fossem comparadas com alguma aparição anterior, saindo portanto do objetivo do projeto, que busca a maior independência possível na análise de cada incremento.

IV.3 - A Pilha de Blocos

Será mantida em tempo de execução para controlar a entrada e a saída de blocos. É inicializada com informações so-

bre o segmento principal.

Em seguida a cada for, gosub, call ou uso de funções de múltiplas linhas definidas pelo usuário, é inserida a informação correspondente nessa pilha, constando de:tipo de bloco, quando for possível a sua identificação.

Toda vez que aparecer um comando de fechamento de bloco, é verificado se o bloco correspondente consta dessa pilha. Se for o caso, a informação sobre esse bloco é retirada da pilha; em caso contrário, é acusado erro e o programa é cancelado.

Quando for executado um for, seu descritor é colocado nessa pilha. Ao ser encontrado um next, é verificado se a variável de controle coincida com a que está nesse descritor. Se for o caso, desvia-se para a rotina que incrementa e testa essa variável, caso contrário, é acusado erro. Sempre que houver algum desvio, também deve-se verificar se não sai da malha FOR/NEXT, comparando-se o endereço de desvio com os endereços iniciais e finais da malha. O descritor do FOR sai da pilha de blocos quando:

a) a condição de fim foi atingida

b) há um desvio para fora do FOR

c) houve erro.

Essa pilha conterá também informações sobre as subrotinas BASIC que estejam sendo executadas (através do comando gosub). Quando for executado um return, é verificado se o topo da pilha de blocos contem uma indicação de que anteriormente foi executado um gosub. Caso não tenha sido, é acusado erro.

Uma pilha é a estrutura ideal para conter as informações mencionadas pois a entrada em um segmento ou em um bloco tipo for ou gosub segue o esquema: o primeiro a entrar é o último a sair (first-in-last-out), isto é, o retorno a um bloco ou segmento se dará somente depois que todos os blocos internos ou os segmentos que forem chamados tenham sido executados.

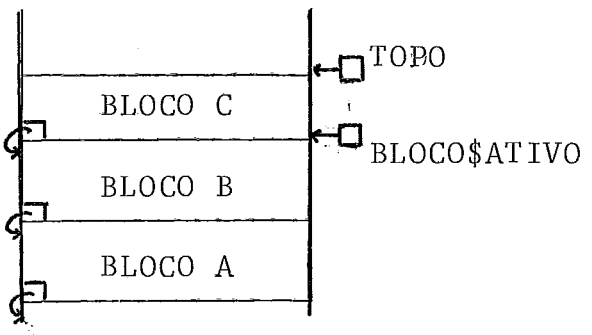
O acesso à essa pilha se dará através dos ponteiros:

TOPO - indicará a próxima posição livre na pilha

BLOCO$ATIVO - apontará para o descritor do bloco corrente

Além das informações sobre o bloco haverá também um apontador para o bloco anterior.

Em um determinado momento da execução a pilha de blocos pode ter o seguinte aspecto:



IV.4 - Passagem de Parâmetros

Em um compilador incremental, nem sempre será possível saber, no início do segmento, se se trata de uma subrotina, função ou o programa principal.

Portanto, no decorrer da análise e geração de código, não se sabe necessariamente quais são os parâmetros formais ou quais as variáveis locais, pois a facilidade de edição vai permitir também que um segmento se transforme em uma função ou subrotina.

No caso das funções, em que a passagem de parâmetros é por valor, os argumentos formais serão tratados como variáveis locais ao segmento de definição. Quando for feita uma referência à função, antes de se iniciar a sua execução, inicializam-se as variáveis locais com os valores dos parâmetros reais correspondentes. Somente o valor final da função é devolvido ao ponto de chamada. Por motivo de economia de tempo de execução, bem como de espaço, não será permitido o nome de variável subscrita como parâmetro, devido ao custo na inicialização da função.

Quanto às subrotinas, serão implementadas por se constituirem em mais uma ferramenta para o usuário. No seu caso, não será devolvido ao ponto de partida somente um valor, mas vários. As alterações que forem feitas nos parâmetros formais deverão ser feitas nos parâmetros reais correspondentes. Além disso, será permitido ter variáveis subscritas como parâmetro.

Neste caso a inicialização dos parâmetros formais não é tão simples quanto para as funções. Aqui, as variáveis locais que estejam na lista de parâmetros formais de subrotinas serão apontadores para os correspondentes parâmetros reais. A inicialização se constituirá portanto na passagem dos endereços dos parâmetros reais para a área de dados da subrotina. Se algum argumento for dado como uma expressão, seu valor é calculado e guardado em uma temporária, cujo endereço é então passado para a subrotina. Em resumo, a passagem é por referência.

Pelo que já foi visto, tem-se que os parâmetros formais e as variáveis locais terão tratamentos diferentes. Como só se saberá se um determinado segmento conterá uma subrotina quando for encontrada a declaração SUB, o usuário deverá ser informado de que todo o segmento deverá ser re-compilado. De modo análogo, quando essa declaração for deletada do segmento.

Apesar de ser pouco eficiente, em termos de compilação, e de fugir um pouco dos objetivos do projeto, tal não afetará muito a maioria dos usuários, pois este é um recurso mais complexo, que deverá ser utilizado por usuários com maior experiência.

