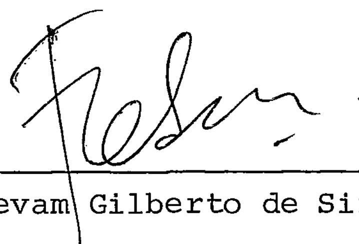
"PLMIX - UM MODELO DE LINGUAGEM DE MÉDIO NÍVEL E SEU COMPILADOR

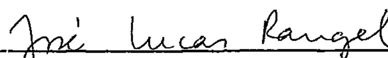
Lígia Barros Caúla

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.).

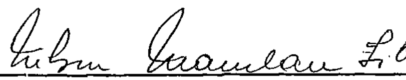
Aprovada por:



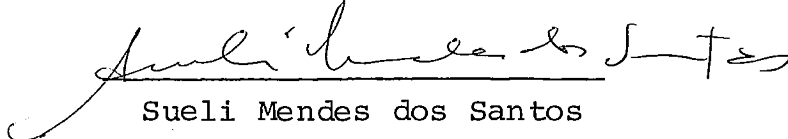
Estevam Gilberto de Simone



José Lucas Mourão Rangel Netto



Nelson Maculan Filho



Sueli Mendes dos Santos

RIO DE JANEIRO, RJ - BRASIL

NOVEMBRO DE 1978

CAÚLA, LÍGIA BARROS

PLMIX - Um Modelo de Linguagem de Médio Nível e seu Compilador|Rio de Janeiro| 1978

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

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

1. Um Modelo de Linguagem de Médio Nível e seu Compilador. I.COPEE/UFRJ. II. PLMIX - Um Modelo de Linguagem de Médio Nível e seu Compilador.

À Meus Pais,

Minha Avó Arlinda,

e Eduardo.

AGRADECIMENTOS

Ao Professor Estevam Gilberto de Simone pelas idéias e paciente orientação ministrada.

Aos Professores José Lucas Mourão Rangel Netto e Jano Moreira de Souza pelas valiosas sugestões.

Ao Professor Gerhard Schwarz e à colega Lilian Markenzon, pelo interesse e colaboração.

Aos colegas do Programa de Engenharia de Sistemas e Computação pelo apoio que recebi.

A meus amigos, em especial meu marido, pelo incentivo.

À COPPE, CAPES e FINEP pelos recursos oferecidos durante a execução deste trabalho.

RESUMO

Para permitir especificar os critérios de projeto para linguagens de programação de médio nível, foi criada a linguagem PLMIX para o computador hipotético MIX, definido por D.E. Knuth.

É apresentada uma conceituação de linguagens de médio nível e propostos critérios que justificam as decisões tomadas na elaboração da sintaxe, sobre os tipos de variáveis, ações semânticas e geração de código.

Paralelamente são apresentadas a estrutura do compilador para a linguagem-usando na análise sintática o método de matriz de transição - e a descrição de algumas rotinas semânticas.

Finalmente é feita uma avaliação da linguagem comparando-se algoritmos escritos em MIXAL, por Knuth, com as traduções para MIXAL dos mesmos algoritmos escritos em PLMIX, e julgando as diferenças encontradas no código.

A B S T R A C T

A language PLMIX for the hypothetical computer MIX created by D.E. Knuth, was defined just only as a means to specify the criteria for designing medium level programming languages.

Besides presenting a definition of medium level programming languages we propose criteria that justify the specific form taken by the syntax of PLMIX, its variable types, semantic actions and code generation.

We present also the structure of the compiler and the description of some semantic routines. The compiler uses the transition matrix method for the parsing.

Finaly to evaluate the performance of PLMIX we compare the length of the code generated by MIXAL with the code generated by PLMIX for exactly the same algorithms.

INDICE

CAPÍTULO I - Introdução 1

CAPÍTULO II - Descrição do MIX 5

II.1. A Máquina 5

II.1.1. Memória 5

II.1.2. Registros 6

II.1.3. Indicadores 7

II.1.4. Dispositivos de Entrada e Saída ... 8

II.2. Linguagem de Máquina 9

II.2.1. Formato da Instrução 9

II.2.2. Notação Utilizada 12

II.2.3. Instruções 12

II.2.3.1. Instruções de Carga 12

II.2.3.2. Instruções de Armazenamento 13

II.2.3.3. Instruções Aritméticas .. 14

II.2.3.4. Instruções Imediatas 15

II.2.3.5. Instruções de Comparação. 15

II.2.3.6. Instruções de Desvio 16

II.2.3.7. Instruções de Entrada e Saída 17

II.2.3.8. Instruções de Converção.. 19

II.2.3.9. Outras Instruções 20

CAPÍTULO III - Decrição da Linguagem PLMIX 23

III.1. Estrutura da Linguagem 23

III.2. Notação para a Sintaxe 25

III.3. Especificação da Linguagem 26

CAPÍTULO IV - Estrutura do Compilador 87

IV.1. Aspectos Gerais 87

IV.2. Analisador Léxico 88

IV.3. Analisador Sintático 95

IV.3.1. Método de Compilação 95

IV.3.2. Gramáticas de Matriz de Transição ... 96

IV.3.2.1. Definição 96

IV.3.3. Determinação da Matriz de Transição.. 99

IV.4. Código Objeto e Algumas Rotinas Semânticas.. 99

IV.4.1. Estrutura dos Dados no Compilador ... 100

VI.4.2. Algoritmo do Compilador 109

IV.4.2.1. Algoritmo de "Parsing" 110

IV.4.2.2. Esquema do Programa 110

IV.4.2.3. Algumas Rotinas Semânticas. 113

IV.5. Tratamento de Erros 130

CAPÍTULO V - Avaliação da Linguagem 135

V.1. Exemplo Número 1 135

V.1.1. Programa T 137

V.1.2. Programa em PLMIX para T 139

V.1.3. Programa em PLMIX Estruturando o Algoritmo 141

V.2. Exemplo Número 2 143

V.2.1. Programa S 143

V.2.2. Programa em PLMIX para S 144

V.2.3. Programa em PLMIX Estruturando o Algoritmo 147

V.3. Exemplo Número 3 149

V.3.1. Programa B 149

V.3.2. Programa em PLMIX para B 150

V.3.3. Programa em PLMIX Estruturando o Algoritmo 153

V.4. Exemplo Número 4 155

V.4.1. Programa A 156

V.4.2. Programa em PLMIX para A 158

V.4.3. Programa em PLMIX Estruturando o Algoritmo 162

CAPÍTULO VI - Conclusões 167

APÊNDICE Nº 1 170

APÊNDICE Nº 2 173

BIBLIOGRAFIA 187

C A P Í T U L O I

I.1. INTRODUÇÃO

Projetos de novas linguagens de programação surgem com a evolução da utilização do computador tanto no aspecto quantitativo como qualitativo. Além disso, há uma variedade de características de arquitetura que fazem com que certas linguagens sejam adequadas ou mais facilmente traduzidas em determinadas máquinas, enquanto que para outras arquiteturas seria desejável uma linguagem com estrutura diferente.

Para o usuário final as linguagens tendem a evoluir para o mais alto nível possível, voltadas para as aplicações desejadas e o mais independentes da máquina.

A programação de "software" básico e de suporte tem sido normalmente feita com linguagem dependente da máquina.

Durante muito tempo a linguagem dependente da máquina utilizada era a linguagem montada. Ela permite ao programador uma gerência sobre memória e registros e código eficiente. Porém a sintaxe usual faz com que se produzam textos longos, de difícil leitura e acompanhamento da lógica e depuração demorada.

A linguagem de alto nível impede este gerenciamento pois o "software" necessário tanto para compilação como para execução dispõe da memória e dos registros de maneira variável, imprevisível ou incontrolável para o programador. Além disso estas linguagens não fornecem, em geral, mecanismos para que o programador atue diretamente sobre a máquina, por exemplo, impedindo ou tornando complexa a introdução de código de linguagem montada no texto do programa.

Com os conceitos de programação estruturada e a definição dos tipos básicos de estruturas necessárias e suficientes para se escrever programas estruturados (Wulf [1]) surgiram projetos de linguagens, que visam permitir a criação destas estruturas básicas de modo simples sem perderem as características da linguagem montada em gerência de memória e registros e eficiência de código gerado.

Chamaremos esta classe de linguagens de médio nível apresentando as seguintes características:

- permitem ao programador a gerência de memória e registros, tal como as linguagens montadas;

- deverão procurar englobar todo o potencial das instruções de máquina, seja em comandos estruturados , funções ou introdução direta do código em linguagem montada;

- sua sintaxe é do tipo da sintaxe das linguagens de alto nível, mais difundidas, com blocos, procedimentos, expressões, instruções condicionais, iterativas, de chaveamento e declaração de tipos;

- os tipos permitidos são os previstos pelo

hardware ou estruturados com simplicidade (sem necessidade de descritores) a partir dos tipos simples;

- o código gerado deve ser o mais eficiente e o mais próximo possível da linguagem montada;

- a tradução dos comandos tem uma sequência de códigos fixa e deverá ser apresentada ao programador.

Cumpre ressaltar que a terminologia linguagem de médio nível não é restrita à linguagens com as características acima. Ela é usada para linguagens cuja utilização da memória e registros é feita sem conhecimento do programador , com a tradução variável dos comandos, etc. Entretanto as características da linguagem não permitem que sejam consideradas de alto nível.

O objetivo deste trabalho é estabelecer critérios para o desenvolvimento de projetos de linguagens de médio nível, levando-se em consideração as notas sobre projetos de linguagens escritas por Hoare [2].

Em vez de se fazer uma enumeração desses critérios foi projetada uma linguagem onde a apresentação das instruções, dos tipos permitidos, as justificativas das decisões tomadas e as explicações complementares caracterizam estes critérios.

A escolha da máquina recaiu sobre o MIX (Knuth [3]) por possuir um jogo de instruções que permitem um projeto amplo, por ser utilizado em cursos de graduação e pós-graduação em várias instituições de ensino e por existir um simulador desenvolvido por Markenzon [6] e implementado no

computador MITRA 15 do Laboratório de Automação de Sistemas e Simulação da COPPE-UFRJ, como parte do projeto de um Laboratório de Ensino de Computação.

CAPÍTULO II

DESCRIÇÃO DO MIX

O MIX é um computador hipotético criado com fins didáticos por Donald E. Knuth [3] e apresentado no livro "Fundamental Algorithms" no capítulo I, seções 1.3.1 e 1.3.2 .

Devido a finalidade proposta o projeto apresenta uma estrutura e uma linguagem simples e de fácil aprendizagem e no entanto poderosa o suficiente para permitir que sejam escritos pequenos programas para a maioria dos algoritmos propostos por Knuth [3~4~5].

II.1. A MÁQUINA

II.1.1. MEMÓRIA

A memória MIX é composta de 4000 palavras, cada uma com 5 bytes mais sinal "+" ou "-".

O byte é a unidade básica de informação sendo capaz de armazenar no mínimo 64 valores e no máximo 100 valores distintos. Esta variação é decorrente da possibilidade do MIX poder ser binário ou decimal. É importante ressaltar que

o programador deve escrever programas que possam ser executa dos em qualquer implementação, isto é, assumindo que o byte não pode representar mais que 64 valores.

Na implementação do simulador existente no MITRA 15 do Laboratório de Sistemas da COPPE-UFRJ, feita por Markenzon [6] assumiu-se o máximo de 64 valores e assim o byte é composto de 6 bits.

II.1.2. REGISTROS

O MIX possui 9 registros sendo que dois com cinco bytes mais sinal (registro A e registro X) e sete com dois bytes mais sinal (registros I1, I2, I3, I4, I5, I6 e registro J).

- Registro A (rA)

É o registro acumulador no qual são efetuadas das operações aritméticas, de deslocamento e conversão.

- Registro X (rX)

Extensão à direita do acumulador. É usado junto com o registro A para formar dez bytes necessários às operações de multiplicação, divisão e conversão, ou para armazenar informações produzidas por deslocamento à direita do registro A.

- Registros I (rI1, rI2, rI3, rI4, rI5 e rI6)

São registros de índice usados geralmente como contadores e para modificar endereços de acesso à memória.

- Registro J (rJ)

Contém o endereço da instrução seguinte a uma instrução de desvio e é usado basicamente para retorno de subrotina.

II.1.3. INDICADORES

São dois os indicadores:

- bit de overflow (OVF)

Modificado pelas instruções aritméticas , "jump on overflow" (JOV) e "jump on no overflow" (JNOV).

- indicador de comparação

Assume valores menor, igual ou maior e é controlado pelas operações de comparação (CMPA, CMPX e CMPi).

II.1.4. DISPOSITIVOS DE ENTRADA E SAÍDA

Os dispositivos dependem dos existentes na máquina onde será feita a simulação. Para a implementação feita por Markenzon [6] temos as seguintes especificações.

- Disco MIX

As especificações do disco são fornecidas por Knuth na página 362 do livro "Sorting and Searching"[5].

Sua capacidade de armazenamento é de vinte milhões de caracteres, distribuídos em duzentos cilindros de vinte trilhas, cada uma com 5.000 caracteres.

As informações transmitidas em operações de entrada e saída são armazenadas em blocos de cem palavras. A referência a um particular bloco é especificada pelo conteúdo dos dois bytes menos significativos de rX.

- Fita MIX

As características da unidade de fita se encontram na página 320 do livro "Sorting and Searching" [5].

A unidade lê e escreve 800 caracteres por polegada de fita, com velocidade de 75 polegadas por segundo. Isto significa que um carater é lido ou escrito cada 1/60 ms. Cada carretel contém 2.400 pés de fita.

As informações são armazenadas por blocos na fita, sendo que cada instrução de leitura ou impressão acarreta a transmissão de um único bloco. Cada bloco tem 100

palavras e o intervalo entre os blocos é de 480 caracteres.

- Teletipo

Definida conforme as características do modelo ASR33 [7] que apresenta velocidade de impressão de 10 caracteres por segundo. Note-se que cada registro possui 70 caracteres, equivalente a 14 palavras MIX.

A transmissão é feita caracter a caracter, onde cada byte representa um caracter.

- Leitora de Cartões

Definida conforme as especificações de modelo LC300 [7] que lê cartões standard de 80 colunas (16 palavras MIX) à velocidade de 300 cartões por minuto.

A transmissão é feita caracter a caracter.

II.2. LINGUAGEM DE MÁQUINA

II.2.1. FORMATO DA INSTRUÇÃO

A maioria das instruções MIX permite ao programador um acesso direto à partes (bytes) da palavra e para permitir identificar os bytes a serem acessados eles são

numerados da seguinte forma:

0	1	2	3	4	5
±	byte	byte	byte	byte	byte

As palavras que contêm instruções são codificadas segundo o seguinte formato:

0	1	2	3	4	5
±	AA		I	F	C

onde:

- ± AA : endereço.

A instrução MIX utiliza endereçamento direto. O sinal da palavra pertence ao endereço.

- I : especificação de índice

É usado para alterar o valor do endereço. Se I = 0, ± AA é usado sem modificação. Os outros valores permitidos variam de 1 a 6 correspondendo aos registros de índice existentes na máquina. Neste caso o valor do registro de índice indicado é somado algebricamente ao valor de ± AA sendo o resultado usado como endereço. Este processo de indexa

ção ocorre para qualquer instrução.

- F : modificação do código de instrução

Em geral F representa uma especificação de campo (L:R), onde L é o número do byte mais à esquerda e R do byte mais à direita. Para permitir a representação desta especificação F é calculado através da fórmula 8*L + R.

Algumas instruções utilizam o campo F com outro significado tal como número do dispositivo de entrada e saída, modificador do código de desvio incondicional, tornando-o condicional, tipo do deslocamento a ser efetuado, etc.

- C : código de operação

Na descrição da linguagem PLMIX algumas vezes serão referenciadas as instruções de máquina correspondentes às instruções PLMIX. A representação da instrução será feita da forma

OP ± AA, I (F)

onde OP é o mneumônico do código de instrução (c). Admite-se opcionalmente as seguintes alterações:

- se I = 0, pode ser omitido
- se F é a especificação padrão da instrução, pode ser omitido.

II.2.2. NOTAÇÃO UTILIZADA

rR : registros quaisquer

rAX: registros rA e rX formando 10 bytes

i : número de 1 a 6

M : endereço resultante após indexação

C(M): conteúdo da posição de memória M

campo: especificação de bytes de uma palavra

V : valor de um campo específico de C(M)

PC : endereço da próxima instrução

A ← B: A recebe o valor de B.

II.2.3. INSTRUÇÕES

A tabela com os códigos e as especificações padrão de F está no Apêndice número 1.

II.2.3.1. INSTRUÇÕES DE CARGA

"load rR": LDA, LDX, LDi

ação : rR ← C(M)

"load rR negative": LDAN, LDXN, LDiN

ação : rR ← - C(M)

Em todas as operações onde um campo parcial é utilizado o sinal é usado se ele faz parte do campo (especificação (0:k), $0 \leq k \leq 5$), caso contrário assume-se sinal +. O campo é deslocado para a direita do registro onde será carregado.

Na atribuição aos registros Ii como este só tem 2 bytes, se o campo englobar mais de dois bytes, os outros deverão ser necessariamente iguais a zero. Caso contrário a instrução será considerada indefinida.

II.2.3.2. INSTRUÇÕES DE ARMAZENAMENTO

"store rR": STA, STX, STi, STJ

ação: M ← rR

"store zero": STZ

ação: M ← 0

Nas instruções de armazenamento o número de bytes do campo desejado é tomado do lado direito do registro e inserido na posição especificada pelo campo.

Os registros rIi se comportam como se tivessem 5 bytes mais sinal, porém com os bytes 1, 2 e 3 iguais a zero. Para rJ o sinal é sempre positivo e só são alterados os bytes 1 e 2, pois eles representam um endereço.

II.2.3.3. INSTRUÇÕES ARITMÉTICAS

ADD (adição)

ação: rA ← rA + V

Se o resultado precisar de mais de 5 bytes para sua representação, OVF é ligado e rA armazena os cinco bytes à direita do resultado e seu sinal.

SUB (subtração)

ação: rA ← rA - V

Poderá ocorrer transbordo e será tratado como na adição.

MUL (multiplicação)

ação: rAX ← rA * V

Os sinais de rA e rX recebem o sinal algébrico do produto, isto é, "+" se rA e rV são do mesmo sinal e "-" caso contrário.

DIV (divisão)

ação: rA ← rA/V ; rX ← (rAX) MOD V

O sinal de rAX é o de rA. Se V = 0 ou o resultado é maior que 5 bytes rA e rX ficam com valores indefinidos e OVF é ligado. Caso contrário rA recebe o quociente

e o sinal algébrico da operação, rX é substituído pelo resto e o sinal de A (de antes da operação).

II.2.3.4. INSTRUÇÕES IMEDIATAS

"enter rR": ENTA, ENTX, ENTi

ação: $rR \leftarrow M$

Se $M = 0$ o sinal da instrução é carregado.

"enter negative rR": ENNA, ENNX, ENNi

ação: $rR \leftarrow - M$

"increment rR": INCA, INCX, INCi

ação: $rR \leftarrow rR + M$

"decrement rR": DECA, DECX, DECi

ação: $rR \leftarrow rR - M$

II.2.3.5. INSTRUÇÕES DE COMPARAÇÃO

"compare rR": CMPA, CMPX, CMPi

ação: comparar um campo especificado de rR com o mesmo campo de C(M).

Se $rR > C(M)$ então $CI \leftarrow$ MAIOR

Se $rR < C(M)$ então $CI \leftarrow$ MENOR

Se rR = C(M) então CI ← IGUAL

Uma comparação com campo (0:0) sempre fornece resultado IGUAL pois + 0 é igual a - 0.

II.2.3.6. INSTRUÇÕES DE DESVIO

O registro J é alterado toda vez que ocorrer um desvio (exceto na instrução JSJ), recebendo o endereço da instrução seguinte aquela onde se deu o desvio.

JMP (jump)
ação: rJ ← PC;
PC ← M;

JSJ (jump save J)
ação: rJ inalterado
PC ← M

JOV (jump on overflow)
ação: se OVF = true então rJ ← PC
PC ← M
senão;

JNOV (jump on no overflow)
ação: se OVF = false então rJ ← PC
PC ← M
senão OVF ← false

JL, JE, JG, JGE, JNE, JLE (jump on less, equal, greater, non-less, non-equal, non-greater).

ação: o desvio ocorre se o indicador de comparação apresenta a condição indicada.

JAN, JAZ, JAP, JANN, JANZ, JANP (jump A negative, zero, positive, nonnegative , nonzero, nonpositive).

JXN, JXZ, JXP, JXNN, JXNZ, JXNP (jump X negative, zero, positive, nonnegative, nonzero, nonpositive).

JiN, JiZ, JiP, JiNN, JiNZ, JiNP (jump Ii negative, zero, positive, nonnegative, nonzero, nonpositive).

ação: o desvio ocorre se o conteúdo de rR satisfizer a condição, caso contrário nada ocorre.

II.2.3.7. INSTRUÇÕES DE ENTRADA E SAÍDA

IN (entrada)

ação: é iniciada a transferência de informação da unidade especificada para posições consecutivas da memória a partir de M. O número de células da memória afetada é igual ao tamanho do bloco desta unidade.

OUT (saída)

ação: são transferidas informações a partir da posição M da memória para o dispositivo indicado.

Para IN e para OUT a máquina aguarda a liberação do dispositivo se ele estiver executando uma operação. A transferência de informação dura um intervalo de tempo que depende da velocidade da unidade que está sendo utilizada. Assim não é aconselhável fazer referências, sem precauções, às posições de memória que estão sendo alteradas, até que a operação seja completada.

IOC (controle de entrada e saída)

ação: uma operação de controle é executada, dependendo do dispositivo:

Fita magnética: Se M = 0 a fita é reenrolada.

Se M < 0 a fita volta M registros ou volta para o início da fita, o que ocorrer primeiro.

Se M > 0 a fita avança (a operação será ignorada e a unidade suspensa se avançar mais do que o último registro escrito na fita).

Disco: M deve ser zero. Posiciona o dispositivo de acordo com rX para economizar tempo da próxima instrução IN ou OUT.

JRED (jump ready)

ação: o desvio ocorre se a unidade especificada terminou a operação iniciada por IN, OUT ou IOC.

JBUS (jump busy)

ação: o desvio ocorre se a unidade requerida não estiver liberada.

II.2.3.8. INSTRUÇÕES DE CONVERSÃO

NUM (conversão a numérico)

ação: converte código de caracteres para código numérico. rAX é considerado um número de 10 bytes escrito em caracteres que são convertidos a um número decimal armazenado em rA. O valor de rX e o sinal de rA permanecem inalterados. É possível ocorrer tansbordo e neste caso o resto é armazenado.

CHAR (conversão a caracteres)

ação: converte código numérico a código de caracteres, necessário para a saída em cartões ou teletipo. O valor contido em A é convertido a um número decimal de 10 bytes em rAX. Os sinais de rA e rX não sofrem alterações.

II.2.3.9. OUTRAS INSTRUÇÕES

MOVE

ação: transfere o número de palavras especificadas por F começando da posição de memória M para a posição de memória dada pelo conteúdo de rIl. É transferida uma palavra de cada vez e rIl é incrementado do valor de F ao fim da operação. Se F = 0 nada acontece.

SLA, SRA, SLAX, SRAX, SLC, SRC (shift left A, shift right A, shift left AX, shift right AX, shift left AX circular, shift right AX circular).

ação: rR é movimentado o número de bytes especificado por M.

Os sinais de rA e rX não são afetados nas operações. SLA e SRA não afetam rX, os outros operadores alteram rX. Nas instruções SLA, SRA, SLAX e SRAX bytes "desaparecem" de um lado enquanto zeros são colocados do outro lado . SLC e SRC efetuam deslocamentos circulares, isto é, os bytes que "desaparecem" de um lado, "surgem" do outro.

NOP (no operation)

ação: nada ocorre

HLT (halt)

ação: a máquina para

As instruções abaixo não constam da definição original. Foram acrescentadas no simulador projetado por Markenzon [6] visando facilitar a depuração e a medida do desempenho dos programas. Para estas instruções o relógio do simulador não é incrementado, não afetando assim a medida do tempo de processamento.

Para maiores detalhes sobre estas instruções ver páginas 20, 64 e 65 da referência [6].

CLOC

ação: o relógio MIX é impresso

LOOP

ação: o campo (18 bits) formado pelas partes AA e I da instrução é incrementado de 1 e o novo valor guardado no mesmo campo desta instrução.

OUTL

ação: imprime o campo formado pelas partes AA e I do endereço mencionado.

TRCE (trace)

ação: imprime a partir deste ponto, em cada instrução: a instrução corrente, e os conteúdos dos registros rA, rX, rI1, rI2, rI3, rI4, rI5, rI6.

NTRC (no trace)

ação: desligar o rastreamento, se ligado. Caso contrá
rio nada ocorre.

C A P Í T U L O III

DESCRIÇÃO DA LINGUAGEM PLMIX

III.1. ESTRUTURA DA LINGUAGEM

PLMIX é uma linguagem de médio nível com as características apresentadas na introdução (páginas 2 e 3) e com uma sintaxe semelhante a do ALGOL 60 [21].

Um programa PLMIX é um bloco, isto é, é um conjunto de instruções, precedidas por declarações e encerradas entre 'begin' e 'end'.

As declarações fornecem informações sobre as variáveis e estabelecem a alocação de memória. A alocação é estática e a semântica de cada declaração especifica como é feita a ocupação de memória para aquela variável, permitindo assim ao programador uma gerência sobre os endereços.

Os tipos de variáveis são os existentes no 'hardware' e alguns facilmente estruturados a partir destes. Esta exigência é para evitar que seja necesário o uso de descritores e instruções extras introduzidas no código para cálculo de acesso a endereço. Deste modo é evitado o uso de posições de memória pelo compilador sem o conhecimento do

programador.

Os procedimentos podem ou não ter parâmetros e o corpo é um bloco ou um comando composto ou apenas uma instrução. Quando houver declarações de variáveis estas são locais ao procedimento, havendo verificação de escopo . Não é permitida a declaração de procedimento no corpo de outra, porém é permitida a chamada desde que esta já tenha sido declarada. A passagem dos parâmetros é por valor -resultado ou por valor [24]. Na chamada os parâmetros formais são inicializados com os valores dos parâmetros reais correspondentes. No retorno os parâmetros reais do tipo valor-resultado, recebem o valor do parâmetro formal correspondente. Os demais permanecem inalterados.

As instruções podem ser compostas, encerradas entre 'begin' e 'end' ou '[' e ']'. Algumas refletem diretamente instruções da máquina, como por exemplo, as instruções de deslocamento, aritméticas, transferência, conversão. Outras são compostas por várias instruções, como os comandos iterativos, condicionais, de chaveamento (CASE), porém é definida na semântica uma tradução fixa de modo que o programador possa saber exatamente o código gerado por determinada instrução.

Os procedimentos podem ser chamados de qualquer ponto do programa.

A atribuição de expressões às variáveis de memória é feita indicando-se na sintaxe em que registro será efetuado o cálculo, para evitar que sejam tomadas deci

sões pelo compilador. É feita exceção apenas para cálculo de expressão em uma condição. Porém a ausência da especificação do registro indica que será utilizado o registro acumulador (rA).

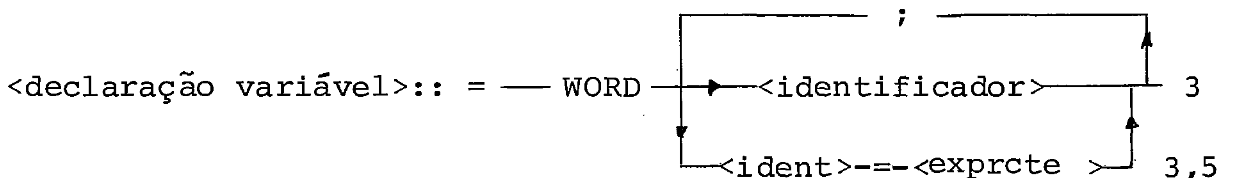
As instruções introduzidas por Markenzon [6] no simulador também tem declarações e instruções correspondentes em PLMIX.

Não há necessidade da introdução de código montado no programa pois todas as instruções de máquina do MIX tem, direta ou indiretamente, equivalente em PLMIX.

III.2. NOTAÇÃO PARA A SINTAXE

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

O exemplo abaixo utiliza todas as regras da representação.



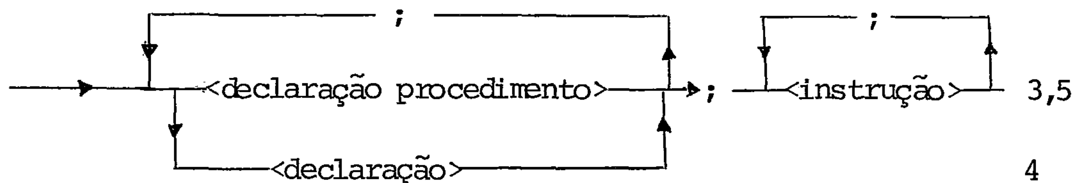
- as categorias sintáticas são as palavras delimitadas por "<" e ">".
- as palavras reservadas são escritas com letras maiúsculas.
- o sinal :: = é lido como "é definido por".
- o sinal → é lido como "é seguido por".
- o sinal ↑ é lido como "ou seguido por".
- o(s) número(s) ao lado de cada linha indica(m) o(s) número (s) da(s) definição (ões) da(s) categoria(s) sintática (s) referenciada(s) naquela linha.

III.3. ESPECIFICAÇÃO DA LINGUAGEM

1 <programa>:: =

—BEGIN ——<lista>——END 2

2 <lista>:: =



O corpo do programa é constituído de declarações separadas por ";" e seguidas por instruções também separadas por ";".

As declarações dão informações sobre as características das variáveis que serão utilizadas no programa - tipo, área de memória associada, valor inicial, etc. É feita uma associação entre o identificador e uma ou mais posições de memória, um registro, um grupo de instruções, uma tabela de endereços, etc.

Um identificador só pode aparecer uma vez nas declarações "globais". Dentro da declaração de procedimento é permitido declarar variáveis com um nome já declarado , pois elas são locais ao procedimento.

Todas as variáveis devem ser declaradas.

A instrução é a unidade de operação da linguagem. O conjunto das instruções é a representação do algoritmo proposto para resolução de um problema.

Normalmente as instruções são executadas sequencialmente, podendo esta ordem ser modificada por um comando explícito de desvio ou uma alteração do resultado de um teste de condição em um comando condicional. Um desvio incondicional é feito indicando-se o rótulo da instrução a ser executada após o desvio.

3 <declaração procedimento>:: =

→ PROCEDURE → <identificador> → ───────────────────── → ; <corpoproc> 6,7
└→ (-<lista param. formais>-) ┘ 8

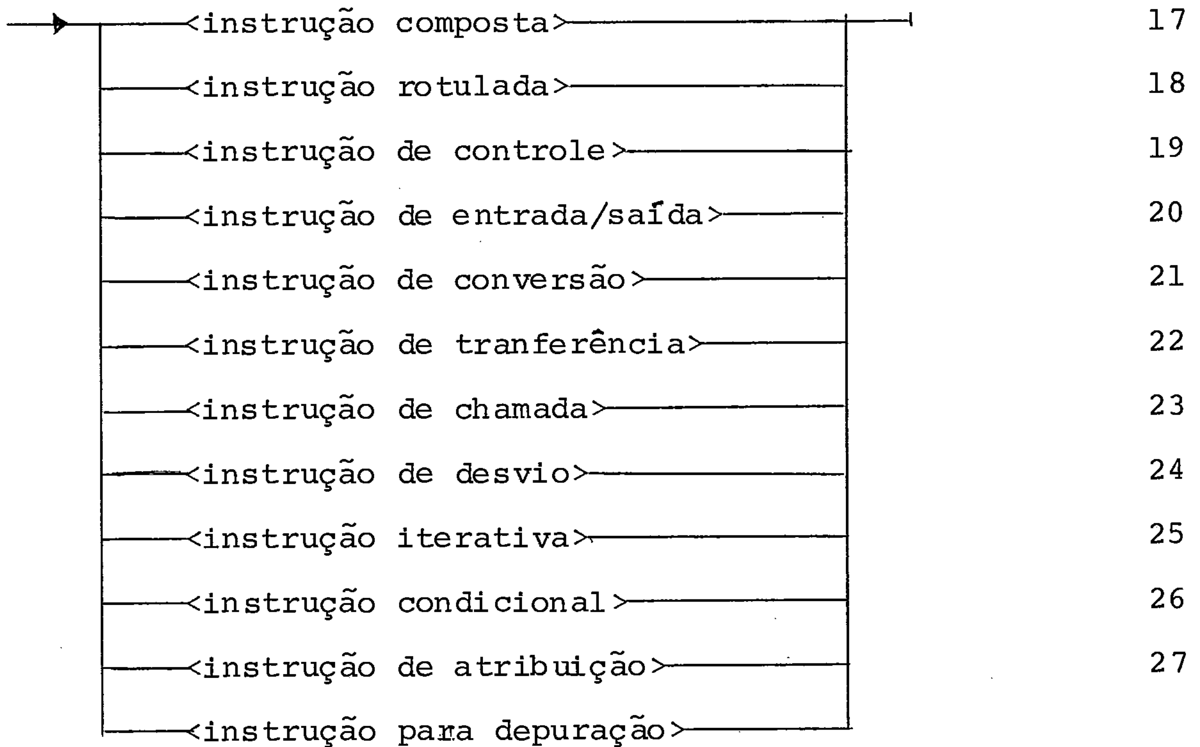
A declaração de procedimento associa um identificador a uma instrução ou a uma sequência de instruções. O identificador pelo qual o procedimento é referenciado é o que aparece na declaração.

Não existe o procedimento tipificado (função). Isto é devido ao fato de, para o cálculo da função que está sendo chamada em uma expressão, haver a possibilidade de estarem sendo usados e eventualmente alterados, sem estar visível para o programador, os mesmos registros que aparecem nesta expressão. Para evitar estes problemas seria necessário guardar os valores destes registros em variáveis temporárias antes de executar a função e restaurá-los após o término do cálculo. Este tipo de solução não se enquadra nas características propostas para linguagem de médio nível, por não ser permitido o uso de temporárias pelo compilador.

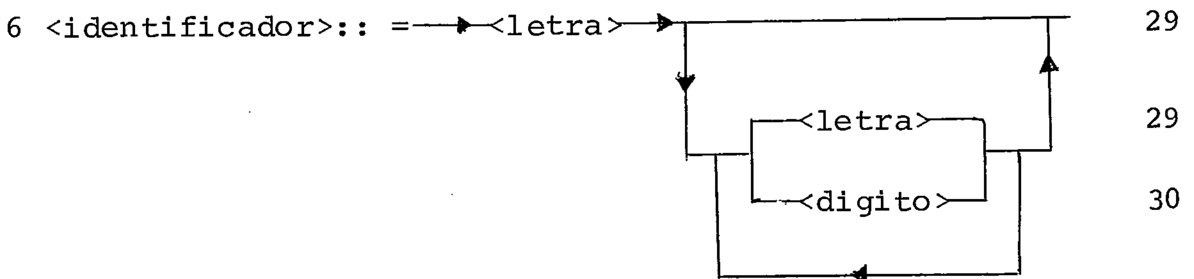
4 <declaração>: : =

<declaração de constante>	9
<declaração de variável simples>	10
<declaração de record>	11
<declaração de variável equate>	12
<declaração de vetor>	13
<declaração de rótulo>	14
<declaração de tabela de desvios>	15
<declaração de contadores>	16

5 <instrução>:: =

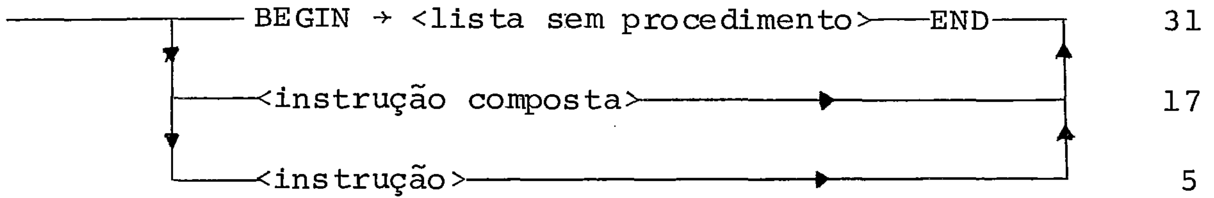


6 <identificador>:: =



O identificador deve começar por uma letra e pode ter até 10 caracteres, entre letras e digitos.

7 <corpo do procedimento>: : =



Um procedimento pode ou não conter declaração de variáveis. Se houver declarações estas serão locais ao procedimento e cada referência à variável é testada para verificação do escopo. A alocação destas variáveis é estática.

A alocação de memória por um procedimento é feita do seguinte modo:

1. uma palavra para cada parâmetro formal
2. área para as variáveis locais (se houver)
3. uma instrução para aguardar o endereço de retorno
4. instruções do procedimento
5. instrução de desvio para a instrução após à chamada ao procedimento.

