AMBIENTE DE PROGRAMAÇÃO PASCAL: FORMA INTERMEDIÁRIA E INTERPRETADOR/DEPURADOR SIMBÓLICO

Antonio Carlos de Oliveira

TESE SUBMETIDA AO CORPO DOCENTE DOS PROGRAMAS DE PÓS-GRADUAÇÃO EM 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:

José Lucas Mourão Rangel Netto
Presidente

Estevam Gilberto de Simone

Edil Severiano Tavares Fernandes

Paulo Mário Bianchi França

RIO DE JANEIRO, RJ - BRASIL

MAIO DE 1983

OLIVEIRA, ANTÔNIO CARLOS DE

Ambiente de Programação PASCAL: Forma Intermediária e Interpretador/Depurador Simbólico |Rio de Janeiro| 1983.

VIII, 188 , 29,7 cm (COPPE-UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1983).

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

1. Compiladores e Linguagens de Programação. I. COPPE/UFRJ. II. Título (série).

"Tu te tornas eternamente responsável por aquilo que cativas."

(Antoine de Saint-Exupéry)

À Elisabete,
minha mulher.

AGRADECIMENTOS

A meus amigos Lígia Alves Barros, Estevam G. De Simone e José L.M. Rangel Netto, que se sucederam na orientação desta Tese, dando-me todo apoio necessário,

A todos que contribuiram com seu incentivo e, em alguns casos, com importantes sugestões,

À Denise Schwartz, pelo excelente trabalho de datilografia e de decifração de hieroglifos,

À UFRJ, pelo suporte financeiro indispensável,

A meus filhos, pela motivação fundamental,

Muito Obrigado.

CURRICULUM VITAE SUMÁRIO

Antônio Carlos de Oliveira é Engenheiro Eletricista, formado pela Escola de Engenharia da Universidade Federal do Rio de Janeiro, em 1971.

Trabalhou na Burroughs Eletrônica Ltda. e para o Serviço Federal de Processamento de Dados - SERPRO, tendo ocupado funções técnicas e gerenciais.

Lecionou na Fundação Getúlio Vargas (CADEMP) em cursos na área de Processamento de Dados.

É professor do Instituto de Matemática da UFRJ, desde abril de 1971, lotado no Departamento de Ciência da Computação.

Leciona, desde agosto de 1982, no Programa de Engenharia de Sistemas e Computação da COPPE/UFRJ.

ÍNDICE

Pág.

RESUMO

Apresentamos aqui a especificação para implementação de um Interpretador/Depurador PASCAL, que se pretende venha a fazer parte de um Ambiente de Programação Pascal, projeto em andamento na COPPE/UFRJ. Tal projeto deverá incluir também um Analisador Léxico/Sintático e um Gerador/Otimizador de código.

Para esse fim, está especificada aqui uma proposta de Forma Intermediária Pascal (árvore de sintaxe abstrata, com atributos parcialmente coletados, e tabela de símbolos). Além disso estão descritos os programas correspondentes ao Interpretador/Depurador simbólico, cuja função é percorrer a Árvore de Sintaxe Abstrata durante a interpretação, oferecendo ao usuário diversos recursos de depuração a nível fonte.

ABSTRACT

We present here the specification for implementation of a PASCAL Interpreter/Debugger, which is meant to be part of a Pascal Programming Environment, presently being developped at COPPE/UFRJ. This projet will also include a Parser/Scanner and a Code generator/optimizer.

In order to do that, we have specified here a proposal of a Pascal Intermediate Form (abstract syntax tree, with partially collected attributes, and symbol table). We have also included the description of the programs which compose the Interpreter/Symbolic Debugger, which traverses the Abstract Syntax Tree during interpretation, providing the user with several debugging facilities at source level.

CAPÍTULO I

I. AMBIENTE DE PROGRAMAÇÃO PASCAL - DESCRIÇÃO DO PROJETO

Temos observado que a maioria dos compiladores disponíveis no mercado, tanto para máquinas nacionais, quanto para máquinas estrangeiras, cumpre quase estritamente sua função básica de traduzir os programas para linguagem objeto (para execução direta) ou para linguagem intermediária (para interpretação), sendo em geral muito pobre nas facilidades oferecidas à equipe de projeto, do programador ao gerente, para depuração, teste, manutenção e controle.

Em nossa experiência como professor da UFRJ, em cursos básicos de computação que ministramos, temos observado os alunos iniciantes frequentemente "perdidos" diante de mensagens de erro inteiramente inadequadas.

Em nossa experiência como analista de sistemas, no mercado de trabalho convencional, constatamos, não raro, bons programadores debruçados por horas sobre listagens, rastreando erros de lógica, e outros, certamente não tão bons programadores, depurando seus programas pelo condenável método de tentativa e erro, pela falta de uma ferramenta de depuração confortável.

Certamente tais problemas se constituem em prejuízo: do aluno, que muitas vezes se desestimula; e da empresa, que paga pelos homens-hora de trabalho desnecessários.

Recentemente surgiu a idéia de Ambientes de Desenvolvimento (AD). Segundo Anthony I. Wasserman em seu artigo Automated

Development Enviroments [10], o objetivo destes ambientes é aumentar a produtividade do pessoal envolvido e prover uma série de ferramentas que simplifiquem o processo de produção de software, devendo conter facilidades tanto para os membros individuais do grupo de projeto quanto para seu gerente geral, requerendo um ambiente de desenvolvimento completo a inclusão do sistema operacional e de seus utilitários, de linguagens de programação e seus tradutores, de recursos de depuração a tempo de execução, de editores de texto e facilidades de documentação e de ferramentas gerenciais, inclusive para acompanhar a produtividade do pessoal.

A presente Tese nasceu do interesse de um grupo de professores e alunos da COPPE/UFRJ em desenvolver o projeto de um ambiente de desenvolvimento adequado a mini-computadores de fabricação nacional. Sendo o pessoal interessado favorável à idéia de um projeto em etapas, para que mais rapidamente os resultados fossem alcançados, optamos por uma fase inicial menos ambiciosa, a partir da qual se poderia chegar ao objetivo almejado.

Quanto à linguagem, procurou-se escolher alguma adequada ao ensino universitário, moderna, pequena, porém rica nas estruturas de dados providas. Com tais definições, optamos quase inevitavelmente pelo PASCAL.

Quanto às facilidades oferecidas ao programador, optamos, entre outras, por um bom recuperador de erros, com mensagens sempre que possível claras e precisas, um editor e um indentador capazes de formatar o programa de maneira organizada e um depurador não-iterativo, capaz de socorrer o usuário, durante a execução, com uma série de informações a nível simbóli-

co.

Não tivemos a pretensão de alterar procedimentos de Sistema Operacional e não incluimos ferramentas para controle gerencial de projeto. Assim, podemos dizer que, nesta fase inicial dos trabalhos, estaremos desenvolvendo o projeto de um Ambiente de Programação PASCAL (APP) que poderá ser expandido, a ponto de ser considerado, futuramente, um Ambiente de Desenvolvimento PASCAL (ADP).

Ainda para esta primeira etapa do projeto, uma série de simplificações foram feitas e serão facilmente constatáveis no corpo deste trabalho. Assim, optamos por não implementar algumas características da linguagem cuja ausência não a comprometessem. E, em relação ao projeto geral, optamos por um depurador não iterativo, a ser substituido futuramente por outro, mais poderoso, no qual, a tempo de execução, no terminal, o usuário poderá interromper seu programa, solicitando procedimentos de depuração e até mesmo alterando o texto fonte. Finalmente, optamos por um conjunto básico de recursos de depuração a ser posteriomente expandido.

Está o projeto dividido em três módulos, que têm como ponto comum a Forma Intermadiária PASCAL. A figura (I.1) dá a idéia geral do mesmo.