Um modo de evitar a re-compilação seria tornar qualquer referência às variáveis como endereçamento indireto, mas tal tornaria a execução de qualquer segmento ineficiente, o que prejudicaria aos usuários que não se utilizassem desse recurso.

IV.5 - Chamada de Segmento

Como foi visto no capítulo anterior, a passagem de parâmetros nas funções e subrotinas seguem procedimentos diferentes.

Para as funções, serão executados os seguintes passos:

1. colocar na pilha do sistema (pilha utilizada pelas micro-rotinas do PLTI): o valor e o tipo de cada parâmetro real, o endereço do descritor e o endereço de retorno.

2. chama a rotina que verifica se o uso é válido, quanto ao número e tipos dos parâmetros. Caso haja algum erro, o programa é cancelado e é enviada uma mensagem para o usuário. Se tudo estiver correto, é deixado no topo da pilha: o endereço de retorno, o valor dos parâmetros e o endereço de início.

3. inicializa o segmento da maneira abaixo:

   3.1 - coloca-se o descritor na posição indicada por TOPO. O valor do BLOCO$ATIVO é guardado junto ao restante das

informações e esse valor é atualizado para o atual valor do TOPO. Este é modificado em seguida, de modo a indicar a próxima posição disponível na pilha de blocos.

3.2 - inicializa a área de dados com os valores dos parâmetros reais (os parâmetros formais no caso das funções, são variáveis locais ao seu segmento de definição).

4. passa-se o controle à primeira instrução executável do segmento.

Ao ser encontrado um CALL, são executados os seguintes passos:

1. calcula-se o endereço de cada parâmetro real e colocando-o na pilha do sistema, bem como o endereço de retorno;

2. procede-se como no item 2, referente à funções.

Na inicialização do segmento:

- insere-se o descritor da subrotina na pilha de blocos, como em (3.1);

- guardar na área de dados do segmento os endereços dos parâmetros reais que se encontram na pilha do sistema;

- passar o controle à primeira instrução do segmento.

IV.6 - Saída de um Segmento

Quando acaba um segmento o sistema deve ser restabelecido para a condição anterior à chamada do segmento, ou seja:

a) o TOPO passará a ter o valor atual de BLOCO$ATIVO;

b) BLOCO$ATIVO é atualizado para voltar a indicar o bloco anterior;

c) continua a execução a partir do ponto indicado pelo endereço de retorno, que está na pilha do sistema;

d) para as funções, seu valor é deixado no topo da pilha do sistema.

IV.7 - Tratamento de Interrupções

Há 4 tipos de interrupção da execução:

a) por erro de programa

b) externa: através de chave no teclado

c) interna: comando STOP foi executado

d) operação de E/S: tela cheia

No caso (a), o programa é cancelado, a mensagem correspondente é emitida, juntamente com o número da linha e o nome do segmento onde ocorreu o erro e aguarda-se a próxima ação do usuário. O programa sai da memória, juntamente com a pilha de blocos. A pilha do sistema é esvaziada.

O caso (d) foi visto no item anterior; e os casos (b) e (c) são explicados a seguir.

O usuário pode interromper a execução de seu programa para fornecer comandos de edição ou de modo imediato. Em qualquer dos casos, o estado atual da execução deve ser guardado, pois no caso de execução em modo imediato, o usuário pode continuar o programa do ponto em que parou, ou de outro ponto qualquer, desde que seja dentro do segmento que foi interrompido. O usuário pode editar qualquer segmento dentro do programa, só que neste caso a re-execução demora mais, pois é feito um segundo passo no segmento.

Antes de iniciar a execução de um novo incremento, é verificado se há alguma interrupção externa. Caso tal aconteça o sistema passa o controle à Rotina de Tratamento Interrupção que tem como funções:

1º - salvar em área de trabalho:

- a pilha de blocos e os valores de BLOCO$ATIVO e TOPO;
- o conteúdo da pilha do sistema;

2º - copiar de volta para a área temporária de código os segmentos que estão na memória.

Desse modo, se a execução for re-encetada, não será necessário atualizar os endereços dentro dos segmentos.

3º - informar ao usuário o "status" atual da execução, ou seja, o nome do segmento em execução, o número da linha que estava sendo executada o endereço da próxima instrução.

4º - traz para a memória as rotinas do compilador, e do analisador de comandos;

5º - aguarda o próximo passo do usuário.

IV.8 - Variáveis Temporárias

Este tipo de variáveis são utilizadas para conter resultados intermediários de expressões. Os exemplos abaixo mostram onde será necessário o emprego de temporárias no decorrer do programa:

Exemplos:

(i) em operações do tipo: A + B%

- será necessário uma temporária para conter o valor de B% convertido para real, e outra para conter o resultado da soma;

(ii) na chamada de subrotinas: call SUB1(x+y)

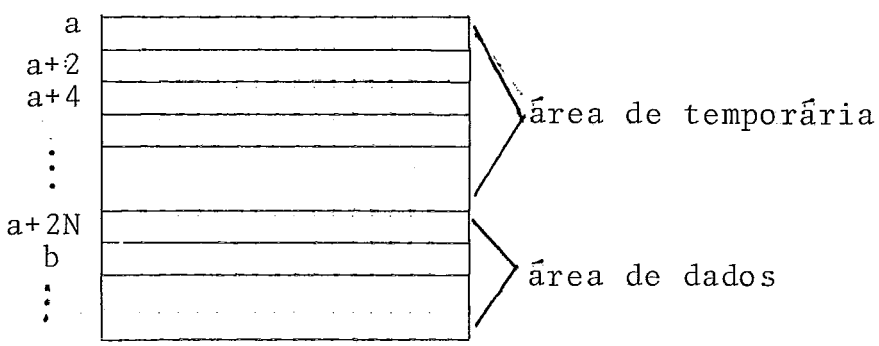
- deve ser criada uma temporária para conter o resultado de x+y, pois os parâmetros são passados por referência. Essa técnica é utilizada pelo compilador PL/I, conhecida como "dummy arguments".