A necessidade da reserva de palavras para os parâmetros é devido ao método de transmissão de parâmetros: a) na chamada ao procedimento o parâmetro real é calculado e seu valor é o valor inicial do parâmetro formal correspondente; b) após a execução os parâmetros reais correspondentes aos parâmetros formais de tipo VALUE não são alterados, os demais

recebem o valor final dos parâmetros formais correspondentes.

Deste modo a transmissão dos parâmetros pode ser "by value" ou "by value-result". O formalismo das definições destes métodos se encontra em Pratt [24] nas seções 6.9 e 6.10.

Estes métodos foram adotados devido ao fato do MIX não possuir registros de indireção. Neste caso o uso de passagem de parâmetros "by reference" se tornaria excessivamente oneroso, pois a cada acesso ao parâmetro teríamos necessariamente uma instrução de carga. A transmissão de parâmetros "by name" é claramente incompatível com os objetivos propostos para linguagens de médio nível, além de apresentar problema semelhante.

De qualquer forma, o uso de subprogramas com parâmetros é claramente desaconselhável, devendo o programador utilizar os registros da máquina para esse fim.

Não é permitida a declaração de procedimento dentro do corpo de outro procedimento. Porém é permitida a chamada a um outro procedimento previamente declarado, não se permitindo recursão. A responsabilidade por evitar recursão fica a cargo do programador.

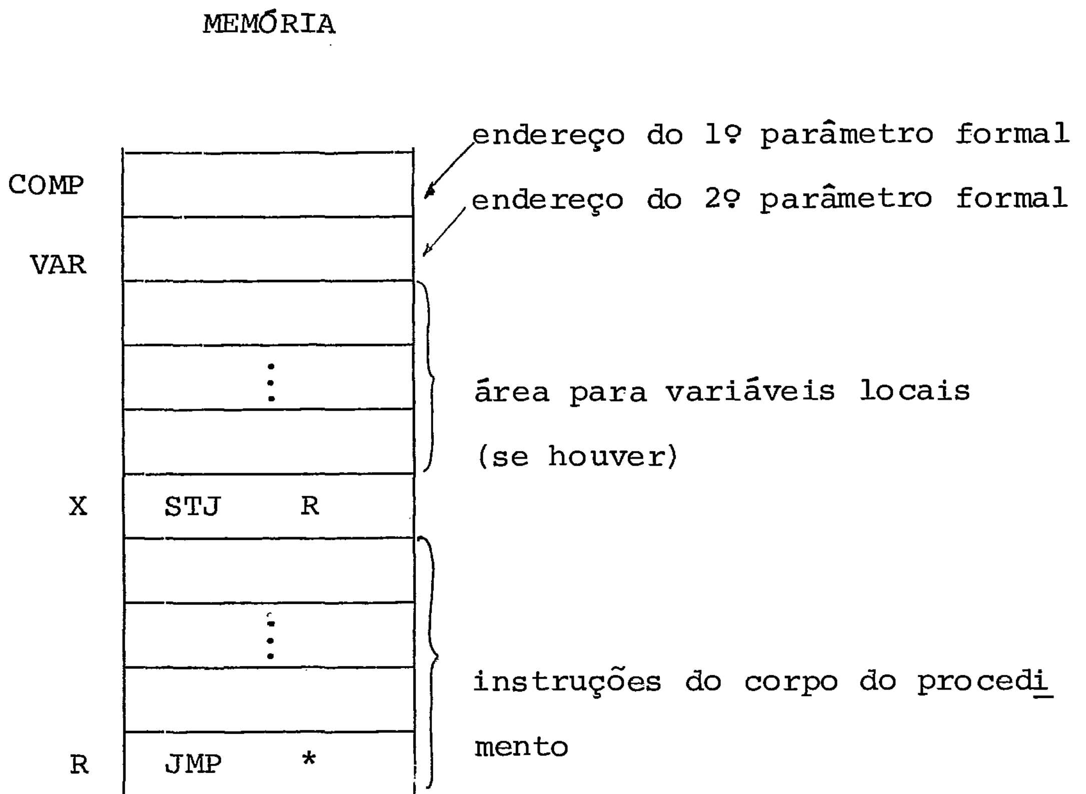
Exemplo:

```
PROCEDURE TESTE (VALUE WORD COMP; WORD VAR);
     BEGIN
      <lista sem procedimento>
     END
```

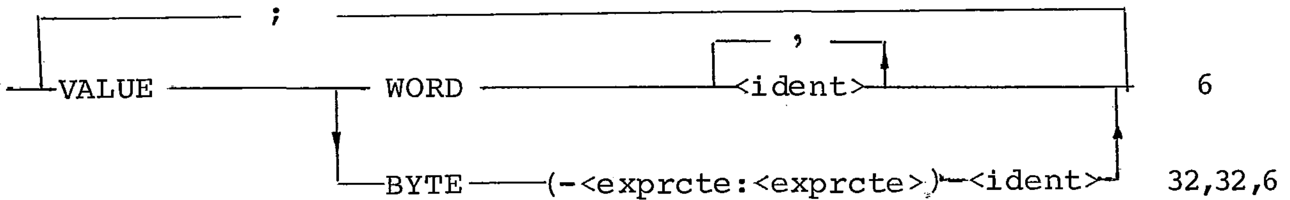
Após a compilação teremos:

TESTE na tabela de símbolos aponta para X.

MEMÓRIA



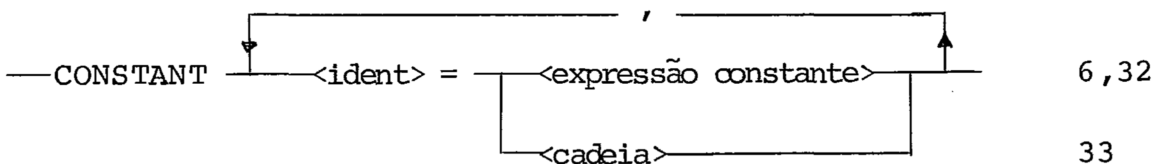
8 <lista dos parâmetros formais>:: =



A palavra VALUE indica transmissão "by value". O parâmetro sem esta especificação é transmitido "by value-result".

Não foi permitido o uso de vetores como parâmetros pois, com o método de transmissão adotado, significaria incentivar desperdício de memória em casos que poderiam ser solucionados de forma mais economica através da utilização adequada de registros, o que acontece com grande frequência.

9 <declaração de constante>:: =



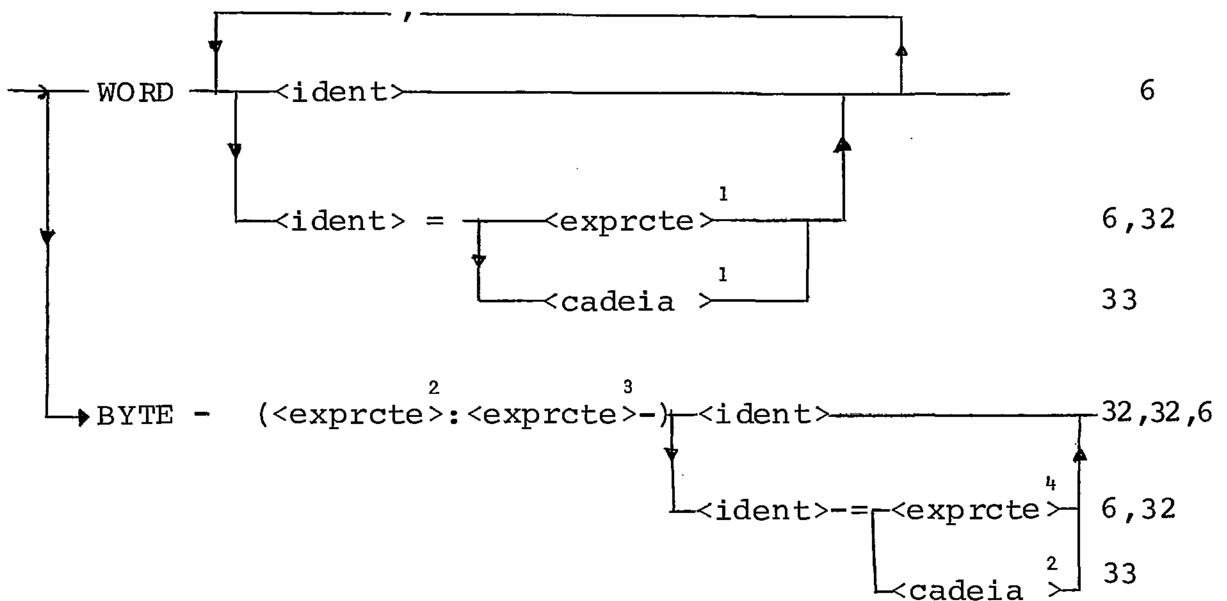
A declaração de constante associa um identificador a um valor, sem alocação de memória. A inicialização de constante obedece às mesmas regras de inicialização de variável tipo WORD que serão vistas em (10).

Exemplo

CONSTANT DEZ = 10, PRINTER = 18;

A compilação desta declaração atribui valor ao identificador na tabela de símbolos e equivale à pseudo-instrução "CON" do assembler MIX.

10 <declaração de variável simples>:: =



A declaração de variável associa um identificador a uma posição de memória podendo ou não ser inicializada.

A variável tipo WORD ocupa os 5 bytes e mais o byte de sinal.

Se a <exprcte>[1] necessitar de mais de 5 by

tes para sua representação o compilador envia uma mensagem de erro. A inicialização com uma cadeia é feita alocando-se cada caracter em um byte da esquerda para a direita. Se a cadeia tiver mais de 5 caracteres só serão utilizados os 5 primeiros. Se tiver menos de 5, os bytes não inicializados ficam com valores indefinidos.

A variável simples tipo BYTE ocupa uma posição de memória mas somente um ou alguns bytes contíguos da palavra estão associados ao identificador. A <exprcte>^{2} indica o byte mais à esquerda e <exprcte>^{3}, o byte mais à direita. Assim devemos ter a relação <exprcte>^{2} $\leq$ <exprcte>^{3} conforme a numeração dos bytes definida em III.2.1.

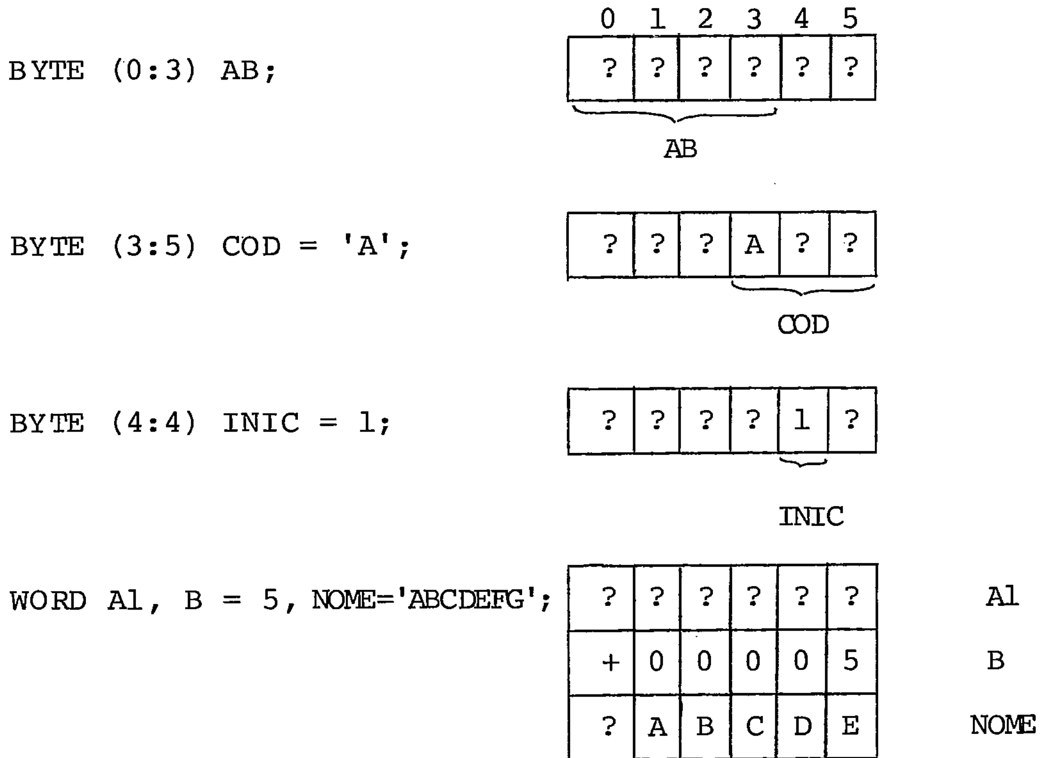
Na inicialização de uma variável BYTE somente os bytes especificados recebem o valor a ser atribuído ficando os outros bytes com valores indefinidos.

O compilador envia mensagem de erro para os seguintes casos: a) tentativa de atribuir um número com sinal a uma variável BYTE onde o byte de sinal (byte Ø) não faz parte da especificação do campo; b) inicializar somente o byte Ø; c) inicializar com um número cuja representação exige um número de bytes maior que o campo.

O byte Ø da palavra de memória ficará com valor indefinido se a inicialização for com uma cadeia.

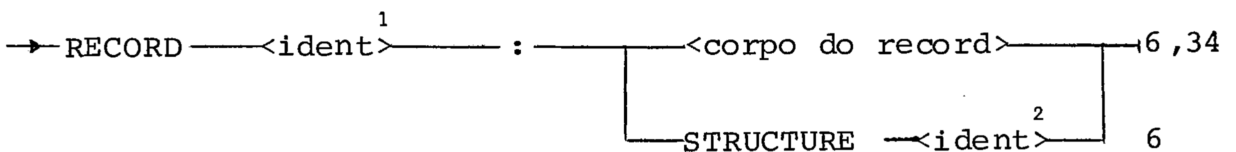
O conteúdo de uma posição de memória não inicializada é indefinido.

Exemplo:



? = valor indefinido

11 <declaração de record>:: =



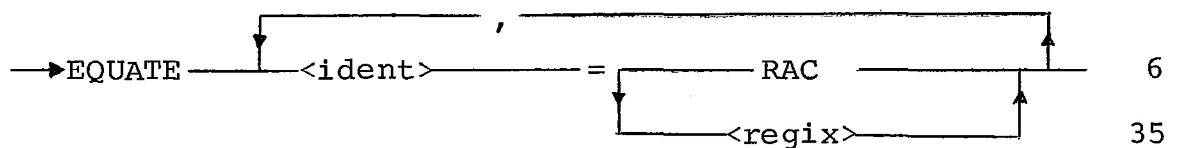
Um record associa uma ou mais palavras a um identificador. Cada uma dessas palavras terá um identificador que permite seu acesso individual. Cada palavra poderá ainda ter bytes contíguos referenciados por outros identificadores. Esta hierarquização será explicada em (34).

A cláusula STRUCTURE permite a criação de um novo record com as mesmas características do record de nome <ident>[2]. Assim para haver uma distinção dos subcampos do record, cada um terá seu nome composto: <ident.record>.<ident nível>* (* será visto em (34)).

Exemplos:

```
RECORD      NO:
    .1      WORD  INFO
    .1      WORD  PONT
            .2 BYTE (1:2) ESQ
            .2 BYTE (3:4) DIR
RECORD      NO1: STRUCTURE NO
```

12 <declaração de variável equate>: : =

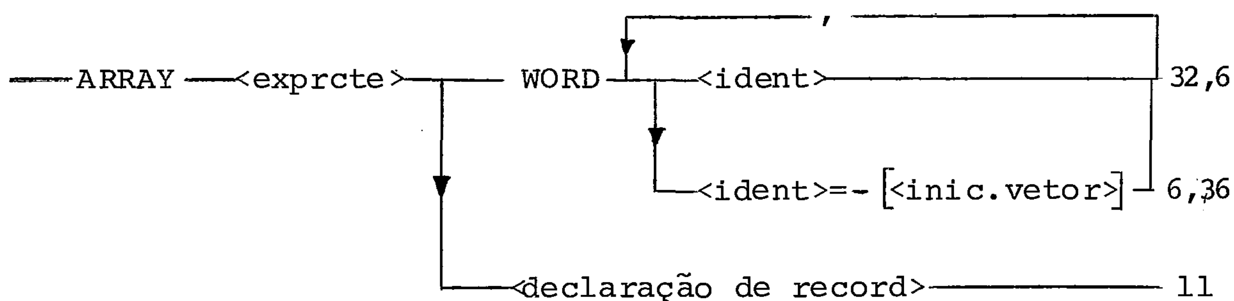


A declaração de equate associa um identificador a um registro. Este identificador não aloca memória. Na compilação qualquer referência a uma variável deste tipo é tratada como uma referência ao registro associado.

Exemplo:

EQUATE INDICE = RI1, ACUMULADOR = RAC

13 <declaração de vetor>:: =



Os arranjos são unidimensionais, com limite inferior igual a zero (0) e limite superior igual ao valor da <exprcte> menos 1.

A restrição a arranjos unidimensionais obedece à filosofia da linguagem de não utilizar palavras da memória sem expresso conhecimento do programador.

O uso de arranjos multidimensionais implicaria, necessariamente, em utilizar memória para descritores e/ou gerar instruções adicionais para acesso, além de implicar obviamente em problemas de otimização de código.

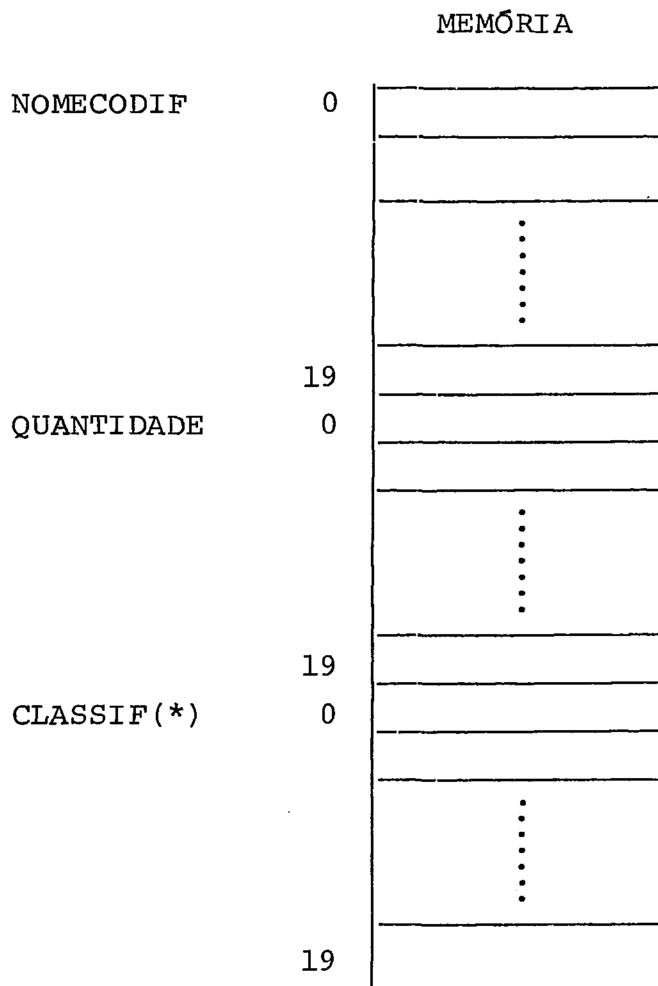
Se o número de palavras a serem inicializadas for menor que o alocado pelo vetor, as palavras não inicializadas tem valor indefinido. Se o número for maior será enviada uma mensagem de erro.

Os arranjos de tipo record são armazenados

de forma a evitar o uso de descritores e geração de instruções para cálculo de acesso. Cada nível é considerado como um vetor em separado. Deste modo o acesso à palavra de índice I de um determinado nível é feito atribuindo-se I a um registro de índice e usando como endereço base o endereço do nível. Temos assim tantos arranjos quanto o número de palavras da estrutura.

Exemplos:

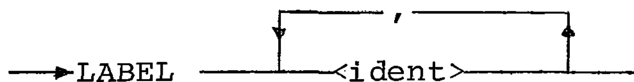
```
CONSTANT    VINTE = 20
ARRAY       VINTE    WORD  ITENS
ARRAY       8        WORD  COD = [1,*2[10],-16,'A']

ARRAY       20      RECORD   MATERIAL:
                  .1 WORD      NOMECODIF
                  .1 WORD      QUANTIDADE
                  .1 WORD      CLASSIF
                     .2  BYTE (1:2) TIPO
                     .2  BYTE (3:5) ESPECIE
```



(*) o endereço de CLASSIF é o mesmo de TIPO e ESPECIE.

Alocação de memória para a declaração do vetor do record.

14 <declaração de rótulo>:: =



6

Os rótulos associam um identificador a uma posição de memória que contém uma instrução. Eles são usados em instruções de desvio incondicional (GOTO).

Exemplo:

LABEL AQUI, FIM.

15 <declaração de tabela de desvios>:: =

→ CASE ——— <ident> ——— : ——— <exprcte> ——— 6,32

Para utilizarmos a instrução de CASE-OF é necessário dispormos de uma tabela com os endereços de cada uma das instruções correspondentes a cada um dos valores do desvio.

Como toda utilização de memória deve ser alocada pelo programador é necessário que ele indique ao compilador a reserva de área necessária para a tabela dos endereços.

A <exprcte> indica o número de opções dentro do CASE-OF porém é utilizada uma palavra a mais. Sendo EØ, E1,... E_{n-1} os endereços de desvio para a primeira, segunda , ..., enésima instrução, E_n é o endereço da primeira instrução após o CASE-OF.

Cada instrução CASE-OF deve ter uma declaração associada.

Exemplo:

CASE CASO1 : 10

16 <declaração de contadores>:: =

6

Um contador é usado como um rótulo e pode ser referenciado como tal.

Sua função é computar quantas vezes foi executada a instrução à qual ele está associado.

Ao encontrar um contador o compilador gera a instrução LOOP antes da instrução a ser controlada. A instrução tem um formato diferente das instruções do "assembler" MIX. O campo ± AA é integrado ao campo I, formando um campo com 3 bytes, que é inicializado com zero. Durante a execução do programa a cada passagem pela instrução é incrementado o campo do contador de 1. Esta instrução não influencia no tempo do relógio simulado.

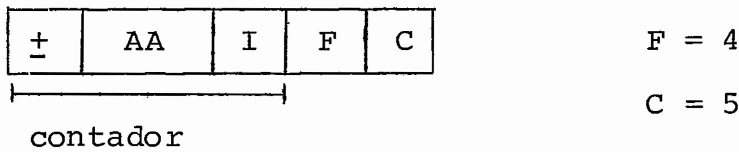
A impressão deste contador através da instrução OUTCOUNTER permite descobrir, por exemplo, quantas vezes foi executado determinado laço de iteração.

Esta facilidade não faz parte da definição do MIX feita por Knuth [3]. Ela foi implementada por Markenzon [6] no simulador do MIX.

Exemplo:

```
COUNTER    LOOP1, LOOP2
```

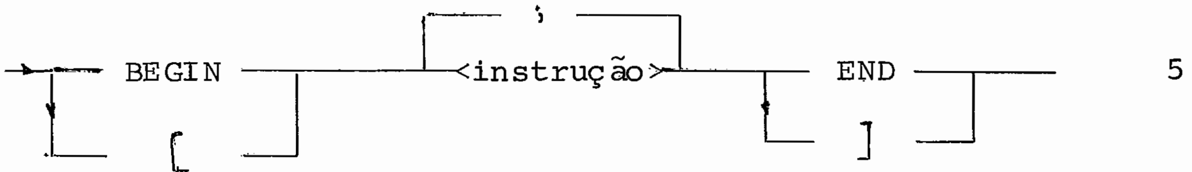
formato da pseudo instrução



F = 4

C = 5

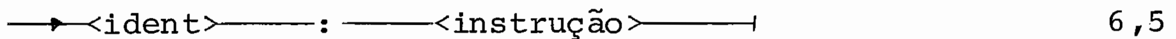
17 <instrução composta>:: =



5

A instrução composta reúne uma ou mais instruções entre delimitadores BEGIN-END ou [-], de modo que elas possam ser tratadas como uma única instrução.

18 <instrução rotulada>:: =

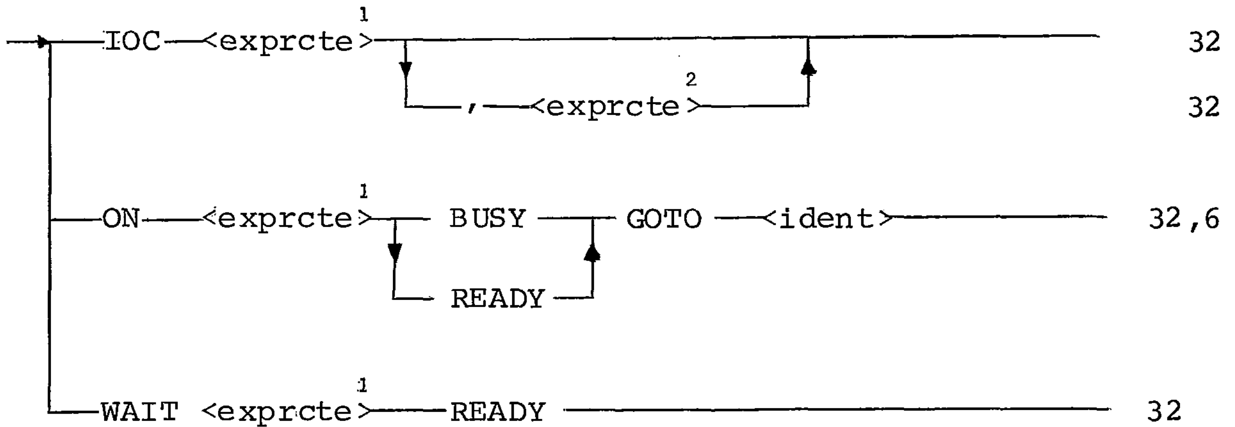


6,5

A instrução rotulada permite que uma instrução possa ser referenciada e que ocorra um desvio da sequência de execução das instruções através de um desvio incondicional.

Se <ident> for um contador, a cada passagem por <instrução>, durante a execução do programa, o contador será incrementado de 1.

19 <instrução de controle>:: =



A instrução IOC indica algumas operações de controle sobre determinado periférico. <exprcte>[1] indica o periférico a ser tratado e <exprcte>[2] informa sobre a operação a ser efetuada. A tradução deste comando é a instrução assem bler IOC com ± AA = <exprcte>[2] e F = <exprcte>[1].

A opção ON faz com que ocorra um desvio para a instrução de rótulo <ident> dependendo do estado do periférico: livre ou ocupado.

A opção WAIT representa uma espera já que é feito um desvio para a mesma posição do teste. Logo a condição do desvio deve ser trocada.

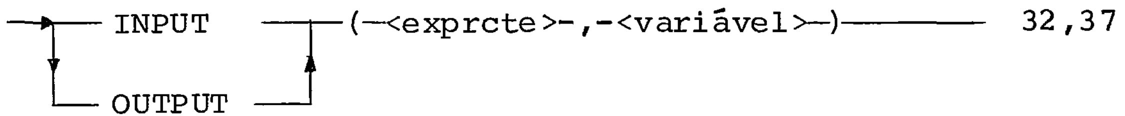
ON e WAIT geram as mesmas instruções JBUS e JRED, diferindo apenas os endereços dos desvios.

Para maiores detalhes ler a seção II.2.3.7.

Exemplos:

PLMIX	MIXAL
IOC 3, -1	IOC -1 (3)
IOC 18	IOC 0 (18)
ON READER READY GOTO VOLTA	JRED 1000(16)
(1000 é o endereço de volta)	
WAIT PRINTER READY	JBUS 500 (18)
(500 é o endereço da instrução JBUS)	

20 <instrução de entrada/saída>:: =



<exprcte> é o periférico onde ocorrerá a operação de entrada/saída. <variável> é nome da área que funciona como "buffer".

Maiores detalhes sobre as operações de entrada/saída estão na seção II.2.3.7.

Exemplos:

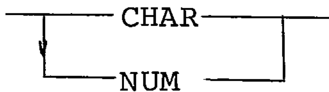
INPUT (6, DADOS [RI1]) IN 1000 (6)

(1000 endereço de DADOS [RI1])

OUTPUT (PRINTER,RESULT [0,RI3]) OUT 2000 (18)

(2000 endereço de RESULT [0])

21 <instrução de conversão>:: =

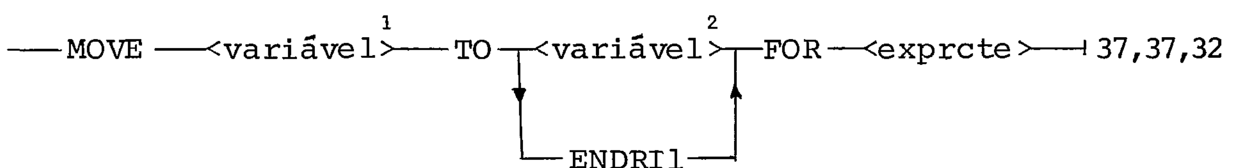


CHAR converte código numérico (em rA) para código caracter com o resultado em rA e rX.

NUM converte 10 bytes (rA e rX) de código caracter para código numérico ficando o valor resultante em rA.

Os detalhes destas operações se encontram na seção II.2.3.8.

22 <instrução de transferência>:: =



A instrução de transferência permite que sejam copiadas informações de uma posição de memória para outra.

Se <variável> for um identificador de variável simples ou indexada são transferidas <exprcte> palavras a partir do endereço de <variável>^{1} para o endereço de <variável>^{2} ou o endereço que está em rI1. Para indicar a segunda opção usa-se a palavra reservada ENDRI1.

Após a execução da instrução, rI1 = endereço de <variável>^{2} + <exprcte> ou rI1 = rI1 + <exprcte>.

Se <variável>^{1} for um identificador de record, MOVE transfere todas as palavras do record. Neste caso <exprcte> deverá ser igual a 1 para evitar erro. No final teremos rI1 = endereço de <variável>^{2} + número de palavras do record.

Se <variável>^{1} for um identificador de vetor de records são transferidos <exprcte> records que correspondem a <exprcte> x (número de palavras do record) posições de memória. Neste caso <variável>^{2} também deverá ser identificador de vetor de record. A opção ENDRI1 não poderá ser usada pois neste caso para gerar código para a instrução é necessário termos o endereço da entrada na tabela de símbolos do nome do record para localizarmos os endereços dos diversos níveis do record.

Exemplos:

PLMIX	MIXAL
MOVE DOADOR[RI3] TO RECEP[RI4] FOR 2	ST1 RECEP, 4
	MOVE DOADOR, 3 (2)
MOVE FONTE[0,RI5] TO ENDR1 FOR 8	MOVE FONTE, 5 (8)

Supondo as seguintes declarações:

```
ARRAY 4 RECORD  ARVORE:
   .1  WORD  INFO
   .1  WORD  PONT
       .2 BYTE (1:2) ESQ
       .2 BYTE (4:5) DIR
ARRAY 4 RECORD  NOVAARV: STRUCTURE ARVORE
```

e a instrução

MOVE ARVORE[RI5] TO NOVAARV[RI6] FOR 4	ST1 N1,6
	MOVE A1,6 (4)
	ST1 N2,6
	MOVE A2,6(4)

Supondo:

A1 = endereço ARVORE.INFO

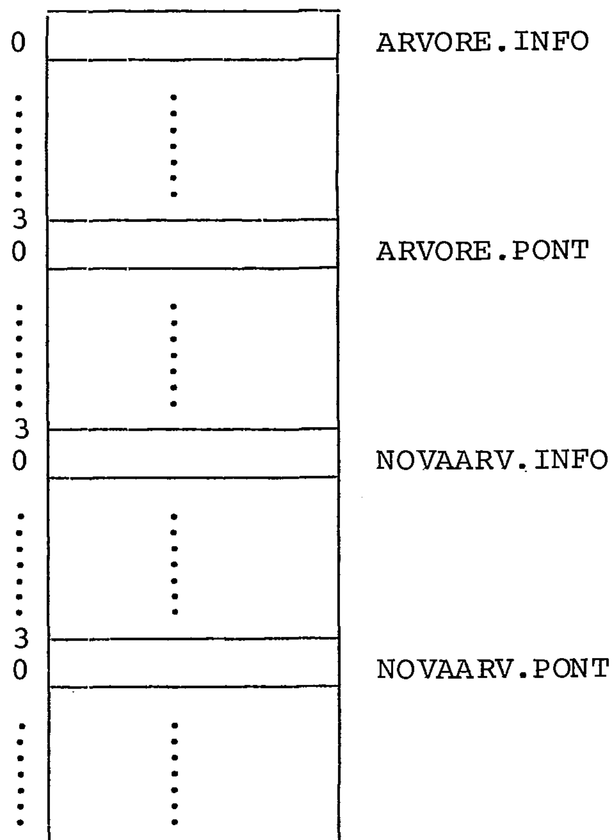
A2 = endereço ARVORE.PONT

N1 = endereço NOVAARV.INFO

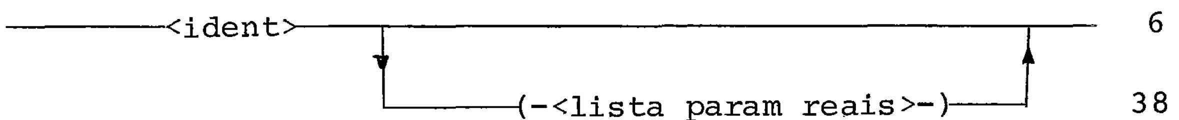
N2 = endereço NOVAARV.PONT

Para melhor compreensão deste último código gerado será mostrada a memória para as declarações dos records acima.

MEMÓRIA



23 <instrução de chamada>:: =



A chamada a um procedimento pode ocorrer em qualquer ponto do programa, inclusive numa declaração de procedimento. Não é permitida uma chamada recursiva, porém não é feito teste para deteção deste erro.

Se não há passagem de parâmetros a chamada se reduz a uma instrução de desvio incondicional para o início do corpo do procedimento.

Havendo passagem de parâmetros a inicialização dos parâmetros formais, e as atribuições de retorno dos parâmetros de tipo "value-result", se faz através de rA. Neste caso a instrução de chamada se traduz por: a) atribuição a cada um dos parâmetros formais; b) instrução de desvio para o início do corpo do procedimento; c) atribuição aos parâmetros transmitidos "by value-result".

Exemplos:

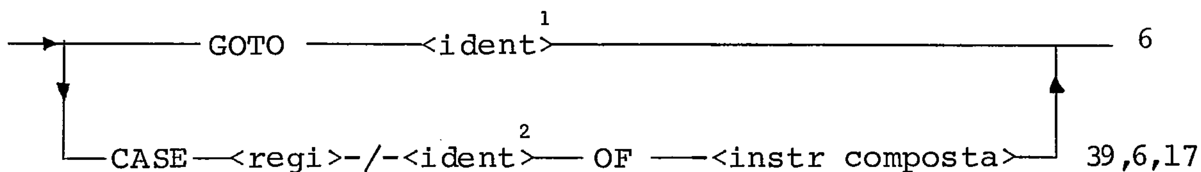
na declaração:

```
PROCEDURE IMP;
<corpo procedimento>

PROCEDURE SOMA (VALUE WORD CONST; WORD RESULT);
<corpo do procedimento>
```

na chamada:

IMP;	JMP	IMP
SOMA(10,VAR)	ENTA	10
	STA	CONST
	LDA	VAR
	STA	RESULT
	JMP	SOMA
	LDA	RESULT
	STA	VAR

24 <instrução de desvio>:: =



A instrução GOTO gera um desvio incondicional para a instrução de rótulo <ident>[1]. Ela pode ocorrer em qualquer ponto do programa, inclusive em um corpo de procedimento, estando o rótulo dentro ou fora dele.

A instrução CASE-OF trata dos desvios através de uma tabela de endereços. <ident>[2] é o nome da tabela e deve estar na declaração de CASE associada.