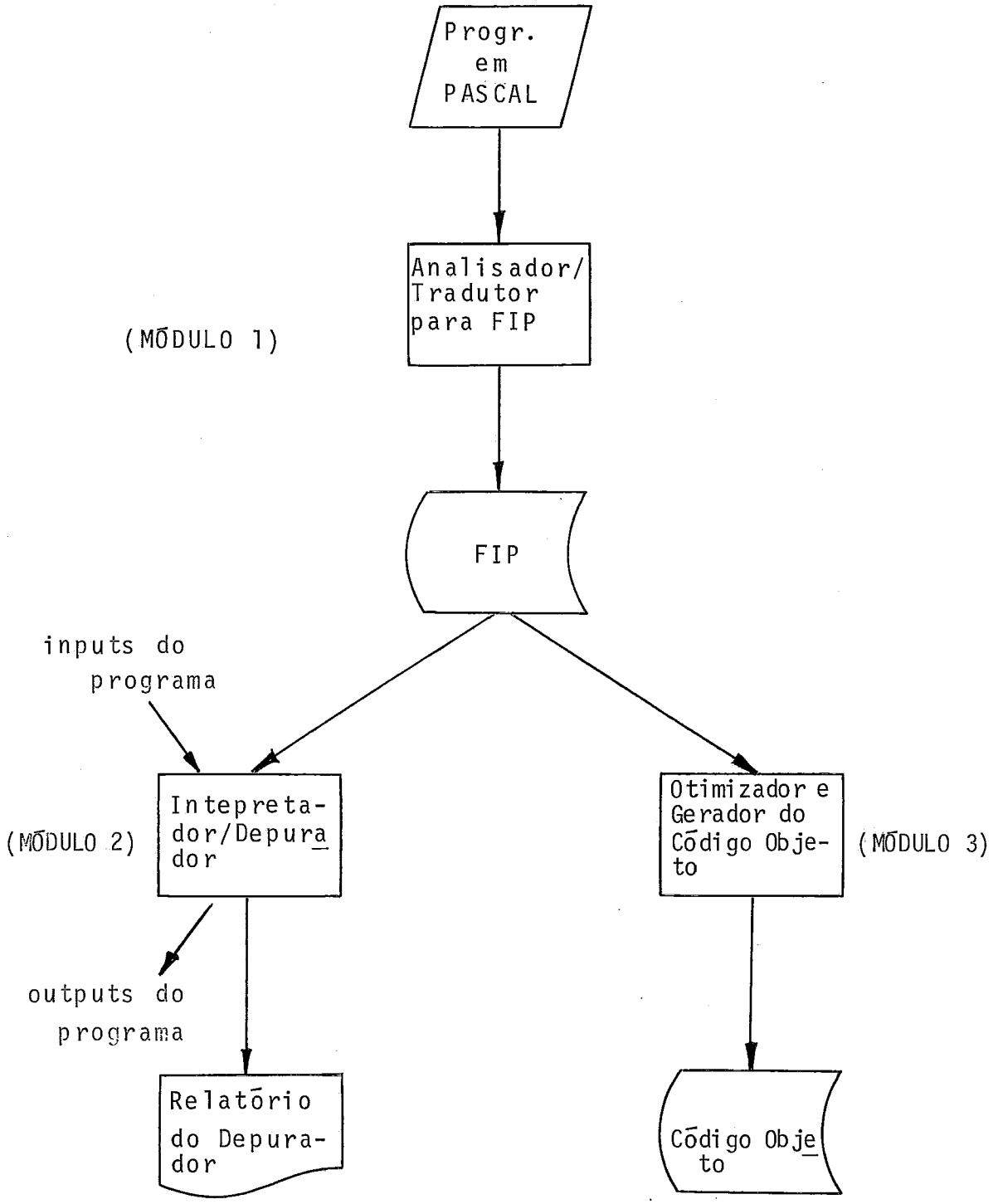

Progr.
em
PASCAL
Analisador/
Tradutor
para FIP
(MÓDULO 1)
FIP
inputs do
programa
Intepreta-
dor/Depura
dor
(MÓDULO 2)
Otimizador e
Gerador do
Código Obje-
to
(MÓDULO 3)
outputs do
programa
Relatório
do Depura-
dor
Código Obje
to

Figura - I.1.

O módulo 1 analisa o programa fonte, escrito em PASCAL, e o traduz para a Forma Intermediária PASCAL (doravante conhecida como FIP), dando como saída (se o programa estiver correto), esta FIP e uma série de tabelas associadas.

O módulo 2 recebe como entrada a FIP passada pelo módulo 1, as tabelas associadas e os inputs normais do programa, executando-o na forma interpretativa, fornecendo como saída os outputs normais do programa e um relatório com informações para depuração do programa a nível fonte.

O módulo 3 otimiza a FIP recebida do módulo 1, gerando o código de máquina correspondente.

A modularização do projeto facilita a divisão dos esforços e atende às restrições de memória da classe de máquinas a que se destina.

A razão de estarem previstas duas possíveis formas de execução (via interpretação e via código objeto) é dar total flexibilidade ao usuário. Assim, na fase de depuração e testes do programa, o usuário deverá recorrer à forma interpretativa, quando o volume de informações de auxílio à depuração disponíveis é grande; e na fase de produção o usuário deverá recorrer à versão do programa em código objeto, que naturalmente permitirá uma execução mais eficiente.

A nossa Tese trata da Forma Intermediária e do módulo 2, Interpretador/Depurador PASCAL, parte do projeto geral que nos coube. Os outros dois módulos deverão corresponder a duas outras teses de alunos da COPPE/UFRJ.

As teses tratam da fase de definições do projeto. Seu desenvolvimento, na versão inicial, dependerá do interesse de

alguma instituição em patrocinã-lo. De qualquer modo, constituem-se as teses numa contribuição, a nível teórico, na área de Ambientes de Programação.

Para elaboração desta Tese, uma ampla literatura foi consultada, sendo parte dela mencionada na bibliografia anexa. Este material serviu, fundamentalmente, para ampliar nossa base teórica, o que nos possibilitou maior desenvoltura na execução deste trabalho.

A idéia básica deste projeto resulta de uma concepção relativamente recente, de Ambientes de Desenvolvimento e de Ambientes de Programação.

Assim, poderíamos dizer que, com o surgimento da linguagem ADA, esta idéia ganhou substância e o artigo "ADA Debugging and Testing Support Enviroments", de Richard E.Fairley, publicado em SIGPLAN NOTICES, volume 15, número 11, de 1980 [5], serviu de ponto de partida e de incentivo ao nosso trabalho. De resto, a mencionada publicação nos auxiliou, através de outros artigos, em diferentes aspectos do projeto.

Um conjunto de três artigos anteriores, datados de 1978, de autoria de Kin-Man Chung e Herbert Yuen, denominados A "Tiny" Pascal Compiler [7,8,9] serviu também de subsídio ao nosso projeto, quanto à sua concepção geral.

Finalmente, para melhor compreensão quanto à implementação de características específicas do PASCAL, foram consultados, entre outros, diversos artigos do Simpósio da Universidade de Southampton, de março de 1977 [6] e mesmo listagens de compiladores PASCAL de uso corrente.

CAPÍTULO II

II. ASPECTOS GERAIS

Neste capítulo serão tecidas considerações sobre vários aspectos do projeto que servirão de base à compreensão dos capítulos seguintes.

II.I. QUANTO À FORMA INTERMEDIÁRIA

A Forma Intermediária Pascal-FIP-retrata ao máximo as características do programa fonte, tendo o aspecto de uma árvore sintática com atributos parcialmente coletados. Esta árvore é alinhavada, o que possibilita seu percurso eficiente, a tempo de interpretação.

A figura (II.1) esquematiza a árvore de um programa completo. A figura (II.2) apresenta um pequeno trecho de programa em Pascal e a figura (II.3) a FIP correspondente a ele.

Da figura (II.3) destacamos três classes de nós: de delimitador (;), de operação (atribuição de real a inteiro ou i: = r) e de variável (X).

Nós de variáveis apontam para a tabela de símbolos. Nós de operações têm atributos coletados. Assim, por exemplo, o nó i + r avisa que será feita uma operação de soma entre uma variável inteira e uma variável real, o que, naturalmente, implicará numa conversão de tipo a tempo de interpretação. A razão de se haver coletado alguns atributos na árvore é ganhar

tempo de interpretação. De outra forma, por exemplo, seriam necessárias buscas adicionais na tabela de símbolos (TS), para os tipos dos operandos, no caso da operação i + r vista. Todavia, não fica totalmente evitada a ida à TS, já que é de lá que se vai buscar o endereço, na forma [Número do Registro de Ativação, deslocamento], de cada variável. Esta ida à TS poderia ser evitada se nós de variáveis tivessem, além do endereço da TS, o próprio endereço da variável. Todavia, existe um compromisso entre tempo e memória. No caso, optou-se por economia deste último recurso.

Como se pode concluir da descrição anterior, a TS (na verdade parte dela) é requerida a tempo de interpretação. Além dela, algumas outras tabelas acompanham também a FIP.

O objetivo deste tópico foi introduzir ao leitor a FIP, apresentada, neste estágio, de forma bastante geral. Será visto, ao longo deste trabalho, que o projeto da FIP foi quase sempre um processo da decisão: escolhia-se uma alternativa, com base em algum critério, abandonando-se outras também viáveis. Por outro lado, algumas características da FIP foram nela inseridas apenas para atender futura expansão do projeto.