(iii) na manipulação com cadeias alfanuméricas, conforme está mostrado em (IV.2.5).

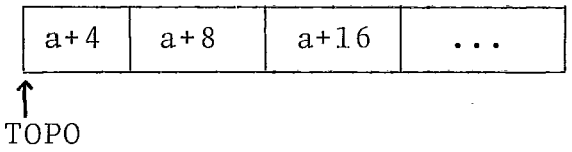
Uma parte da área estática de dados será reservada para conter as temporárias. Para o controle da alocação e liberação, será usada uma pilha: toda vez que for usada uma nova temporária, seu endereço é tirado do topo dessa pilha. Quando ela for liberada, seu endereço volta à pilha. Caso seja necessária uma temporária e a pilha esteja vazia, a execução é abortada, pois foi esgotada a área de temporárias.

Para melhor compreensão do processo, supor o exemplo seguinte:

- área estática de dados



- supor que em um ponto qualquer de execução, a pilha de temporárias está assim:



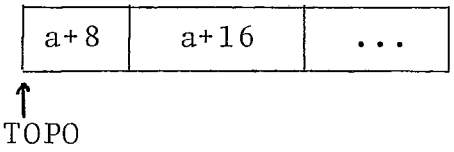
e está sendo executado o seguinte código, referente ao trecho de expressão: A * (B + C%)

lia A

lia B

la C%

call FLOAT → aloca o endereço (a+4); a pilha de temporárias ficará então:



call FADD → libera (a+4);
aloca (a+4) para conter a soma

call FMUL → libera (a+4);
aloca (a+4) para conter o produto

V. PASSO 2

Esse passo é executado sempre que o usuário fornecer o comando FIM, indicando que acabou a edição no segmento.

Haverá uma área na memória, disponível tanto em tempo de compilação quanto de execução, contendo as seguintes informações:

- identificação do segmento que está sendo compilado;

- identificação do segmento que está em execução;

- endereço de retorno para execução (quando e onde a execução foi interrompida)

- chave de edição: 1 bit, que será ligado quando for dado um comando de edição e desligada, quando for executado o Passo 2.

De posse dessas informações, é feito um acesso à Tabela de Segmentos para determinar a entrada referente ao 1º incremento no Dicionário de Incrementos. Para cada entrada, é feito o seguinte processo:

1. verifica se o campo de endereço do código contem alguma informação. Caso esteja vazio, o número da linha é colocado na Lista de Rótulos Indefinidos, pois significa que essa linha foi referenciada mas não foi fornecida pelo usuário. Ao final da

análise, todo o conteúdo dessa lista é mostrado no vídeo; caso ela contenha pelo menos uma entrada, o programa não poderá ser executado.

2. verifica se o tipo da instrução é FOR. Em caso afirmativo, é colocado na Pilha Auxiliar o número da linha e o endereço do descritor.

3. verifica se o tipo da instrução é def ou sub. Procede-se então do seguinte modo:

   3.1 - verificar se é a linha de menor número dentro do segmento;

   3.2 - verificar se a Pilha Auxiliar contem uma destas declarações. Se for o caso, acusar erro, pois não é permitida a definição de funções ou subrotinas embutidas.

4. verifica se o tipo da instrução é next. Se for:

   4.1 - verificar se o topo da Pilha Auxiliar contem um for;

   4.2 - completar o descritor do FOR com o endereço da instrução após o NEXT;

   4.3 - retirar a linha da Pilha Auxiliar.

5. caso seja um SUBEND ou FNEND:

5.1 - verificar se o topo da Pilha Auxiliar contem uma linha DEF ou SUB

5.2 - retirar essa linha da Pilha Auxiliar.

Depois de percorrer todo o Dicionário de Incrementos, é feita a alocação de variáveis subscritas. A lista de símbolos indefinidos, citada em (III.4.3) é percorrida. A cada nó referente a matriz/vetor, é feito um acesso a Tabela de Símbolos para obter descritor, onde serão citadas informações sobre as dimensões da variável. Completa o descritor com os valores "default", caso o(s) limite(s) da(s) dimensão(oes) não sejam(s) conhecido(s). No caso de não ser possível saber quantas dimensões tem a variável, é acusado erro e o nó correspondente continua nesta lista.

No caso de ser encontrado qualquer erro nesse passo, a chave de erro da Tabela de Segmentos é ligada e o programa não poderá ser executado.

VI. PROCESSAMENTO DE COMANDOS IMEDIATOS

O uso de comandos imediatos é particularmente útil na depuração de programas, pois o usuário poderá examinar os dados, imprimindo seus valores ou alterá-los e em seguida continuar a execução.

Todos os comandos executáveis do BASIC são permitidos. Do mesmo modo que qualquer outro comando, eles são analisados pelo compilador. A distinção entre uns e outros é que os comandos imediatos não têm número de linha. O código gerado para eles fica em uma área separada. Após a execução de um ou mais desses comandos, o sistema volta ao modo de entrada, ficando o controle com o segmento corrente.