É feito um teste antes de ser executada a instrução para evitar que seja dado um desvio para uma instru

ção fora do CASE-OF introduzindo erros de maneira incontrolá vel e às vezes imperceptível para o programador. Deste modo são geradas as seguintes instruções antes do desvio do CASE-OF:

suponha a seguinte declaração:

CASE <ident>[1] : <numero>

e a instrução

CASE <regi>/<ident>[1] OF <instr composta>

teremos:

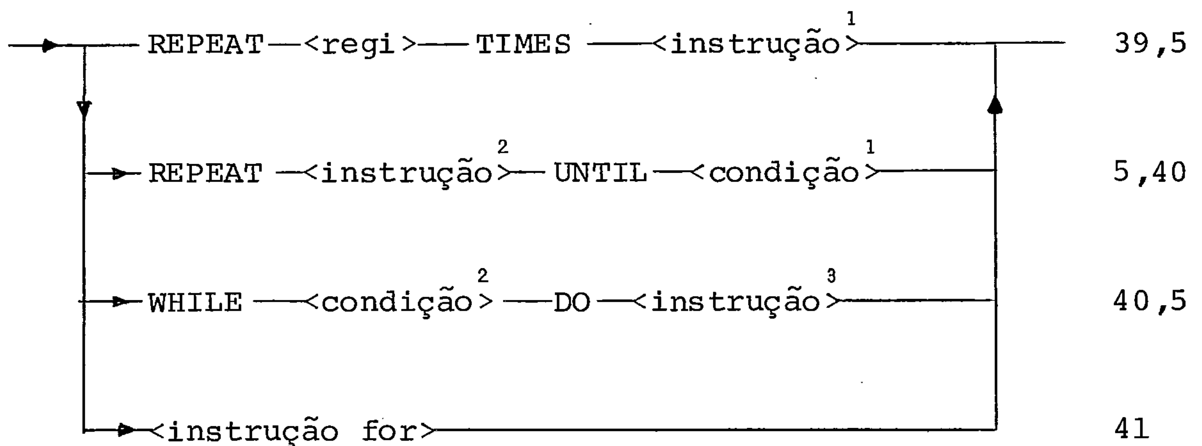
```
     J<regi> NN      * + 2           % testa se ≥ 0
     JMP             ERRO
     DEC<regi>       <numero>        % testa se c(rI:)<numero>
     J<regi> NN      ERRO
     INC<regi>       <numero>        % restaura <regi>
     JMP             * + 2
ERRO ENT<regi>       <numero>        % desvia la. intr após CASE-OF
     JMP          <ident>,<regi>     % desvio para tabela
```

Os valores permitidos para rIi variam de 0 a <numero>-1. Após cada instrução correspondente a uma opção é gerado um desvio para primeira instrução após o CASE-OF.

Exemplos

```
GOTO   SAI                             JMP    SAI

CASE RI2/CASO1 OF                      J2NN * + 2
     BEGIN                             JMP  5F
        VAL,RAC:=3; % RI2=0            DEC2 3
       BEGIN      % RI2=1              J2NN 5F
         RI5: = VAL + MAX;             INC2 3
         VAL: = 0                      JMP  * + 2
     END;                           5H ENT2 3
     VAL,RAC:=27;%RI2=2                JMP  CASO1,2
                                       <instr 0>
                                       JMP  6F
                                       <instr 1>
                                       JMP  6F
                                       <instr 2>
                                    6H
```

25 <instrução iterativa>:: =



A instrução iterativa permite a execução de uma instrução, em geral composta, um determinado número de vezes sem que estas instruções sejam instruções com contadores ou testes de saída (explícitos). A instrução gera um comando de desvio condicional que é utilizado de modos diferentes dependendo da instrução.

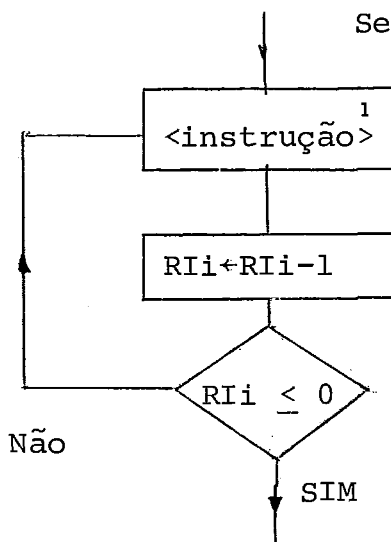
A instrução REPEAT <regi> TIMES <instrução>^{1} faz com que a <instrução>^{1} seja executada um número de vezes igual ao conteúdo do registro Ii associado à instrução.

O registro deve ser carregado antes da instrução. Se o conteúdo do registro for zero ou negativo, <instrução>^{1} será executada uma vez.

Para que o número de iterações seja igual ao valor carregado inicialmente em rIi este não deve ser alterado em <instrução>^{1}. Porém não é feito nenhum teste para verificar a alteração do conteúdo do registro.

A cada execução de <instrução>^{1} o valor do registro é decrementado de 1, interrompendo a iteração quando este valor for zero ou negativo.

Segundo o esquema abaixo a compilação será:

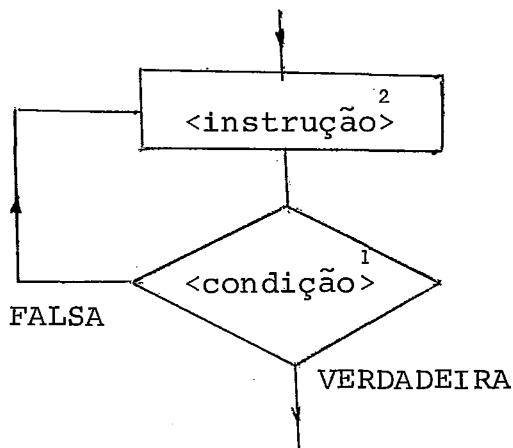



```
XXX <instrução>¹
    DEC<regi>1
    J<regi>P  XXXX
```

A instrução REPEAT -<instr>[2] UNTIL <condição>[1] faz com que <instrução>[2] seja executada enquanto a <condição> for falsa, sendo que a instrução é executada pelo menos uma vez.

Para que a iteração termine é necessário que pelo menos um dos elementos que fazem parte de <condição>[1] tenha seu valor alterado no corpo de <instrução>[2].

Para o esquema da instrução o código gerado será:



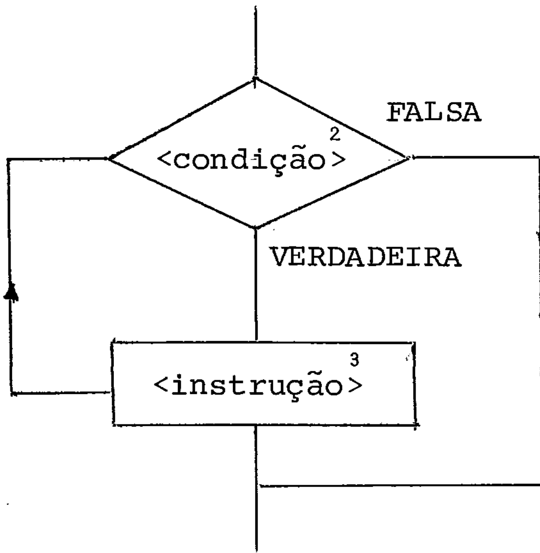
XXX <instrução>[2]
<instr p/teste condição>*
J ¬<condição>[1] XXX

* as instruções que são geradas para o teste de condição serão vistas em (40).

A instrução WHILE<condição>[2] DO <instrução>[3] permite a iteração enquanto a condição for verdadeira. Se no primeiro teste a condição for falsa, <instrução>[3] não é executada nenhuma vez.

Para que a iteração processe um número finito de vezes é necessário que pelo menos um dos elementos que fazem parte da condição seja alterado.

Com o esquema teremos:




```
XXX <instr p/teste condição>
                2
    J ¬ <condição>    ZZZ
             3
    <instrução>
    JMP    XXX
ZZZ
```

26 <instrução condicional>:: =

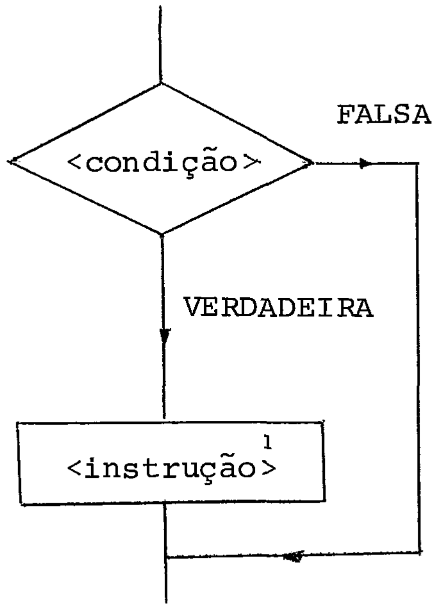
```
                                             1
──►IF ──<condição>── THEN ──<instrução>─┬──────────────────────┬─40,5
                                        │                   2  │
                                        └─ELSE-<instrução>─────┘ 5
```

A instrução condicional permite que após um teste sejam executadas ou saltadas uma ou mais instruções.

Se a condição testada for verdadeira a instrução após o THEN é executada, sendo saltada a seguinte ao ELSE, se houver. Para condição falsa é executada a instrução após o ELSE, se houver.

Se houver várias instruções IF aninhadas, é obedecida a regra que um ELSE se relaciona sempre com o THEN mais próximo. A alteração desta regra é feita usando-se instrução composta.

Para a compilação de IF <cond> THEN <instr>[1] temos:



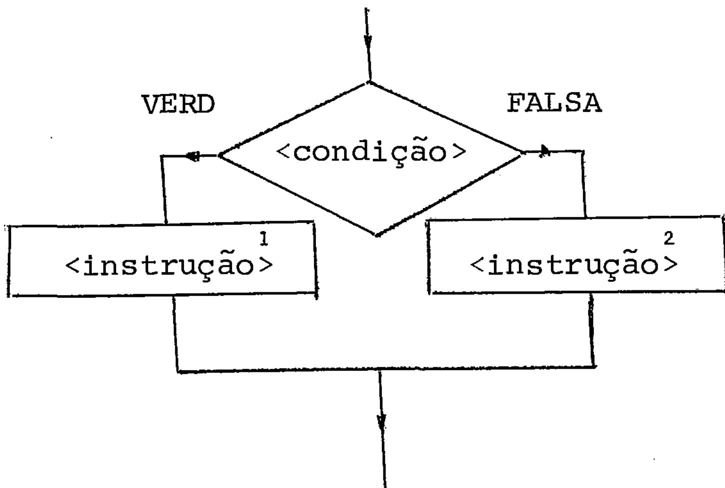
<instr p/teste condição>

J ¬ <condição> XXX

<instrução>^{1}

XXX

Para IF <cond> THEN <instr>^{1} ELSE <instr>^{2} teremos:



<instr p/teste condição>

J ¬ <condição> XXX

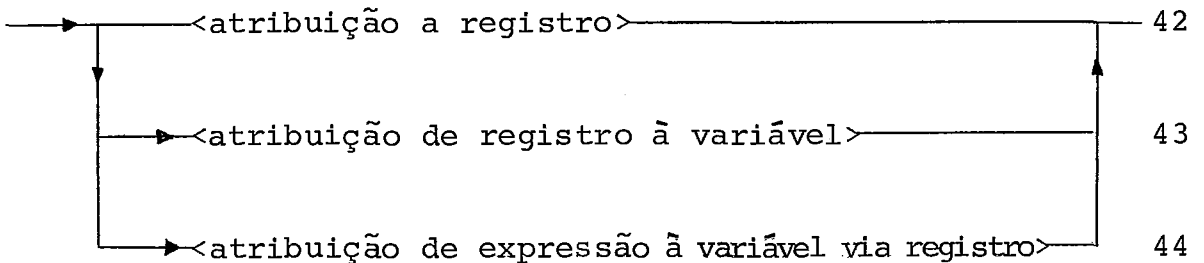
<instrução>^{1}

JMP ZZZ

XXX <instrução>^{2}

ZZZ

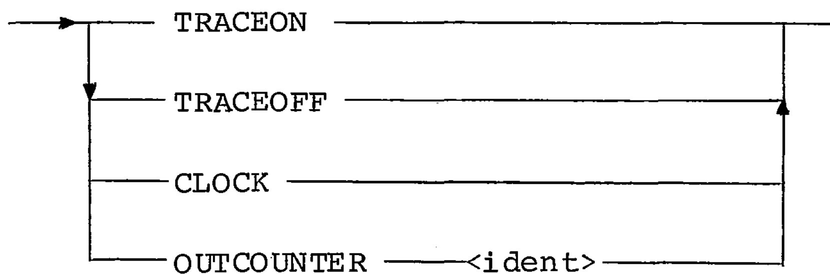
27 <instrução de atribuição>:: =



A instrução de atribuição permite, após o cálculo da expressão à direita do sinal de atribuição (:=) , alterar o conteúdo de um registro, uma ou mais posições de memória associadas a variáveis simples, indexadas ou identificador de nível de record substituindo-o pelo resultado do cálculo efetuado.

Na atribuição à variável indexada não é feito teste para verificação se o acesso esta fora dos limites da declaração.

28 <instruções para depuração>:: =



6

A instrução TRACEON avisa que a partir deste ponto antes de cada instrução serão impressos o código da instrução, e os conteúdos de rA, rX e rIi, $1 \leq i \leq 6$.

A instrução TRACEOFF desativa a instrução TRACEON.

A instrução CLOCK faz com que seja impresso o conteúdo do relógio do simulador.

OUTCOUNTER <ident> imprime o conteúdo des

te contador.

Nenhuma destas instruções afeta o conteúdo do relógio.

Estas instruções não existem na definição feita por Knuth [3] e foram introduzidas no simulador do MIX por Markenzon [6], cujo trabalho deve ser procurado para maiores detalhes.

Exemplo: PLMIX		MIXAL	
⋮			
COUNTER LOOP1;	PC		
RI2: = 2;	0	ENT2	2
RAC: = 10;	1	ENTA	10
RI1: = 20;	2	ENT1	20
TRACEON;	3	TRCE	
REPEAT RI2 TIMES	4	LOOP	
BEGIN	5	INCA	0,1
LOOP1:RAC:=RAC + RI1	6	ENT3	0,1
RI3:=RI1 - 3	7	DEC3	3
END	8	DEC2	1
TRACEOFF;	9	J2P	4
OUTCOUNTER LOOP1;	10	NTRC	
	11	OUTL	3

Na execução teremos na impressora:

INST = 5	A = 10	I1 = 20	I2 = 2	I3 = ?
	X = ?	I4 = ?	I5 = ?	I6 = ?
INST = 5	A = 10	I1 = 20	I2 = 2	I3 = ?
	X = ?	I4 = ?	I5 = ?	I6 = ?
INST = 48	A = 30	I1 = 20	I2 = 2	I3 = ?
	X = ?	I4 = ?	I5 = ?	I6 = ?
INST = 51	A = 30	I1 = 20	I2 = 2	I3 = 20
	X = ?	I4 = ?	I5 = ?	I6 = ?
INST = 51	A = 30	I1 = 20	I2 = 2	I3 = 17
	X = ?	I4 = ?	I5 = ?	I6 = ?
INST = 50	A = 30	I1 = 20	I2 = 1	I3 = 17
	X = ?	I4 = ?	I5 = ?	I6 = ?
INST = 42	A = 30	I1 = 20	I2 = 1	I3 = 17
	X = ?	I4 = ?	I5 = ?	I6 = ?
INST = 5	A = 30	I1 = 20	I2 = 1	I3 = 17
	X = ?	I4 = ?	I5 = ?	I6 = ?

```
INST = 48     A = 50     I1 = 20     I2 = 1     I3 = 17
              X = ?      I4 = ?      I5 = ?     I6 = ?

INST = 51     A = 50     I1 = 20     I2 = 1     I3 = 20
              X = ?      I4 = ?      I5 = ?     I6 = ?

INST = 51     A = 50     I1 = 20     I2 = 1     I3 = 17
              X = ?      I4 = ?      I5 = ?     I6 = ?

INST = 50     A = 50     I1 = 20     I2 = 0     I3 = 17
              X = ?      I4 = ?      I5 = ?     I6 = ?
```

** LOOP : 2

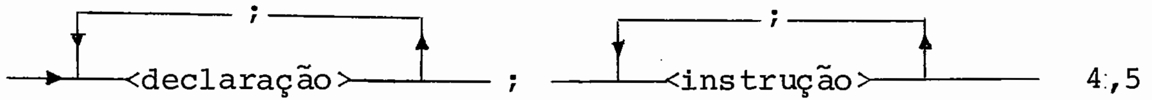
29 <letra>:: =

```
 ┬── A ──┬
 ├── B   │
 │   :   │
 └── Z ──┘
```

30 <digito>:: =

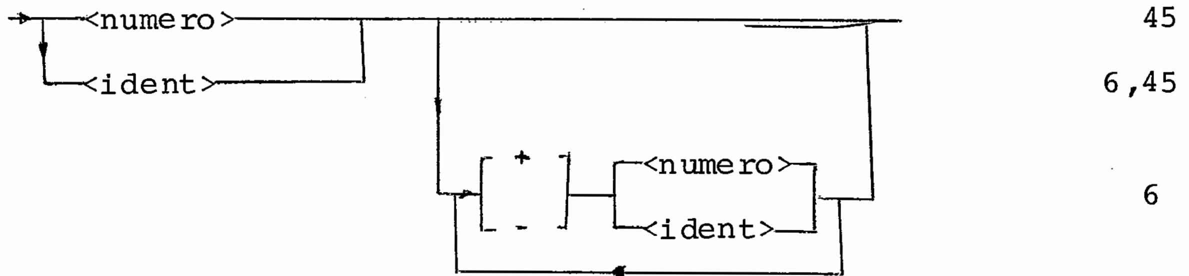
```
 ┬── 0 ──┬
 ├── 1 ──┤
 │   :   │
 └── 9 ──┘
```

31 <lista sem procedimento>:: =



A <lista sem procedimento> aparece na definição do <corpo do procedimento> para indicar na sintaxe, a impossibilidade de declaração de subprograma no corpo de outro.

32 <expressão constante>:: =



Uma expressão constante é uma expressão que pode ser computada em tempo de compilação. Deste modo os componentes da expressão são números ou identificadores de constantes já definidas.

O compilador sempre trabalha com o resultado da expressão.

Exemplo:

PLMIX	MIXAL	
CONSTANT VINTE = 20		
RAC: = VINTE + 17 + RI2	ENTA	37
	INCA	0,2
RAC: = RI2 + VINTE + 17	ENTA	0,2
	INCA	20
	INCA	17

No segundo exemplo, como as expressões são calculadas da esquerda para a direita sem precedência, a presença de RI2 indica não estarmos tratando de expressões constantes, logo o compilador tratará cada constante em separado, o que justifica o código.

33 <cadeia>:: =

46

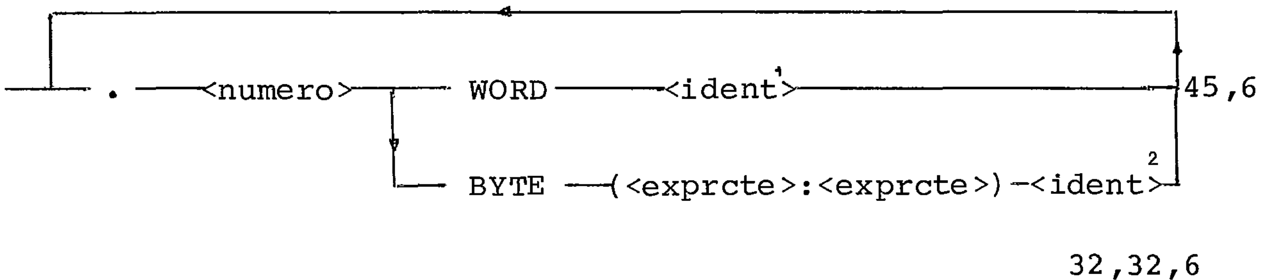
Uma cadeia é uma sequência de caracteres, inclusive branco, encerrada entre plics ('). É permitido o caracter plic dentro da cadeia mas deverão ser escritos dois plics juntos, sem espaço entre eles, e um só será considerado o plic da cadeia, o outro servirá apenas como informação para o analisador léxico.

Exemplo:

'ISTO E" UM EXEMPLO'

a cadeia será ISTO E"UM EXEMPLO.

34 <corpo do record>:: =



O nível superior é aquele da definição tipo WORD. Ela associa <ident>[1] a uma posição de memória.

As declarações de tipo BYTE que seguem a declaração WORD se referem à palavra associada a <ident>[1]. Estas declarações de BYTE dão nomes a bytes contíguos desta palavra. Poderá haver declarações que se superponham, isto é, dar um nome aos bytes (0:3), dar outro nome ao campo (2:3) e um outro a (2:2).

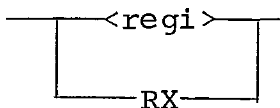
O valor de <numero> é irrelevante para o compilador.

A referência a um nível é feita por <identificador record>.<identificador nível>.

Exemplo:

```
RECORD MATRICULA :
.01 WORD CODIGO
.01 WORD NASCIM
    .02 BYTE (1:4) DATA
        .03 BYTE (1:2) ANO
        .03 BYTE (3:4) MES
    .02 BYTE (5:5) SEXO
```

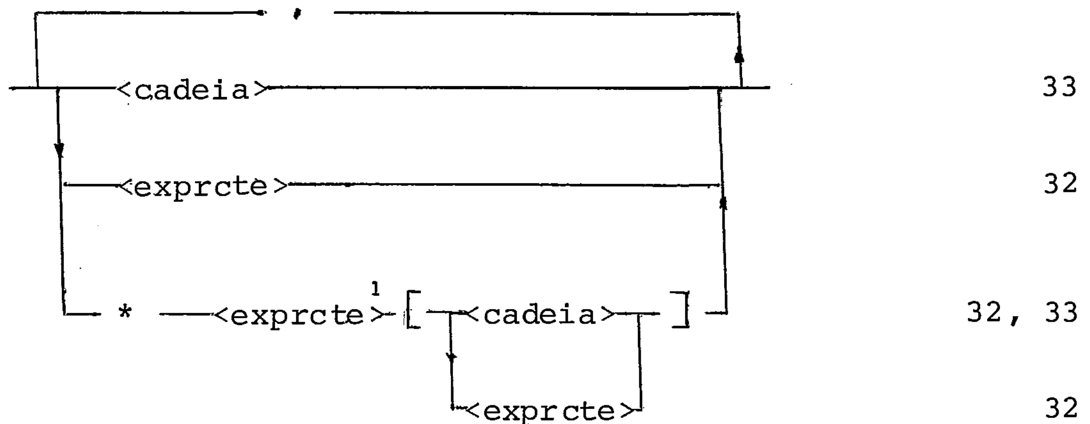
35 <regix>:: =



39

A entidade <regix> engloba referências a um registro de índice ou ao registro RX, que é a extensão à direita do acumulador (rA).

36 <inicialização de vetor>:: =



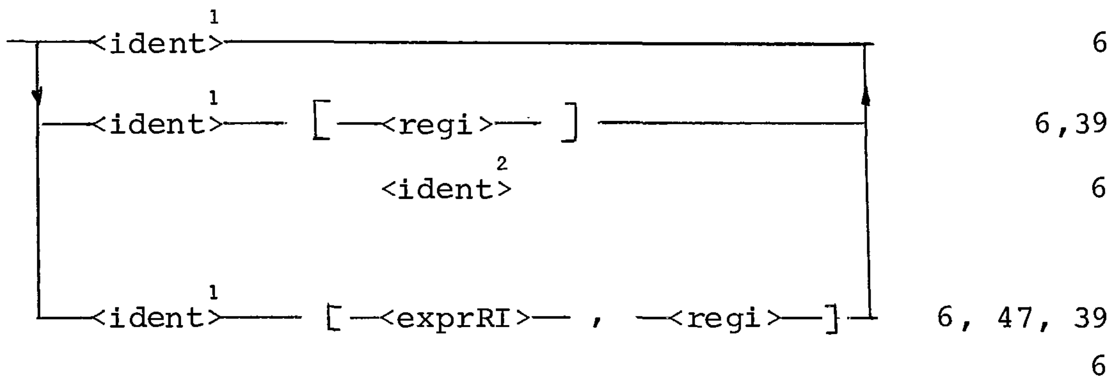
A inicialização de um vetor obedece às mesmas regras que a inicialização de variável simples tipo WORD (10).

Se várias palavras vão ser inicializadas com um mesmo valor, então usa-se o fator de repetição *<exprcte>[1] e o valor a ser repetido entre colchetes ([e]).

Exemplo:

ARRAY 10 WORD VAL = [20,30,-17,*5[31], 'AB', 'CD']

37 <variável>:: =



Uma <variável> é uma variável simples ou indexada.

O índice deve ser um registro de índice , um identificador de equate (associado a um registro de índice), ou uma expressão RI. Uma expressão RI é calculada em um registro de índice e neste caso é necessário indicar em qual registro será efetuado o cálculo. Uma expressão constante também é uma expressão RI. Neste caso após o compilador avaliar

o valor da expressão, ela é atribuida ao registro de índice in dicado.

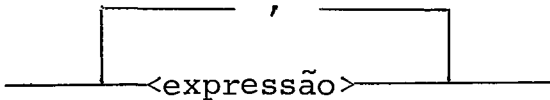
Exemplo:

DADO

VALOR [RI1]

TABELA [RI2 + VINTE - 10, RI4]

38 <lista de parâmetros reais>:: =



48

Os parâmetros reais devem corresponder em tipo e número aos parâmetros formais.

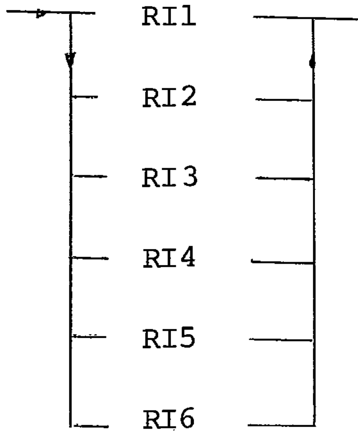
Para os parâmetros de tipo BYTE não é feita verificação se o campo do parâmetro real é igual ao do parâmetro formal.

Na execução do procedimento só é utilizado o campo especificado na declaração dos parâmetros formais.

Exemplo:

TESTE (MAX, 20 + RI3, ITEM [RI4])

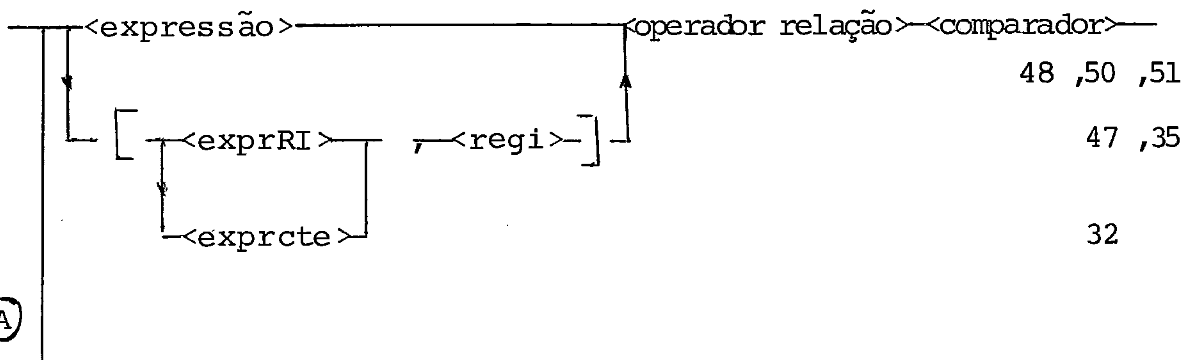
39 <regi>:: =

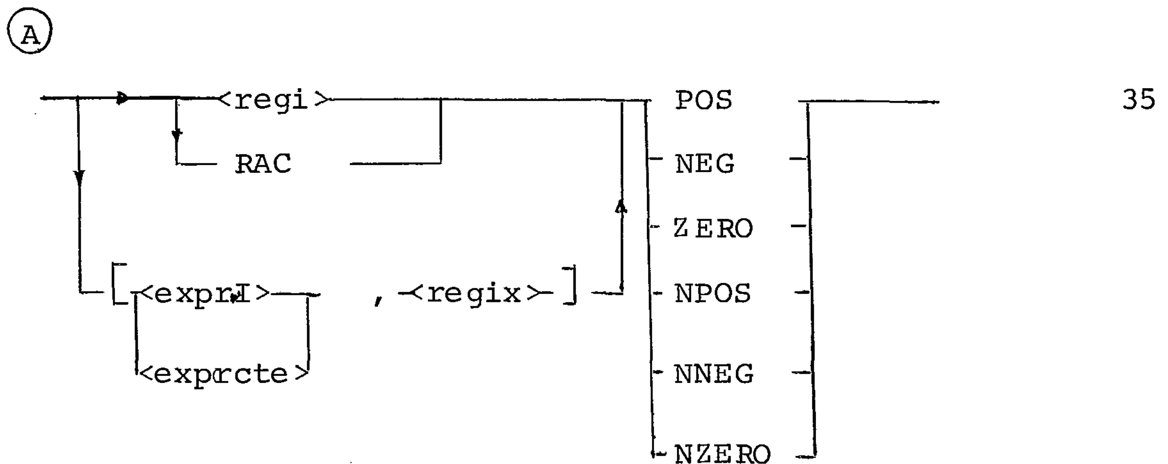


Os registros de índice tem dois bytes mais o sinal e são usados basicamente como contadores e índices nas variáveis indexadas.

A operação de comparação pode ser feita com um dos elementos do teste no registro de índice. Neste caso supõe-se que rIi tem cinco bytes, sendo que os bytes 1, 2 e 3 são iguais a zero.

40 <condição>:: =





Para o teste de condição <expressão> é calculada no registro A e depois executada a instrução de comparação - CMPA - entre o rA e uma posição de memória. Dependendo do resultado é ligado o indicador de MENOR, IGUAL ou MAIOR.

Se <expressão> for somente um registro de índice ou rX ou uma variável equate associada a um desses registros, a comparação é feita com este registro.

A comparação também pode ser feita com expressões RI calculadas em um registro, evitando assim utilizar rA. É necessário então indicar em qual registro será efetuado o cálculo da expressão RI e posterior comparação.

A instrução de desvio não altera o indicador de comparação.

Uma comparação de + 0 e - 0 resulta em IGUAL.

Para desvio usando o resultado no indicador de comparação a compilação gera a negação da relação testada. Assim teremos:

EQ	JNE
NE	JE
GT	JLE
GE	JL
LT	JGE
LE	JG

Os registros podem ser testados para condições: positivo, negativo, zero, não positivo, não negativo e não zero. Para tais condições teremos na compilação:

POS	J <reg> NP
NEG	J <reg> NN
ZERO	J <reg> NZ
NPOS	J <reg> P
NNEG	J <reg> N
NZERO	J <reg> Z

onde <reg> pode ser A, X ou Ii, $1 \leq i \leq 6$.

Exemplos:

PLMIX	MIXAL
IF INDICE GT MAX	LDA INDICE
THEN <instrução>	CMPA MAX
	JLE XXX
	<instrução>
	XXX

PLMIX	MIXAL		
WHILE [FIM + 16 + RI6, RI4] LT FINAL		LD4	FIM
DO <instrução>		INC4	16
		INC4	0,6
		CMP4	FINAL
		J4GE	ZZZ
		<instrução>	
		JMP	XXX
	ZZZ		
REPEAT <instrução>	XXX	<instrução>	
UNTIL RAC ZERO		LDA	TESTE
		JANZ	XXX

41 <instrução for>

```
          1                1             1            2
— FOR —< ident>— := —<expressão>— STEP —<exprcte>— UNTIL —<expressão>(A)
                                                          6, 48, 32, 48

       (A) — DO ——<instrução>——                             5

                          1                 1                 2
— FOR —<regix>— := ┬<exprRI>─┬ STEP —<exprcte>— UNTIL ┬<exprRI>─┬(B)  35,47,32,47
          2        │         │                        │         │
       <ident>     └<exprcte>┘                        └<exprcte>┘     6,32,32

       (B) — DO ——<instrução>——
```

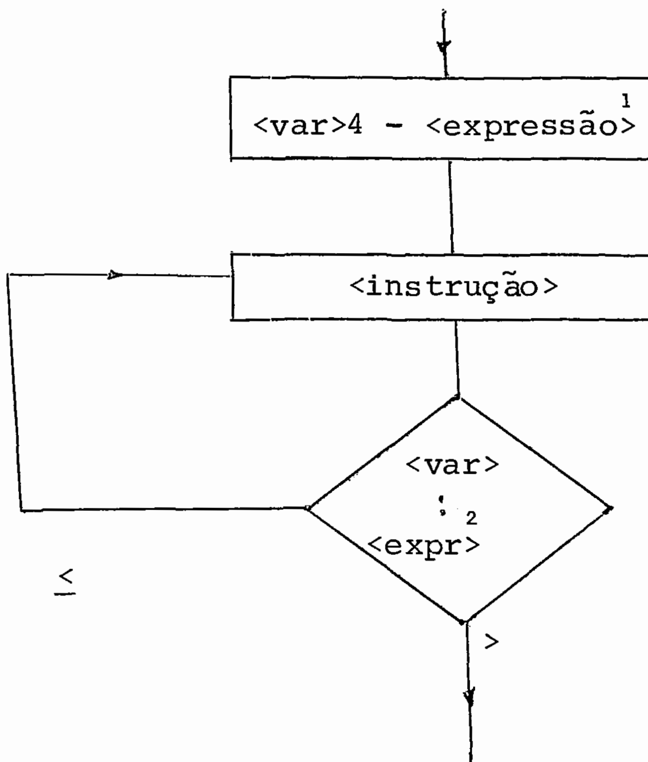
A <instrução for> faz com que <instrução> seja executada uma ou mais vezes, dependendo de uma variável ou registro de controle.

Esta variável ou registro recebe um valor inicial <expressão>^{1} ou <exprRI>^{1} (<exprcte>^{1}) antes do início da iteração. Após cada execução de <instrução>, a variável ou registro de controle é alterada de um valor constante , chamado passo. (<exprcte>^{1}). A iteração é repetida até que a variável (ou registro) seja inferior ao valor final <expressão>^{2}, ou superior ao valor final, para passo positivo.

A variável (ou registro) de controle pode ser usada em <instrução>, porém a alteração explicita (pelo programador) de seu valor, acarretará na modificação no número de iterações.

O esquema abaixo é geral e serão mostrados códigos gerados para variável de controle sendo uma variável simples e sendo um registro

supondo passo positivo



Código gerado para <ident>, expressão no rA e passo positivo

```
      <instr para cálculo <expressão1> >
      STA         XXX
      <instr para cálculo <expressão2> >
      STA         YYY
      JMP         ZZZ
YYY
ZZZ   <instrução>
      LDA         XXX
      INCA        <passo>
      CMPA        YYY
      JLE         ZZZ
```

XXX endereço da variável de controle

YYY posição de memória que contém o valor limite. É necessário guardar esta informação na memória, pois só é possível fazermos comparação entre registro e posição de memória.

Código gerado para <regi> (ou <ident> é uma variável equate), passo negativo.

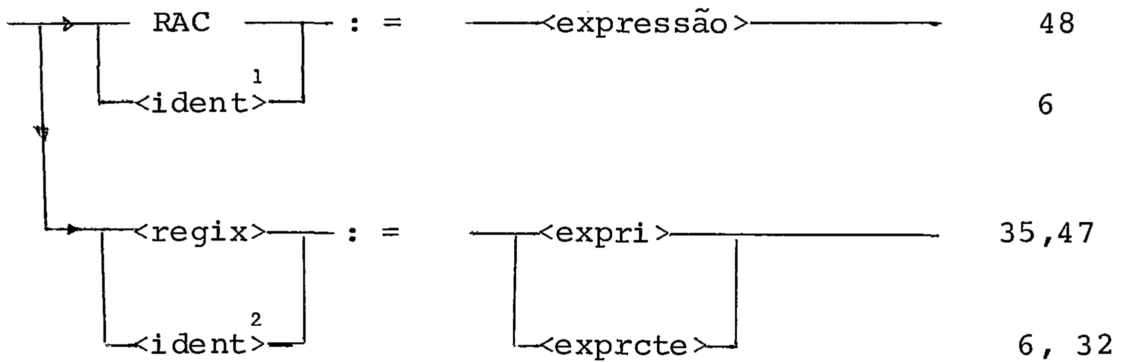
```
     <instr para cálculo <exprRI¹> >        (1)
     <instr para cálculo <exprRI²> >        (2)
      STA        YYY
      JMP        ZZZ
YYY
ZZZ   <instrução>
      DECIi      <passo>
      CMPIi
      JGE        ZZZ
```

(1) a <exprrI>^{1} será calculada no <regix> usado como variável de controle. Deste modo o valor inicial já esta nele e não é preciso instrução de carga.

(2) <exprI>^{2} será calculada no rA para evitar destruir o conteúdo do registro de controle. Deste modo qualquer instrução "for" altera o conteúdo de rA.

O segundo exemplo de código gerado é igual para o rX.

42 <atribuição a registro>: : =



A operação de atribuição a registro, é uma indicação do registro sobre o qual será calculada a expressão à direita do sinal de atribuição, e onde ficará o resultado . Quando <expressão> for uma <expressão constante>, esta será calculada pelo compilador e depois carregada no registro através da instrução do tipo ENT <reg>.

<ident>^{1} e <ident>^{2} são variáveis do tipo equate, sendo <ident>^{1} associada ao rA e <ident>^{2} associada a rX ou rIi.

Só é feita indicação no "bit de overflow " se este ocorrer no registro rA. Para os registros rX e rIi não é fornecida nenhuma informação pela máquina.

Na atribuição de uma variável tipo BYTE o conteúdo do campo especificado é atribuido ao registro (rA , rX ou rIi) ajustado à direita e os outros bytes do registro são zerados. Se o campo de uma variável BYTE não inclui o byte de sinal, após a atribuição o registro fica com sinal positivo.