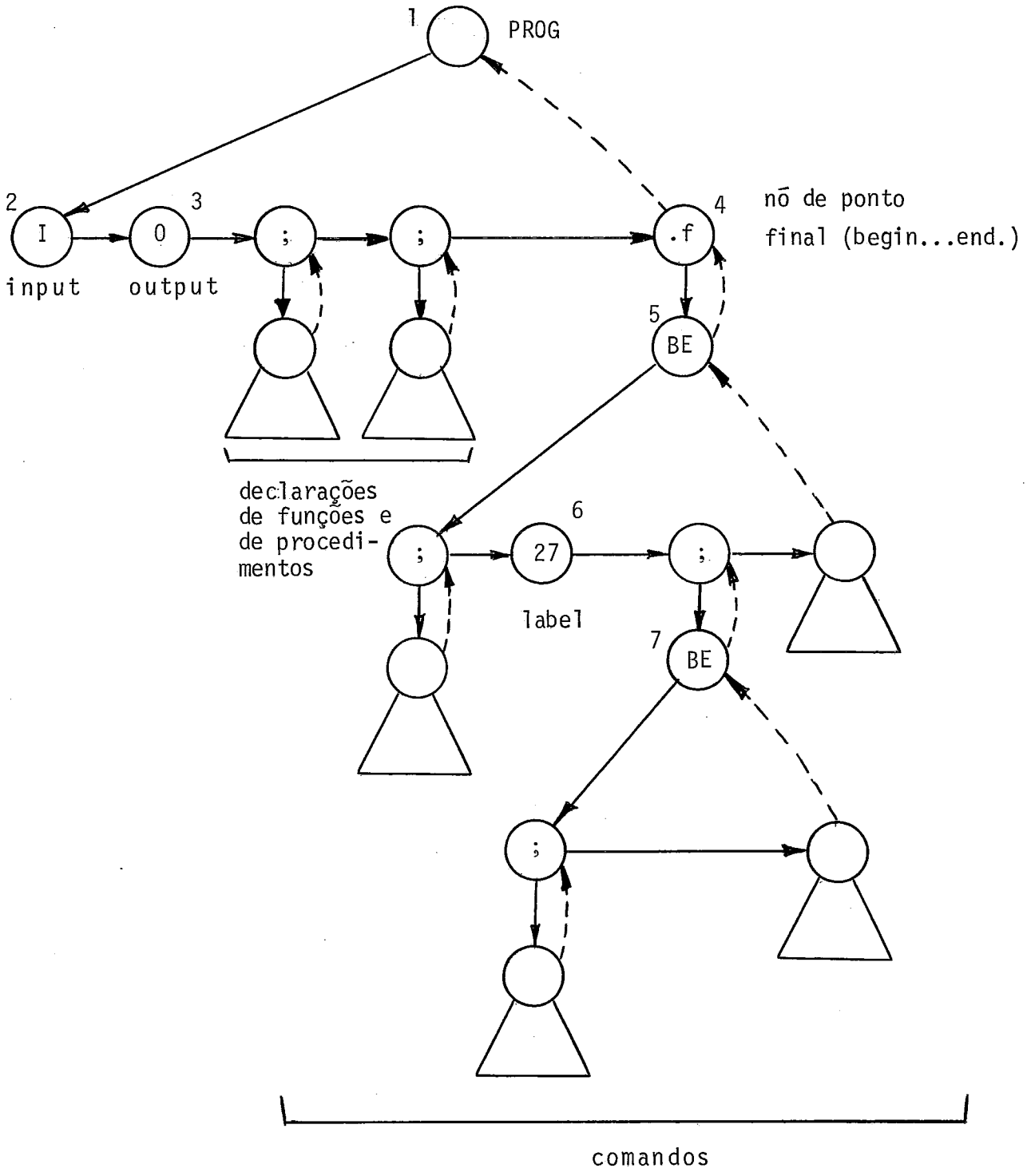

1
PROG
2
I
input
3
0
output
;
;
4
.f
nó de ponto
final (begin...end.)
5
BE
declarações
de funções e
de procedi-
mentos
6
27
label
;
;
7
BE
;
comandos

Figura - II.1

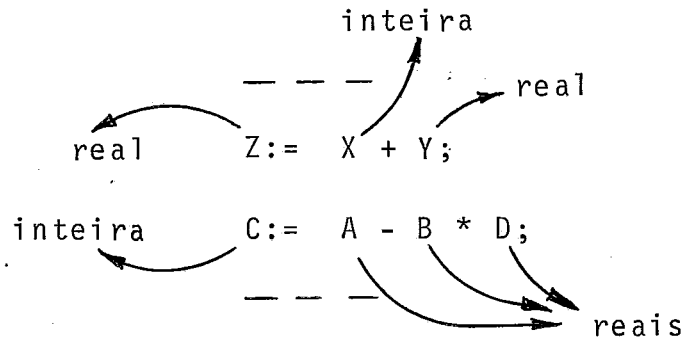


Figura - II.2

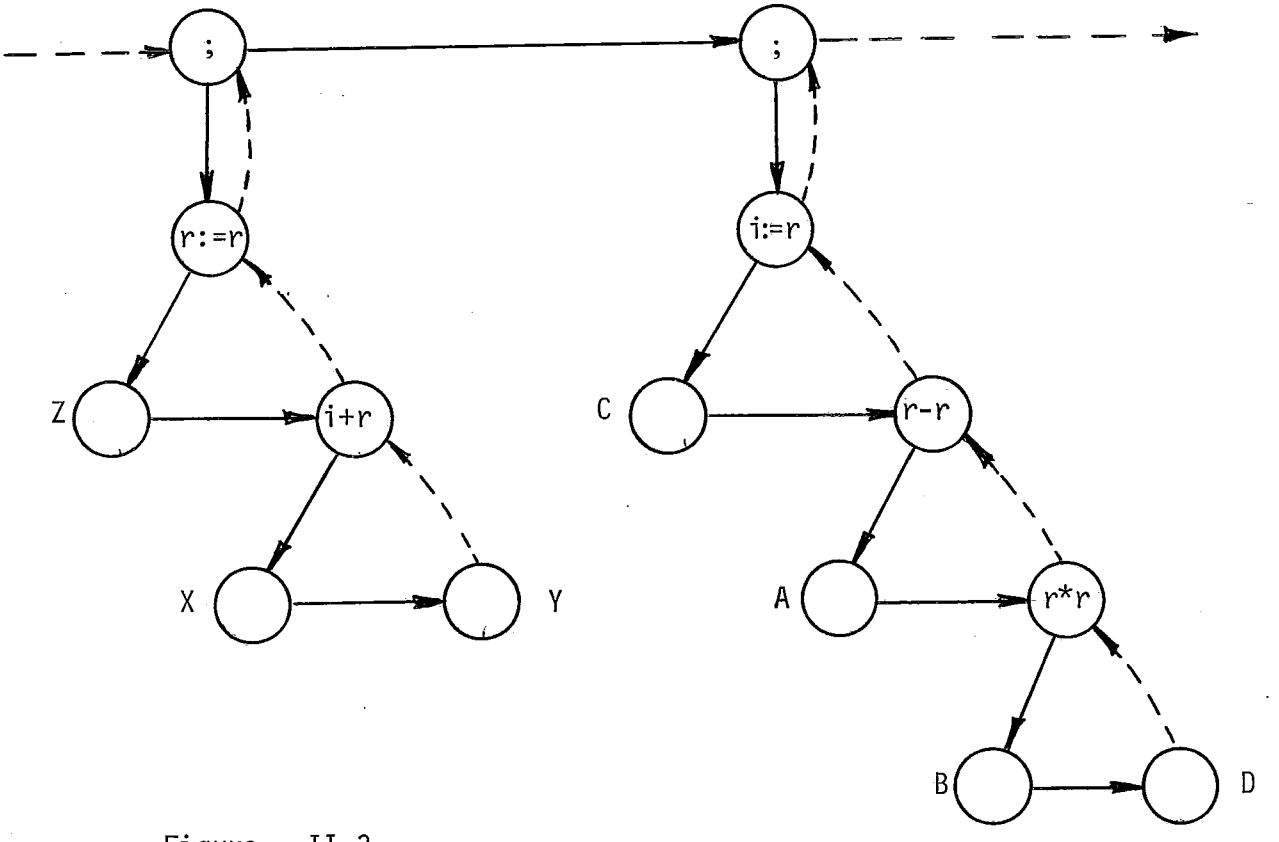


Figura - II.3

II.II. QUANTO AOS NÓS

Embora a quantidade de informações varie com o tipo de nó, não havendo portanto uniformidade, não deveremos trabalhar com nós de tamanho variável: eles serão padronizados em função do maior nó possível.

Genericamente, um nó pode ser visto de acordo com a estrutura mostrada na figura (II.4).

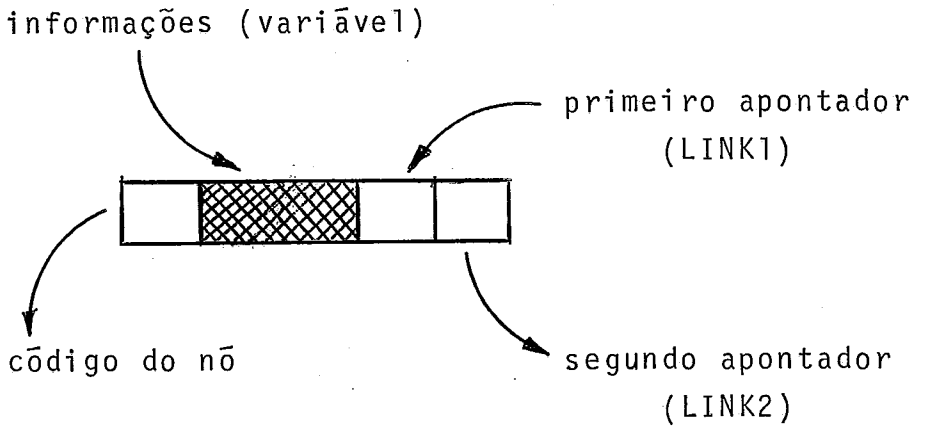


Figura - II.4

Todo nó terá um código, que identifica seu tipo: nó de variável, nó de soma de inteiro com real, nó de ";", etc.

Dependendo do tipo de nó, será requerido um determinado conjunto de campos de informações: endereço da TS se nó de variável; nada, se nó de soma de inteiro com real; etc. O número de campos de informações é variável, sendo o comprimento - como visto - dimensionado pelo máximo, podendo haver sobras.

Todo nó terá ainda dois campos de apontadores: LINK1 e LINK2. Estes são os campos que permitem o percurso da árvore

de forma eficiente. O primeira apontador (LINK1) aponta sempre para o nó filho, valendo zero se este não existir. O segundo apontador (LINK2) aponta para o nó irmão se positivo e é alinhavo se negativo.

II.III. QUANTO À ESTRUTURA DO PROGRAMA INTERPRETADOR

O programa interpretador deverá dispor de duas procedures, a que denominamos SUBINDO e DESCENDO. Estas procedures estão esquematizadas na figura (II.5).

```
PROCEDURE SUBINDO (NO, ENDEREÇO);

BEGIN
CASE  NO [ENDEREÇO].OP  OF
1:
2:
     ≷
END  CASE
END;
```

```
PROCEDURE DESCENDO (NO, ENDEREÇO);

BEGIN
CASE NO [ENDEREÇO].OP OF
1:
2:
     ≷
END  CASE
END;
```

Figura - II.5

Uma rotina inicial do interpretador chama a procedure DESCENDO, após ter atribuído à variável ENDEREÇO o endereço do primeiro nó. Durante o processo de interpretação, em que ora se sobe, ora se desce nos diversos nós da árvore, vão sendo chamadas aquelas duas procedures. Na execução de qualquer delas, desvia-se para um dos parágrafos do respectivo CASE, em função do código do nó, aqui tratado por NO[ENDEREÇO].OP.