Para retornar a execução, o usuário deve fornecer o comando CONT (ou CONTINUE), caso queira continuar do ponto onde houve a interrupção ou de outro ponto qualquer do segmento. Esses comandos só são válidos se o usuário tiver interrompido a execução de algum segmento. Caso a interrupção se tenha dado em fase de compilação esses comandos não são válidos.

Se ocorreu algum erro nesse modo, o usuário é avisado de que a execução do programa deve ser re-iniciada. Através do comando EXEC.

VII. CARGA DE PROGRAMAS

Essa rotina é acionada sempre que o usuário der o comando EXEC. Neste ponto já se terá conhecimento de toda a estrutura do programa: o total de segmentos, formação de cada segmento.

Todo o programa é percorrido, ou melhor, as estruturas que o definem: a Tabela de Segmentos, o Dicionário de Incrementos. Será verificado:

- se a lista de Símbolos Indefinidos, citada em III.4.3, não estiver vazia, significa que há funções ou subrotinas que ainda não foram definidas. O conteúdo dessa lista deve ser mostrado ao usuário;

- se há algum segmento com erro, ou seja, se a chave de erro na tabela está ligada.

Caso não haja erros, deverão ser executados ainda os seguintes passos:

(a) cria a Tabela de Arquivos com informações obtidas nas tabelas de Designadores e na de Símbolos, bem como do diretório de arquivos em disco;

(b) aloca a Pilha de Blocos, inicializando-a com informações sobre o segmento principal;

(c) carrega o código gerado para o programa.

VIII. EDITOR

O usuário contará com facilidades de edição que lhe permitirão apagar, listar trechos ou todo um segmento, bem como resequenciá-lo. Os comandos de edição terão a forma seguinte:

$$\text{APAGA} \left\{ n_1, n_2, \ldots, n_i \right\}$$
$$\text{AP} \quad \left\{ n\text{-}m \right\}$$

- apaga uma ou mais linhas do segmento. No primeiro formato, são apagadas as i linhas cujos números estão indicados. No segundo caso, são apagadas todas as linhas entre os números n e m inclusive.

$$\text{SELEC} \left\{ n_1, n_2, \ldots, n_i \right\}$$
$$\text{SEL} \quad \left\{ n\text{-}m \right\}$$

- mostra no vídeo os trechos selecionados do segmento corrente. Caso seja usado o primeiro formato, são mostradas as i linhas indicadas. Na outra forma, todo o trecho entre as linhas n e m inclusive é visualizado.

RESEQ [n[, i]]
RES

- remunera as linhas do segmento corrente. Caso não seja fornecido nenhum parâmetro, é assumido 10 para o número da linha inicial e 10 o valor do incremento.

Deve ser fornecido um comando por linha. Cada linha é analisada e o comando executado, desde que não haja erros. Após sua execução é emitida uma mensagem ao usuário indicando que o comando foi executado corretamente e aguarda-se o próximo comando de edição.

As linhas que são deletadas são re-analisadas, para que se possa atualizar o contador de referências na Tabela de Símbolos. No Dicionário de Incrementos, o campo referente ao endereço de código é esvaziado, caso o campo de referências não contenha um valor nulo, é acusado erro: está sendo deletada uma linha para a qual há referências.

Para o resequenciamento de um segmento, é verificado antes se alguma linha virá a ter um valor fora dos limites permitidos. Por exemplo, supor que um segmento S tem 100 linhas (o número de linhas pode ser obtido da Tabela de Segmentos) e o usuário forneça o comando:

RES 30000, 500

O valor máximo permitido para um número de linha é 32767. Assim sendo, o segmento poderia ter no máximo 4 linhas para que o novo valor da última linha não ultrapassa o limite máximo. Como S tem 100 linhas, seria enviada uma mensagem ao usuário, informando que o resequenciamento é inválido.

Em seguida é criada uma cópia do Dicionário de Incrementos, alterando-se os números de linha conforme a indicação do usuário.

Ao final da operação, o endereço do novo Dicionário de Incrementos é guardado na Tabela de Segmentos, e a área ocupada pelo antigo é liberada.

De modo análogo, é criada uma cópia do programa fonte e do código objeto, atualizando-se os números de linha. As áreas antigas são depois liberadas caso a operação tenha sido executada com sucesso.

Os comandos de edição não precisarão ser fornecidos necessariamente por ordem do número de linha, pois na criação do programa o usuário poderá fornece-las em qualquer ordem, e os registros do arquivo objeto estão de acordo com essa ordem. Portanto não vai implicar necessariamente em uma redução dos movimentos do disco se as linhas forem sendo editadas ordenadamente.

IX. DISCUSSÃO E CONCLUSÃO

O compilador descrito nesse trabalho apresenta uma série de diferenças em relação ao método tradicional, conforme foi citado no item referente à metodologia e pode ser percebido no decorrer do texto.

Uma comparação entre um compilador desenvolvido pelo método tradicional e o incremental só pode ser discutida aqui em termos teóricos, de vez que não me foi possível implementá-lo.