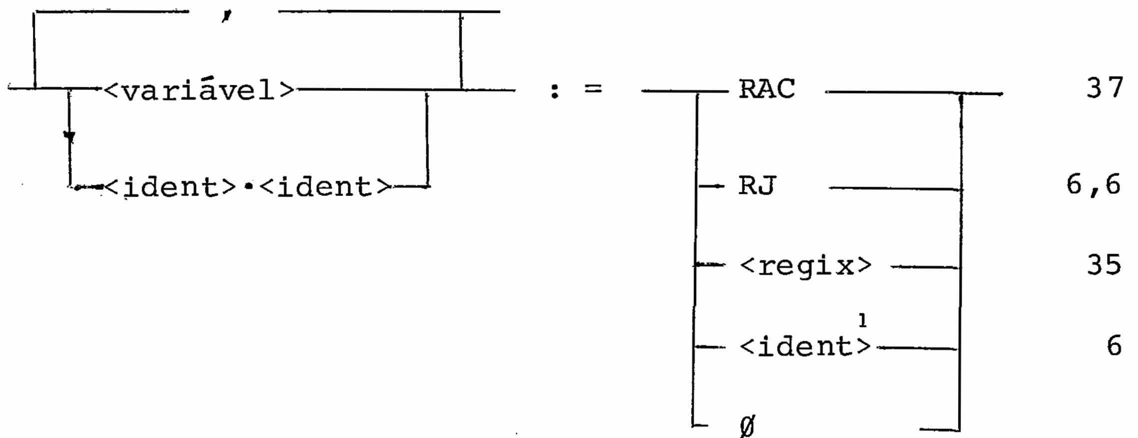
Exemplos:

PLMIX	MIXAL	
RAC: = VAR * COD [RI1] - 13 SLA 2	LDA	VAR
	MUL	COD,1
	DECA	13
	SLA	2

Suponha DELTA e MAX constantes e CONT variável

RI1: = CONT + DELTA - MAX + RI3	LD1	CONT
	INC1	DELTA
	DEC1	MAX
	INC1	0,3

43 <atribuição de registro a variável>:: =



A variável tipo WORD tem todos os seus bytes alterados quando recebe rA, rX, rIi ou uma variável declarada EQUATE (<ident>[1]) com um desses registros. Na atribuição do registro rJ somente são alterados os bytes 1 e 2 e o sinal, que é sempre positivo. Atribuindo-se um registro de índice os

bytes 1,2 e 3 serão zerados e só recebem valores os bytes 4 e 5.

Se a variável tipo BYTE só ocupar alguns bytes da palavra, somente estes serão alterados, permanecendo os demais com o conteúdo anterior à operação. A atribuição é feita tomando-se os bytes do registro à partir da direita e efetuando um deslocamento para à esquerda, se necessário, para inserir no campo especificado.

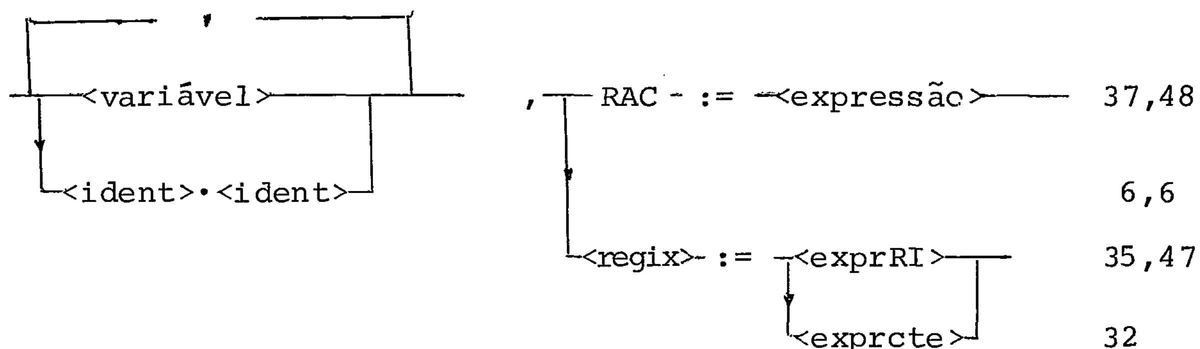
Além dos registros poderá ser feita a atribuição do valor zero, que corresponde à instrução STZ (store zero). Na inicialização de uma variável com este valor, é aconselhável esta atribuição pois ela gasta apenas uma instrução, enquanto que uma atribuição via registro demandará duas instruções - uma para carregar o registro e outra para transferir o conteúdo do registro para a memória.

O mesmo registro pode ser atribuido a mais de uma variável e neste caso se fará da esquerda para a direita.

Exemplos:

```
VALTR: = RAC                      STA     VALOR

A, B, D [RI1]: = RI2              ST2     A
                                  ST2     B
                                  ST2     D,1

NO.LINK,INICIO[RI4]: = 0          STZ     NO(LINK)
                                  STZ     INICIO,4
```

44 <atribuição de expressão a variável via registro>:: =



Esta atribuição permite que uma ou mais variáveis recebam uma expressão. Como a atribuição à memória só pode ser feita através de registro, é necessário indicar qual será usado. A expressão deve poder ser calculada no registro referenciado.

A atribuição na lista será feita da esquerda para a direita, sendo válidas as observações feitas em (43) sobre atribuição de registro à variável.

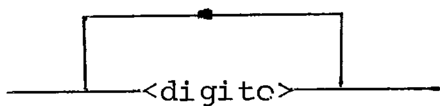
Exemplos:

PLMIX	MIXAL	
VAL,DADO[RI4], RAC:= MATRIZ[RI4]* DEZ + 5	LDA	MATRIZ,4
	MUL	DEZ
	INCA	5
	STA	VAL
	STA	DADO,RI4

TESTE[10 + MAX,RI4], RI1: = CONT + 5

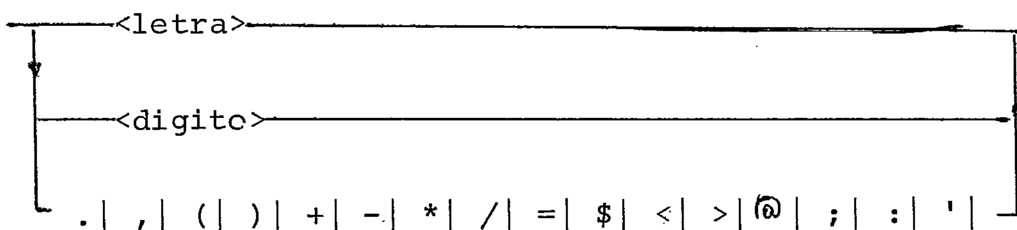
```
LD1     CONT
INC1    5
ENT4    10
INC4    MAX
ST1     TESTE,4
```

45 <número>:: =



30

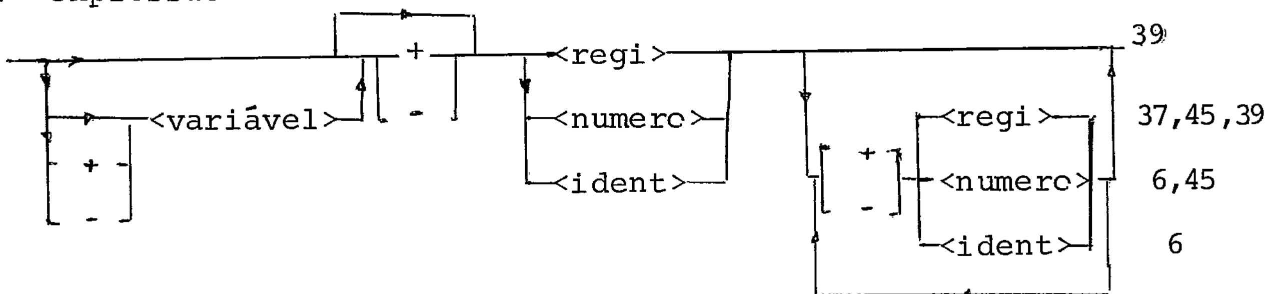
46 <caracter>:: =



29

30

47 <expressão RI>:: =



A expressão de registro I só admite os operadores de soma e subtração que correspondem às instruções INC e DEC. Os fatores só podem ser registros, números ou identificadores de constante ou equate.

Uma variável só pode aparecer como primeiro elemento da expressão.

A expressão é avaliada da esquerda para a direita.

O registro que está à esquerda do sinal de atribuição só poderá aparecer na expressão se for o primeiro elemento desta, caso contrário o compilador enviará uma mensagem de erro e não calculará a expressão.

Se o resultado da operação não puder ser colocado em 2 bytes o registro fica com valor indefinido e não ocorrerá ação sobre o indicador de "overflow".

Exemplos:

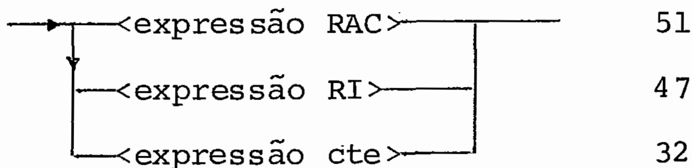
Supondo as declarações:

PLMIX | MIXAL

```
WORD VAR;
BYTE (3:4) ABC;
CONSTANT MAX = 100;
ARRAY 5 WORD A;
RECORD NO:
     .1 WORD INFO
     .1 WORD PONT
```

PLMIX	MIXAL	
.2 BYTE (1:2) ESQ		
.2 BYTE (3:4) DIR;		
ARRAY 3 RECORD TIPO: STRUCTURE NO;		
RI1 : = MAX	ENT1	100
RI6 : = NO.ESQ	LD6	NO(ESQ)
RI5 : = RI5 - 39	DC5	39
RI3 : = - RI2	ENN3	0,2
RI4 : = A[RI1] + RI2 - 15	LD4	A,1
	INC4	0,2
	DEC4	15
RI2: =TIPO.DIR[MAX-RI1,RI3]+ RI5	ENT3	MAX
	DEC3	0,1
	LD2	TIPO,3(DIR)
	INC2	0,5
VAR, RI6: = VAR + MAX	LD6	VAR
	INC6	MAX
	ST6	VAR

48 <expressão>



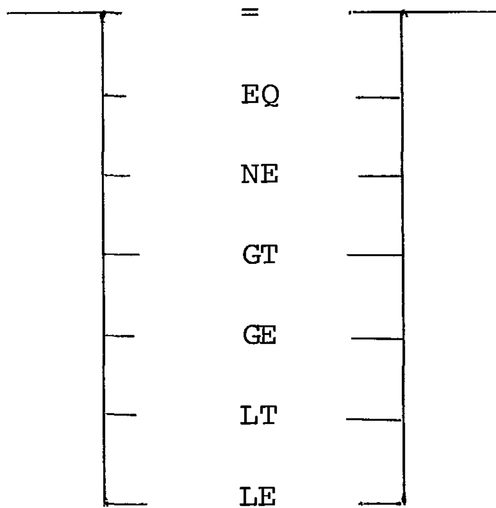
51

47

32

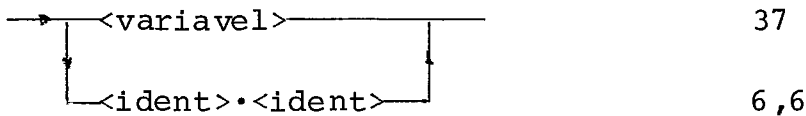
Uma expressão deverá ser calculada no acumulador.

49 <operador de relação>:: =



Na compilação das instruções com condição a relação testada será a negação da relação escrita no programa (vide (40)).

50 <comparador>:: =



37

6,6

O segundo elemento de uma comparação deverá ser sempre referenciável por um endereço de memória.

Esta restrição deriva dos testes serem sempre entre registro e variável, e da impossibilidade de serem usadas variáveis temporárias pelo compilador. Se fossem permitidas expressões no segundo membro da comparação, a utilização

de variáveis temporárias seria inevitável, conforme o exemplo:

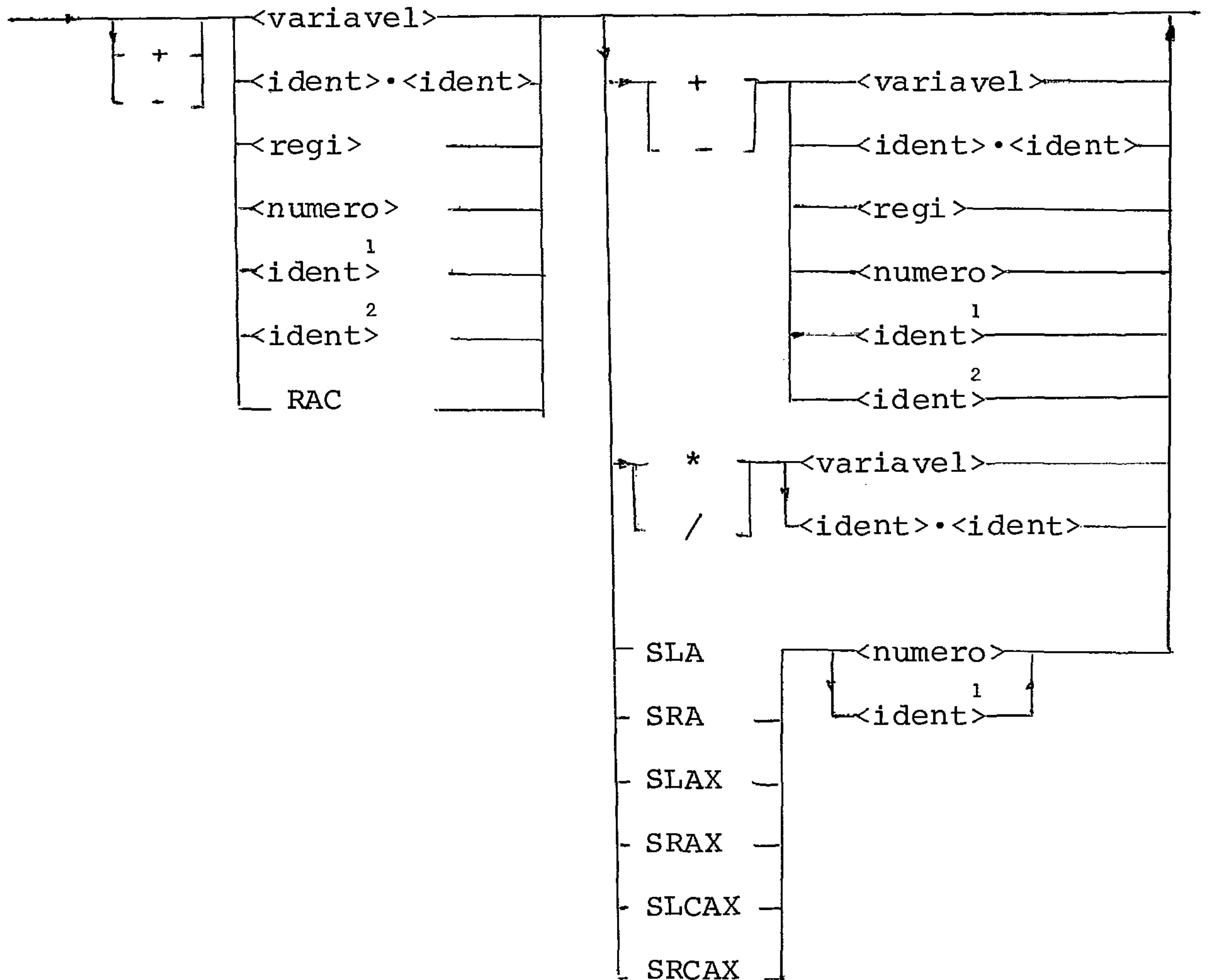
A * B > C * D

exigiria sua transformação em

A * B - C * D > 0

com uso obrigatório de temporária.

51 <expressão RAC>:: =



<ident>[1] - identificador de constante

<ident>[2] - identificador de variável equate

A expressão é calculada da esquerda para a direita sem prioridade entre os operadores. Cada uma das operações tem um tratamento sendo que o resultado fica em rA. A operação seguinte trabalha com o novo conteúdo de rA.

Atuação dos operadores:

adição: se a magnitude do resultado for maior que o permitido o indicador de "overflow" é ligado e permanece no acumulador o valor remanescente do resultado que couber em 5 bytes. A parte mais significativa ficará em um registro à esquerda de A.
Se o resultado for nulo o sinal do acumulador não é alterado.

subtração: mesmo procedimento da adição.

multiplicação: a multiplicação é fetuada sempre o rA e uma variável. O resultado obtido é armazenado em 10 bytes (rA e rX) e os bytes menos significativos ficam em rX. Os bytes de sinal recebem o sinal algébrico do produto.

divisão: o valor em rA e rX, tratado como um número de 10 bytes com o sinal de rA, é dividido

pelo valor da variável. Se a variável for igual a zero ou o quociente precisar de mais de 5 bytes para armazenar seu valor, rA e rX ficam com valores indefinidos e é ligado o indicador de "overflow". Caso contrário o quociente fica em rA e o resto em rX. O sinal de rA é o sinal algebrico da operação e rX recebe o sinal de rA, anterior à divisão.

deslocamento: em PLMIX os deslocamentos são operadores binários. Maiores detalhes sobre atuação dos operadores estão em II.2.3.9.

Exemplos

Suponha as declarações:

```
WORD VAR;
BYTE (3:4) ABC;
CONSTANT MAX = 100;
ARRAY 5 WORD A
RECORD NO:
      .1 WORD INFO
      .1 WORD PONT
            .2 BYTE (1:2) ESQ
            .2 BYTE (3:4) DIR;
ARRAY 3 RECORD TIPO: STRUCTURE NO;
EQUATE I = RI1;
```

RAC : = - ABC	LDN	ABC (3:4)
RAC : = RAC * A[RI4]	MUL	A,4
RAC : = RAC SLA 2 + NO.DIR	SLA	2
	ADD	NO (DIR)
RAC : = TIPO.DIR[3,RI1]/A[MAX+5,RI2]	ENT1	3
	LDA	TIPO,1(DIR)
	ENT2	MAX
	INC2	5
	DIV	A,2
RAC : = VAR+MAX*ABC-A[I] + I	LDA	VAR
	INCA	MAX
	MUL	ABC(3:4)
	SUB	A,1
	INCA	0,1
RAC : = TIPO.PONT[A[I]-MAX-RI3,RI2]	LD2	A,1
	DEC2	MAX
	DEC2	0,3
	LDA	TIPO,2 (PONT)
RAC : = - 350 + MAX / VAR	ENNA	250
	DIV	VAR
RAC : = -I + RI4 - A[RI5]	ENNA	0,1
	INCA	0,4
	SUB	A,5

C A P Í T U L O IV

ESTRUTURA DO COMPILADOR

IV.1. ASPECTOS GERAIS

O compilador PLMIX é um tradutor de um passo. Em uma passagem pelo programa a ser compilado reconhece os símbolos, realiza análise sintática, constroi tabela de símbolos, resolve referências, assinala erros e gera código - objeto.

A compilação em um passo foi possível devido às características da linguagem de ter todas as variáveis declaradas e dispensar otimização de código. Para cada comando da linguagem existe uma tradução definida, de modo que a otimização de código deverá ser feita pelo próprio programador. O uso previsto de linguagens de médio nível, tipo PLMIX, exige que a otimização de subexpressões e alterações da estrutura do programa visando otimização de código sejam tão radicais que não é possível efetuá-los automaticamente durante a compilação, a custos razoáveis (Knuth [25]).

A exigência de declaração de todas as variáveis e rótulos a serem usados no programa, facilitou a escrita do compilador permitindo saber, a cada momento, o códi

go a ser gerado.

IV.2. ANALISADOR LÉXICO

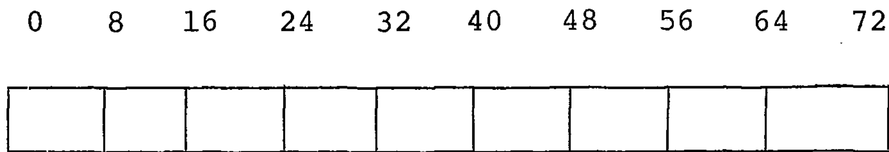
A função básica do analisador léxico é examinar o texto fonte e devolver o próximo símbolo terminal ("token") ao analisador sintático. Os comentários e os brancos são reconhecidos pelo analisador léxico e não são submetidos ao analisador sintático.

Entidades sintáticas como identificadores, constantes numéricas e cadeias são reconhecidas e devolvidas ao analisador sintático com uma codificação numérica, assim como as palavras reservadas e os símbolos especiais que também são definidos como símbolos terminais.

Para determinar se uma sequência de caracteres representa um identificador, uma palavra reservada ou um identificador de constante, o analisador léxico consulta a tabela de símbolos, que guarda informações sobre uma determinada sequência de caracteres.

No que concerne ao analisador léxico a tabela de símbolos é uma tabela tipo "hashing" com a estrutura definida em IV.4.1. e que utiliza a função abaixo para calcular o endereço de entrada na tabela e o incremento para caso de colisão (endereço aberto com hash duplo). Os 10 bytes reservados para o identificador são numerados a seguir, para fa

cilitar a descrição do cálculo.



NOME[0:16] = NOME[0:16] ⊕ NOME[16:16] ⊕ NOME[32:16] ⊕ NOME[48:16] +
⊕ NOME[64:16]

NOME[0:16] = NOME[2:6] * NOME[10: 6]

ENDEREÇO = NOME[4:8]

INCREMENTO = NOME[12:4]

No início da compilação a tabela de símbolos contém todas as palavras reservadas, guardadas em endereços calculados pelo método de Brent [23].

O analisador léxico reconhece os elementos gramaticais da linguagem definidos pelas seguintes regras

	regra	código	informação adicional
1	<letra> { <letra> | <digito> }	identificador	entrada tabela símbolos
		ident.constante	valor constante
2	<digito> { <digito> }	numero	valor
3	' <carater> { <carater> } '	caracter em cadeia	carater

<table>
<tr><th></th><th>regra</th><th>código</th><th>informação adicional</th></tr>
<tr><td>4</td><td>$</td><td>$</td><td></td></tr>
<tr><td>5</td><td>;</td><td>;</td><td></td></tr>
<tr><td>6</td><td>:</td><td>:</td><td></td></tr>
<tr><td>7</td><td>|</td><td>|</td><td></td></tr>
<tr><td>8</td><td>|</td><td>|</td><td></td></tr>
<tr><td>9</td><td>(</td><td>(</td><td></td></tr>
<tr><td>10</td><td>,</td><td>,</td><td></td></tr>
<tr><td>11</td><td>)</td><td>)</td><td></td></tr>
<tr><td>12</td><td>/</td><td>/</td><td></td></tr>
<tr><td>13</td><td>: =</td><td>: =</td><td></td></tr>
<tr><td>14</td><td>=</td><td>=</td><td></td></tr>
<tr><td>15</td><td>+</td><td>+</td><td></td></tr>
<tr><td>16</td><td>-</td><td>-</td><td></td></tr>
<tr><td>17</td><td>.</td><td>.</td><td></td></tr>
<tr><td>18</td><td>*</td><td>*</td><td></td></tr>
<tr><td>19</td><td>'</td><td>'</td><td></td></tr>
<tr><td>20</td><td>BEGIN</td><td>palavra reservada</td><td></td></tr>
<tr><td>21</td><td>END</td><td></td><td></td></tr>
</table>

	regra	código	informação adicional
22	ARRAY	palavra reservada	
23	RECORD		
24	STRUCTURE		
25	CASE		
26	ON		
27	GOTO		
28	WAIT		
29	INPUT		
30	OUTPUT		
31	CHAR		
32	NUM		
33	OF		
34	MOVE		
35	TO		
36	FOR		
37	ENDRI1		
38	REPEAT		
39	UNTIL		

	regra	código	informação adicional
40	TIMES	palavra reservada	
41	WHILE		
42	DO		
43	STEP		
44	IF		
45	THEN		
46	TRACEON		
47	TRACEOFF		
48	CLOCK		
49	OUTCOUNTER		
50	RJ		
51	PROCEDURE		
52	BYTE		
53	WORD		
54	LABEL		
55	EQUATE		
56	CONSTANT		
57	COUNTER		

	regra	código	informação adicional
58	READY	palavra reservada	
59	BUSY		
60	IOC		
61	RI1		
62	RI2		
63	RI3		
64	RI4		
65	RI5		
66	RI6		
67	OVFL		
68	NOTOVF		
69	EQ		
70	NE		
71	GT		
72	GE		
73	LT		
74	LE		
75	ELSE		
76	RX		

	regra	código	informação adicional
77	SLA	palavra reservada	
78	SRA		
79	SLAX		
80	SRAX		
81	SLCAX		
82	SRCAX		
83	RAC		
84	VALUE		
85	POS		
86	NEG		
87	ZERO		
88	NPOS		
89	NNEG		
90	NZERO		

Para as regras número 1, 2, 3, 6, 13 e 14 são utilizados automatos finitos. Para as regras número 4 , 5, de 7 a 12 e de 15 a 19 é utilizada a tradução direta, visto serem símbolos com um só elemento e que não dão origem a dúvidas.

Para as regras a partir do número 20, as palavras reservadas inicialmente são reconhecidas como identificadores e depois recebem codificação correta após consulta à tabela de símbolos. Procedimento identico ocorre para identificador de constante.

É feita uma deteção dos erros que independem do contexto, tais como caracteres inválidos e identificadores com mais de 10 letras.

IV.3. ANALISADOR SINTÁTICO

IV.3.1. MÉTODOS DE COMPILAÇÃO

O analisador sintático usa o método da matriz de transição (Gries [9]) para fazer o "parsing" do programa. Trata-se de um método "bottom-up" com análise feita determinando-se repetidamente o "handle" (u) da forma sentencial que está sendo analisada e reduzindo-o a um não-terminal U, usando a regra U:: = u. O método é essencialmente determinístico e o tempo de "parsing" é proporcional ao comprimento do texto fonte.

O método exige que a gramática original esteja na forma de gramática de operadores (Gries [9]) e que pertença à classe de gramática tratáveis por matriz de transição. Inicialmente é feita uma transformação na gramática de modo a reduzir o comprimento dos lados direitos das produções a limi

tes inferiores ou iguais a 3 símbolos, através da adição de novos não-terminais denominados "não-terminais estrelados".

IV.3.2. GRAMÁTICAS DE MATRIZ DE TRANSIÇÃO

Para usar o método de matriz de transição é necessária que a gramática de definição da sintaxe da linguagem seja uma gramática que chamaremos de "matriz de transição" (GMT). Uma GMT é uma gramática de operadores (90) com algumas restrições.

IV.3.2.1. DEFINIÇÃO

a) Seja $G(N, \Sigma, P, S)$ uma gramática livre de contexto onde:

1. N é um conjunto finito de símbolos não terminais
2. Σ é um conjunto finito de símbolos terminais e $N \quad \Sigma = \emptyset$.
3. P é um subconjunto finito de $N \times (N \cup \Sigma)^*$; um elemento (α, β) em P é escrito como $\alpha \rightarrow \beta$ e é chamado de produção
4. S é um símbolo de N chamado de símbolo inicial.

b) Sejam U, V e W, a representação de símbolos não terminais; a, b, c a representação de símbolos terminais e α, β, γ, δ a representação de formas sentenciais que são subconjuntos de $(\Sigma \cup N)^+$.

c) Seja U* a representação de um não- terminal estrelado. Estes não-terminais são gerados quando a gramática original é transformada conforme o algoritmo descrito por Gries [9], página

d) Seja PR (T) o conjunto dos predecessores de T:

$PR(T)=\{T_1 | \ldots T_1 T \ldots$ é uma forma sentencial$\}$

e) GRAMÁTICA DE OPERADORES

G é uma GO se não existe em P nenhuma produção da forma U → ... VW ... , onde V e W são não terminais.

f) GRAMÁTICA DE MATRIZ DE TRANSIÇÃO

G é uma GMT se ela for uma GO e obedecer às seguintes condições adicionais:

condição 1:

- para cada par (U*,T) no máximo uma das

relações abaixo é válida

a) existe a regra

$U \rightarrow U^*$, $V \in PR(T)$, U é único

b) existe a regra

$U_1^* \rightarrow U^*T$

c) existe a regra

$U_1^* \rightarrow T$, $U^* \in PR(U_1^*)$

condição 2:

- para cada tripla (U^*,U,T) no máximo uma das três relações abaixo prevalece:

  a) existe a regra

  $U_1^* \rightarrow U^*U_2$, $U_1 \in PR(T)$; $U_2 \overset{*}{\Rightarrow} U$;

  U_1 e U_2 únicos

  b) existe a regra

  $U_1^* \rightarrow U^*U_2T$, $U_2 \overset{*}{\Rightarrow} U$; U_2 é único

  c) existe a regra

  $U_1^* \rightarrow U_2T$, $U^* \in PR(T)$, U_2 é único

IV.3.3. DETERMINAÇÃO DA MATRIZ DE TRANSIÇÃO

O cálculo da matriz de transição envolve, inicialmente, a transformação na gramática original atráves da introdução dos não-terminais estrelados. A gramática transformada, chamada de gramática aumentada de operadores é equivalente à gramática original.

Utilizamos para o cálculo da matriz de transição o gerador de analisadores sintáticos "NHÃONHÃO" (Simone [10]) que nos fornece a gramática aumentada, a matriz de transição compactada e seu método de acesso.

IV.4. CÓDIGO OBJETO E ALGUMAS ROTINAS SEMÂNTICAS

Para orientar o projeto da linguagem e a geração do código foram utilizados os postulados feitos por Wirth [8] para o projeto do PL360:

1 - instruções que expressam operações sobre dados devem corresponder, de maneira óbvia, às instruções de máquina. Sua estrutura deve ser tal que possa ser decomposta em elementos estruturais, cada um correspondendo diretamente a uma única instrução.

2 - nenhuma utilização de memória deve ser ocultada do programador. Em particular o uso de registros deve ser explicitado no programa.

3 - o controle de sequência deve estar expressa implicitamente na estrutura de certos comandos.

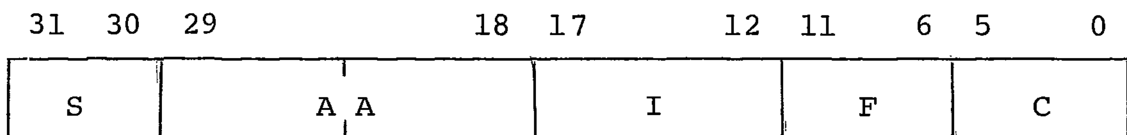
IV.4.1. ESTRUTURA DOS DADOS NO COMPILADOR

1. MEMÓRIA

array MEMO [0:3999]

significando um vetor de 4000 posições com elementos de tamanho de 32 bits.

1.1. Representação para instrução



S - sinal (00 - positivo e 10 negativo)

AA - endereço

I - registro de índice

F - modificador

C - código

No compilador será usada a seguinte notação:

S(X) = MEMO[X].[31:2]

AA(X) = MEMO[X].[29:12]

I(X) = MEMO[X].[17:6]

F(X) = MEMO[X].[11:6]

C(X) = MEMO[X] [5:6]

1.2. Representação para dados

31 30	29 24	23 18	17 12	11 6	5 0
0	1	2	3	4	5
S					

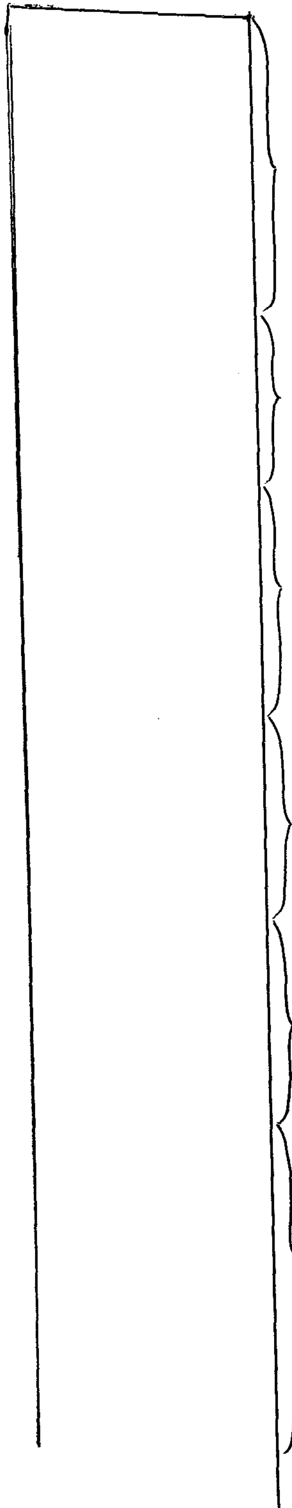
1.3. Alocação da Memória

As variáveis são alocadas a partir do endereço zero da memória.

São alocadas todas as variáveis e procedimento (variáveis e instruções) e após são alocadas as instruções do programa propriamente ditas.

As variáveis que representam expressão limite de comandos "FOR" ocupam posições de memória entre as instruções.

Uma alocação típica de memória é apresentada no esquema.



(1) endereços de variáveis tipo
WORD, BYTE, ARRAY, RECORD
ARRAY RECORD, TABELA CASE

parâmetros
formais

variáveis locais
tipo (1)
(se houver)

instruções

procedimento

variáveis tipo (1)

instruções

2. Pilha do compilador

array STACK [0:200]

Representando um arranjo de 200 elementos com comprimento 9 bits cada. Esta pilha conterá os não-terminais estrelados (U*) indicativos da porção esquerda do texto ainda não completamente reduzida, para permitir o tratamento de um "handle" mais à direita no texto. O valor 200 poderá ser alterado durante geração do compilador.

3. Pilha de Ponteiros

array STACKPONT [0:200]

Representando 200 elementos cada um de 12 bits. O STACKPONT é uma pilha paralela a STACK. Para cada estrelado na pilha STACK, STACKPONT aponta para uma área auxiliar com informações que foram guardadas quando o estrelado entrou na pilha. STACKPONT [x] = 0 significa que não há informação na área auxiliar.

4. Pilha com informações

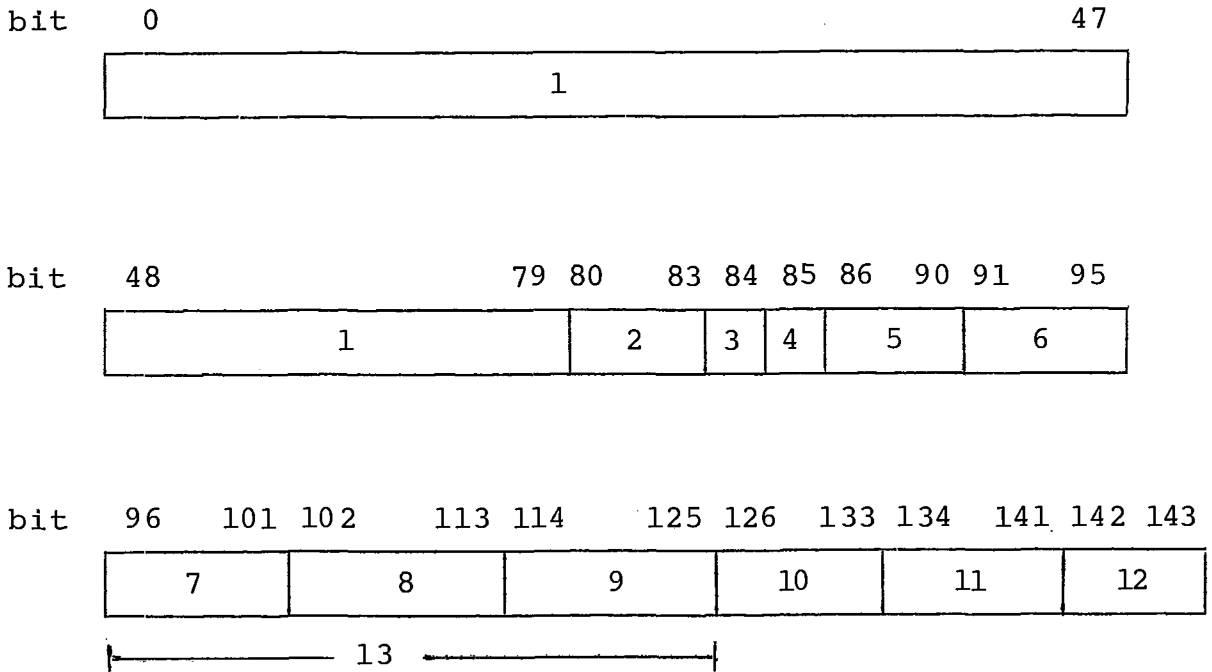
array AREA [0:600]

Representando 600 elementos de 32 bits. Guarda informações de atributos herdados (Kennedy [18]), ponteiros para memória ou para a tabela de símbolos.

5. Tabela de símbolos

array TABSIM [0:255]

Representando um arranjo com 256 elementos com 144 bits cada um. A tabela de símbolos contém informações sobre as palavras reservadas e os identificadores declarados. A distribuição das informações é feita nos seguintes campos:



(1) NOME(x) =TABSIMB[X].[0:80] - nome do identificador

(2) TIPO(x) =TABSIMB[X].[80:4] - tipo da variável

(3) JADEF(x)=TABSIMB[X].[84:1] - se igual a 1 estão completas as informações sobre o identificador. Caso contrário, é igual a D.