Observe-se que estamos aqui imaginando a árvore simulada num array de records, de nome NO, sendo OP uma componente. Outras serão os campos de informações e os de apontadores, LINK1 e LINK2. Naturalmente, se a linguagem de implementação não dispuser dos recursos aqui mencionados, a solução deverá ser adotada considerando tais restrições (exemplo: a linguagem pode não dispor do comando CASE ou não admitir estruturas tipo array de records).

II.IV. QUANTO À TABELA DE SIMBOLOS

A TS se compõe, basicamente, de três partes, como esquematiza a figura (II.6).

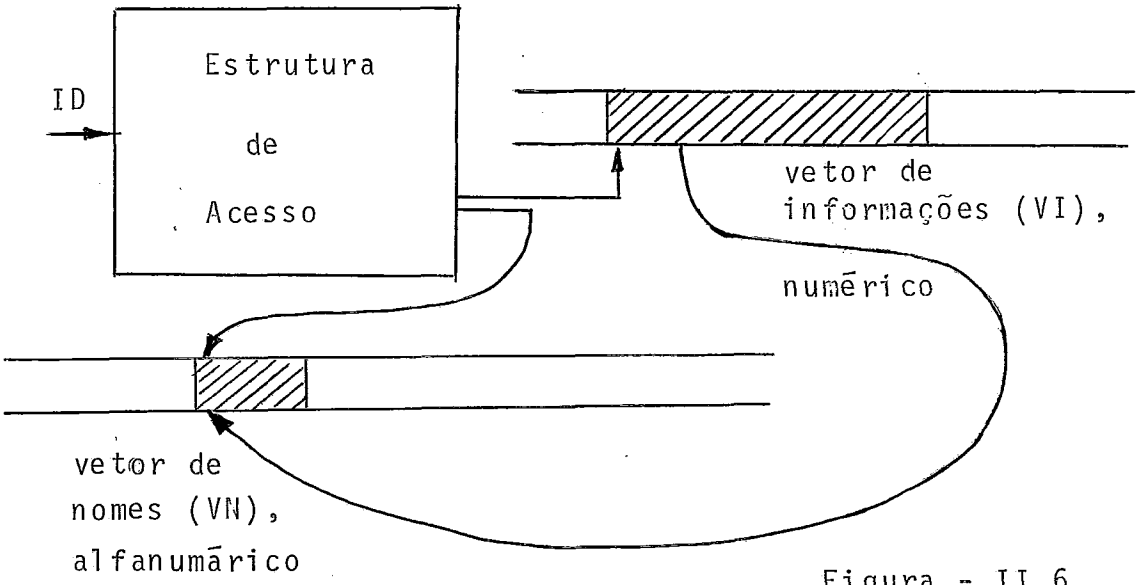


Figura - II.6

A estrutura de acesso interessa apenas ao módulo 1 do projeto. É através dela que, dado um identificador, chega-se ao seu bloco de informações no vetor VI ou ao seu nome no vetor VN. Ao interpretador interessam apenas os vetores VI e VN. Da figura (II.6) verifica-se que nomes de identificadores são duplamente apontados: direto da estrutura de acesso para que, na compilação, obtido um identificador e aplicada a função hash, se verifique, imediatamente, se o endereço gerado corresponde a ele, ou se ocorreu colisão; de um campo do bloco de informações do identificador para que o módulo de depuração do interpretador tenha acesso aos nomes dos objetos. Já foi dito que nó de variável aponta para a TS. Este ponteiro é o endereço do início do bloco de informações do identificador da variável em VI.

II.V. QUANTO A CONSTANTES

O módulo 1 do projeto (analisador/tradutor para FIP) deverá criar uma tabela de constantes. Cada entrada nesta tabela consistirá no descritor da constante seguido de seu valor.

Nós de constantes apontarão para aquela tabela. A figura (II.7) ilustra um trecho da tabela de constantes com duas entradas apontadas de dois nós da FIP.

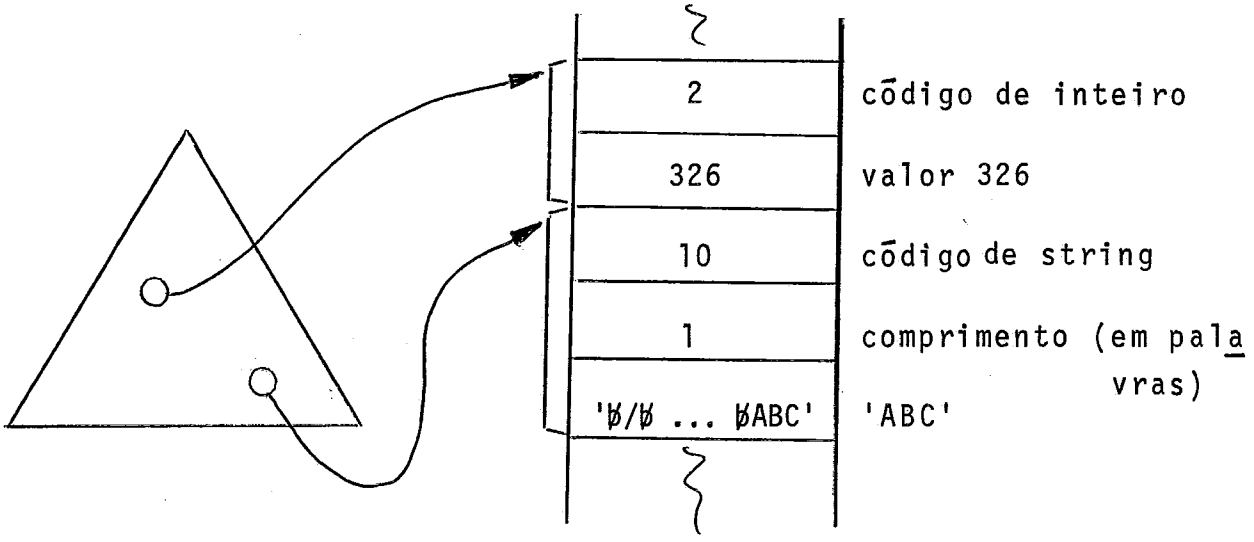


Figura - II.7

O primeiro exemplo daquela figura corresponde à constante tipo inteiro, valor 326. O segundo exemplo corresponde à constante tipo string, valor 'ABC'. O string é armazenado em forma compactada (packed).

Devido à heterogeneidade dos dados, caso a linguagem de implementação imponha uma tabela homogênea (por exemplo, usando um vetor), então valores reais, caráter e string serão substituídos por ponteiros para outras áreas. Poderão também ser criadas tabelas por tipo.

II.VI. QUANTO À ÁREA DE DADOS

O Pascal admite dados estáticos e dinâmicos. Para os primeiros, será reservada área nos chamados Registros de Ativação (RAs), havendo um para o programa principal e um por ativação de procedimento/função. Para os segundos, a área reservada chama-se Heap.

Além disto,o próprio interpretador vai requerer uma área de trabalho para uma série de variáveis, tabelas, buffers,

etc.

Assim, sem preocupação com a realização física, dependente da máquina em que o projeto for implementado, a idéia geral da distribuição de dados na memória é a esquematizada na figura (II.8).

Quanto aos RAs, serão eles gerenciados através de um mecanismo de pilha, a pilha de RAs (PRA). Assim, cada vez que se chama um procedimento ou se referencia uma função, o RA correspondente é carregado para conter os dados não dinâmicos locais àquela ativação de procedimento ou função. Dados não dinâmicos (incluídas variáveis tipo pointer, que permitem adentrar estruturas do Heap) têm seus endereços na forma [número do RA em que foi declarado, deslocamento no RA] armazenados nos blocos de informações correspondentes a seus identificadores no vetor VI da tabela de símbolos.

A figura (II.9) apresenta um esquema alternativo ao da figura (II.8), certamente mais conveniente, por reduzir o risco de "estouro" na PRA ou no Heap. Este é o esquema que deverá ser implementado. As discussões deste trabalho, entretanto, baseiam-se no esquema da figura (II.8) por nos parecer mais simples a manipulação de endereços crescentes.

Cumpre ressaltar que ambos os esquemas dizem respeito apenas ao interpretador. Se o usuário solicitar recursos de depuração, será convocado o interpretador/depurador, que consome mais memória para o código e cujo esquema da alocação de dados inclui estruturas específicas.

Finalmente, cabe,para a PRA,observação análoga à feita para a tabela de constantes,quanto à heterogeneidade dos dados.