Berthaud e Griffiths[3] compararam o desempenho dos compiladores padrões da IBM para as linguagens ALGOL e PL/I com os respectivos desenvolvidos pelo método incremental. Os compiladores incrementais se mostraram por volta de 10 vezes mais rápidos que os outros, o que já era esperado, de vez que seu trabalho é bem menor. Em termos de velocidade de execução, o compilador padrão naturalmente gera um código mais rápido, de vez que é a própria linguagem de máquina, e mais eficiente. No compilador incremental, a velocidade vai depender do tipo de instrução mas no caso do PL/I, em média, o fator foi da ordem de 100. Comparando-se as duas linguagens, no caso da IBM, o código gerado para um programa em ALGOL, em geral, tem um desempenho ruim se comparado com um PL/I. Usando-se incrementalismo, a situação se inverteu; o interpretador ALGOL é de ordem 2 vezes mais veloz que o de PL/I. Principalmente devido à variedade de tipos de dados em PL/I, as rotinas para avaliação de expressões ficaram muito lentas. Além disso, o PL/I também sai perdendo em termos de espaço,

porque a descrição das variáveis é mantida também em tempo de execução.

Em resumo, essa técnica é bastante útil em ambiente de ensino, pois os usuários que estão aprendendo uma linguagem tendem a ter um tempo de execução pequeno comparado com o de depuração. Para fins comerciais, esse método se mostraria anti-econômico.

No nosso caso específico, as diferenças principais entre o compilador incremental BASIC e os tradicionais que foram implementados no T.I., podem ser resumidas em:

- o controle da execução não é feito pelo SOCO, portanto muitas das facilidades aí disponíveis tiveram que ser desenvolvidas pelo próprio sistema, particularmente no que diz respeito à entrada e saída;

- a passagem de parâmetros, especialmente nas subrotinas, terá tratamento especial, não podendo utilizar a micro-rotina ent, disponível no interpretador PLTI;

- foi necessário criar uma pilha extra, para controle de entrada e saída de blocos, especialmente no que se refere a evitar a recursividade, de vez que não é possível fazer as verificações necessárias em fase de compilação;

- associação entre um símbolo e sua declaração será feita em tempo de execução;

- o código gerado não ocupa posições contíguas no arquivo, devido à facilidade de edição, sendo por vezes necessário compactar a área que ele ocupa, de modo a que se possa vir a re-utilizar os espaços vazios.

CONSIDERAÇÕES FINAIS

As rotinas necessárias para a implementação do presente trabalho encontram-se na forma "quase PLTI", de modo que qualquer programador com experiência nessa linguagem possa implementá-las.

Procurou-se desenvolver todo o projeto de forma modular, de modo a facilitar tanto a parte de programação quanto as futuras alterações que venham a ocorrer no sentido de otimizá-lo.

BIBLIOGRAFIA

| [1] | Griffiths, Peccoud e Peltier - "Incremental Interactive Compilation", Proc. I.F.I.P., 1968.

| [2] | Bethaud e Griffiths - Incremental Compilation and Conversational Interpretation", Annual Review in Automatic Programming, Volume 7, parte 2, 1973.

| [3] | Gries, David - Compiler Construction for Digital Computers, Wiley International Edition, 1971.

| [4] | Aho, Alfred V. e Ullman, Jeffrey D. - Principles of Compiler Design, Addison Wesley, 1977.

| [5] | Bauer, F. L. et al. - Compiler Construction, Springer-Verlag, 1976.

| [6] | Wirth, Nicklauss - Algorithms + Data Structures =Programs, Prentice-Hall, 1976.

| [7] | Knuth, Donald E. - Fundamental Algorithms, Addison Wesley, 1976.

| [8] | Knuth, Donald E. - Storting and Searching, Addison Wesley, 1975.

| [9] | Madnick, Stuart E. e Donovan, John J. - Operating Systems, McGraw-Hill, 1974.

|10| IBM DOS/VC Entry Time-Sharing System/II, IBM Deutschland GmbH, Central Programming Support Group, Boeblingen, West Germany, 1977.

|11| DEC - PDP-11 Basic Plus, Digital Equipment Corporation,Maynard, Massachussets, 1974.

|12| Burroughs - B6700 Basic Language, Information Manual, 5000383, 1971.