(4) OCUPADO(X) = TABSIMB[X].[85:1] - se igual a 1 indica entrada na tabela ocupada. Caso contrário igual a 0.

(5) BYTESQ(X) = TABSIMB[X].[86:5] - número do bit mais a esquerda do campo da palavra.

(6) NBYTES(X) = TABSIMB[X].[91:5] - número de bytes a serem utilizados.

(7) F1(X) = TABSIMB[X].[96:6] - valor do campo F do código do record.

NPAL(X) = TABSIM[X].[96:6] - número de palavras por nó do record.

NPARM(X) = TABSIMB[X].[96:6] - número de parâmetros do procedimento.

REGISTRO(X)=TABSIM[X].[96:6] - número do registro associado à variável tipo EQUATE.

(8) ENDEREÇO(X)=TABSIMB[X].[102:12]- endereço na memória

TAMARREC(X)=TABSIMB[X].[102:12]- número de elementos de um vetor de records.

CODIGO(X)= TABSIMB[X].[102:12]- código da palavra reservada

(9) LINK(X) = TABSIMB[X].[114:12] - ligação para a memória

DIMENS(X) =TABSIMB[X].[114:12]- número palavras vetor WORD

(10)LINKT(X) =TABSIMB[X].[126:8] - ligação para tabela de símbolos.

(11)REPETIDO(X)=TABSIMB[X].[134:8]- se diferente de HEX"FF" aponta para entrada na tabela de símbolos de identificador com mesmo nome.

(12) SINALCTE(X)=TABSIMB[X].[143:2] - sinal de uma constante

(13) VALORCTE(X)=TABSIMB[X].[96:30] - valor de uma constante

Alguns campos se superpõem, porque são mutuamente exclusivos, minimizando a largura da área ocupada.

Para cada tipo de variável admissível na linguagem são apresentadas as informações armazenadas na tabela de símbolos, além do nome.

As observações abaixo referem-se à tabela na próxima folha.

(1) se dentro de record, aponta para próxima definição de nível

(2) se dentro de record, aponta para nome do record
se dentro de procedure, aponta para próxima definição de variável local.

(3) ligação para primeira palavra do record

(4) se dentro de procedure, aponta para próxima definição de variável local

(5) ligação para primeiro parâmetro formal, se houver. Se não, ligação para a primeira declaração de variável local, se houver.

(6) ligação para a memória com a primeira instrução com referência futura.

							13					
	1	2	3	4	5	6	7	8	9	10	11	12
PALAVRA RESERVADA	id	0	1	x				CÓDIGO				
WORD	id	1	1	x	1	5	Fl = 5	x ENDEREÇO	x LINK(1)	x(2)	x	
BYTE	id	2	1	x	x	x	Fl = X	x ENDEREÇO	x LINK(1)	x(2)	x	
RECORD	id	3	x	x			x NPAL		x LINK(3)	x(4)	x	
ARRAY WORD	id	4	x	x	1	5	Fl = 5	x ENDEREÇO	x DIMENS	x(4)	x	
ARRAY RECORD	id	5	x	x			x NPAL	x TAMARREC	x LINK(3)	x(4)	x	
PROCEDURE	id	6	x	x			x NPARM	x ENDEREÇO		x(5)	x	
LABEL	id	7	x	x				x ENDEREÇO	x LINK(6)	x(4)	x	
EQUATE	id	8	1	x			x REGISTRO			x(4)	x	
CONSTANT	id	9	1	x				X		x(4)	x	x
TABELA CASE	id	10	1	x				x ENDEREÇO	x DIMENS	x(4)	x	
COUNTER	id	11	x	x				x ENDEREÇO	LINK(6)	x(4)	x	
PARAM.FORM.WORD	id	12	1	x	1	5	Fl = 5	x ENDEREÇO		x(2)	x	
PARAM.FORM.BYTE	id	13	1	x	x	x	x Fl	x ENDEREÇO		x(2)	x	

x - representa presença de informação com conteúdo variável

1 - NOME

2 - TIPO

3 - JADEF

4 - OCUPADO

5 - BYTESQ

6 - NBYTES

7 - F1,NPAL, NPARM,REGISTRO

8 - ENDEREÇO,TAMARREC, CÕDIGO

9 - LINK,DIMES

10 - LINKT

11 - REPETIDO

12 - SINALCTE

6. Tabela de códigos

array CODIGO [0:7]

Representando 7 elementos de 48 bits cada. A tabela geral de códigos se encontra no apêndice. Para se compreender as informações em algumas rotinas é mostrada a estrutura da tabela, em se tratando de valores dos códigos em relação aos registros.

Exemplo: LOAD

0	6	12	18	24	30	36	42
RAC	RI1	RI2	RI3	RI4	RI5	RI6	RX

CODIGO [1].[0:6] - LDA

CODIGO [1].[i*6:6] - LDi, $1 \leq i \leq 6$

CODIGO [1].[42:6] - LDX

IV.4.2. ALGORITMO DO COMPILADOR

O compilador recebe do gerador de analisadores sintáticos "NHÃONHÃO" (Simone [10]) a matriz de transição com as informações sobre a relação entre cada (U*,U,T) e a redução a ser efetuada. Além disso o gerador fornece uma indicação do tipo de redução a ser efetuada:

> se usadas as condições a.1. ou a.2. da Sec.IV.3.2.1 fls. 97 e 98.

= se usadas as condições b.1. ou b.2. da Sec.IV.3.2.1, fls. 97 e 98.

< se usadas as condições c.1. ou c.2. da Sec.IV.3.2.1 fls. 97 e 98.

A compilação é iniciada com o delimitador de programa ($) na pilha STACK. Este delimitador é o usado na produção extra da gramática: <S> → $ <programa> $ para permitir informar ao compilador a ocorrência do final físico do programa.

A compilação é processada "token" a "token" e quando é encontrado o "token" $ o analisador léxico informa ao analisador sintático o fim da entrada. A compilação é encerrada, estando o código objeto pronto para a execução assim que é encontrado "$", caso o programa esteja correto.

IV.4.2.1. ALGORITMO DE PARSING

O algoritmo de "parsing" utiliza uma pilha para não-terminais estrelados, uma variável para não-terminal simples e a matriz de controle. A cada momento tem-se o "estrelado" (U*) no topo da pilha, a variável que representa a última redução efetuada (U) e o símbolo enviado pelo analisador léxico (T). Fazendo-se uma entrada na matriz pela linha correspondente a U* e coluna equivalente a T, obtém-se a redução a ser usada. Para cada redução tem-se uma rotina semântica associada e o teste para o não-terminal U. Isto é o que chamaremos uma rotina para a tripla (U*, U, T).

A decomposição do analisador em diversas rotinas, cada uma encarregada do tratamento de uma situação precisa (U*, U, T) facilita a determinação dos erros, gerando mensagens claras ao usuário e procedimentos de recuperação de erros convenientes.

IV.4.2.2. ESQUEMA DO PROGRAMA

Será mostrado um esquema simplificado do compilador usando a linguagem ALGOL para representá-lo.

```
begin
    SCANNER;
    while NAOACABOU do
       begin
           TRIPLA;
           Case RELAÇÃO of                    % <, >, =
               begin
               1: % relação MENOR
                  SEMÂNTICA(URED);            % URED fornecido pe
                  PUSH;                         lo gerador
                  U:= EPS;                    % EPS significa au-
                                                sência de símbolo
                  SCANNER;                    % em U.
               2: % relação MAIOR
                  SEMÂNTICA(URED);
                  U: = URED;
                  POP;
               3: % relação IGUAL
                  SEMÂNTICA(URED)
                  POP;
                  PUSH;
                  U: = EPS
                  SCANNER;
               end;
         end
  end
```

SCANNER fornece o próximo símbolo, como explicado na seção IV.2. e informa quando termina a cadeia de entrada.

A rotina TRIPLA fornece a relação entre U* e T e a redução URED. Para uma relação MENOR, URED:: = UT | T; para MAIOR, URED:: = U* U | U* e para IGUAL, URED:: = U*UT|U*T.

SEMÂNTICA é a rotina onde são tomadas as ações semânticas e gerado o código objeto. Em geral a cada URED corresponde uma ação semântica. Algumas vezes para um mesmo URED são tomadas ações semânticas diferentes em função de U* ou U.

A estrutura da rotina semântica se resume a:

```
case URED of
   begin
      URED1:
      URED2: case U of
                begin
                  U1:
                  U2:
                   :
                end;
      URED3:
      URED4: case U* of
                begin
                  :
                end
       :
      UREDN
   end
```

IV.4.2.3. ALGUMAS ROTINAS SEMÂNTICAS

Serão apresentadas as rotinas semânticas relativas a certas produções da gramática aumentada, refletindo alguns comandos da linguagem e mostrando o código gerado, a situação da pilha do compilador, das duas pilhas auxiliares e da memória.

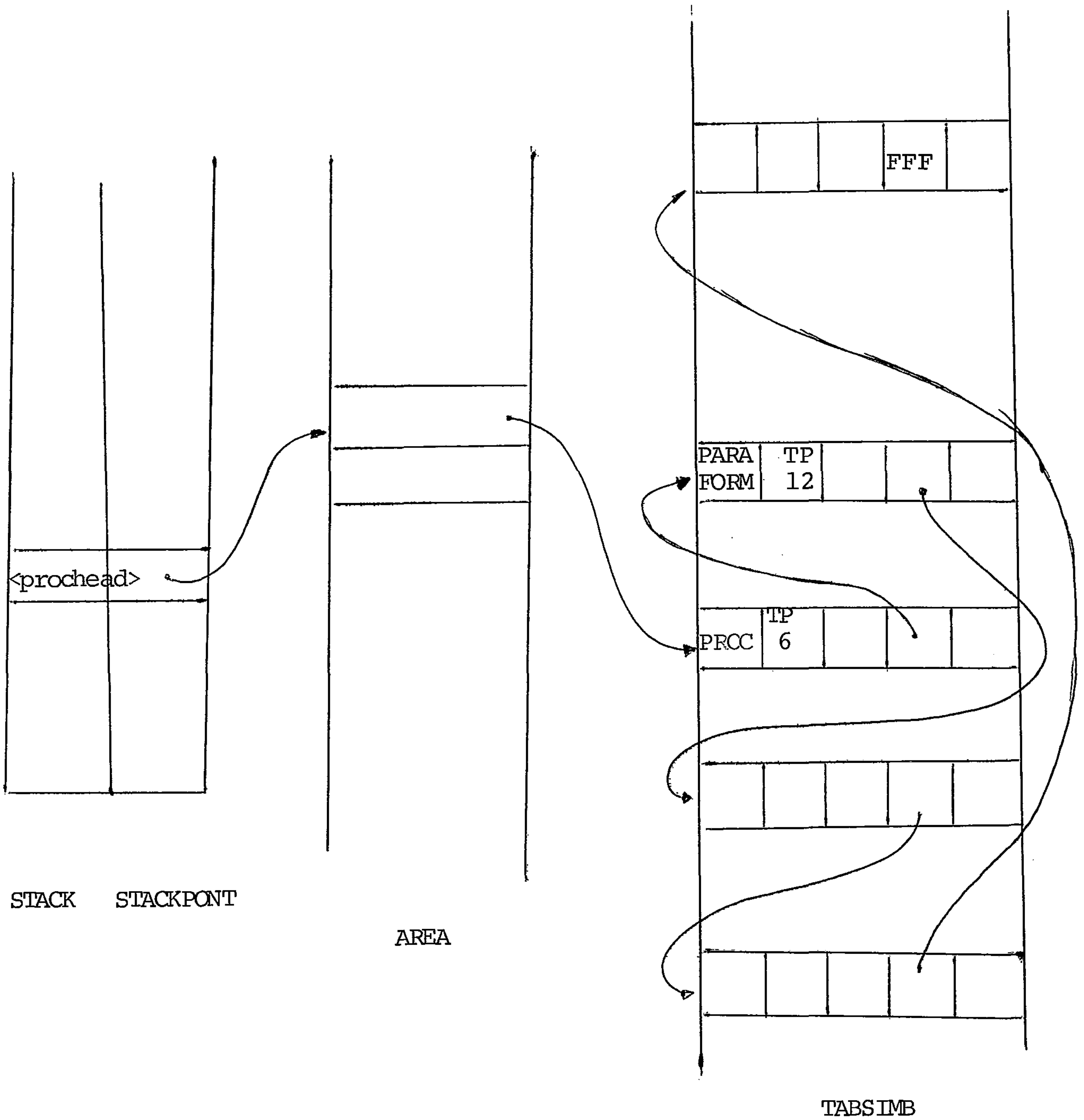
A tradução dos comandos foi apresentada na definição da linguagem, porém achamos conveniente repetir este esquema para melhor compreensão das rotinas descritas.

Esta seção pretende apenas evitar a exposição completa do compilador neste trabalho, devido à sua extensão. O programa completo estará disponível quando de sua implementação no MITRA 15 do Laboratório de Automação e Simulação de Sistemas da COPPE-UFRJ.

92: % <DECPROC> ::= <PROCHEAD> : <CORPOPROC>

```
        DCL: = true; % acabou declaração de procedure e pos
                     % so receber novas declarações

        % retirar as variáveis locais da tabela de símbolos

        DISP: = STACKPONT[TOP] ; % apontador para área   onde
                                 % estão informações auxilia
                                 % res.
```

```
LK: = LINKT (AREA[DISP]); % aponta para primeiro parâ
                          % metro formal (se houver )
                          % ou para primeira variável
                          % local.
TP: = TIPO (AREA[DISP]):  % tipo do identificador
while LK      = 4" FF"
do begin
   % TP = 6 - procedure
   % TP = 12 - parâmetro formal word
   % TP = 13 - parâmetro formal byte
   if (TP ¬ = 6 OR TP¬= 12  OR TP ¬ = 13)
   then begin % retirar da tabela
        OCUPADO (LK) = 0 ;
        if REP: = REPETIDO (LK)¬ = 4"FF" % no caso de
        then REPETIDO (REP): = 4"FF"     % variável com
        end;                             % mesmo nome
                                         % na tabela
   LK: = LINKT (LK) ;
   TP: = TIPO (LK) ;
   end;
```



95: % <instrução>

175: % <instr>:: <u>GOTO</u> <ident>

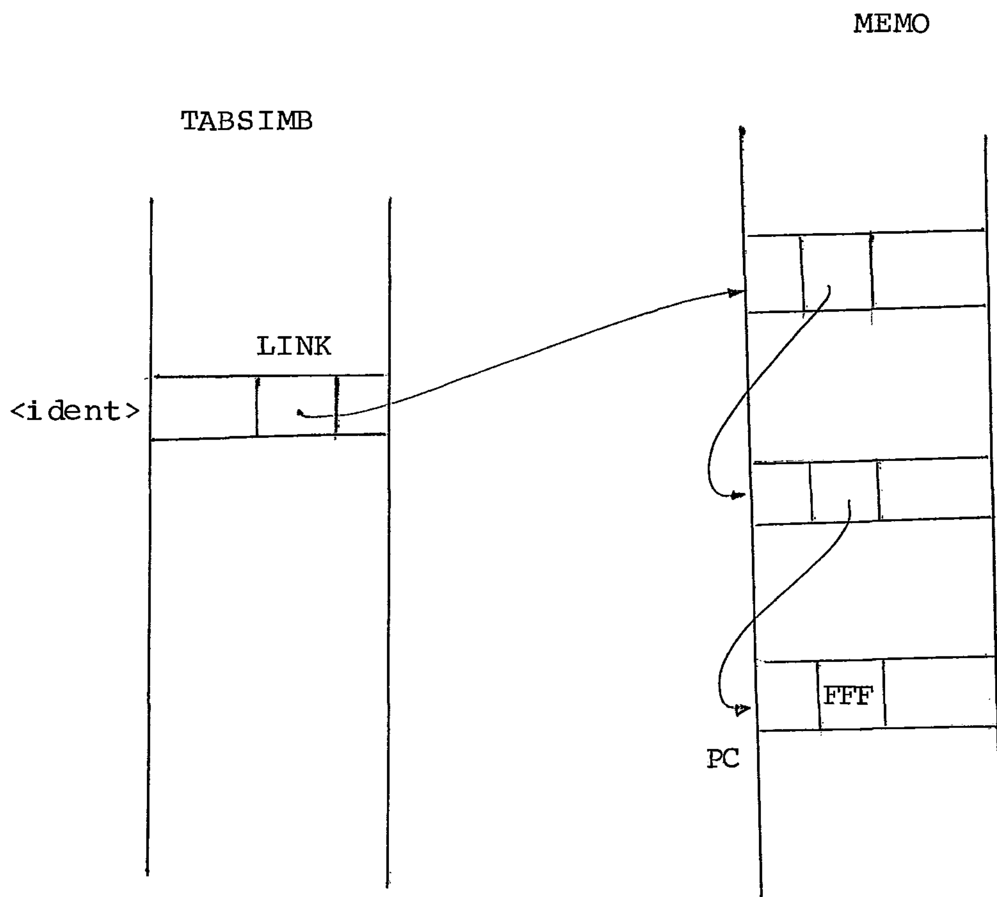
if AA1: = ROTULO (ENDER) ¬ = - 1

then GERACODIGO (PC, 0, AA1, 0, 0, 39) % IMP

else ERROM ( ) ;

observação: a rotina ROTULO devolve em AA1 o endereço do label <ident> se este já tiver aparecido no programa. Se ele liga o último desvio não resolvido a este, sendo que este recebe 4"FF" no campo de endereço.
Se <ident> não estiver numa declaração LABEL, AA1 recebe o valor - 1.

Esquema quando rótulo ainda não foi definido:



95: % <instrução>

188: % <instr>:: = REPEAT <instr> UNTIL <condição>

% completar informação na última instrução gerada

AA (PC - 1): = AREA [STACKPONT [TOP]];

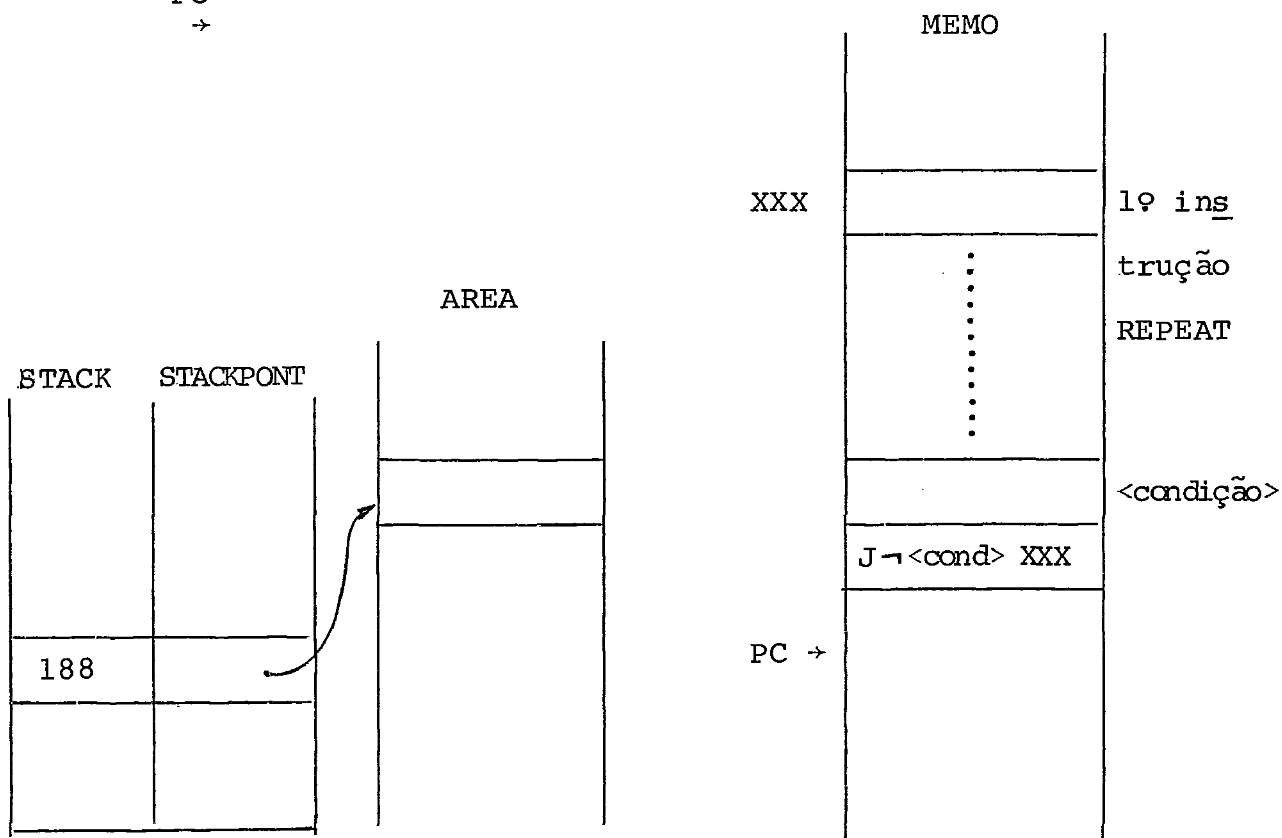
Esquema:

XXX <instr>

<cond>

J ¬ <cond> XXX

PC →



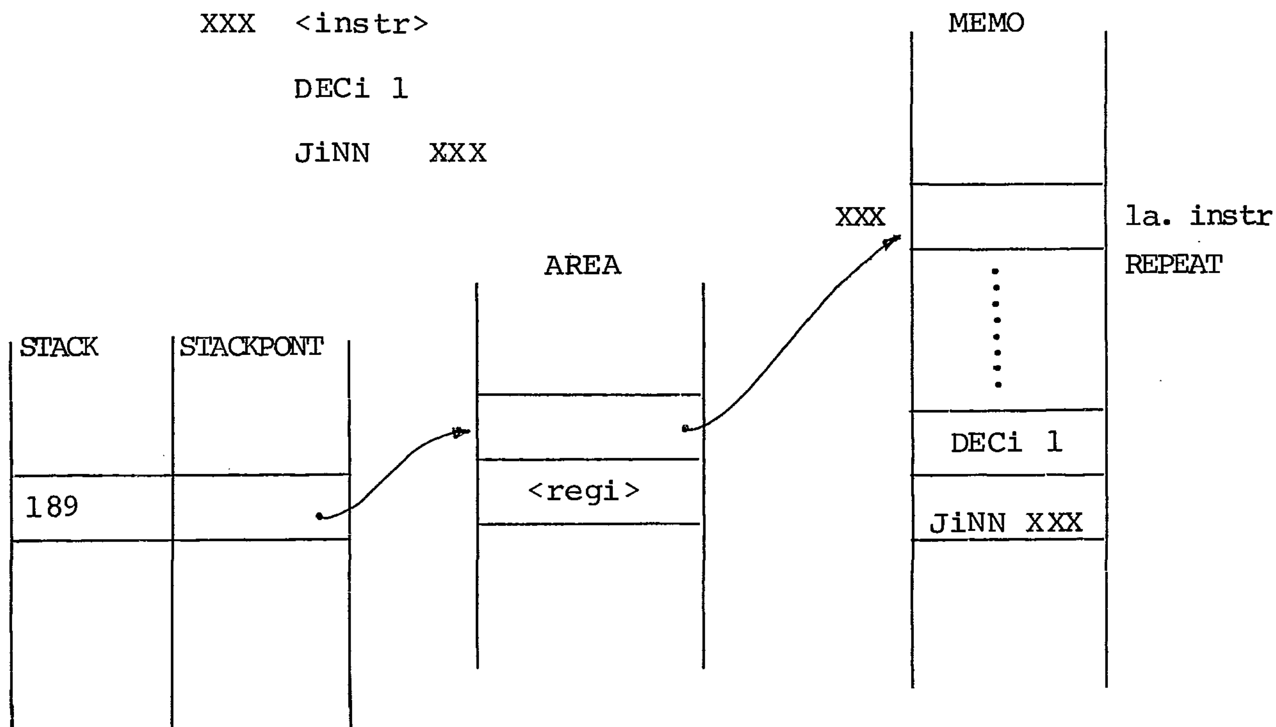
188 = REPEAT <instr> UNTIL

95: % <instr>

189: % <instr>:: <u>REPEAT</u> <<u>regi</u>> <u>TIMES</u> <instr>

```
DISP: = STACKPONT[TOP];

C1: = CODIGO[7,AREA[DISP + 1]]; % código para decre
                                % mentar em função
                                % de <regi>.

GERACODIGO(PC,0,1,0,1,C1); % DEC <regi> 1

C1: = CODIGO[6,AREA[DISP + 1]]; % código de desvio
                                % em função do <regi>

GERACODIGO(PC,0,AREA|DISP|,0,3,C1); % J <regi> NN
```

Esquema:



189 = <u>REPEAT <regi> TIMES</u>

95: % <instr>

191: % <instr>: : = <u>WHILE <cond> DO</u> <instr>

DISP: = STACKPONT [TOP]

GERACODIGO(PC,0,AREA[DISP],0,0,39); % JMP

% completar J ¬ <cond> XXX

AA (AREA[DISP + 1]): = PC ; % XXX = PC

Esquema:

XYZ "LDA"

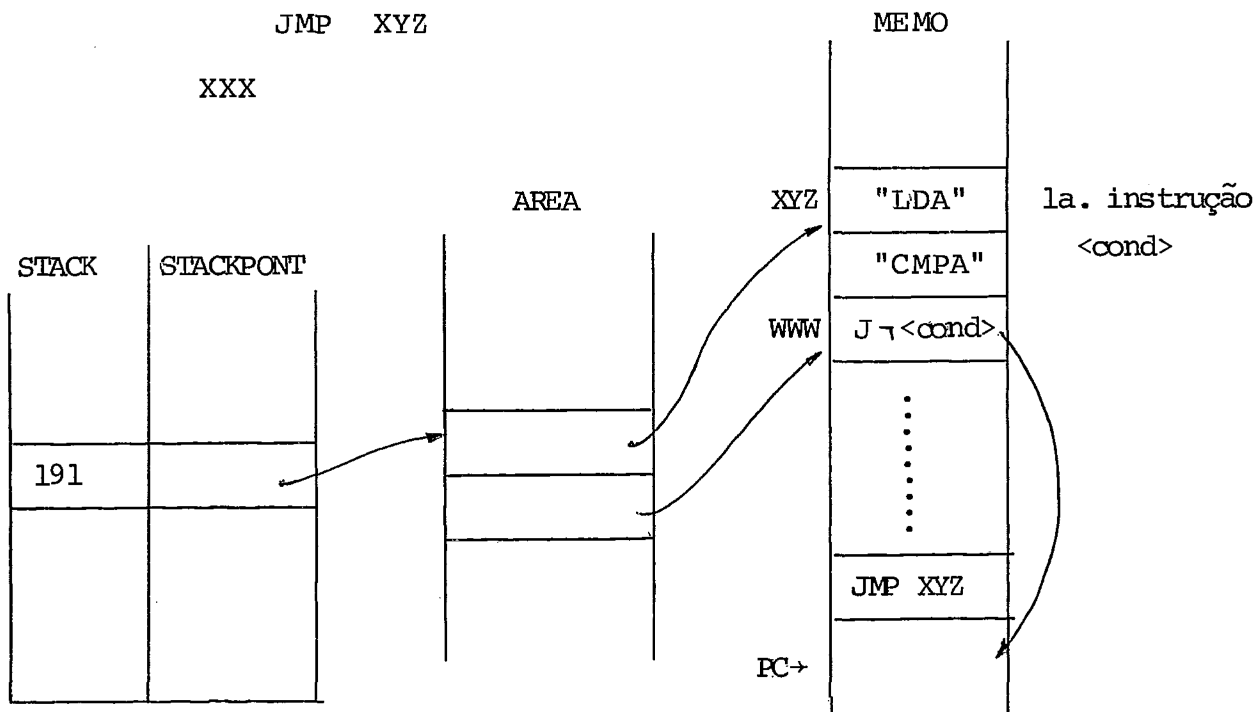
"CMPA"

WWW J ¬<cond> XXX

<instr>

JMP XYZ

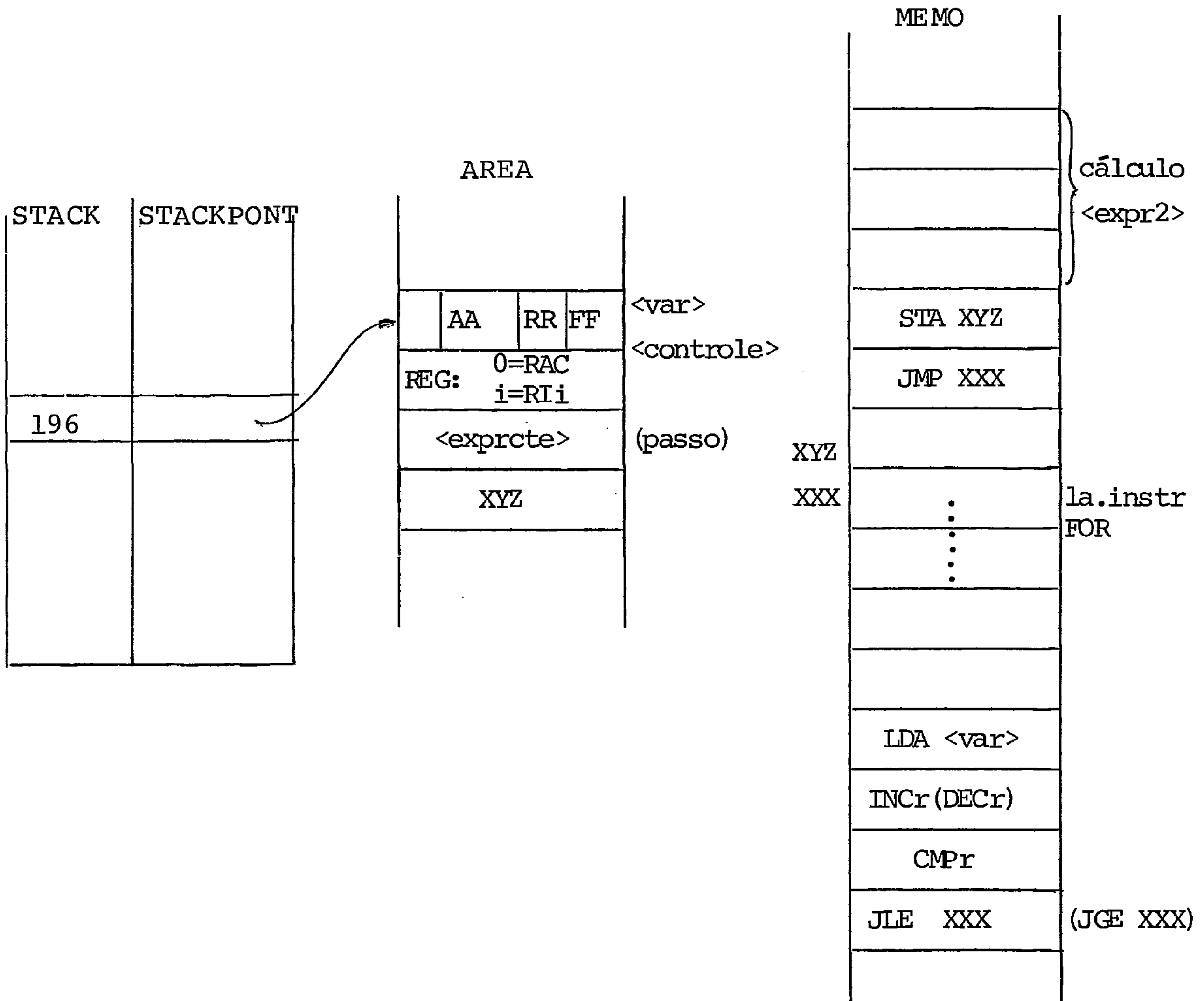
XXX



191 = <u>WHILE <cond> DO</u>

```
95: % <instr>

    196: % <instr>:: FOR<ident>:<expr2>STEP<exprcte>UNTIL<expr2>DO<instr>

    200: % <instr>:: FOR<regi>:=<exprl>STEP<exprcte>UNTIL<exprl>DO<instr>

       DISP: = STACKPONT[TOP];

       if AREA[DISP]  = 1  % se = 1 → var controle é <regi>

       then begin

            AA1: = AREA[DISP].EE; % especificação dos campos

            REG: = AREA[DISP].RR; % da variável de controle.

            FD : = AREA[DISP].FF;

            GERACODIGO(PC,0,AA1,REG,FD,8); % LDA <var>

    C1: = CODIGO[7,AREA[DISP + 1]]; % INC ou DEC em função
                                    % do registro de cálculo

    if AREA[DISP + 2] > 0 % passo

    then begin

            F1: = 0; % INC

            F2: = 9; % JLE

            end

    else begin

            F1: = 1; % DEC

            F2: = 7; % JGE

            end;

    GERACODIGO(PC,AREA[DISP + 2],0,F1,C1); % INC<reg> passo
                                           % ou DEC<reg>passo
```

```
C1: = CODIGO[8,AREA[DISP + 1]]; % CMP em função de re
                                % gistro

GERACODIGO(PC,0,AREA[DISP + 3],0,FD,C1); % CMP<reg>
                                         % LIMITE

GERACODIGO(PC,0,AREA[DISP + 3]+1,0,F2,39); % JLE ou
                                           % JGE

Esquema para estrelado 196

"LDA" <expr2>1  % calcula valor inicial

STA <var>       % inicializa var controle

"LDA" <expr2>2  % calcula LIMITE


      STA         XYZ
     +---------------+
XYZ  |               |  % endereço onde está armazenado LIMITE
     +---------------+
XXX    <instrução>


LDA <var>

INCA <exprcte>  % ou DECA

CMPA XYZ        % compara com valor LIMITE

JLE  XXX        % ou JGE
```



196 = FOR <ident>: = <expr2> STEP <exprcte> UNTIL <expr2> DO

95: % <instr>

202: % <instr>:: IF <cond> THEN <instr>

% <instr>:: IF < > THEN <elsecl>

% atualizar J¬ <cond>

AA (AREA[STACKPONT[TOP]]): = PC;