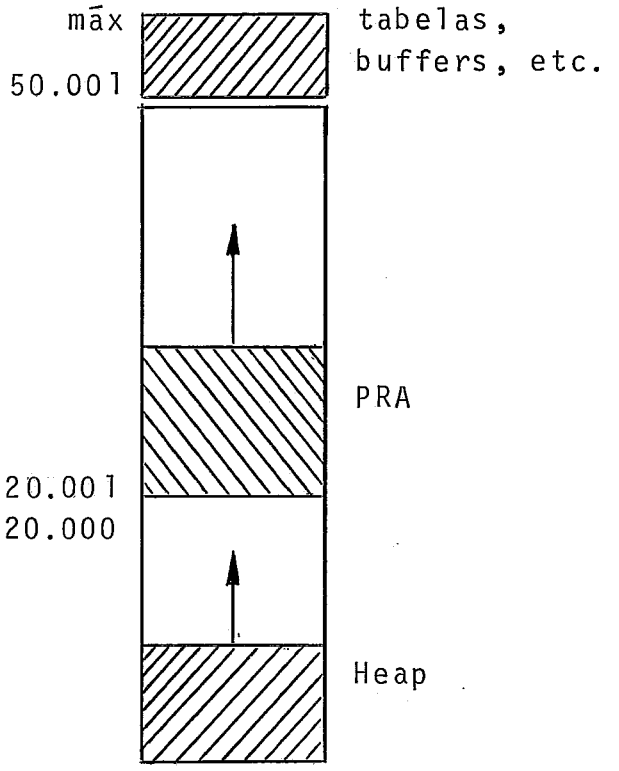


Figura - II.8

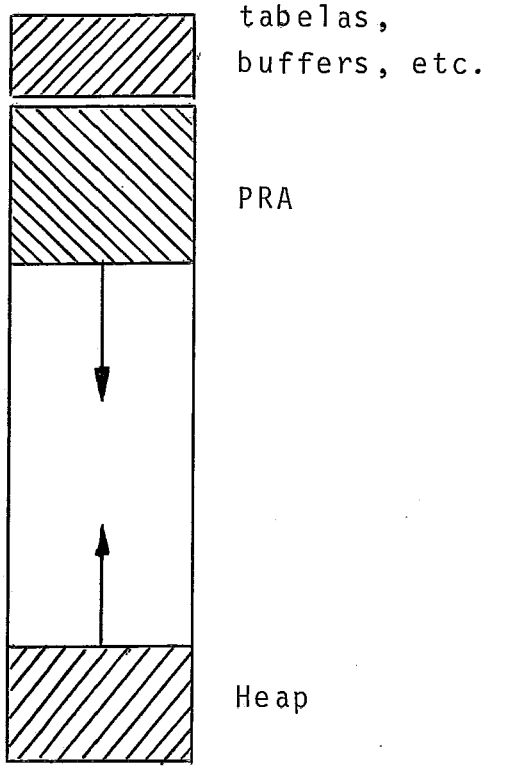


Figura - II.9

II.VII. QUANTO À GERÊNCIA DA PILHA DE RAs

Seja o trecho de programa em PASCAL esquematizado na figura (II.10).

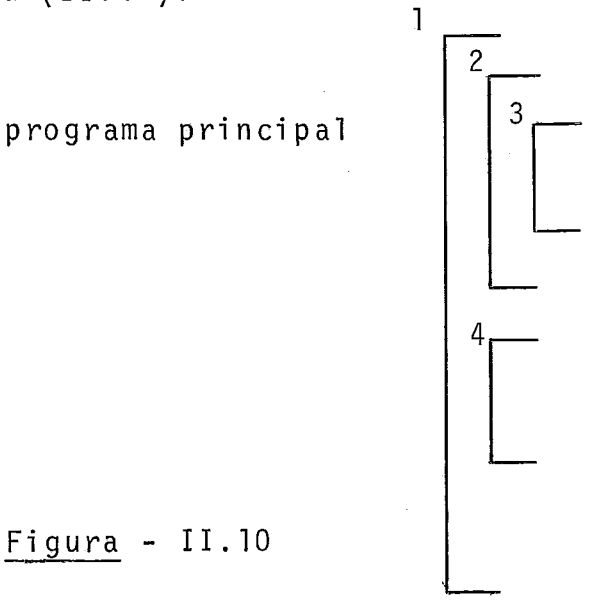


Figura - II.10

O módulo 1 do projeto deverá atribuir aos diversos Registros de Ativação uma numeração crescente, à medida que for encontrando as declarações dos vários módulos que compõem o programa, como mostra aquela figura.

Suponhamos, agora, que a tempo de execução ocorram chamadas na seguinte ordem: 1 → 2 → 2 → 3. Isto implicará no empilhamento de RAs mostrado na figura (II.11).

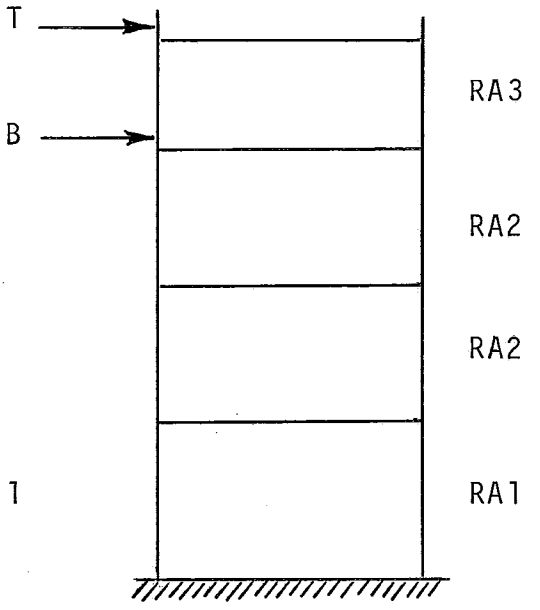


Figura - II.11

Cada RA conterá, como visto, os dados não dinâmicos locais àquela ativação de procedimento/função. Assim, existem duas versões do RA do procedimento 2 porque este foi chamado duas vezes, a segunda de forma recursiva.

Surge agora uma questão: o que fazer no retorno do procedimento 3? Naturalmente, o ponteiro T, de topo da pilha de RAs, deverá ser baixado, o mesmo ocorrendo com o ponteiro B, de base do último RA.

Daí nasce a necessidade da chamada cadeia dinâmica,que

é uma cadeia ligando os RAs na ordem inversa de chamada. Assim, facilmente se descobre os novos valores dos ponteiros: T receberá o valor de B e B o próprio valor do ponteiro da cadeia dinâmica que se acha no RA do procedimento em conclusão. A figura (II.12) ilustra tal cadeia. Daquela figura pode-se observar que, por RA, além da entrada do ponteiro da cadeia dinâmica, existe outra entrada com o número do mesmo. A razão desta segunda entrada será adiante explicada.

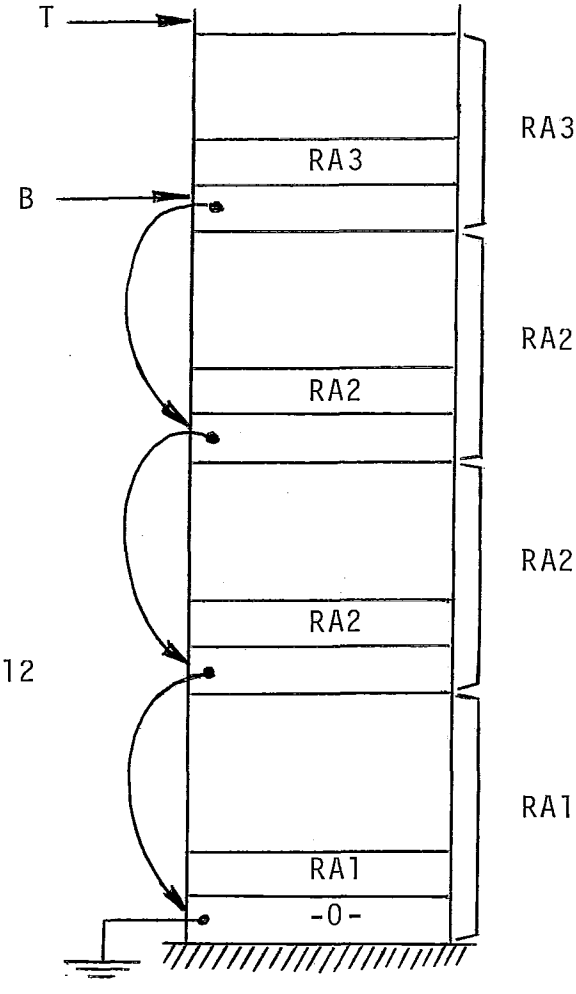
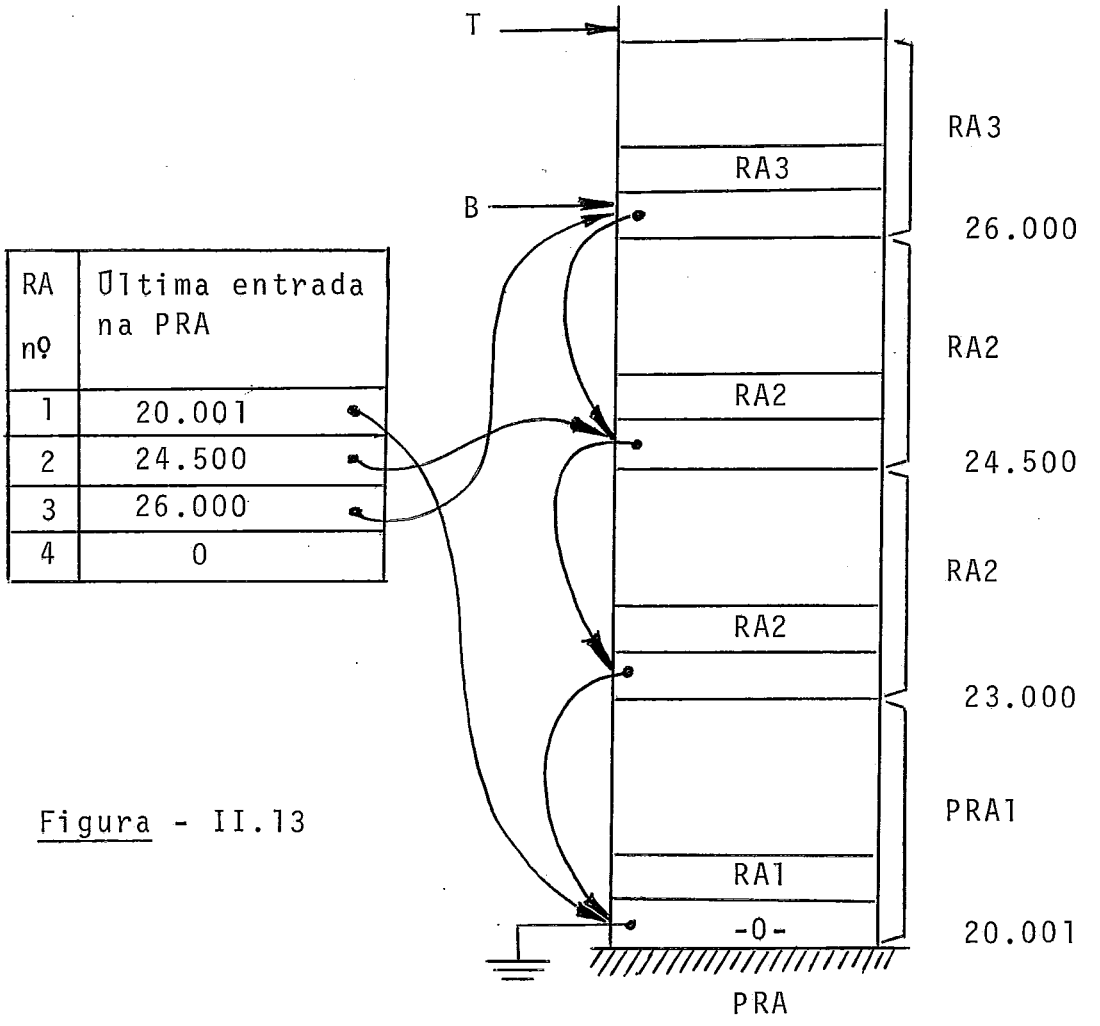