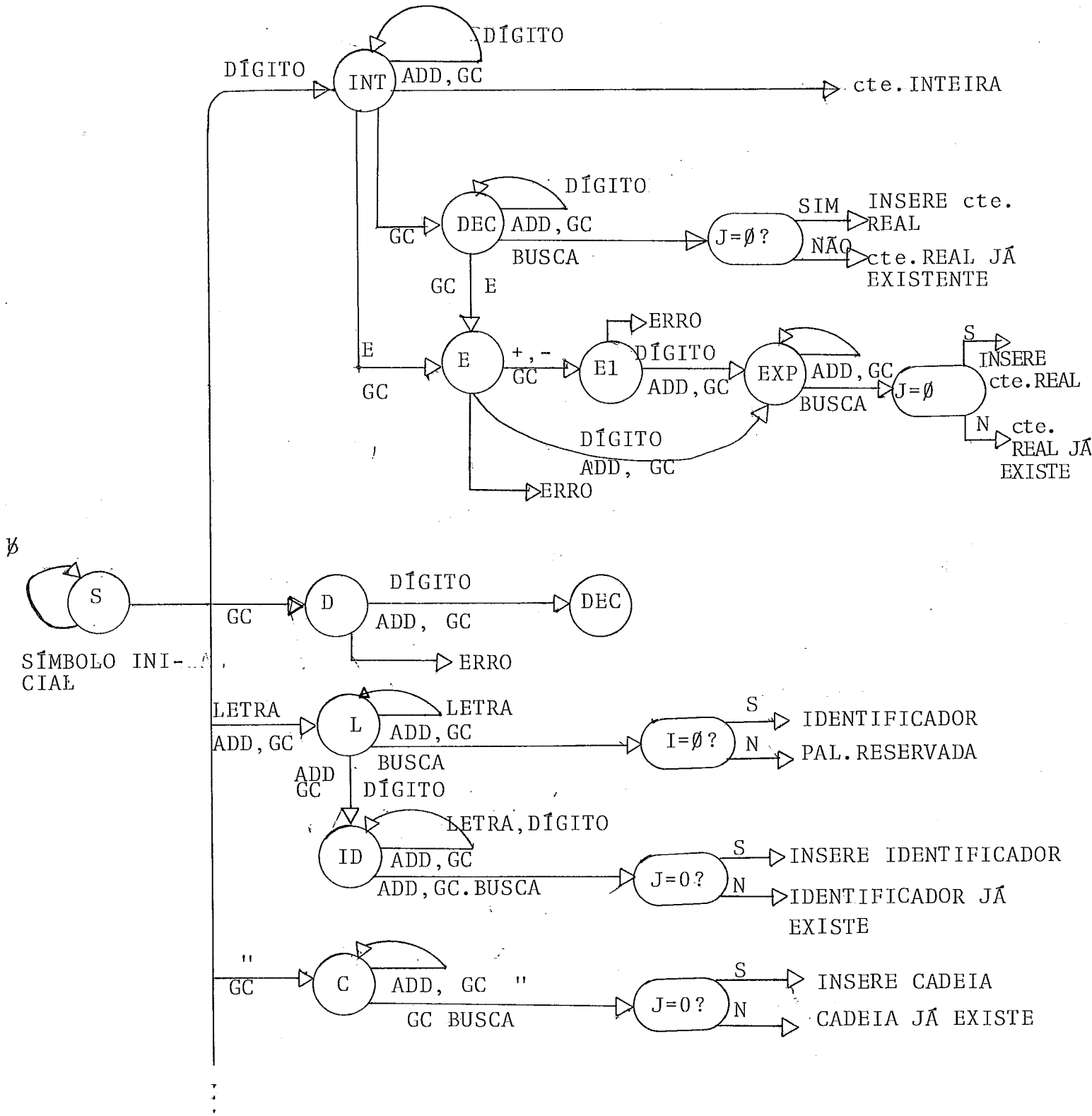
|13| Rodrigues, Guilherme C. - Histórico do Desenvolvimento Tecnológico no NCE, Publicação do Vº Seminário Integrado de Software e Hardware Nacionais, 1978.

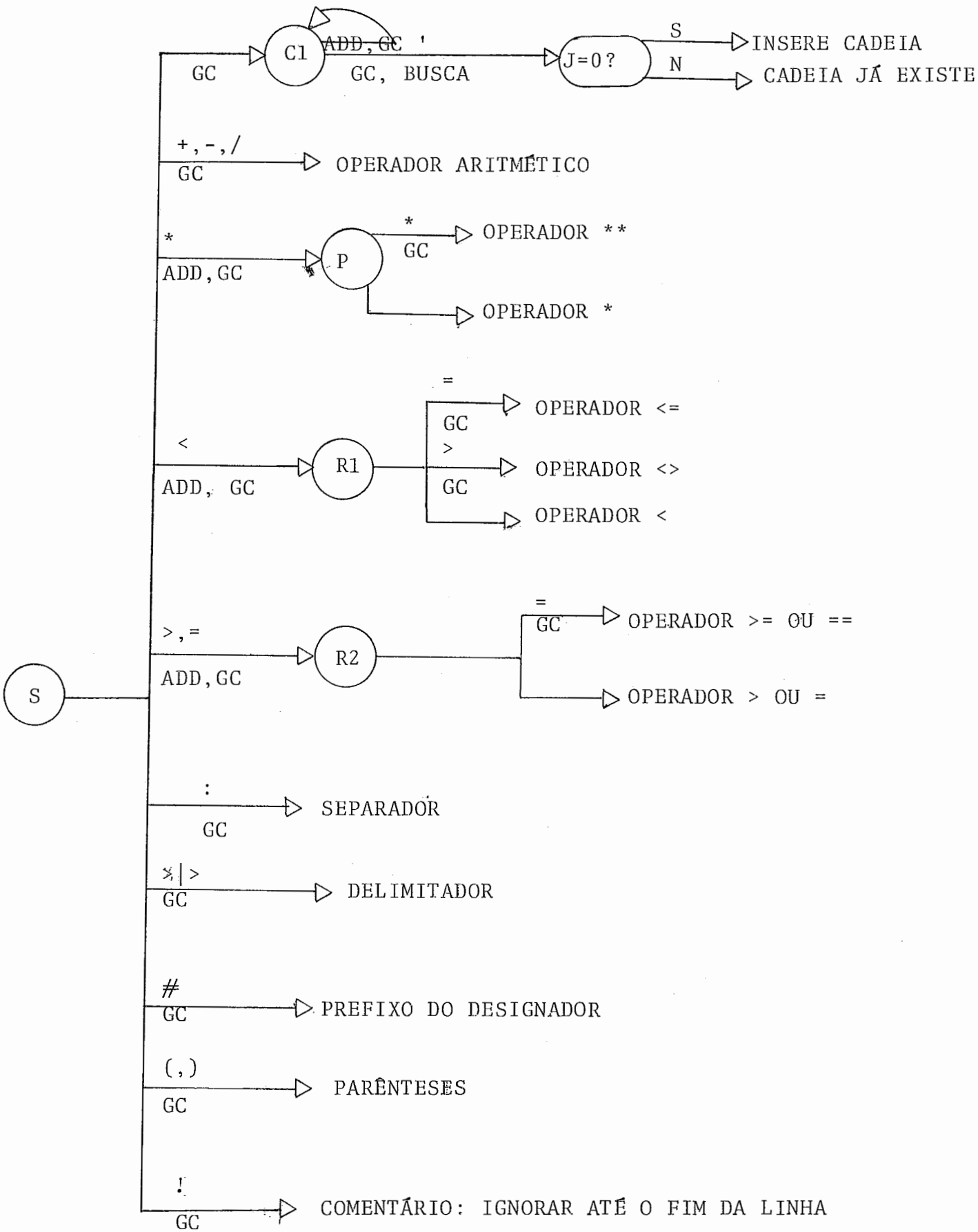
|14| NCE - Descrição da Linguagem PLTI, Publicação Interna do Núcleo de Computação Eletrônica da UFRJ.