Esquema

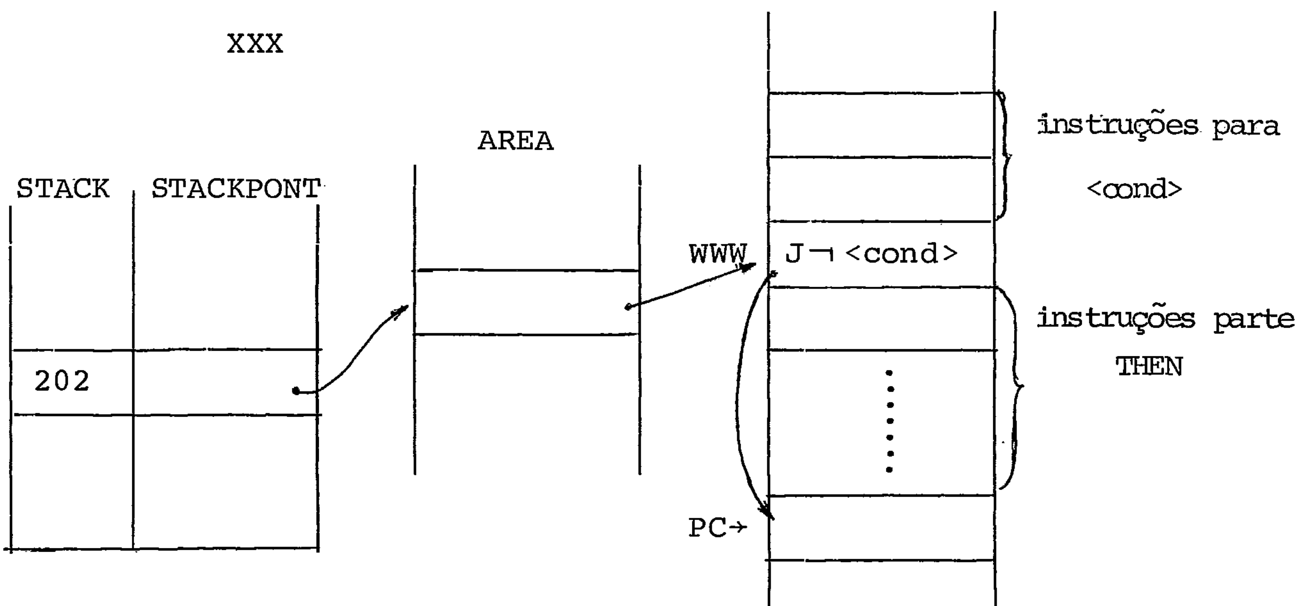
a) IF <cond> THEN <instr>

<cond>

WWW J ¬<cond> XXX

<instr>

XXX



202: IF <cond> THEN

b) IF <cond> THEN <elsecl>

<cond>

J <cond> XYZ

<instr>

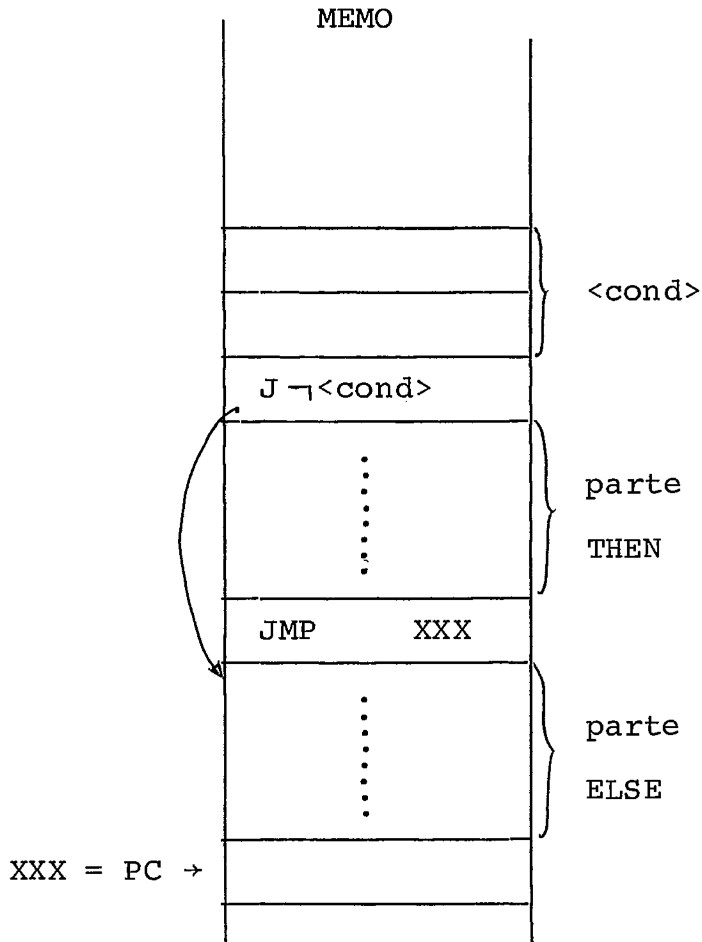
WWW JMP XXX

XYZ <instr>

XXX

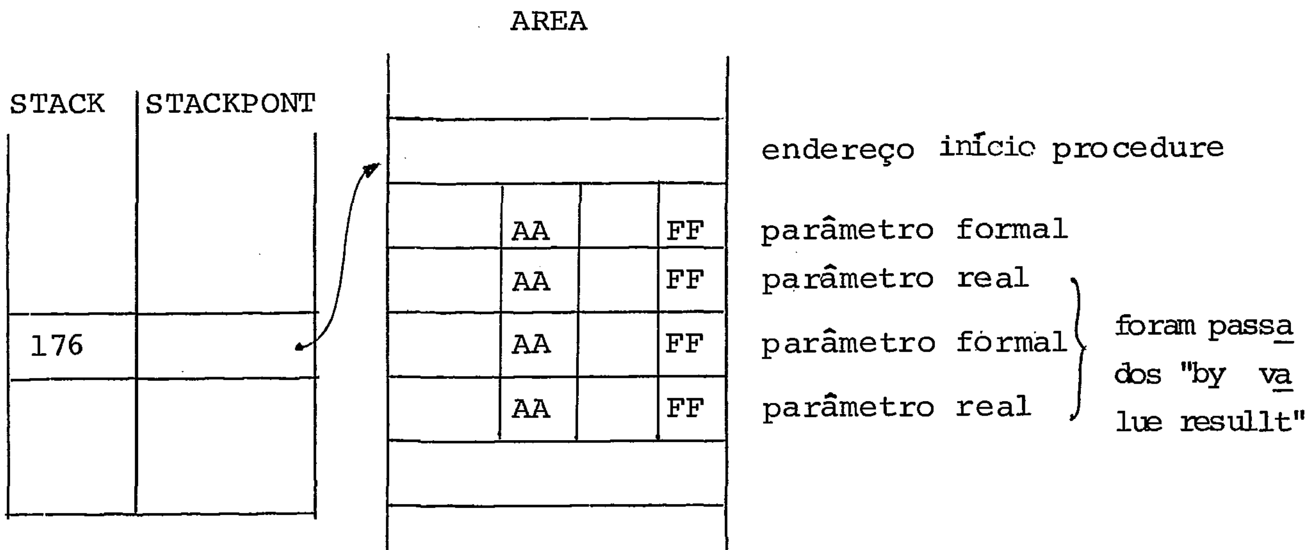
STACK, STACKPONT e AREA

iguais ao anterior




```
144 : % <lparm>:: = <expr2>
      % <lparm>:: = <lparm>,<expr2>

          NP: = * + 1:
          DISP: = STACKPONT[TOP] + 1;
          if NP > NPARM
          then ERROM (    )
          else begin

               if MEIO = 103  % <ident>
               then begin
```

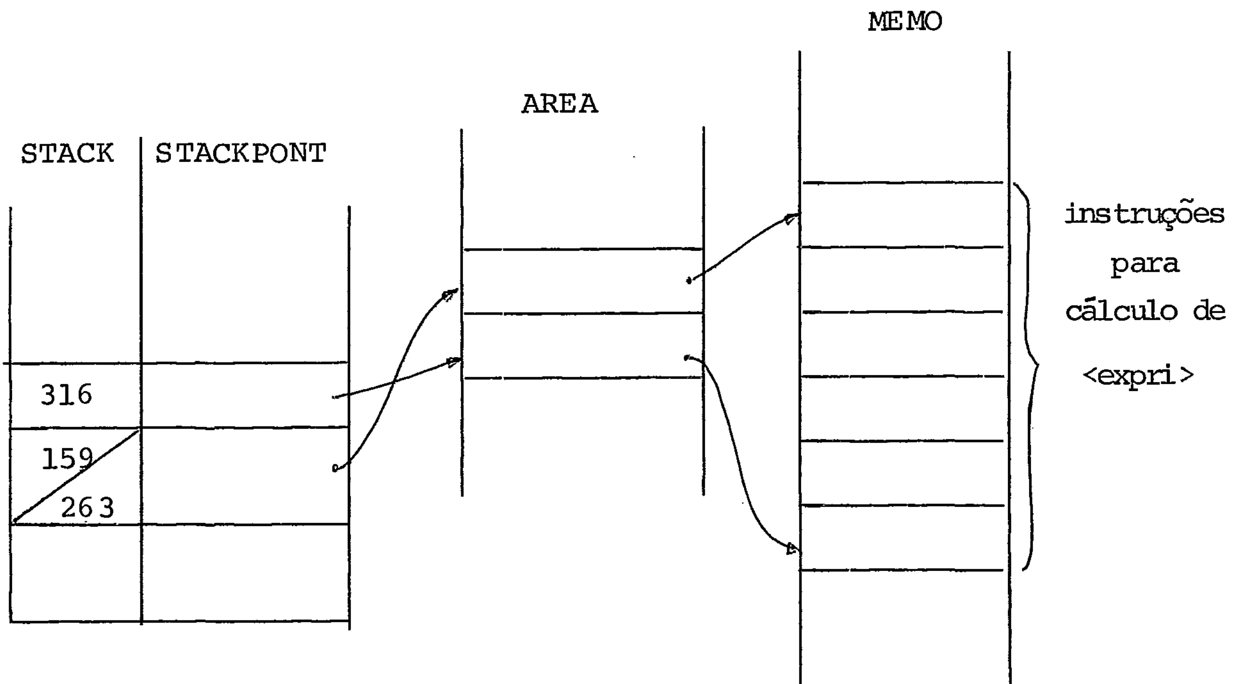
```
AREA[DISP].EE: = ENDEREÇO(END1); % END1 entrada
AREA[DISP].FF: = F1(END1);        % na TABSIMB
                                  % parâmetro formal
AREA[DISP + 1].EE: = ENDEREÇO(ENDER); % ENDER entra
AREA[DISP + 1].FF: = F1(ENDER)        % da na TABSIMB
DISP: = * + 2;                        % param. real
RET: = * + 1; % num param. com passagem "by value -
              % result"
end;
END1: = LINKT(END1); % próximo parâmetro formal
```



176 - <ident>

```
137: % <express>:: = <expri>,<regi>
     % o registro onde será efetuado o cálculo da expressão
     % só é conhecido após gerado o código.
     % Deste modo o código é gerado supondo ser calculado em
     % RAC e depois é somado a cada código o valor do
```

```
% registro de índice para dar o código real.
for I: = AREA[STACKPONT[TOP - 1]] step 1    until
          AREA[STACKPONT[TOP]]
do C(I): = * + REG;
```



159 = [*

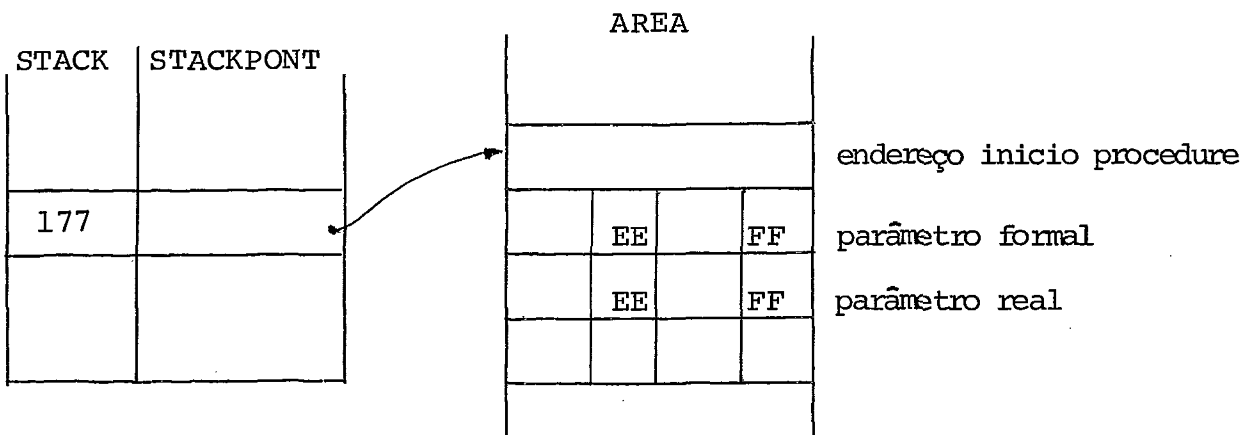
ou

263 = <ident>[*

316 = <expri>,

177: % <u><ident> (<ℓparm>)</u>

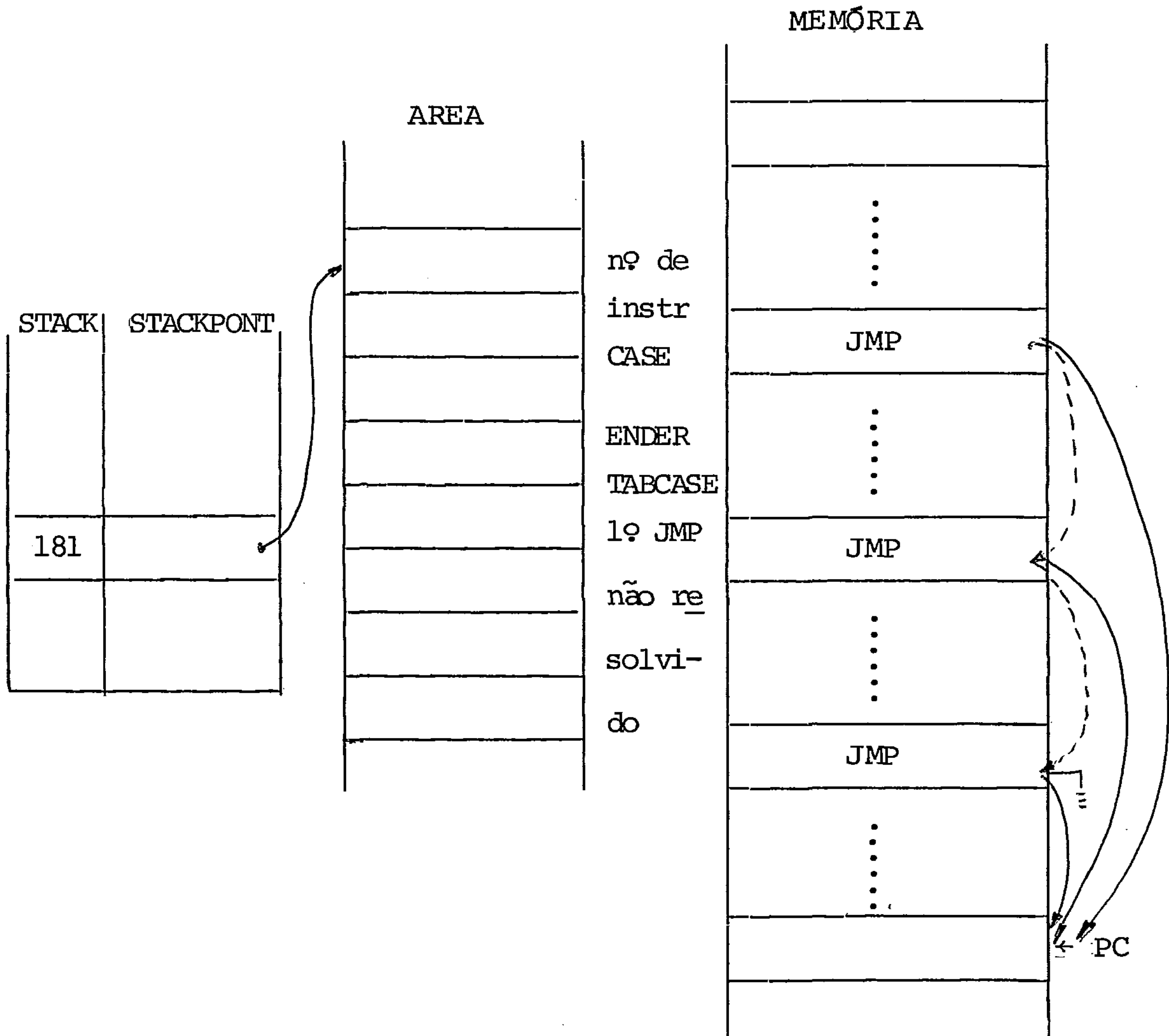
```
if NP = NPARM
then begin
      DISP: = STACKPONT[TOP]
      GERACODIGO(PC,0,AREA[DISP],0,0,39); % JMP para sub
                                          % rotina
      % atribuição de retorno aos parâmetros reais
      % que são passados por valor-resultado
      for J: = 1 step 1 until RET
      do begin
         % LDA @ PARAM FORMAL
         AA1: = AREA[DISP + J].EE;
         FD : = AREA[DISP + J].FF;
         GERACODIGO(PC,0,AA1,0,FD,8);
         % STA @ PARAM REAL
         AA1: = AREA[DISP + J + 1].EE;
         FD : = AREA[DISP + J + 1].FF;
         GERACODIGO(PC,0,AA1,0,FD,24);
         end;
```



177 <ident> (<ℓparm>)

181: % CASE <regi>/<ident> OF BEGIN <listainstr> END

```
% primeira entrada na tabela case é o endereço da pró
  xima instrução após o CASE

DISP: = STACKPONT[TOP] + 1;

MEMO[AREA[DISP]]: = PC;

% atualizar endereços dos desvios de saída para cada
% opção do CASE

END1: = AREA[DISP + 2]; % do primeiro desvio não resolvi-

do    begin
      T: = END1;
      END1: = AA(END1); % próximo desvio
      AA(T): = PC;
      end
until END1 = 4"FFF";
ECASE: = false;
```



203: <ident>:

```
if TIPO(ENDER)¬ 7  or TIPO(ENDER)¬ = 11
then ERRO (  )
else if JAEXISTE(ENDER)= 1  % dupla ocorrência
     then ERRO  (  )
     else begin
          JAEXISTE(ENDER): = 1;
          ENDEREÇO(ENDER): = PC;
          if TIPO(ENDER) = 1 % contador
          then begin % gera codigo instr. LOOP
```

```
        MEMO[PC].[31:20]: = 0; % inicializa contador
        MEMO[PC].[11:6]: = 4; % FD = 4
        MEMO[PC].[5:6]: = 5; % C = 5
        end;
    % resolvendo as referências futuras
    END1: = LINK(ENDER);
    if END1¬= 4"FFF"
    then    do    % existem referências
          begin
           T: = AA(END1);
           AA(END1): = PC;
           END1: = T
           end
           until END1 = 4"FFF";
```

Observação: mesmo esquema de GOTO<ident>.

IV.5. TRATAMENTO DE ERROS

Tratamento de erros é o processo de determinar como continuar a analisar o programa fonte após ter sido encontrado um erro.

Os erros podem ser classificados em erros léxicos, sintáticos ou semânticos, dependendo da fase da análise em que ocorrem.

a. Erros na fase de Análise Léxica

A função do analisador léxico é transformar uma sequência de caracteres, que constituem o programa fonte, em uma sequência de símbolos ("tokens"). Cada classe de símbolos tem uma especificação que é um conjunto regular. Se após algum processamento o analisador descobre que o caracter ou conjunto de caracteres não pertence a nenhuma classe então é chamada uma rotina de erro. O tratamento de erros na fase léxica se reduz a saltar os caracteres errados até achar um novo símbolo. Este tratamento simples é devido a falta de especificação no nível léxico da linguagem.

b. Erros na Fase de Análise Sintática

Frequentemente a parte relevante da deteção e tratamento de erros no processo de compilação ocorre na análise sintática. Uma das razões é o alto grau de precisão que pode ser alcançado na especificação da sintaxe das linguagens usando-se gramáticas livres de contexto.

O analisador sintático deteta um erro quando não há uma transição legal de sua configuração para outra , que é determinada por seu estado, o conteúdo da pilha e o símbolo recebido.

Existem vários estudos sobre tratamento de erros incluíndo:

1. Correção de nomes de identificadores com letras aparentemente trocadas ou com omissão ou inserção de uma letra. A recuperação foi tratada por Freeman [12] e Morgan [13] com complexidade que só é aplicável em estudos teóricos.

2. Tratamento baseado na distância de Hamming (Gries [14]) com algoritmos da ordem $n^3 \log n$ de tempo, sendo n a cadeia que está sendo analisada.

3. Definição de gramática com produções extras para efetuar transições de estado mesmo na presença de erro. Neste caso a gramática torna-se muito extensa acarretando compiladores que ocupam muito espaço e lentos.

4. Introdução de sequência de símbolos terminais na cadeia de entrada ou de símbolos na pilha. Este esquema pode introduzir uma sequência imprevista de erros que podem não serem detetados na compilação e gerarem um programa objeto com lógica diferente da pretendida pelo programador e imperceptível para este.

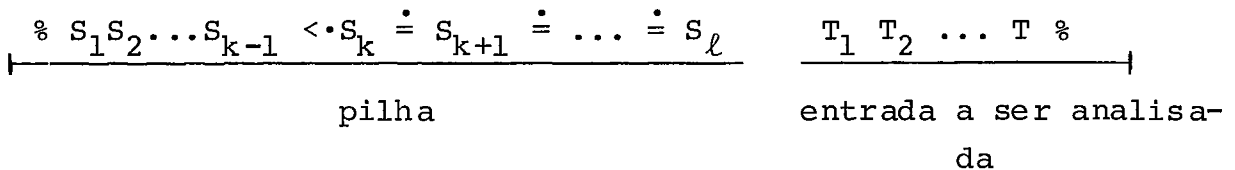
5. "Panic Mode" - subtração de símbolos da entrada e da pilha até achar uma configuração topo da pilha - símbolo de entrada que permita o prosseguimento da análise . O método é simples de implementar garantindo uma rapidez de análise. Entretanto podem ser introduzidos erros devido a eliminação de algumas informações e da presença de algumas ações semânticas que já haviam sido tomadas e que não foram desfeitas. Esta técnica foi utilizada no compilador XPL desenvolvido por McKeeman e Horning [15]. O nome ironico foi dado ao método por Gries [14].

A utilização do método de matriz de transição permite que o erro seja detetado o mais cedo possível evitando reduções e ações semânticas desnecessárias. O acesso à matriz através da linha que corresponde ao estrelado no to

po do stack e da coluna referente ao terminal recebido acusa ou uma relação correta ($<\cdot$, $\doteq$, $\cdot>$) ou uma relação vazia ($\cdot$). Este último caso representando um erro.

Segundo Gries [9] mais da metade dos elementos da matriz representam pares ilegais e assim podem ser detetadas várias condições de erros diferentes.

Supondo um passo típico do analisador no método de matriz temos:



Podemos distinguir três tipos de erros:

1. $S_\ell \quad \cdot \quad T$

2. $S_\ell \cdot> T_1$ mas não existe nenhuma regra do tipo $U::= S_k \ldots S_\ell$ tal que S_{k-1} se relacione com U e com T_1. Isto quer dizer não existe a tripla $U^*\, UT_1$.

3. $S_\ell \doteq T_1$ mas $S_k \ldots S_\ell T$ não é prefixo do lado direito. Isto é não existe $U_1^* ::= U^* T_1$, onde U^* é $S_k \ldots S_\ell$.

Para os três tipos de erros o compilador usa a técnica de "panic mode" procurando na entrada o símbolo END ou ; , fazendo a variável U = ' ' e descendo a pilha até achar

um estrelado U* tal que exista uma relação entre U* e T_1.

c. Erros na Fase de Análise Semântica

O tratamento de erros na fase semântica ainda é feito de um modo ad hoc. A recuperação é feita basicamente para identificadores não declarados que neste caso são incluídos na tabela de símbolos pelo analisador, com as informações mais adequadas ao contexto onde se encontra, porém com uma informação extra - de que foi colocado sob condição de erro (Aho [16]).

O tratamento proposto neste compilador baseia-se na estratégia de "panic mode" sendo que existem duas rotinas distintas de tratamento de erro.

A ERROM é chamada quando ocorre erro semântico na redução maior (•>). Sintáticamente a frase está correta mas existem informações incompatíveis. A ação tomada é abandonar a redução efetuada (U ← ' ') e prosseguir a análise com o símbolo do topo da pilha e o símbolo lido.

Quando o erro ocorre no início ou no meio de uma produção, isto é, numa relação <• ou $\doteq$ devem ser eliminados terminais da entrada e elementos da pilha até que seja encontrada uma tripla U*UT que permita o prosseguimento da análise. A rotina ERRO recebe informações sobre quais terminais podem ser aceitos e paralelamente quais estrelados devem ser acessados na pilha para se ter uma tripla correta. Esta informação permite procurar símbolos além de END e ; tais como ' ,),].

CAPÍTULO V

AVALIAÇÃO DA LINGUAGEM

Serão apresentados alguns algoritmos que se encontram nos livros de Knuth [3~5] e os programas escritos em MIXAL [3] (Seção 1.3.2.) para resolvê-los.

Para cada algoritmo proposto serão escritos dois programas em PLMIX - um deles seguirá exatamente o algoritmo e o outro resolverá o problema segundo as técnicas de programação estruturada. Após cada programa será apresentado o código por ele gerado e serão feitos alguns comentários, porém não será analisado o tempo de execução para estes programas.

V.1. EXEMPLO NÚMERO 1

Algoritmo para atravessar uma árvore binária em ordem simétrica. O algoritmo está na página 317 e o programa na página 323 da referência [3].

Algoritmo T.

T é o ponteiro para a árvore binária segundo a figura V.1. O algoritmo visita todos os nós da árvore em ordem simétrica, fazendo uso de uma pilha auxiliar A.

T1. [inicializar]. Tornar à pilha A vazia.

Inicializar a variável de ligação

P ← T

T2. [P = λ?] Se P = λ vá para T4.

T3. [pilha ← P] (agora P aponta para uma árvore binária não vazia a ser atravessada).

A <= P

P <= LLINK(P)

vá para T2

T4. [P <= pilha] se A estiver vazia o algoritmo termina; caso contrário faça P <= A.

T6. [visite P] "visitar" NO[P].

P ← RUNK(P). vá para T2.

P ← LLINK(P)

vá para T2

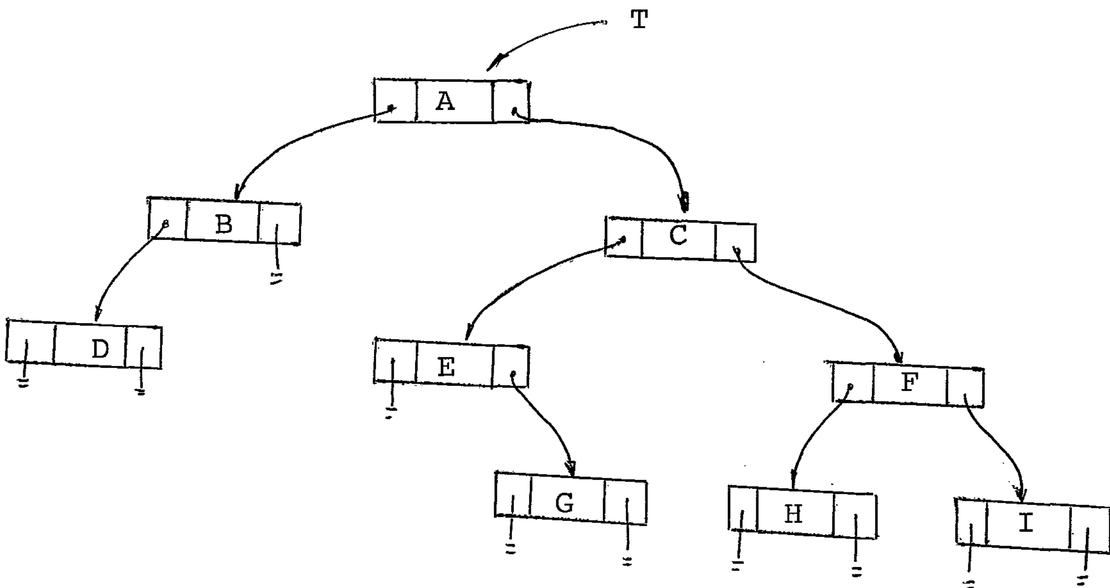


Figura V.1.

V.1.1. PROGRAMA T

A pilha fica nas posições A + 1, A + 2, ... A + MAX; Pode ocorrer "OVERFLOW" se a pilha cresce demais. rI6 é o ponteiro para a pilha. rI5 ≡ P. O programa está ligeiramente diferente do algoritmo T, e deste modo não é feito o teste para pilha vazia quando vai de T3 para T2 para T4.

Estrutura do nó de informação

LTAG	LLINK	INFO1
RTAG	RLINK	INFO2

```
          LLINK   EQU     1:2
          RLINK   EQU     1:2
1         T1      LD5     HEAD(LLINK)     T1 inicializar P ← T
2         T2B     J5Z     DONE               para se P = λ
3                 ENT6    0
4         T3      DEC6    MAX             T3 pilha <=   P
5                 J6NN    OVERFLOW           pilha cheia ?
6                 INC6    MAX+1              Se não, incrementar
                                             ponteiro
7                 ST5     A,6                guarde P na pilha
8                 LD5     0,5(LINK)       P ← LLINK (P)
9         T2A     J5NZ    T3                 para T3 se P ≠ λ
10        T4      LD5     A,6             T4  P <==  pilha
11                DEC6    1                  decresce ponteiro pilha
```

```
12      T5      JMP     VISIT     T5 visite P
13              LD5     1,5       P ← RLINK(P)
14      T2      J5NZ    T3        T2 P = λ?
15              J6NZ    T4        teste se pilha vazia
       DONE     ----
```

V.1.2. PROGRAMA EM PLMIX, SEGUINDO O ALGORITMO T

```
begin
    constant MAX = ? , TAMARV = ? ; % indefinido no
    array TAMARV record ARVORE:        programa MIX
          .1 WORD ESQ
             .2 BYTE (0:0) LTAG
             .2 BYTE (1:2) LLINK
             .2 BYTE (3:5) INFO1
          .1 WORD DIR
             .2 BYTE (0:0) RTAG
             .2 BYTE (1:2) RLINK
             .2 BYTE (3:5) INFO2;
word MAXIMO = MAX;
record HEAD: structure ARVORE;
array MAX word A;  % pilha
equate PTR = RI6, P = RI5;
label DONE, T3, T4;

HEAD.LLINK: = 0;    % não será mostrado o código gerado
                    % pois Knuth assume esta informação
                    % já pronta no seu programa.
```

```
      % programa para algoritmo T
      P: = HEAD.LLINK;                 % P ← T
      if P zero then goto DONE;        % pare se P = λ
      PTR: = 0;
  T3: PTR: = PTR - MAX;                % verificar transbordo
      if PTR pos then goto OVERFLOW
      A[PTR]: = P;                     % pilha <=   P
      P: = ARVORE.LLINK[P];            % P ← LLINK(P)
      if P nzero then goto T3;         % para T3 se P ≠ λ
  T4: P: = A[PTR];                     % P ← pilha
      PTR: = PTR - 1;
      VISITA;                          % visite P
      P: = ARVORE.RLINK[P];            % P ← RLINK(P)
      if P nzero then goto T3;         % se P ≠ λ vá para T3
      if PTR nzero then goto T4;       %
  DONE:
end
```

Tradução do programa acima em MIXAL

```
1         ST5     HEAD (1:2)
2         J5NZ    * + 2
3         JMP     DONE
4         ENT6    0
5    T3   DEC6    MAX
6         J6NP    * + 2
7         JMP     OVERFLOW
8         INC6    MAX + 1
9         ST5     A,6
```

```
10        LD5     0,5 (LLINK)
11        J5Z     * + 2
12        JMP     T3
13   T4   LD5     A,6
14        DEC6    1
15        JMP     VISITA
16        LD5     1,5 (RLINK)
17        J5Z     * + 2
18        JMP     T3
19        J5Z     * + 2
20        JMP     T4
     DONE ...
```

<u>Observações</u>:

São omitidas algumas informações no programa MIXAL como por exemplo as declarações dos procedimentos "OVERFLOW" e "VISITA" e os valores iniciais das constantes MAX e TAMARV. Estas informações não são dadas no programa do livro pois são consideradas presentes em um programa de inicialização geral. Entretanto, isto não influencia na avaliação.

A tradução do programa em PLMIX produziu 5 instruções a mais. Elas correspondem às saídas dos testes condicionais da estrutura do comando "if then" a seguir

<condição>

J ¬ <cond>

<intr then> ⊟ <desvio incondicional> (goto)

Na linguagem montada o teste é feito evitando-se estes dois desvios seguidos. O compilador entretanto não faz nenhum teste para verificar esta sequência desnecessária de dois desvios.

V.1.3. PROGRAMA EM PLMIX ESTRUTURANDO-SE O ALGORITMO PROPOSTO

As declarações são as mesmas para o programa em V.1.2., com excessão da declaração LABEL que aqui torna-se desnecessária.

```
P: = HEAD = LLINK;
PTR: = 0;
repeat
    begin
       while P nzero do
           begin
               PTR: = PTR + 1;
               if PTR GT MAXIMO then goto OVERFLOW;
               A[PTR]: = P;
               P: = ARVORE.LLINK[P]
           end
    P: = A[PTR];
    PTR: = PTR - 1;
    VISITA;
    P: = ARVORE.RLINK[P]
    end
until PTR zero;
```

Tradução do programa anterior para MIXAL

```
1         LD5    HEAD (1:2)
2         ENT6   0
3    E1   J5Z    E2
4         INC6   1
5         CMP6   MAXIMO
6         J6LE   * + 2
7         JMP    OVERFLOW
8         ST5    A,6
9         LD5    0,6  (LLINK)
10        JMP    E1
11   E2   LD5    A,6
12        DEC6   1
13        JMP    VISITA
14        LD5    1,5  (RLINK)
15        J6NZ   E1
```

Observações

Apesar da alteração da estrutura ela permitiu a obtenção de um código traduzido com o mesmo número de instruções do programa escrito em linguagem montada e com algumas transformações para aumentar a eficiência.

V.2. EXEMPLO NÚMERO 2

Este exemplo se encontra no livro "Sorting and Searching" [5] nas páginas 139 e 140.

O algoritmo proposto é para efetuar ordenação de chaves pelo método de seleção.

Algoritmo S (ordenação por seleção direta).

Os registros $R_1, \ldots, R_n$ são rearranjados e após completada a ordenação as chaves estarão em ordem ($k_1 \leq \ldots \leq k_n$). O método seleciona em primeiro lugar a maior chave.

S1. [laço com j] execute os passos S2 e S3 para $j = N, N - 1, \ldots, 2$.

S2. [ache o máximo de $(k_1, \ldots, k_n)$] busca entre as chaves $k_j, k_{j-1}, \ldots, k_1$ para achar a maior chave. Vamos chamá-la de k_i

S3. [troca com R_j] troque os registros $R_i \leftrightarrow R_j$ (agora os registros $R_j, \ldots, R_n$ estão em suas posições finais).

V.2.1. PROGRAMA S

Os registros estão nas posições de INPUT + 1 a INPUT + N e são ordenados nas mesmas posições de memória que ocupam. A chave ocupa uma palavra. rA ≡ máximo atual, rI1 ≡ J - 1, rI2 ≡ K (índice para busca), rI3 ≡ i. Assume-se

que N ≥ 2.

```
01   START   ENT1   N - 1        S1.laço com j, j ← N
02   2H      ENT2   0,1          S2.ache o máximo k←j-1
03           ENT3   1,1          i ← j
04           LDA    INPUT,3      rA ← ki
05   8H      CMPA   INPUT,2
06           JGE    * + 3        pule se ki ≥ Kk
07           ENT3   0,2          senão i ← k
08           LDA    INPUT,3      rA ← ki
09           DEC2   1            k ← k - 1
10           J2P    8B           repete-se k > 0
11           LDX    INPUT+1,1    S3.troque com Rj
12           STX    INPUT,3      Ri ← Rj
13           STA    INPUT+1,1    Rj ← rA
14           DEC1   1
15           J1P    2B           N ≥ j ≥ 2
```

V.2.2. PROGRAMA ESCRITO EM MIX SEGUNDO O ALGORITMO S

```
begin
   constant N = ?; % indefinido no programa MIX
   array N word ENTRADA
   label INICIO,VOLTA,MAIOR;
   equate J = RI1, K = RI2, I = RI3, MAX = RAC;
```

```
          J: = N;
INICIO:   K: = J - 1;
          I: = J
          MAX: = ENTRADA[I]
VOLTA:    if MAX lt ENTRADA[K];
          then begin
                   I: = K;
                   MAX: = ENTRADA[I]
               end
MAIOR:    K: = K - 1;
          if K pos then goto VOLTA;
          ENTRADA[I], RX: = ENTRADA[I]
          ENTRADA[J]: = MAX;
          J: = J - 1;
          RI4: = J - 2;
          if RI4 nneg then goto INICIO;
end
```

Tradução do programa acima para MIXAL

```
01        ENT1      N
02   2H   ENT2      0,1
03        DEC2      1
04        ENT3      0,1
05        LDA       INPUT,3
06   8H   CMPA      INPUT,2
07        JGE       3F
08        ENT3      0,2
09        LDA       INPUT,3
10   3H   DEC2      1
```

```
11        J2NP    4F
12        JMP     8B
13   4H   LDX     INPUT,1
14        STX     INPUT,3
15        STA     INPUT,1
16        DEC1    1
17        ENT4    0,1
18        DEC4    2
19        J4NI    5F
20        JMP     2B
     5H
```

Observações

As instruções nas linhas 2 e 3 podem ser escritas em um único comando, ENT2 1,1. Entretanto o compilador gera os códigos separadamente porque não há otimização de expressões.

As instruções nas linhas 17 e 18 são instruções para salvar o registro RI1. A comparação com um valor numérico ou expressão constante destroi o valor o registro onde se dá o teste. Deste modo foi usado o registro I4 para manter inalterado o conteúdo de RI1.

As outras duas instruções extras são desvios de saída da construção "if <cond> then <instrução>".

V.2.3. PROGRAMA MIXAL USANDO TÉCNICAS DE PROGRAMAÇÃO ESTRUTURADA

```
   begin
        constant N = ?; % indefinido no programa MIX
        array N word ENTRADA;
        equate J = RI1, K = RI2, I = RI3, MAX = RAC;

        J: = N;
   repeat
      begin
         K: = J - 1
         I: = J;
         MAX: = ENTRADA[I];
         repeat
            begin
               if MAX LT ENTRADA[K]
               then begin
                    I: = K;
                    MAX; = ENTRADA[I]
                    end;
                    K: = K - 1
               end
         until K npos
         ENTRADA[K], RX: = ENTRADA[I];
         ENTRADA[J]: = MAX;
         J: = J - 1;
         RI4: = J - 2
      end
until RI4 neg
end
```

Tradução para MIXAL

```
01       ENT1    N
02  2H   ENT2    0,1
03       DEC2    1
04       ENT3    1
05       LDA     INPUT,3
06  8H   CMPA    INPUT,2
07       JGE     3F
08       ENT3    0,2
09       LDA     INPUT,3
10  3H   DEC2    1
11       J2P     8B
12       LDX     INPUT,1
13       STX     INPUT,3
14       STA     INPUT,1
15       DEC1    1
16       ENT4    1
17       DEC4    2
18       J4NN    2B
```

Observações:

Com este programa foram desnecessários dois testes do tipo "if" deixando-se assim de se usar duas instruções de desvio. Esta é a única diferença do exemplo anterior.