Figura - II.12

Por outro lado, a linguagem PASCAL admite a referência a objetos não-locais. Assim, do procedimento 3 da figura (II.10) pode-se fazer referência a variáveis declaradas no procedimento 2 ou no programa principal, além, é claro, de se po

der referenciar variáveis locais. Por esta razão, o esquema de figura (II.12) precisa ser complementado plea introdução da tabela DISPLAY, apresentada na figura (II.13).



RA nº	Última entrada na PRA
1	20.001
2	24.500
3	26.000
4	0

Figura - II.13

Com a tabela DISPLAY, referências a variáveis locais ou não ficam prontamente resolvidas. É sabido que o endereço de uma variável é da forma [número do RA em que foi declarada, deslocamento]. Assim, na referência a uma variável, vai-se à tabela de símbolos, de onde se busca seu endereço naquela forma; com base no número do RA consulta-se a tabela DISPLAY e descobre-se o endereço na PRA do início do RA em questão; soma-se o endereço

com o deslocamento obtendo-se a localização da variável na PRA. Chamaremos a este procedimento, sumariamente, de "busca o endereço da variável, linearizando-o na PRA".

A tabela DISPLAY é construida dinâmicamente, a tempo de interpretação, sendo alterada a cada entrada em ou saída de procedimento/função: na entrada, simplesmente faz-se a tabela apontar para o início do RA que se está alocando; na saída é preciso consultar a cadeia dinâmica para ver se existe outra versão de RA com aquele número já empilhado e, se existe, faz-se a tabela apontar para o início do mesmo, senão zera-se a entrada (daí a necessidade do número dos RAs na pilha).

CAPÍTULO III

III. DADOS DE TIPO SIMPLES E EXPRESSÕES

Um objetivo desde capítulo é explicar que solução estamos propondo para os diversos tipos de dados simples mantidos pelo PASCAL, com exceção do pointer. Serão abordados o inteiro, o real, o booleano, o caráter, o escalar declarado, o set e o subrange. O pointer, por sua característica particular será exposto em capítulo próprio. Ao apresentarmos os dados de tipo simples somos quase que inevitavelmente conduzidos a uma discussão paralela em torno de expressões.

Durante este trabalho, sempre que estivermos descrevendo ações ao longo do percurso da árvore - FIP (daqui por diante tratada apenas de FIP), associadas a um determinado nó, usaremos a seguinte convenção:

Di - descrição

Di significa que estamos descendo no nó de número i. Descer, no caso, significa que viemos de nó pai ou irmão.

Ver figura (III.1).

Si - descrição

Si significa que estamos subindo no nó de número i. Subir, no caso, significa que viemos de outro nó via alinhavo

Ver figura (III.1).

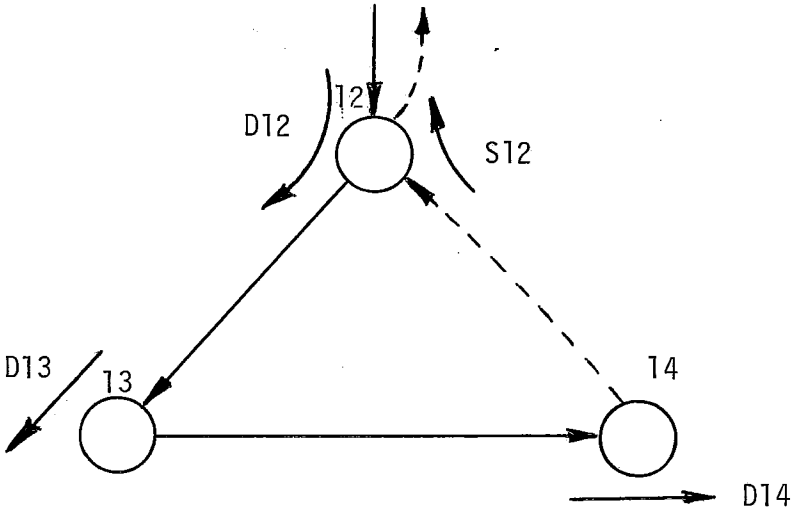


Figura - III.1

Ao numerarmos os nós nos diversos exemplos deste trabalho, não estaremos preocupados com o endereçamento real que deverão receber do módulo 1 do projeto. Estaremos apenas criando uma referência para a descrição associada.

III.I. O TIPO INTEIRO

Será usada a representação própria da máquina para inteiros. Nada a acrescentar.

III.II. O TIPO REAL

Será usada - se existir - a representação própria da máquina para reais. Caso contrário, serão requeridas rotinas que simulem o tipo real na máquina alvo.

III.III. O TIPO BOOLEANO

Assumiremos, para efeito das discussões neste trabalho, a representação em que a cada variável booleana é associada uma palavra de memória sendo TRUE representado pelo valor 1 e FALSE pelo valor 0.

III.IV. O TIPO CARÁTER

Variáveis e constantes do tipo caráter serão representadas em ASCII numa palavra do computador. Tais caracteres ocuparão um byte, contendo brancos os demais bytes.

Aos caracteres, obedecendo à mesma ordem da tabela ASCII correspondente, existirão os chamados números de ordem. Assim, por exemplo, numa determinada máquina, à letra A poderá corresponder o número de ordem 65; à letra B, 66; etc.

Funções como a ORD e a CHR exigirão procedimentos de conversão, respectivamente, da forma de caráter para a de número de ordem e vice-versa.

Por outro lado, esta representação não exigirá conversão na entrada e na saída.

III.V. O TIPO ESCALAR DECLARADO

Existem em PASCAL duas categorias de escalares: os implícitos e os declarados. São escalares implícitos o inteiro, o real, o booleano e o caráter, já tratados. Quanto aos escalares declarados, serão associados aos seus valores números inteiros de Ø a N. Quaisquer operações envolvendo escalares envolverão, em última análise, tais números.

Por exemplo:

"FOR X: = SEGUNDA TO SEXTA ..."

Se o escalar foi declarado como

"VAR X, DIAS: (SEGUNDA, TERÇA, QUARTA, QUINTA, SEXTA, SABADO, DOMINGO);"

então o módulo 1 do projeto terá feito Ø corresponder a SEGUNDA e 4 corresponder a SEXTA. Serão executadas 5 iterações no FOR, com X valendo, internamente, respectivamente, Ø, 1, 2, 3 e 4.

Ainda com relação ao exemplo visto, supondo uma expressão relacional, no programa, do tipo X > DIAS, tendo X o valor TERÇA e DIAS o valor SEXTA, a comparação se dará entre os números 1 e 4 e o resultado da expressão (resultado booleano) será Ø (FALSE).

Cumpre ressaltar, ainda, que escalares declarados não podem ser lidos ou gravados/impressos, não sendo pois requerida conversão a tempo de interpretação. A única conversão necessária é de natureza estática: é feita pelo módulo 1 do projeto que associa a cada valor escalar declarado um número inteiro.

III.VI. O TIPO SET

Os valores de variáveis ou constantes do tipo SET serão armazenados numa palavra de memória (ou mais, se quisermos maior capacidade) da máquina alvo. Assim, supondo um computador cuja palavra de memória tenha 16 bits, serão admitidos conjuntos de até 16 elementos.

O tipo SET é sempre de escalar, declarado ou não. Cada bit da palavra correspondente ao conjunto se relaciona com um elemento escalar. Seja a seguinte sequência de declarações:

```
"TYPE  VOG = (A, E, I, O, U);
       VOGS = SET OF VOG;
 VAR   VOGAIS: VOGS;"
```

Supondo que num determinado ponto do programa o valor-conjunto VOGAIS seja [A, I, O, U], a representação interna na palavra associada à variável VOGAIS seria ØØØØØØØØØØ11101 numa palavra de 16 bits (bit Ø à direita, bit 15 à esquerda).