|15| Meyers, Edmund D. - Time-Sharing Computation in the Social Sciences, Prentice Hall, 1973.

APÊNDICE A

AUTÔMATO LÉXICO




S
GC
C1
ADD, GC '
GC, BUSCA
J=0?
S
N
INSERE CADEIA
CADEIA JÁ EXISTE
+, -, /
GC
OPERADOR ARITMÉTICO
*
ADD, GC
P
*
GC
OPERADOR **
OPERADOR *
<
ADD, GC
R1
=
GC
OPERADOR <=
>
GC
OPERADOR <>
OPERADOR <
>, =
ADD, GC
R2
=
GC
OPERADOR >= OU ==
OPERADOR > OU =
:
GC
SEPARADOR
GC
DELIMITADOR
#
GC
PREFIXO DO DESIGNADOR
(,)
GC
PARÊNTESES
!
GC
COMENTÁRIO: IGNORAR ATÉ O FIM DA LINHA

ondë:

GC - rotina que obtem o próximo caracter

ADD - guarda o caracter recebido

BUSCA - significa o acesso a uma tabela; no caso, é de símbolos ou de palavras reservadas. Devolve J = 0 se não achou o símbolo, e 1 em caso contrário.

APÊNDICE B

DESCRIÇÃO DOS COMANDOS DE CONTROLE

Em seguida será dado o formato de cada comando, bem como a ação referente a cada um. Abaixo de cada um vem a abreviatura mínima permitida.

1. ADEUS

   AD

   - o usuário terminou seu trabalho. O controle é devolvido ao SOCO.

2. CONTINUE [número de linha]

   CONT

   - continua a execução do segmento corrente, a partir do ponto em que foi interrompido, ou de outra linha qualquer cujo número de linha foi fornecido.

3. EXECUTE

   EX

   - inicia a execução de um programa que se encontra na área temporária.

4. GUARDE

   GU nome do programa

   - guarda de modo permanente um programa que se encontra na área temporária, sob o nome que é indicado no comando.

5. INDICE

   IND

   - mostra no vídeo uma lista dos segmentos que constituem um programa que está na área temporária.

6. LISTA

   LIS nome do programa

   - lista no vídeo todo o programa.

7. PEGUE

   PE nome do programa

   - traz para a área temporária um programa que havia sido armazenado anteriormente

8. RENOMEIA

   REN nome antigo, nome novo

   - altera o nome de um programa já existente.

9. REORGANIZE

   REO

   - re-arruma o arquivo de código.

10. RETIRE

   RET nome do programa |nome do segmento|

   - apaga todo um programa ou um segmento de um programa

11. SEGMENTO

SEG nome do segmento

- cria um novo segmento ou permite a alteração de algum já existente. O sistema passa ao modo de entrada.

12. TROCA

TR nome do programa

- substitui o programa mencionado pela versão que se encontra na área temporária.

APÊNDICE C

PALAVRAS RESERVADAS

ADEUS	EXECUTE	LET	RECORD	SUB
AND	FILE	LINE	RENOMEIA	SUBEND
APAGA	FIMSEG	LISTA	REORG	THEN
CALL	FNEND	MAT	RESEQ	TO
CONTINUE	FOR	NEXT	RESTORE	TROCA
DATA	GOSUB	NOT	RETIRA	UNTIL
DEF	GOTO	ON	RETURN	USING
DIM	GUARDA	OR	SEGMENTO	WHILE
ELSE	IF	PEGUE	SELEC	XOR
END	INDICE	PRINT	STEP	
EQV	INPUT	READ	STOP	