V.3. EXEMPLO NÚMERO 3

Algoritmo para efetuar ordenação de chaves pelo método de troca por seleção, conhecido por "bubble sort". O algoritmo se encontra na página 107 da referência [5].

Algoritmo B

Os registros $R_1, \ldots, R_n$ devem ser ordenados, de modo que suas chaves satisfaçam a condição $K_1 \leq \ldots \leq K_N$.

B1. [inicializar BOUND] Faça BOUND ← (BOUND aponta para o registro de maior ordem, ainda não ordenado; neste caso estamos indicando que toda a entrada está, possivelmente, desordenada).

B2. [laço com j] Faça t ← 0. Execute o passo B3 para j = 1,2,...,BOUND - 1, e depois vá para B4.

B3. [comparação/troca R_j: R_{j+1}] Se $K_j > K_{j+1}$ troque $R_j \leftrightarrow R_{j+1}$ e faça t ← j

B4. [alguma troca ?] Se t = 0, o algoritmo acaba. Caso contrário faça BOUND ← t volte para B2.

V.3.1. PROGRAMA B

Os registros serão ordenados nas posições INPUT + 1 a INPUT + N. rI1 ≡ t; rI2 = j.

```
1   START   ENT1   N            B1. inicializa BOUND.t←N
2    1H     ST1    BOUND(1:2)   BOUND ← t
3           ENT2   1            B2. laço com j
4           ENT1   0            t ← 0
5           JMP    BOUND        fim se j ≥ BOUND
6    3H     LDA    INPUT,2      B3.comparação e troca Rj:Rj+1
7           CMPA   INPUT+1,2
8           JLE    2F           não troca se Kj ≤ Kj + 1
9           LDX    INPUT+1,2    Rj + 1
10          STX    INPUT,2      → Rj
11          STA    INPUT+1,2    (Rj antigo) → Rj + 1
12          ENT1   0,2          t ← j
13   2H     INC2   1            j ← j + 1
14  BOUND   ENTX   -*,2         rX ← j-BOUND (instrução modi
                                ficada)
15          JXN    3B           1 ≤ j ≤ BOUND
16   4H     J1P    1B           B4. alguma troca ? vai para
                                B2 se t > 0
```

V.3.2. PROGRAMA EM PLMIX SEGUNDO O ALGORITMO B

```
begin
   constant N = ;
   word BOUND = N ;
   array N word INPUT;
   equate T = RI1, J = RI2
   label INICIO;
```

```
INICIO: T: = 0
        for J: = 1 step 1 until BOUND - 1  do
            if INPUT[J] > INPUT[J + 1, RI3]
            then begin  % troca
                RX: = INPUT[RI3];
                INPUT [RI3]: = RAC; % RAC=INPUT[J]
                INPUT[J]: = RX;
                T: = J
                end
        if T nzero
        then begin
            BOUND: = T
            goto INICIO
            end
end
```

Tradução para MIXAL

```
1    1H    ENT1    0
2          ENT2    1
3          LDA     BOUND
4          DECA    1
5          STA     * + 2
6          JMP     * + 2
7   LIM
8    4H    LDA     INPUT,2
9          ENT3    0,2
10         INC3    1
```

```
11        CMPA    INPUT,3
12        JLE     2F
13        LDX     INPUT,3
14        STA     INPUT,3
15        STX     INPUT,2
16        ENT1    0,2
17   2H   INC2    1
18        CMP2    LIM
19        JLE     4B
20        J2Z     5F
21        ST1     BOUND
22        JMP     1B
     5H
```

Observações:

A instrução "<u>for</u> - <u>step</u> - <u>until</u>" introduziu instruções que não estão presentes na versão do autor, além de ocupar uma posição de memória a mais (linha nº 7) para armazenar o valor do limite superior da iteração. Além disso foi gerada uma instrução a mais para o cálculo de <registro> já que a geração de código para expressão é da esquerda para a direita. Deste modo geramos

```
ENT3    0,2
INC3    1
CMPA    INPUT,3
```

quando poderia ser escrito simplesmente

```
CMPA    INPUT + 1, 2
```

V.3.3. PROGRAMA EM PLMIX, ESTRUTURANDO-SE O ALGORITMO PROPOSTO

```
begin
    constant N = ;
    word BOUND ;
    array N word INPUT;
    equate T = RI1, J = RI2;
    T: = N;
    repeat
         begin
             BOUND: = T;
             T: = 0;
             J: = 1;
    while J < BOUND do
         if INPUT[J] INPUT[J + 1, RI3]
         then begin
              RX: = INPUT [RI3]
              INPUT[RI3]: = RAC; % RAC = INPUT [J]
              INPUT[J]: RX;
              T: = J;
              J: = J + 1
              end
         end
    end
  until T zero
end
```

Tradução para MIXAL

```
1         ENT1      N
2         ST1       BOUND
3         ENT1      0
4         ENT2      1
5   4H    CMP2      BOUND
6         JGE       2F
7         LDA       INPUT,2
8         ENT3      0,2
9         INC3      1
10        CMPA      INPUT,3
11        JLE       3F
12        LDX       INPUT,3
13        STA       INPUT,3
14        STX       INPUT,2
15        ENT1      0,2
16        INC2      1
17  3H    JMP       4B
18  2H    J1NZ      5B
```

Observações:

As instruções a mais (linhas 9 e 10) são para o cálculo do índice [J + 1, RI3], que no programa original é feito em uma instrução.

V.4. EXEMPLO NÚMERO 4

Algoritmo para efetuar a soma de dois polinomios. O algoritmo está na página 273 da referência |3|.

Algoritmo A

Este algoritmo soma o polinomio P ao polinomio Q, assumindo que P e Q são ponteiros para os polinomios da forma da figura V.4.1. A lista P ficará inalterada e na lista Q ficará a soma. Os ponteiros P e Q retornam ao ponto inicial ao término do algoritmo; são usados ponteiros auxiliares Q1 e Q2.

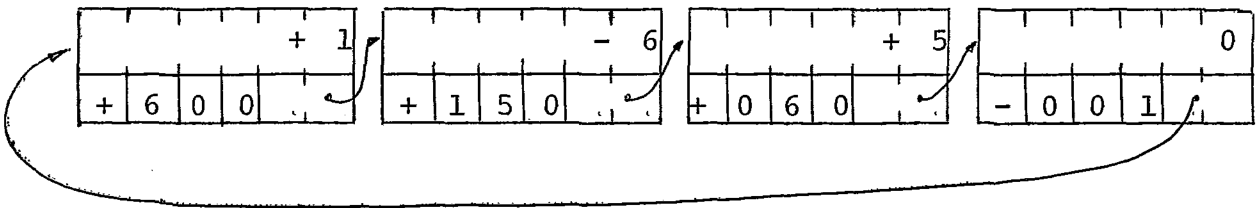


Figura V.4.1.

Cada termo do polinomio ocupa duas palavras- uma para o coeficiente e outra para os expoentes do termo xyz e a ligação para o outro termo, conforme a figura V.4.2.

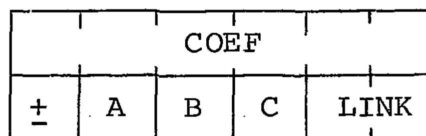


Figura V.4.2.

A1. |inicializações| Faça P ← LINK(P), Q1 ← Q, Q ← LINK(Q)

A2. |ABC(P): ABC(Q)| Se ABC(P) < ABC(Q), faça Q1 ← Q2 e Q ← LINK(Q) e repita este passo. Se ABC(P) = ABC(Q) vá para A3. Se ABC(P) > ABC(Q) vá para A5.

A3. |soma coeficientes| (Encontramos termos com expoentes iguais). Se ABC(P) < 0 o algoritmo termina. Caso contrário faça COEF(Q) ← COEF(Q) + COEF(P). Se COEF(Q) = 0 vá para A4. Senão, faça Q1 ← Q, P ← LINK(P), Q ← LINK(Q) e vá para A2.

A4. |apagar termo igual a zero| Faça Q2 ← Q, LINK(Q1) ← Q ← LINK(Q). AVAIL<= Q2. (foi removido o termo igual a zero, criado em A3). Faça P ← LINK(P) e vá para A2.

A5. |insere novo termo| (O polinomio P tem um termo que não está presente em Q, então ele será inserido em Q). Faça Q2 <= AVAIL, COEF(Q2) ← COEF(P), ABC(Q2) ← ABC(P), LINK(Q2) ← Q, LINK(Q1) ← Q2, Q1 ← Q2 , P ← LINK(P) e retorne para o passo A2.

V.4.1. PROGRAMA A

No código a seguir temos: P ≡ rI1, Q ≡ rI2 , Q1 ≡ rI3 e Q2 ≡ rI6.

```
        LINK    EQU    4:5         definição do campo LINK
        ABC     EQU    0:3         definição do campo ABC
        ADD     STJ    3F          entrada da subrotina
1       6H      ENT3   0,2         A1. Inicialização Q1 ← Q
2       ØH      LD1    1,1(LINK)   P ← LINK(P)
3               LDA    1,1(ABC)    rA(0:3) ← ABC(P)
4       1H      LD2    1,3(LINK)   Q ← LINK(Q1)
5       2H      CMPA   1,2(ABC)    A2. ABC(P):ABC(Q)
6               JE     3F          Se igual, vá para A3.
7               JG     5F          Se maior, vá para A5.
8               ENT3   0,2         Se menor, faça Q1 ← Q
9               JMP    1B          repita
10      3H      JAN    *           A3. Soma coeficientes
                LDA    0,1         COEF(P)
12              ADD    0,2              + COEF(Q)
13              STA    0,2                    → COEF(Q)
14              JANZ   6B          o resultado é zero?
15              ENT6   0,2         A4. Apaga termo zero. Q2←Q
16              LD2    1,2(LINK)   Q ← LINK(Q)
17              LDX    AVAIL      ⎫
18              STX    1,6(LINK)  ⎬ AVAIL ⇐ Q2
19              ST6    AVAIL      ⎭
20              ST2    1,3(LINK)   LINK(Q1) ← Q
21              JMP    ØB          avança P
22      5H      LD6    AVAIL      ⎫
23              J6Z    OVERFLOW   ⎪ A5.Insere novo termo
24              LDX    1,6(LINK)  ⎬ Q2 ⇐ AVAIL
25              STX    AVAIL      ⎭
```

```
26    STA    1,6(ABC)     ABC(Q2) ← ABC(P)
27    LDA    0,1          rA ← COEF(P)
28    STA    0,6          COEF(Q2) ← rA
29    ST2    1,6(LINK)    LINK(Q2) ← Q
30    ST6    1,3(LINK)    LINK(Q1) ← Q2
31    ENT3   0,6          Q1 ← Q2
32    JMP    ØB           avança P
```

V.4.2. PROGRAMA EM PLMIX, SEGUINDO O ALGORITMO A

```
begin
    constant N = ? ; % indefinido no programa MIX
    array N record POLIP:
        .1 word COEF
        .1 word EXPO
            .2 byte (0:3) ABC
            .2 byte (4:5) LINK ;
    array N record POLIQ: structure POLIP;
    equate P = RI1, Q = RI2, Q1 = RI3, Q2 = RI6;
    word AVAIL; % ponteiro para disponível
    label L6, LØ, L1, OVERFLOW;
    L6 : Q1 : = Q;
    LØ : P : = POLIP.LINK[P];
    L1 : Q : = POLIQ.LINK[Q1];
    if POLIP.ABC[P] < POLIQ.ABC[Q]
```

```
then begin
        Q1 : = Q;
        goto L1
        end
else if RAC = POLIQ.ABC[Q] % RAC = POLIP.ABC[P]
     then begin  % somar coeficientes
          POLIQ.COEF[Q], RAC: = POLIP.COEF[P] + POLIQ.COEF[Q],
          if RAC nzero
          then goto L6
          else begin % apagar termo igual a zero
               Q2: = Q;
               Q: = POLIQ.LINK[Q];
               POLIQ.LINK[Q2], RX: = AVAIL;
               AVAIL: = Q2;
               POLIQ.LINK[Q1]: = Q;
               goto L6
               end
          end
     else begin % insere novo termo
          Q2: = AVAIL;
          if Q2 = 0 then goto OVERFLOW;
          AVAIL, RX: = POLIQ.LINK[Q2];
          POLIQ.ABC[Q2]: = RAC; % RAC ainda contém
                                    POLIP.ABC[P]
          POLIQ.COEF[Q2], RAC: = POLIQ.COEF[P];
          POLIQ.LINK[Q2]: = Q;
          POLIQ.LINK[Q1]: = Q2;
          Q1: = Q2;
          goto LØ
          end
```

Antes da tradução será mostrado como está alocada a memória para os polinomios POLIP e POLIQ.

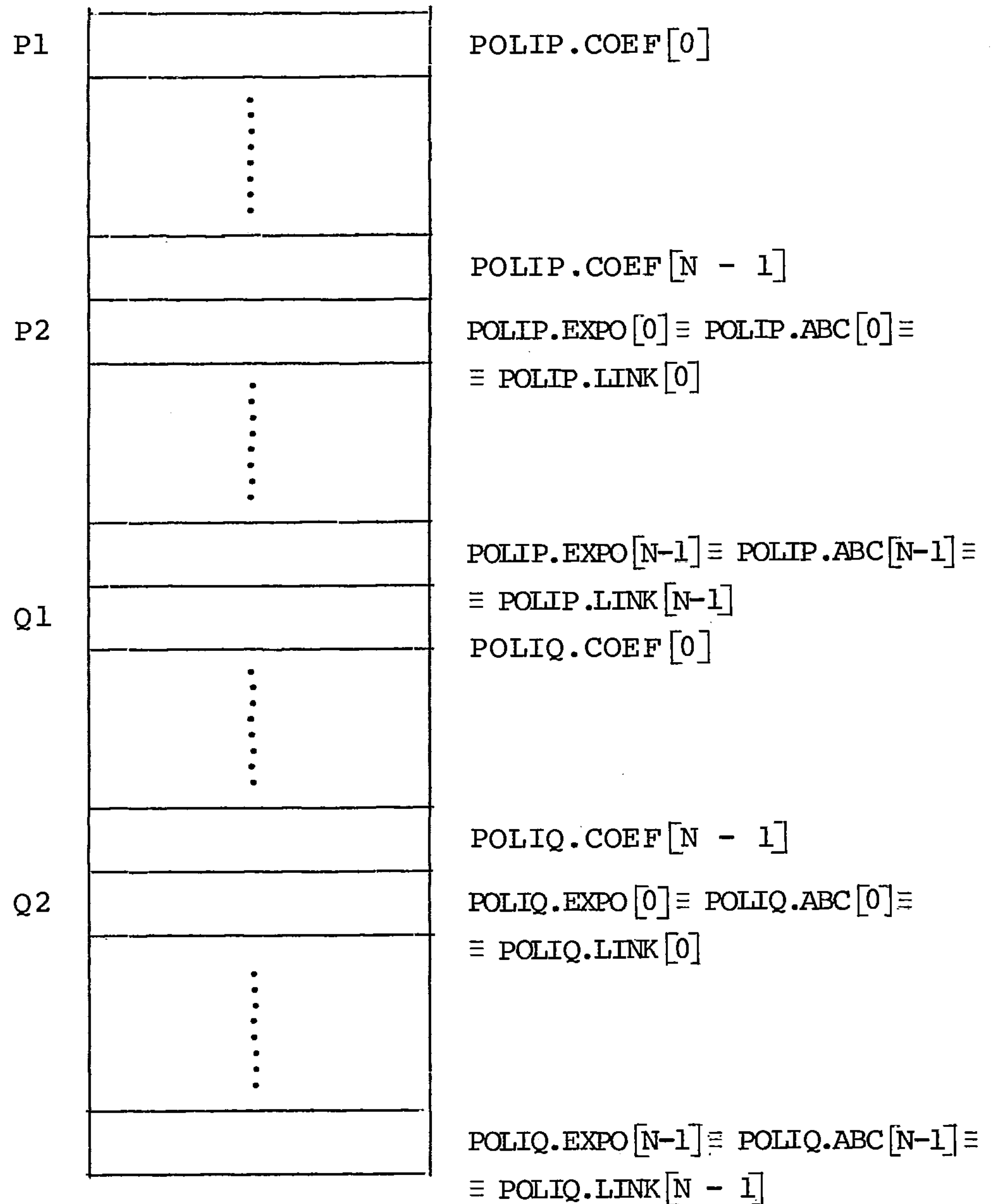


Figura V.4.3.

Os endereços de POLIP.COEF[0], POLIP.EXPO[0], POLIQ.COEF[0] e POLIQ.EXPO[0] serão referenciados por P1, P2, Q1 e Q2 respectivamente, conforme exemplificado na figura V.4.3.

```
1    6H    ENT3    0,2
2    ØH    LD1     P2,1(LINK)
3    1H    LD2     Q2,3(LINK)
4          LDA     P2,1(ABC)
5          CMPA    Q2,2(ABC)
6          JGE     7F
7          ENT3    0,2
8          JMP     1B
9    7H    CMPA    Q2,2(ABC)
10         JNE     5F
11         LDA     P1,1
12         ADD     Q1,2
13         STA     Q1,2
14         JAZ     * + 2
15         JMP     6B
16         ENT6    0,2
17         LD2     Q2,2(LINK)
18         LDX     AVAIL
19         STX     Q2,6(LINK)
20         ST6     AVAIL
21         ST2     Q2,3(LINK)
22         JMP     ØB
23   5H    LD6     AVAIL
24         J6NZ    * + 2
```

```
25        JMP     OVERFLOW
26        LDX     Q2,6(LINK)
27        STX     AVAIL
28        STA     Q2,6(ABC)
29        LDA     P1,1
30        STA     Q1,6
31        ST2     Q2,6(LINK)
32        ST6     Q2,3(LINK)
33        ENT3    0,6
34        JMP     ØB
```

Observações:

Levando-se em consideração o conteúdo do acumulador (rAC) nas diversas comparações, evitou-se carregamento desnecessário do registro, podendo assim termos um texto em PLMIX que ficou com o código gerado praticamente igual ao escrito pelo autor. Foram utilizadas estratégias de movimentação dos ponteiros em pontos do programa que também permitiram essa economia de código.

V.4.3. PROGRAMA EM PLMIX, ESTRUTURANDO-SE O ALGORITMO PROPOSTO

São válidas as declarações feitas para o programa em V.4.2., com as seguintes alterações: - retirar a declaração de label e incluir a declaração:

```
equate NAOACABOU = RI4; % será usada como variável ló
                          gica.
NAOACABOU: = 1;
while NAOACABOU nzero do
   begin
       while POLIP.ABC[P] < POLIQ.ABC[Q] do
           begin
               Q1: = Q;
               Q: = POLIQ.LINK[Q]
           end
       if RAC = POLIQ.ABC[Q]
       then if RAC neg
            then NAOACABOU: = 0
            else begin
                     POLIQ.COEF[Q], RAC: = POLIP.COEF[P]
                                         + POLIQ.COEF[Q]
                     if RAC zero % se coef = ∅
                     then begin % elimina nó
                              Q2: = Q;
                              Q: = POLIQ.LINK[Q];
                              POLIQ.LINK[Q2], RX: = AVAIL;
                              AVAIL: = Q2
                              POLIQ.LINK[Q1]: = Q
                              end
                     else begin % coef  = ∅
                              Q1: = Q;
                              Q: = POLIQ.LINK[Q]
                              end
```

```
                                  P: = POLIP.LINK[P]
                              end
                  else begin % insere novo termo
                       Q2: = AVAIL
                       if Q2 zero then goto OVERFLOW;
                       AVAIL, RX: = POLIQ.LINK[Q2];
                       POLIQ.ABC[Q]: = RAC;
                       POLIQ.COEF[Q], RAC: = POLIP.COEF[P];
                       POLIQ.LINK[Q2]: = Q;
                       POLIQ.LINK[Q1]: = Q2;
                       Q1: = Q2;
                       P: = POLIP.LINK[P]
                       end
         end          % final do while
```

Os endereços dos termos dos polinomios referenciados na tradução para MIX, são os da figura V.4.3.

```
1         ENT4     0,1
2         J4Z      1F
3    2H   LDA      P2,1(ABC)
4         CMPA     Q2,2(ABC)
5         JGE      3F
6         ENT3     0,2
7         LD2      P2,2(LINK)
8         JMP      2B
9    3H   CMPA     P2,2(ABC)
```

```
10          JNE     7F
11          JANN    * + 3
12          ENT4    0
13          JMP     9F
14          LDA     P1,1
15          ADD     Q1,2
16          STA     Q1,2
17          JANZ    4F
18          ENT6    0,2
19          LD2     Q2,2(LINK)
20          LDX     AVAIL
21          STX     Q2,6(LINK)
22          ST6     AVAIL
23          ST2     Q2,3(LINK)
24          JMP     8F
25    4H    ENT3    0,2
26          LD2     Q2,2(LINK)
27    8H    LD1     P2,1(LINK)
28    9H    JMP     6F
29    7H    LD6     AVAIL
30          J6NZ    * + 2
31          JMP     OVERFLOW
32          LDX     Q2,6(LINK)
33          STX     AVAIL
34          STA     Q2,2(LINK)
35          LDA     P1,1
36          STA     Q1,2
37          ST6     Q2,2(LINK)
38          ENT3    0,6
```

```
39          LD1        P2,1(LINK)
40   6H     JMP        2B
```

Observações:

O grande número de instruções a mais é devido a dois fatores básicos: - utilização da instrução "if-then-else" e da necessidade de repetir instruções para permitir estruturar o programa.

CAPÍTULO VI

CONCLUSÕES

A importância deste projeto é devido ao fato das linguagens de médio nível, com as características apresentadas na introdução deste trabalho, tenderem a substituir gradualmente a linguagem montada (assembler). Sua principal vantagem é associar a potencialidade da linguagem montada com as estruturas típicas necessárias para se escrever programas estruturados e a sintaxe semelhante à das linguagens de alto nível mais usuais. Sua maior utilização é em projetos de "software" básico e de suporte.

Voltado para o objetivo de estabelecer critérios que orientem no projeto de novas linguagens, a apresentação do trabalho através da definição de uma linguagem específica , além de conferir um sentido prático com grandes possibilidades de utilização a curto prazo, assume a generalidade de um estudo teórico sobre projetos de linguagens.

A análise da sintaxe e semântica da linguagem através de justificativas das soluções adotadas e explicações de ordem geral, permite que a apresentação adquira características didáticas. Entretanto, este trabalho não se propõe a ser completo, já que uma obra didática é um estudo mais amplo e complexo.

Uma boa avaliação da linguagem está no fato de podermos expressar com facilidade os algoritmos propostos por Knuth na série "The Art of Computer Programming", sem perder em eficiência e sem onerar o tempo de processamento dos programas traduzidos do PLMIX, em relação aos programs escritos em MIXAL pelo autor. O PLMIX permite ainda o acompanhamento pormenorizado dos programas através de rastreamento, impressão dos contadores e verificação do tempo de execução, com a introdução das instruções criadas por Markenzon [6].

Com a utilização em maior escala da linguagem, surgirão certamente propostas para expansões, que poderão ser facilmente introduzidas devido ao método de análise sintática adotado e principalmente por não considerarmos o trabalho totalmente completo, inflexível e hermético.

A escolha do método de matriz de transição para efetuar a análise sintática permitiu-nos avaliar um método que havia sido abandonado com a introdução de novas classes de gramáticas. Após o estudo e avaliação podemos sugerir sua adoção em compiladores de linguagens que se propõem a atingir uma faixa de usuários ainda inexperientes, por permitir uma especificação clara das mensagens de erros e uma recuperação dos erros de uma forma simples e rápida. Além disso o método é de fácil implementação, principalmente com a utilização da geração automática da matriz de transição através do NHÃONHÃO (Simone [10]).

A implementação do compilador será no computador MITRA 15 do Laboratório de Automação de Sistemas e Simulação, visto que nele está implementado o simulador do MIX

(Markenzon |6|). O compilador será escrito em LP15. Como as facilidades de depuração no MITRA 15 são restritas, optamos por escrever o compilador em duas etapas. Na primeira o compilador será implementado no computador Burroughs B6700, no Núcleo de Computação Eletrônica, escrito em ALGOL. O programa terá uma sintaxe a mais próxima possível do LP15, evitando recursos do ALGOL de díficil tradução para o LP15. Deste modo, após completada a primeira etapa, a implementação final será um trabalho simples, rápido, com apenas algumas adaptações.

APENDICE Nº 1

00	1	01	2	02	2	03	10
Não Opera NOP(0)		rA ← rA + V ADD(0:5) FADD(6)		rA ← rA − V SUB(0:5) FSUB(6)		rAX ← rA x V MUL(0:5) FMUL(6)	
04	12	05	1	06	2	07	1 + 2F
rA ← rAX/V rX ← resto DIV(0:5) FDIV(6)		Especial NUM (0) CHAR(1) HLT (2)		desvios bytes SLA (0) SRA (1) SLAX(2) SRAX(3) SLC (4) SLC (5)		MOVE F palavras de M para rI1 MOVE (1)	
08	2	09	2	10	2	11	2
rA ← V LDA(0:5)		rI1 ← V LD1(0:5)		rI2 ← V LD2(0:5)		rI3 ← V LD3(0:5)	
12	2	13	2	14	2	15	2
rI4 ← V LD4(0:5)		rI5 ← V LD5(0:5)		rI6 ← V LD6(0:5)		RX ← V LDX(0:5)	
16	2	17	2	18	2	19	2
rA ← V LDAN(0:5)		rI1 ← V LDIN(0:5)		rI2 ← −V LD2N(0:5)		rI3 ← − V LD3N(0:5)	
20	2	21	2	22	2	23	2
rI4 ← −V LD4N(0:5)		rI5 ← −V LD5N(0:5)		rI6 ← −V LD6N(0:5)		rX ← −V LDXN(0:5)	
24	2	25	2	26	2	27	2
F(M) ← rA STA(0:5)		F(M) ← rI1 ST1(0:5)		F(M) ← rI2 ST2(0:5)		F(M) ← rI3 ST3(0:5)	
28	2	29	2	30	2	31	2
F(M) ← rI4 ST4(0:5)		F(M) ← rI5 ST5(0:5)		F(M) ← rI6 ST6(0:5)		F(M) ← rX STX(0:5)	
32	2	33	2	34	1	35	1 + T
F(M) ← rJ STJ(0:2)		F(M) ← 0 STZ(0:5)		Unidade F ocupada JBUS(0)		Controle,Unidade F IOC(0)	

continuação

36	1+T	37	1+T	38	1	39	1
Entrada,unidade F IN(0)		Saída,unidade F OUT(0)		Unidade F pronta JRED(0)		Desvios JMP(0) JSJ (1) JOV(2) JNOV(3) também | * | abaixo	
40	1	41	1	42	1	43	1
rA:,desvio JA|+|		rI1:0,desvio J1|+|		rI2:0,desvio J2|+|		rI3:0,desvio J3|+|	
44	1	45	1	46	1	47	1
rI4:0,desvio J4|+|		rI5:0,desvio J5|+|		rI6:0,desvio J6|+|		rX:0,desvio JX|+|	
48	1	49	1	50	1	51	1
rA←|rA|? ± M INCA(0)DECA(1) ENTA(2)ENNA(3)		rI1←|rI1|?±M INC1(0)DEC1(1) ENT1(2)ENN1(3)		rI2←|rI2|?± M INC2(0)DEC2(1) ENT2(2)ENN2(3)		rI3←|rI3|?±M INC3(0)DEC3(1) ENT3(2)ENN3(3)	
52	1	53	1	54	1	55	1
rI4←|rI4|?±M INC4(0)DEC4(1) ENT4(2)ENN4(3)		rI5←|rI5|?±M INC5(0)DEC5(1) ENT5(2)ENN5(3)		rI6←|rI6|?±M INC6(0)DEC6(1) ENT6(2)DEC6(3)		rX ← |rX|?±M INCX(0)DECX(1) ENTX(2)DECX(3)	
56	2	57	2	58	2	59	2
rA(F):V → CI CMPA(0:5) FCMP(6)		rI1(F):V → CI CMP1(0:5)		rI2(F):V → CI CMP2(0:5)		rI5(F):V → CI CMP3(0:5)	
60	2	61	2	63	2	63	2
rI4(F):V → CI CMP4(0:5)		rI5(F):V → CI CMP5(0:5)		rI6(F):V → CI CMP6(0:5)		rX(F):V → CI CMPX(0:5)	

rA = registro A

rX = registro X

rAX= registro AX

rIi= index reg. i,1 $\leq$ i $\leq$ 6

rJ = registro J

CI = ind. comparação

|*|: |+|

JL(4) < N(0)

JE(5) = Z(1)

JG(6) > P(2)

JGE(7) $\geq$ NN(3)

JNE(8) $\neq$ NZ(4)

JLE(9) $\leq$ NP(5)

APENDICE Nº 2

ÁRVORE SINTÁTICA

Na descrição de cada categoria sintática serão usadas as seguintes referencias:

A	< categoria sintatica>	(B)
	<descrição da sintaxe>	C,C
		C

A - número da definição da categoria

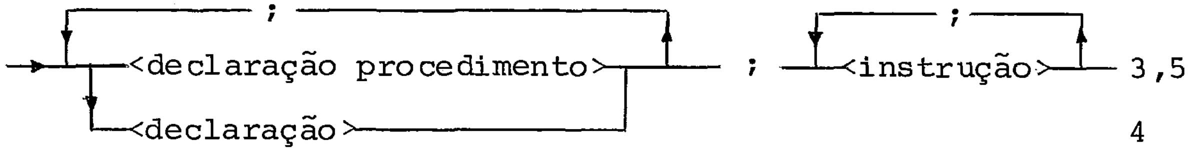
B - página onde se encontra a definição

C - números das definições dos não terminais envolvidos nesta definição.

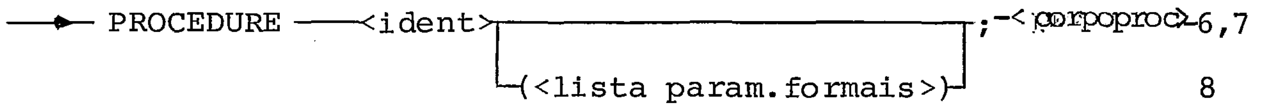
1 <programa>: : = (26)

——►—BEGIN ———<lista>————END 2

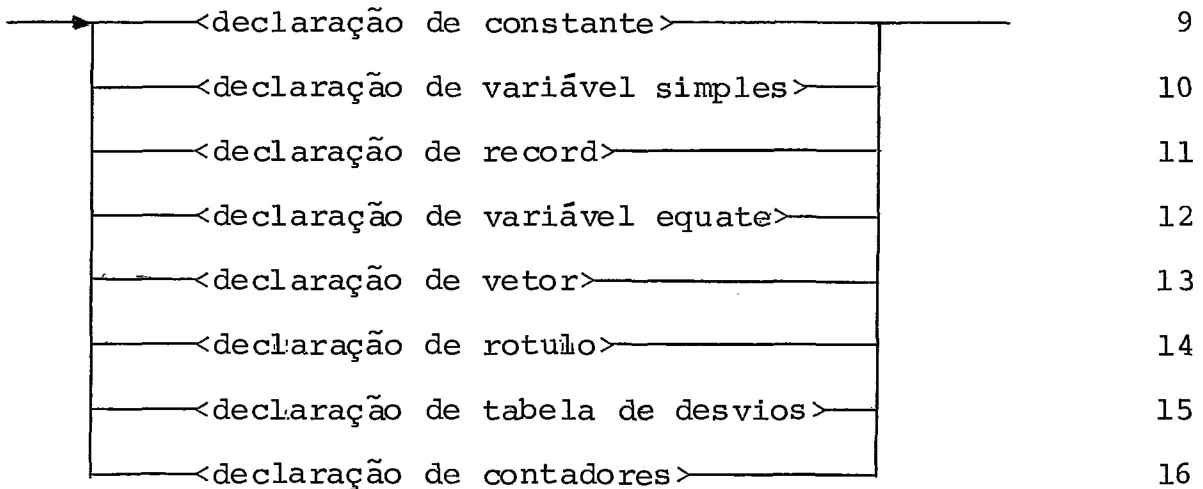
2 <lista>:: = (26)



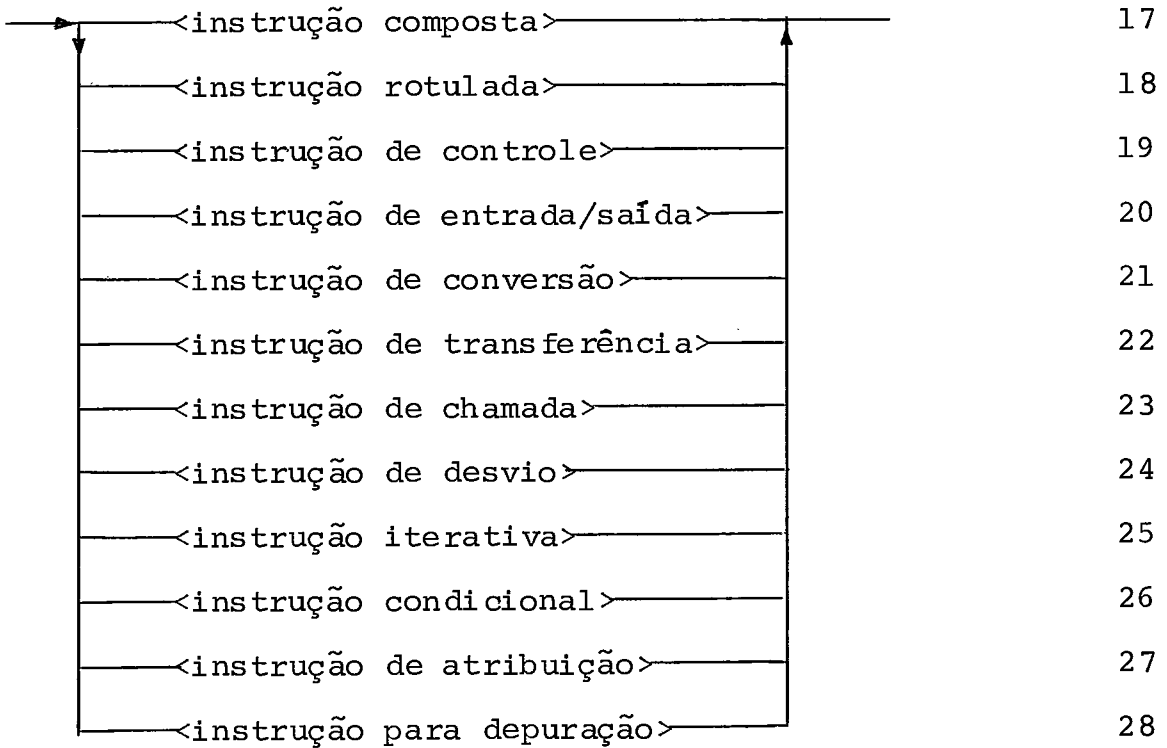
3 <declaração procedimento>:: = (27)



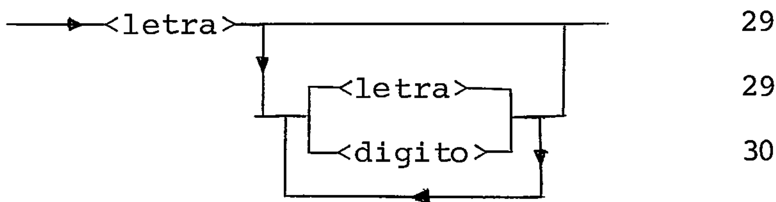
4 <declaração>:: = (28)



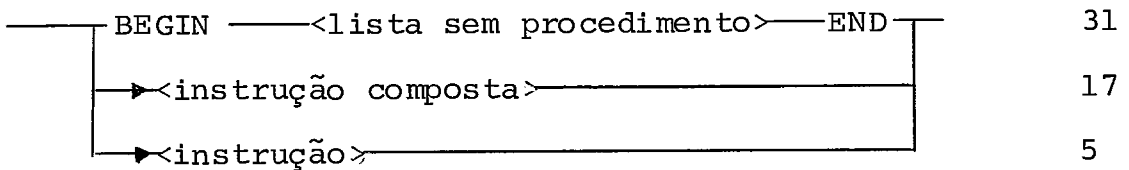
5 <instrução>:: = (29)



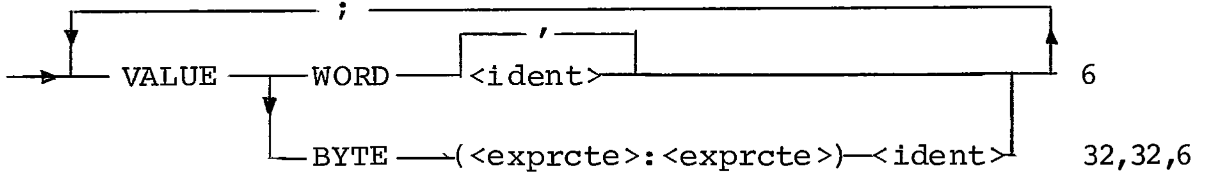
6 <identificador>:: = (29)