Valores tipo SET não podem também ser lidos ou gravados/impressos.

III.VII. O TIPO SUBRANGE

Subranges serão sempre de escalares, declarados ou não, excetuado o real. Assim, a representação interna de valores subrange recairá sempre em casos já discutidos.

III.VIII. EXPRESSÕES ARITMÉTICAS

A figura (III.2) apresenta o exemplo de uma expressão aritmética: (-D) * (A + B + C). Neste exemplo, as variáveis D,

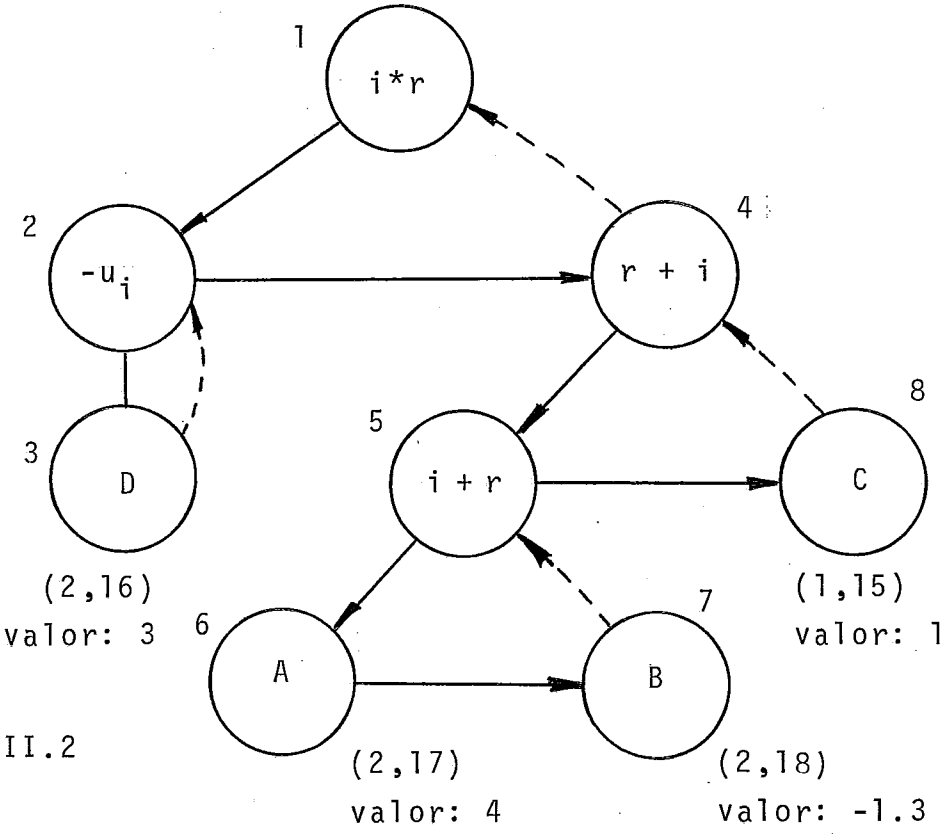


Figura - III.2

A e C são inteiras e a variável B é real. Os nós 3,6,7 e 8 são de variáveis. Como visto, eles endereçam a tabela de símbolos de onde se busca o endereço na forma [RA, d]. Junto de cada um daqueles nós foi escrito o suposto endereço das diversas variáveis. Suporemos que a ordem de chamadas tenha sido 1 → 2 onde 1 corresponde ao programa principal e 2 ao procedimento chamado. As variáveis A, B e D são locais a este procedimento. A variável C foi declarada no programa principal, sendo, pois, não local ao procedimento 2. Acompanhemos, agora, a sequência de ações no percurso da subárvore mostrada na figura (III.2). Na

quele exemplo, T, ponteiro de topo da PRA é suposto inicialmente igual a 23201.

D1 - Reserva espaço para a temporária que conterá o resultado incrementando a variável T (ponteiro de topo da PRA).

D2 - Análogo à ação D1.

D3 - Busca o endereço da variável linearizando-o na PRA. Empilha, na PRA, o conteúdo deste endereço.

Ver figura (III.3).

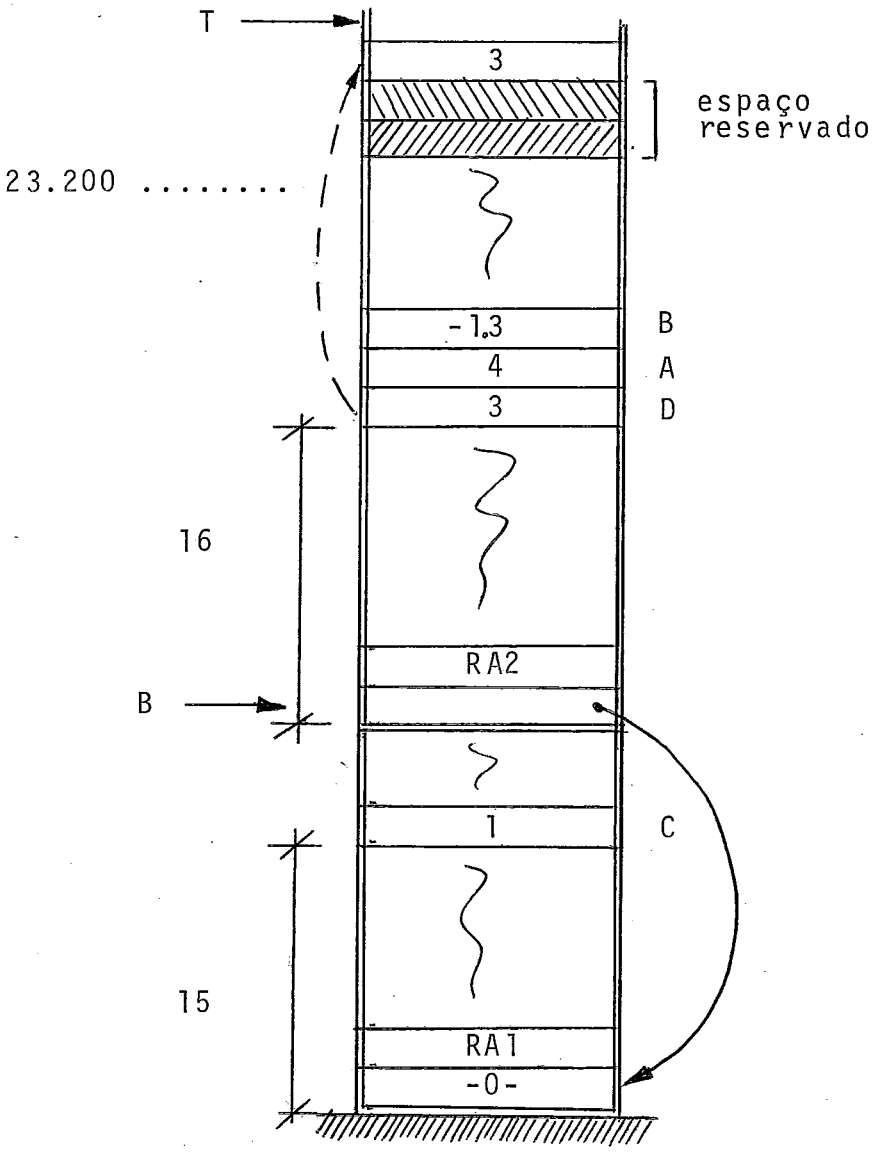


Figura - III.3

S2 - Opera, colocando em (T-2) da PRA o valor do conteúdo de (T-1) da PRA com sinal trocado. Após, decrementa T. Ver figura (III.4).

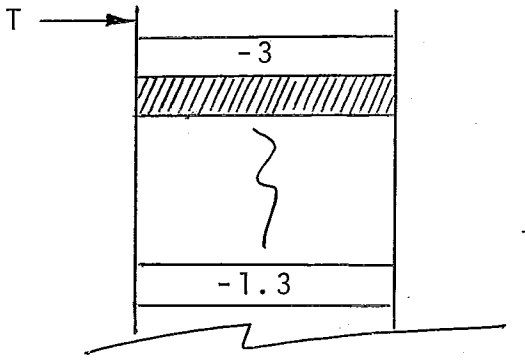


Figura - III.4

D4 - Análoga a D1.

D5 - Análoga a D1.

D6 - Análoga a D3.

D7 - Análoga a D3.

Ver figura (III.5).

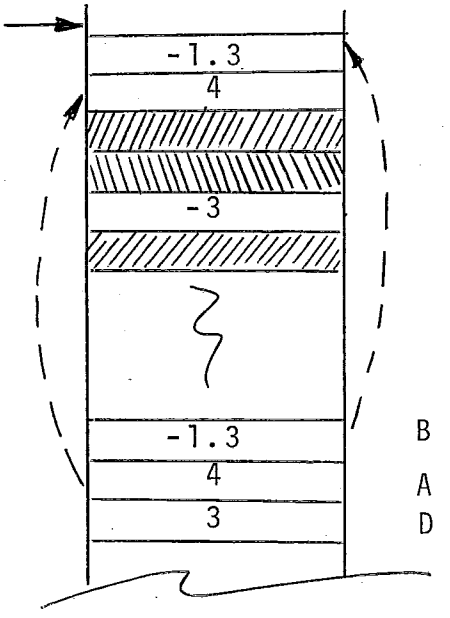


Figura - III.5

S5 - Opera, convertendo o valor em (T - 2) para a forma real, e empilha o valor convertido na PRA.