APÊNDICE D

DESCRIÇÃO DAS MICRO-ROTINAS DO PLTI

```
 100
 200
 300
 400
 500
 600
 700        SIMBOLOGIA:
 800
 900 S     -TOPO DA PILHA DE OPERANDOS
1000 SO, S1, ..., S|N| - POSIÇÕES DA PILHA, AO SEREM REFERENCIADAS SAEM DA PILHA
1100                     DAS SAEM DA PILHA
1200 ==    -ATRIBUIÇÃO DE VALOR BYTE, SE NA PILHA ZERA PARTE SUPERIOR DOS 16 BITS. SE NA MEMÓRIA PEGA SOMENTE PARTE
1300        SUPERIOR DOS 16 BITS. SE NA MEMÓRIA PEGA SOMENTE PARTE
1400        INFERIOR DO TOPO
1500 =     -ATRIBUIÇÃO DE VALOR ADDRESS.
1600 X1, X2, ..., X|N| - PRIMEIRO, SEGUNDO, ..., ENESIMO OPERANDO DA INSTRUÇÃO
1700        )
1800                     X = B SE OPERANDO FOR BYTE OU
1900                     X = A SE OPERANDO FOR ADDRESS
2000 B(X) -CONTEÚDO BYTE DA MEMÓRIA X.
2100 A(X) -CONTEÚDO ADDRESS DA MEMÓRIA X E X+1.
2200 /SO/, /S1/, ..., /S|N|/ - POSIÇÕES DA PILHA, AO SEREM REFERENCIADAS NÃO SAEM DA PILHA.
2300                          RENCIADAS NÃO SAEM DA PILHA.
2400 PC    -CONTADOR DE INSTRUÇÕES.
2500 POP   -TIRA O TOPO DA PILHA.
2600 COMANDOS VÁLIDOS NA LINGUAGEM PLTI SÃO TAMBÉM USADOS.
2700
2800
2900
3000        MICRO-ROTINAS DO TIPO LOAD:
3100
3200 LIB   :S = B1
3300 LIA   :S ==A1
3400 LB    :S = B(A1)
3500 LA    :S ==A(A1)
3600 LXB   :S = B(A(A1))
3700 LXA   :S ==A(A(A1))
3800 LXSB  :S = B(A(A1)+B2)
3900 LXSA  :S ==A(A(A1)+B2)
4000 LAS   :S ==A(A1) + B2)
4100 INDB  :S = B(SO + S1)
4200 INDA  :S ==A(SO + S1)
4300
4400
4500
4600        MICRO-ROTINAS DO TIPO STORE:
4700
4800 SB     : B(S1) = SO
```

```
4900 SA     : A(S1) == SO
5000 SEB    : B(S1) =  /SO/
5100 SEA    : A(S1) == /SO/
5200
5300
5400
5500        MICRO-ROTINAS AUXILIARES:
5600
5700 ADDR   : S == S1 + SO <- 1
5800 LIAC   : S == PC + 1
5900 NOP    :
6000
6100
6200
6300        MICRO-ROTINAS ARITMÉTICAS:
6400
6500 ADD    : S == SO + S1
6600 MUL, DIV, MOD, OR, XOR, AND,
6700 EQ, NE, LT, GT, LE, GE - SEMELHANTES A ADD.
6800 NOT    : S == NOT SO
6900 MINU   : S == - SO
7000 SLR    : S == SO → S1
7100 SLL    : S == SO ← S1
7200
7300
7400
7500        MICRO-ROTINAS DE COMANDOS:
7600
7700 GOT    : PC == A1
7800 IF     : IF NOT SO THEN PC == A1
7900 GOTC   : DO I = 1 TO B1; POP; END; PC == A2
8000 DOTB   : B(A1) = /S2/;
8100            IF/S1/=/S2/ OR NOT (/SO/>0 XDR/S2/>/S1/)
8200            THEN DO; POP; POP; POP; PC == A2; END
8300 DOTA   : A(A1) == /S2/;
8400            IF/S1/=/S2/ OR NOT (/SO/>0 XOR/S2/>/S1/)
8500            THEN DO; POP; POP; POP; PC == A2; END
8600 END    : /S2/ == /S2/ + /SO/
8700 MOV    : TAM == SO; ORIG == S1; DEST == S2;
8800            DO TAM = TAM TO 0 BY - 1; B(ORIG) = B(DEST);
8900            ORIG == ORIG + 1; DEST == DEST + 1; END
9000 CASE   : N == SO; IF N ← B1 THEN PC == A|N+1|
9100 EXIT   : GO TO EXIT
9200 ABO    : LIMPA PILHA; PC == A1
9300
9400
9500
9600        MICRO-ROTINAS DE ENTRADA E SAÍDA:
9650
9700 READ   : CALL A1; DÁ VALOR AO STATUS
9800 WRITE, REWRITE, REWIND, EOF, SEEK: SEMELHANTES
9900
10000
10100
10200       MICRO-ROTINAS DE SUBROTINAS:
10300
```

```
10400 ASM    : CARREGA REGISTROS; CALL A1; SALVA REGISTROS
10500 ENT    : DO I = 0 TO B1 = 1; A|4+I| == S|I|; END;
10600          DO I = 0 TO B2 - 1; A|4+B1+I| == A(/S|B1+I|)/;END;
10700          PC == A3
10800 RET    : PC == SO; IF PC = O THEN DO: CARREGA REGISTROS;
10900          RETURN; END
11000 RETV   : S == S1; RET
11100 SR     : DO I = 0 TO B1 - 1;
11200          IF B|I+3| = 0 THEN B(S|I|) = B(A2+I<-1+1);
11300                        ELSE A(S|I|) = A(A2+I<-1); END
11400
11500
11600
11700        MICRO-ROTINAS DE DEPURAÇÃO:
11800
11900 SET, RESET, DUMP; PARE: CHAMAM SUBROTINAS ESPECIFICADAS.
12000
```