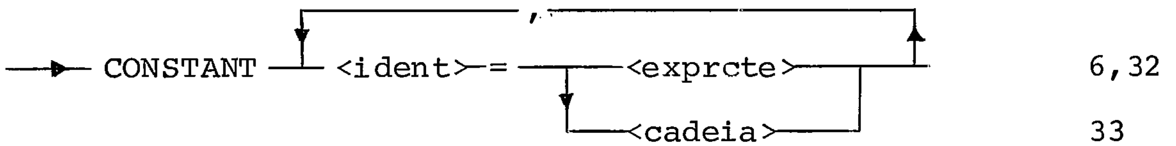
7 <corpo do procedimento>:: = (30)



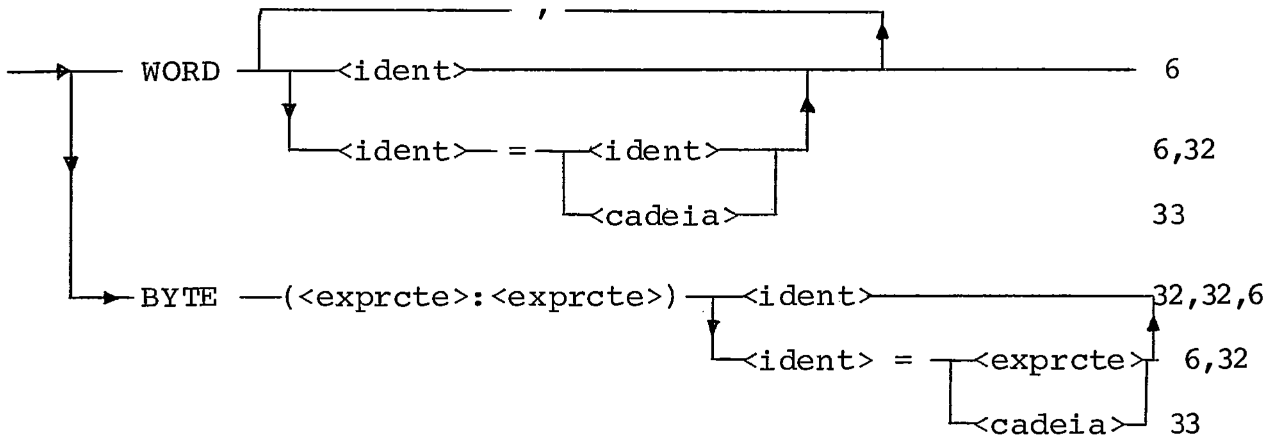
8 <lista dos parâmetros formais>:: = (33)



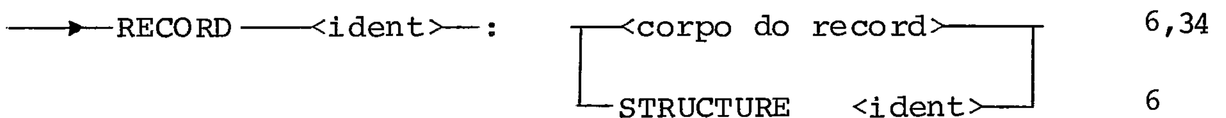
9 <declaração de constante>:: = (33)



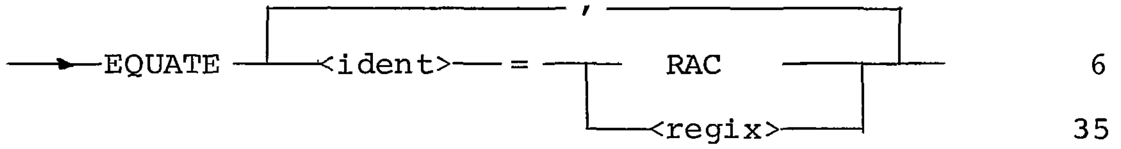
10 <declaração de variável simples> (34)



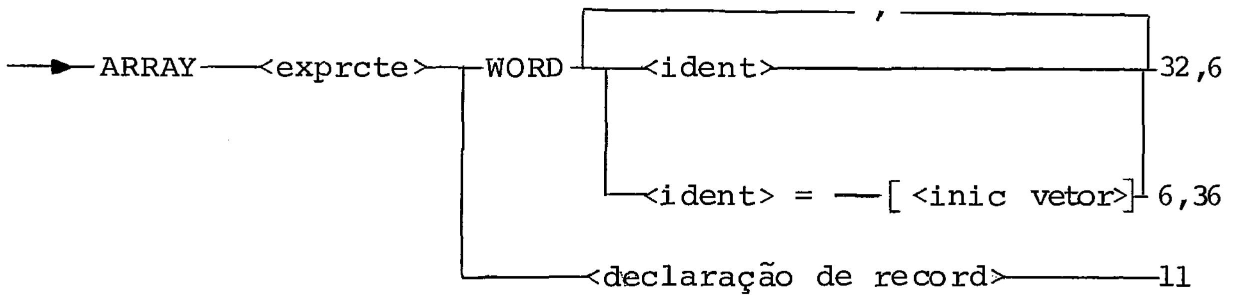
11 <declaração de record>:: = (36)



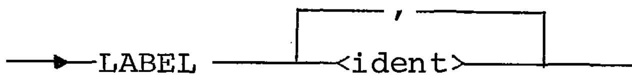
12 <declaração de variável equate>:: = (37)



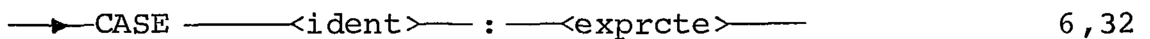
13 <declaração de vetor>:: = (38)



14 <declaração de rotulo>:: = (40)

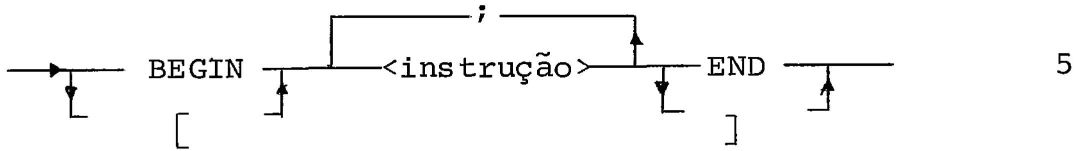


15 <declaração de tabela de desvios>:: = (41)

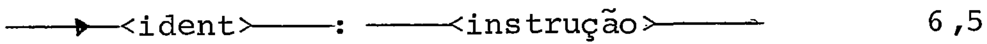


16 <declaração de contadores>:: = (42)

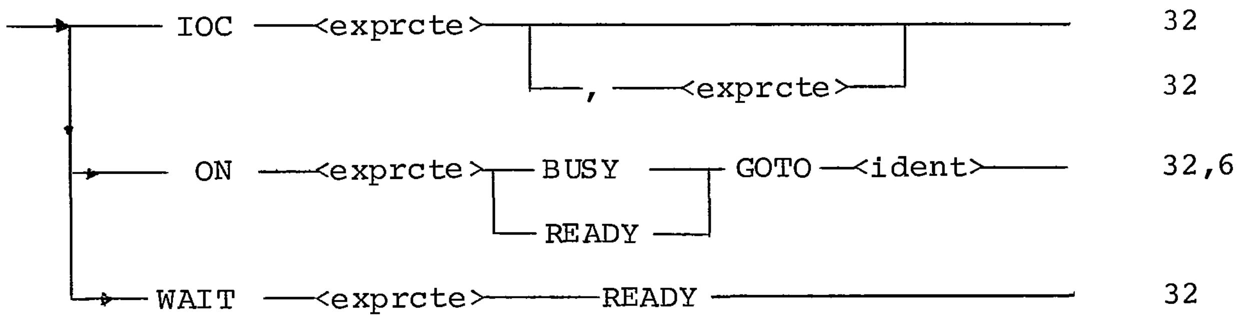
17 <instrução composta>:: = (43)



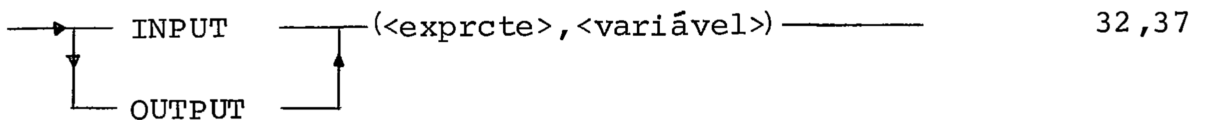
18 <instrução rotulada>:: = (43)



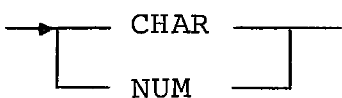
19 <instrução de controle>:: (44)



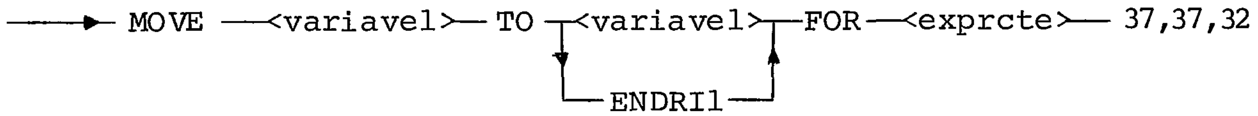
20 <instrução entrada/saída>:: = (45)



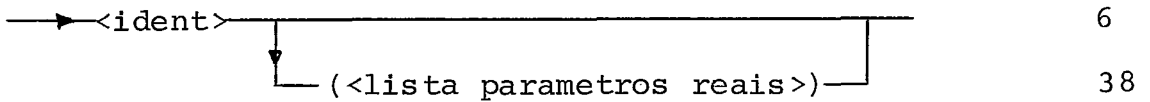
21 <instrução de conversão>:: = (46)



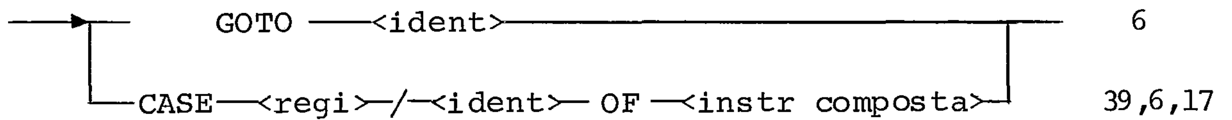
22 <instrução de transferencia>:: = (46)



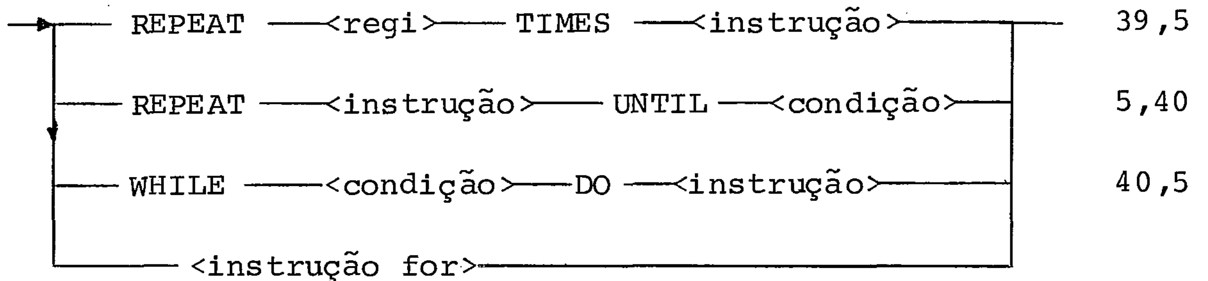
23 <instrução de chamada>:: = (48)



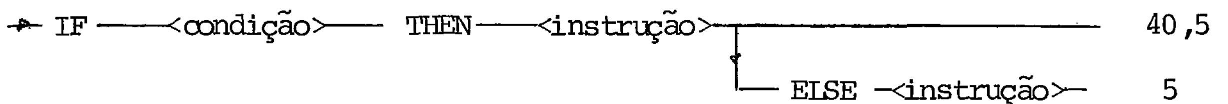
24 <instrução de desvio>:: = (51)



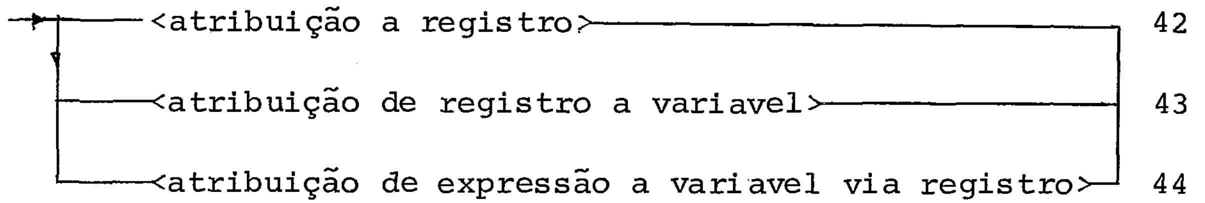
25 <instrução iterativa>:: = (53)



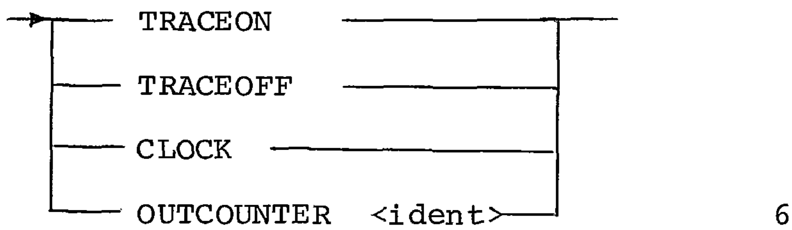
26 <instrução condicional> :: = (56)



27 <instrução de atribuição>:: = (57)

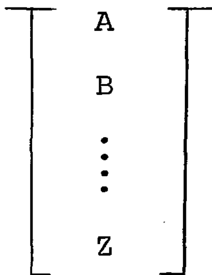


28 <instruções para depuração>:: = (58)

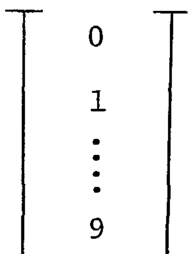


6

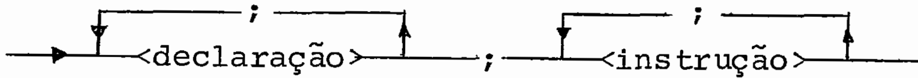
29 <letra>:: = (61)



30 <digito>:: = (61)

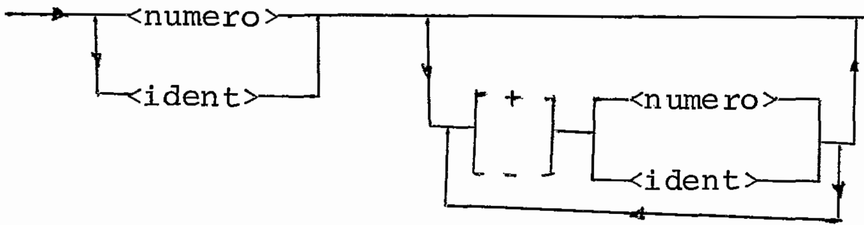


31 <lista sem procedimento>:: = (62)



4,5

32 <expressão constante>:: = (62)



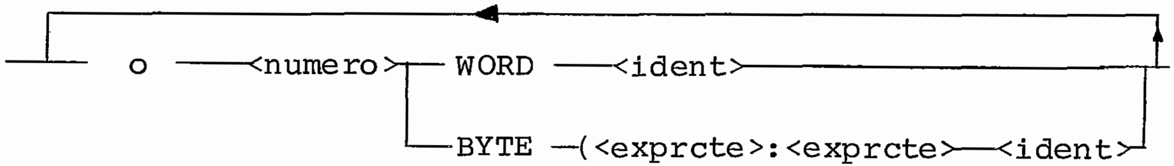
45
6
45

6

33 <cadeia>:: = (63)

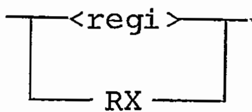
46

34 <corpo do record>:: = (64)

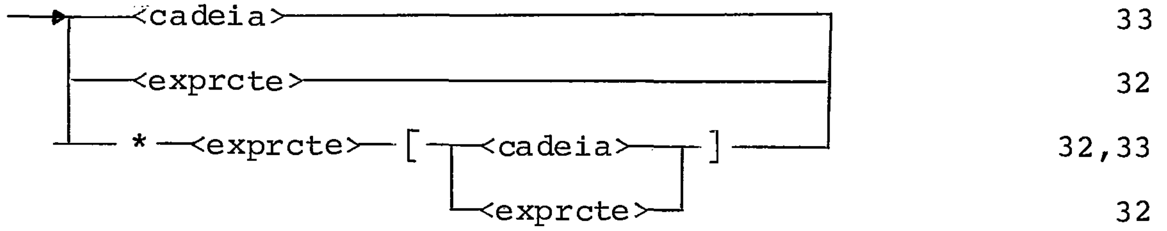


32,32,6

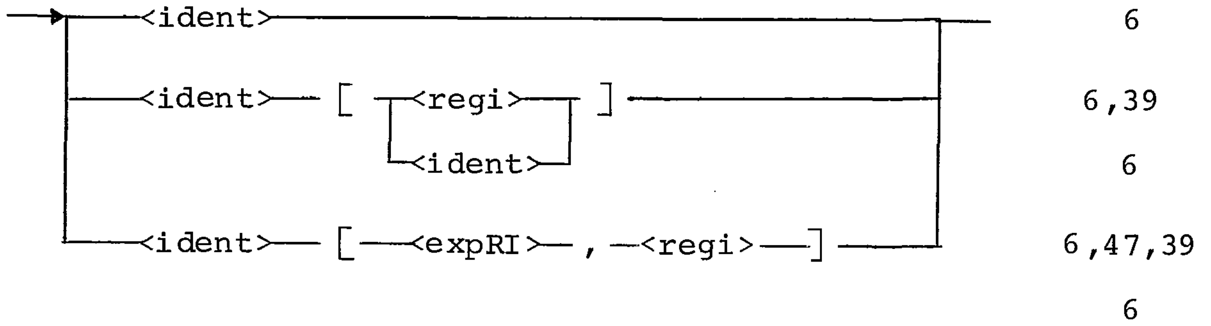
35 <regix> (65)



36 <inicialização do vetor>:: = (65)

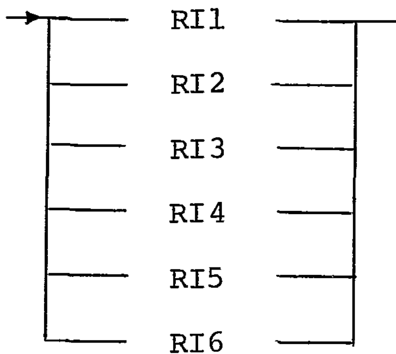


37 <variavel>:: = (66)

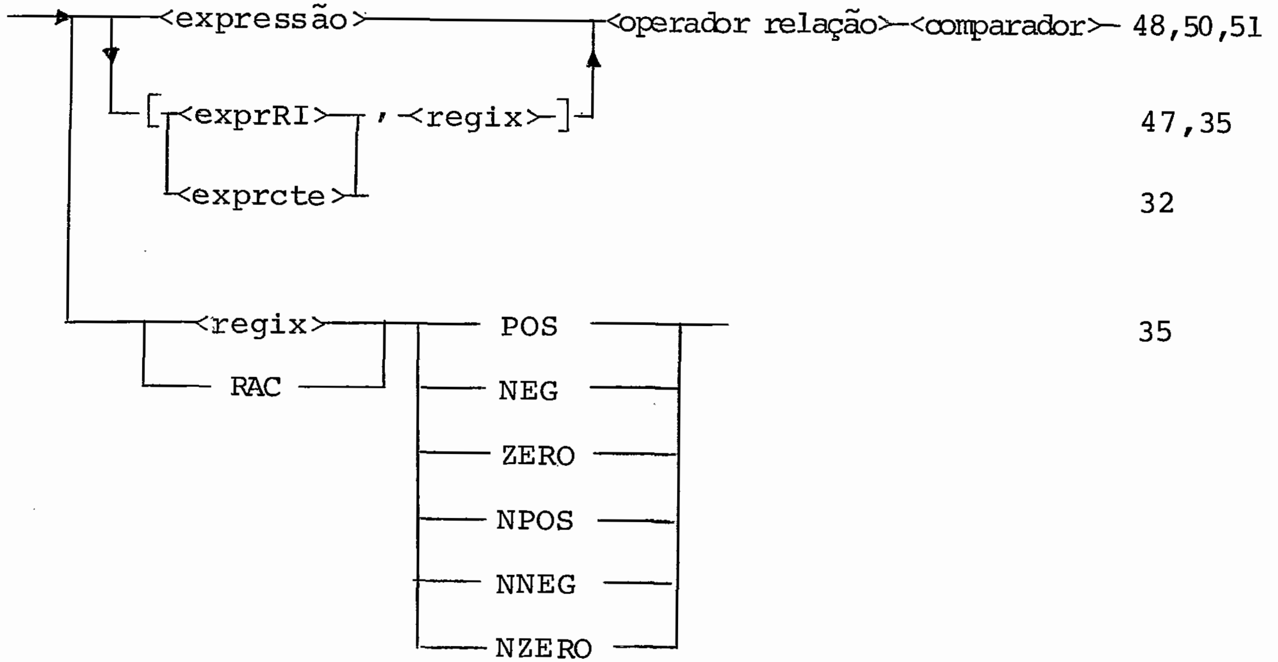


38 <lista parametros reais>:: = (67)

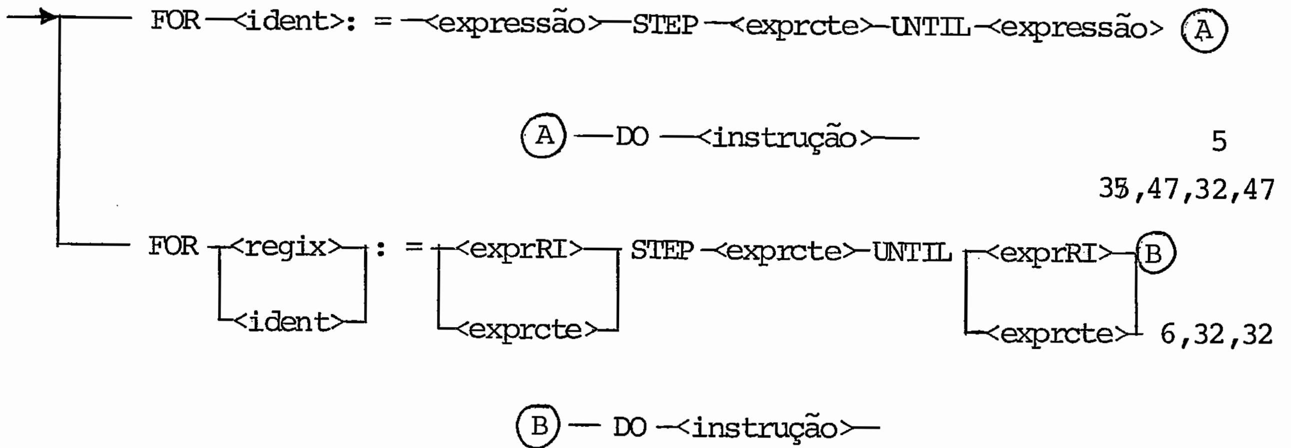
39 <regi>:: = (68)



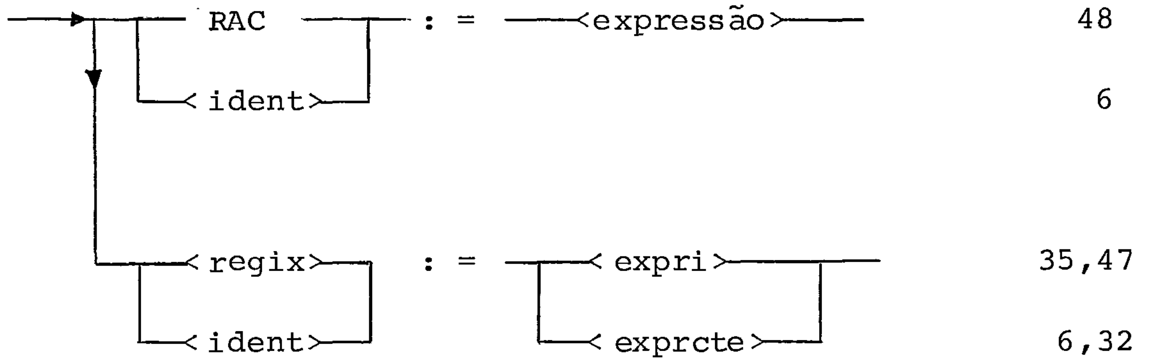
40 <condição>:: = (68)



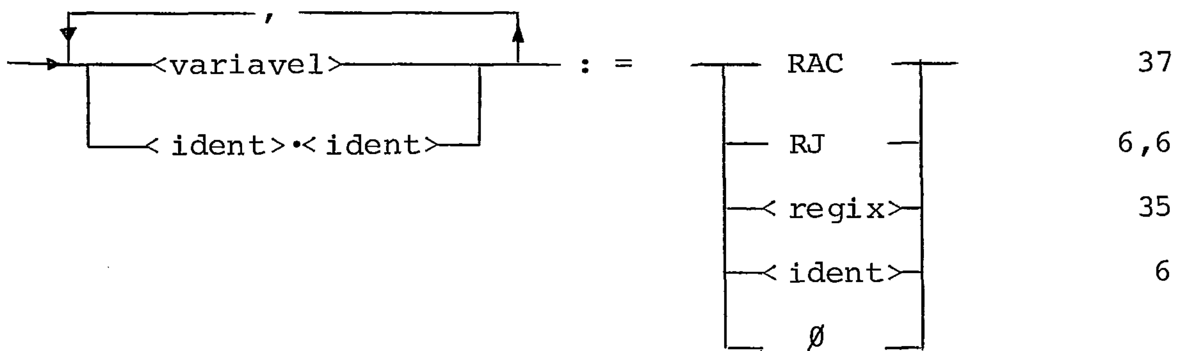
41 <instrução for>:: = (71)



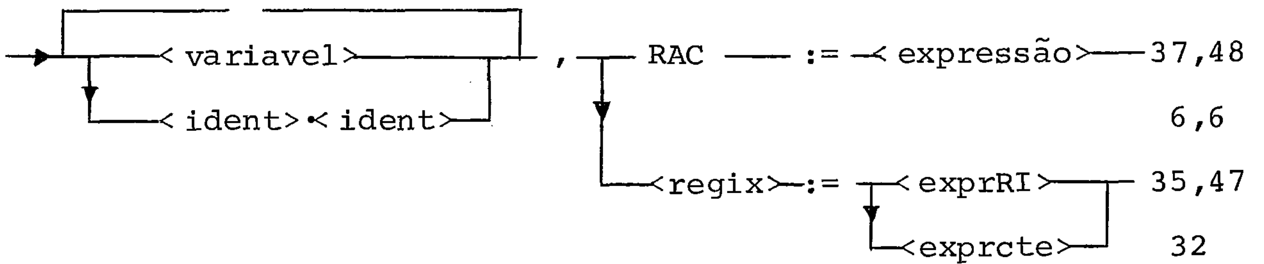
42 <atribuição a registro>:: = (74)



43 <atribuição de registro a variavel> (76)

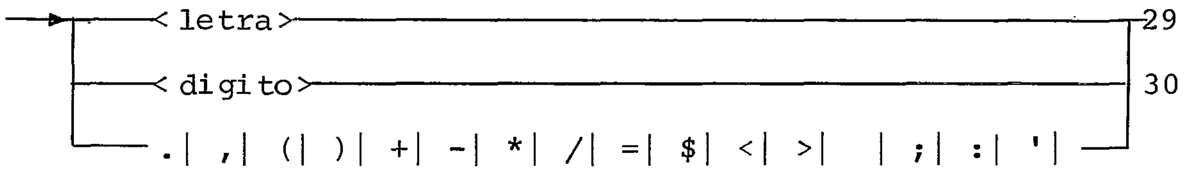


44 <atribuição de expressão a variavel via registro>:: = (78)

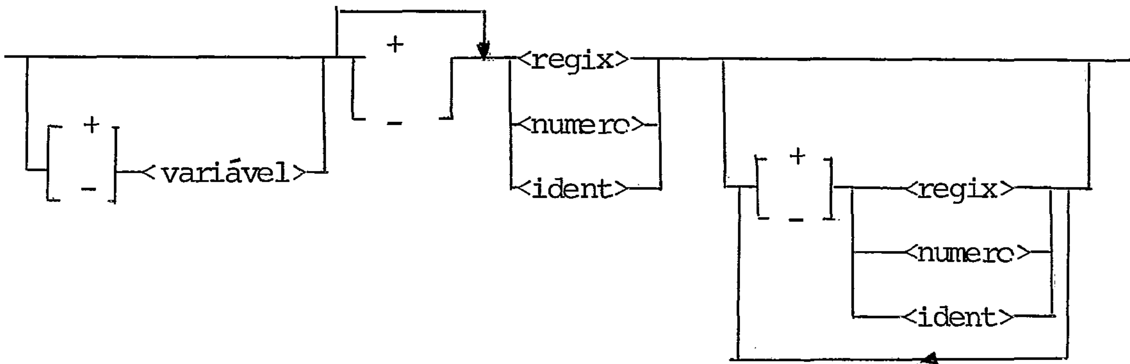


45 <numero>:: = (79)

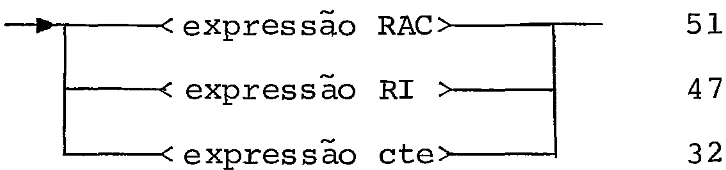
46 <carater>:: = (79)



47 <expressão RI>:: = (79)



48 <expressão>:: = (81)

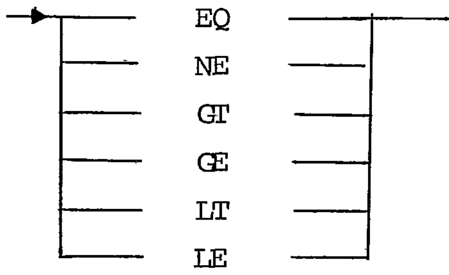


51

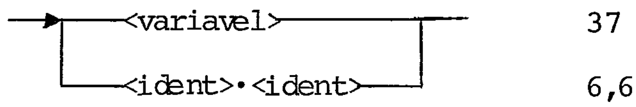
47

32

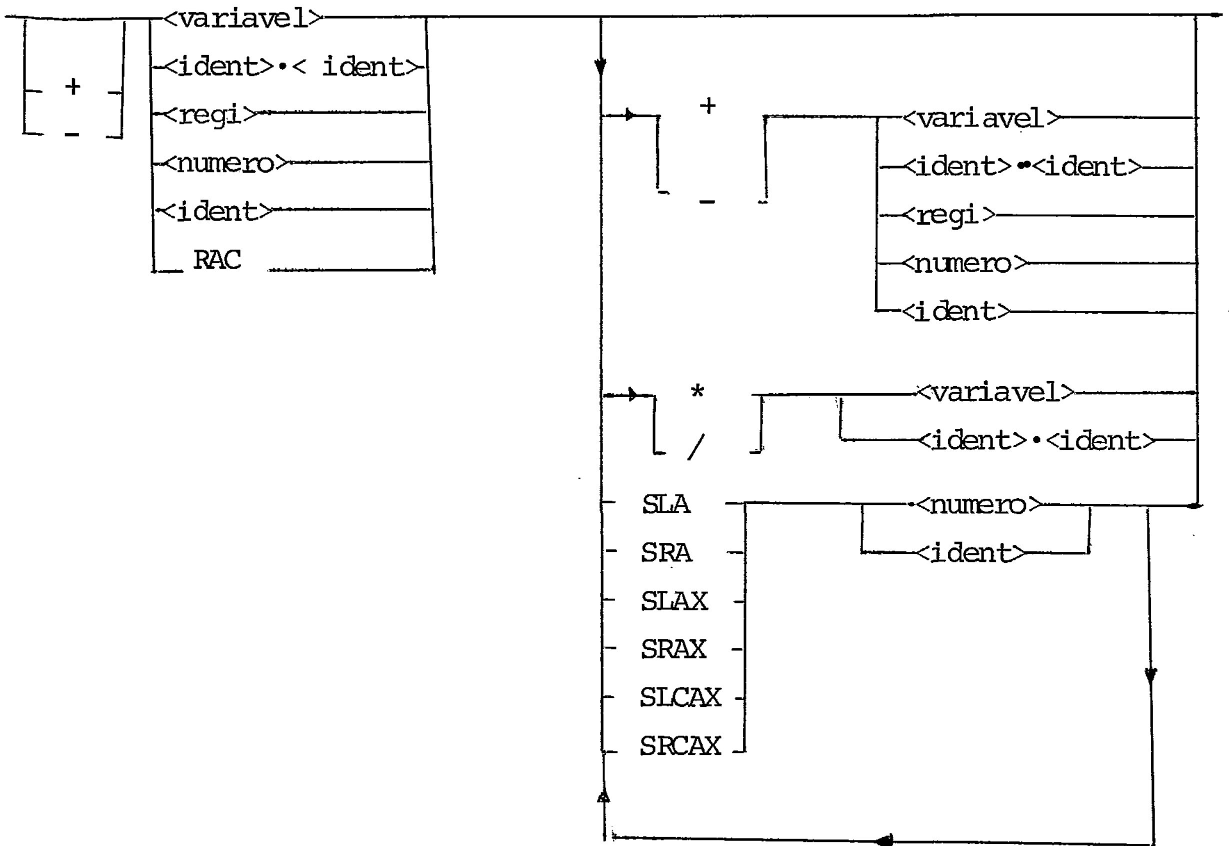
49 <operador relação>:: = (82)



50 <comparador> (82)



51 <expressão RAC> (83)



B I B L I O G R A F I A

|1| Wulf, W. -"A Case Against the GOTO" - Proc. ACM (1972), 791 - 796.

|2| Hoare, C.A.R. - "Hints on Programming Language Design" - Stanford Artificial Intelligence Laboratory - MEMO AIM (1973)

|3| Knuth, D.E. - "The Art of Computer Programming" - vol. 1: Fundamental Algorithm, Addison-Wesley (1968)

|4| Knuth, D.E. - "The Art of Computer Programming" - vol. 2: Seminumerical Algorithms, Addison-Wesley (1968)

|5| Knuth, D.E. - "The Art of Computer Programming" - vol. 3: Sorting and Searching, Addison-Wesley (1968).

|6| Markenzon, L. - "Simulador/Montador MIX em Mini-Computador com Sistema de Tempo Compartilhado" - Tese nº 1009 - Departamento de Sistemas e Computação, COPPE/UFRJ (1978).

|7| MITRA 15 - "Manuel d'Utilization: Périphériques" - Companhie Internacionale pour l'Informatique, 1973.

|8| Wirth, N. - "PL360 - A Programming Language for the 360 Computers" - JACM 15 (jan 1968), 37-74.

|9| Gries, D. - "Compiler Construction for Digital Computers"- Wiley (1971).

|10| De Simone, E. - "Tese de Doutoramento, a publicar".

|11| Knuth, D.E. - "An Empirical Study for Fortran Programs"- Software Practice and Experience 1, (1971) 105-133.

|12| Freeman, D. - "Error Correction in CORC: The Cornell Computing Language" - Tese, Cornell University (1963).

|13| Morgan, H.L. - "Spelling Correction in System Programs"- CACM 13 (fev. 1970), 90-94.

|14| Gries, D. - "Error Recovery and Correction - An Introduction to the Literature" - Compiler Construction - an advanced Course (628-631), Springer-Verlag (1976).

|15| MacKeeman et al. - "A Compiler Generator " - Prentice-Hall (1970).

|16| Aho, A.U. et al. - "Principles of Compiler Design" - Addison-Wesley (1978).

|17| Hartmann, - " A Concurrent Pascal Compiler for Mini-computers" - Lectures Notes on Computer Science nº Springer-Verlag (1977).

|18| Kennedy, K. et al. - "Automatic Generation of Efficient Evaluators for Attribute Grammars" - Conference of the Third ACM Symposium on Principles of Programmin Languages (1976).

|19| Bauer, F.L. et al. - "Compiler Construction - an Advanced Course" - Springer-Verlag (1976).

|20| MITRA 15 - "Manuel d'Utilization: Langage LP15, LP15,E"- Compagnie Internationale pour l'Informatique (1975).

|21| Naur, P. et al. - "Revised Report on the Algorithmic Language ALGOL 60" - CACM 6 (Jan 73), 1-16

|22| Gries, D. - "Use of Transition Matrices in Compiling" - CACM 11 (jan 1968), 26-31.

|23| Brent, R.P. - "Reducing the Retrieval Time of Scatter Storage Tecniques" - CACM 16 (fev. 1973), 105-109.

|24| Pratt, T. - "Programming Languages: Design and Implementation" - Prentice-Hall (1975).

|25| Knuth, D.E. - "An Empirical Study of Fortran Programs"- Software - Practice and Experience 1, (1971), 105-133.