Ver figura (III.6).

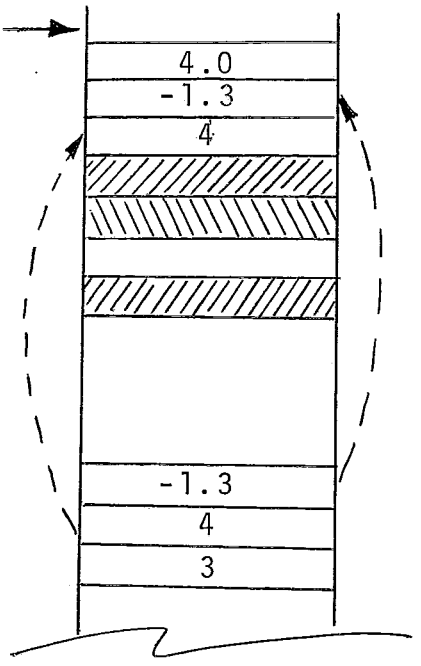


Figura - III.6

Opera, somando os dois valores reais em (T-1) e (T-2) da PRA, colocando o resultado em (T-4) da mesma pilha. Após, decrementa T.

Ver figura (III.7).

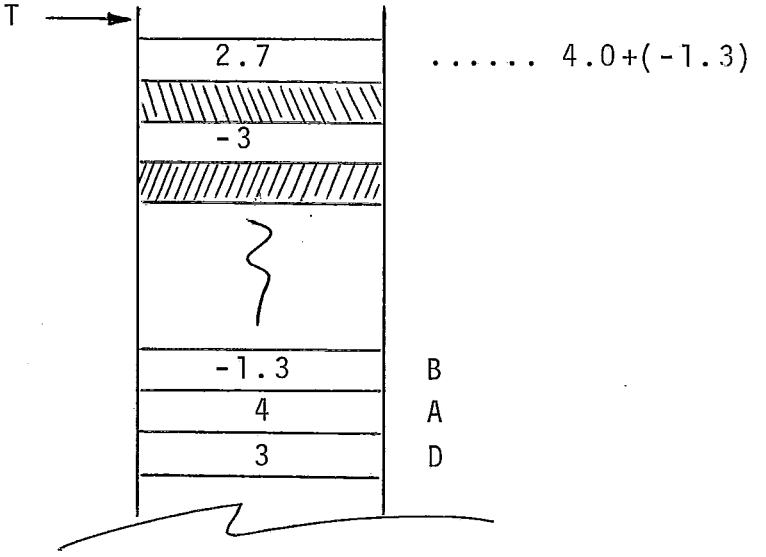


Figura - III.7

D8 - Análogo a D3.

Ver figura (III.8).

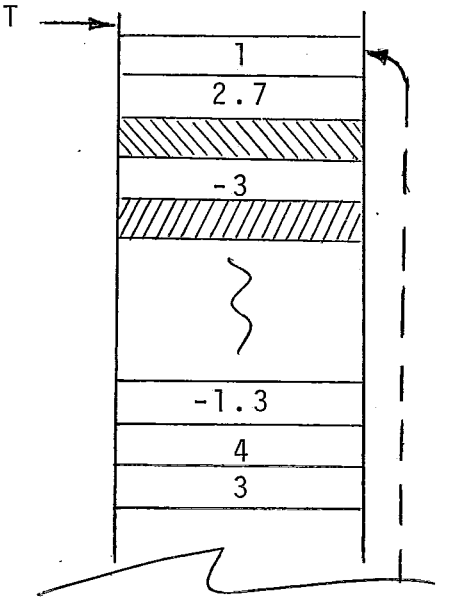


Figura - III.8

S4 - Opera, convertendo o valor em (T-1) para a forma real, e empilha o valor convertido na PRA.

Ver figura (III.9).

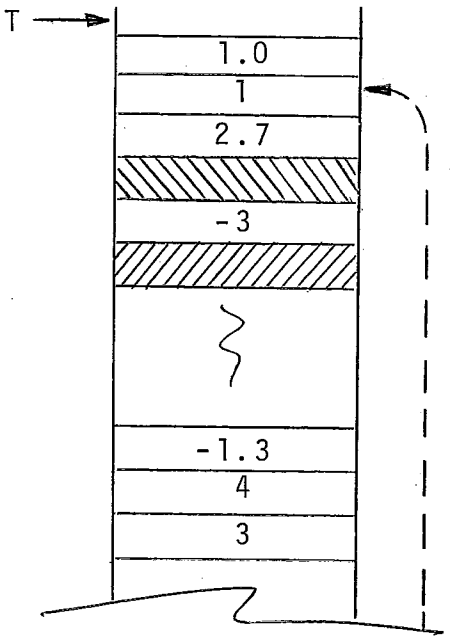


Figura - III.9

Opera, somando os dois valores reais em (T-1) e (T-3) da PRA, colocando o resultado em (T-4) da mesma pilha. Após, decrementa (de 3) T.
Ver figura (III.10).

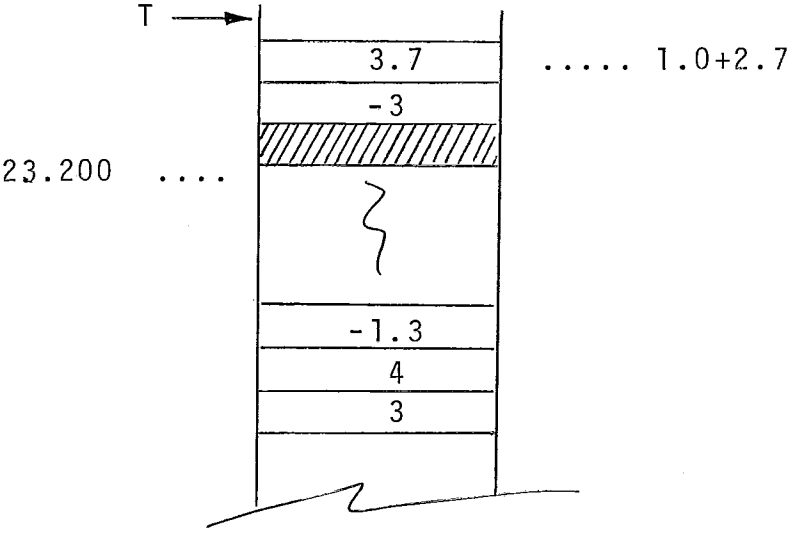


Figura - III.10

S1 - Análogo a S5, substituindo-se á operação de soma por produto.

Ver figura (III.11).

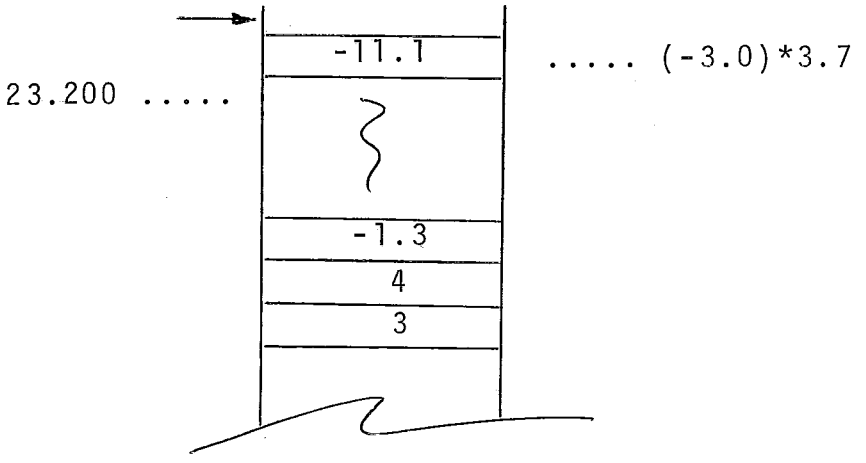


Figura - III.11

Da forma como estamos resolvendo expressões aritméticas, seu valor ficará invariavelmente em (T-1) da pilha de Registros de Ativação - PRA. Isto ocorrerá inclusive nos casos particulares, em que a expressão aritmética se reduza a uma simples referência a variável ou constante. No primeiro caso, ao descer no nó da variável, seu valor - como já foi visto - será transferido de seu endereço na PRA para o endereço T, sendo T incrementado. No segundo caso, ao descer no nó de constante, esta será transferida da tabela de constantes (TC) para o endereço T, sendo T incrementado. Esta uniformização no tratamento de expressões aritméticas é importante para comandos como atribuição ou chamada de procedimento a serem tratados adiante.

Cumpre, neste ponto, formularmos uma observação. É

possível que, na máquina alvo, valores inteiros requeiram, por exemplo, uma palavra e valores reais duas. Assim, os espaços reservados não serão necessariamente constantes.

III.IX. EXPRESSÕES RELACIONAIS

A figura (III.12) apresenta a FIP para um exemplo de expressão relacional.

Os procedimentos no percurso da árvore de uma expressão relacional (que inclusive envolve expressões aritméticas) são absolutamente análogos aos do percurso da árvore de uma expressão aritmética. Por esta razão, não julgamos necessário detalhar o percurso da árvore exemplo. O resultado da operação relacional será booleano, e, portanto, Ø ou 1. Na verdade a expressão relacional nada mais é que um caso particular de expressão booleana. Expressões relacionais não envolvem necessariamente inteiros e/ou reais. Podemos ter operações entre escalares declarados, caracteres ou subranges. A figura (III.13) ilustra o exemplo de uma expressão relacional entre dois caracteres.

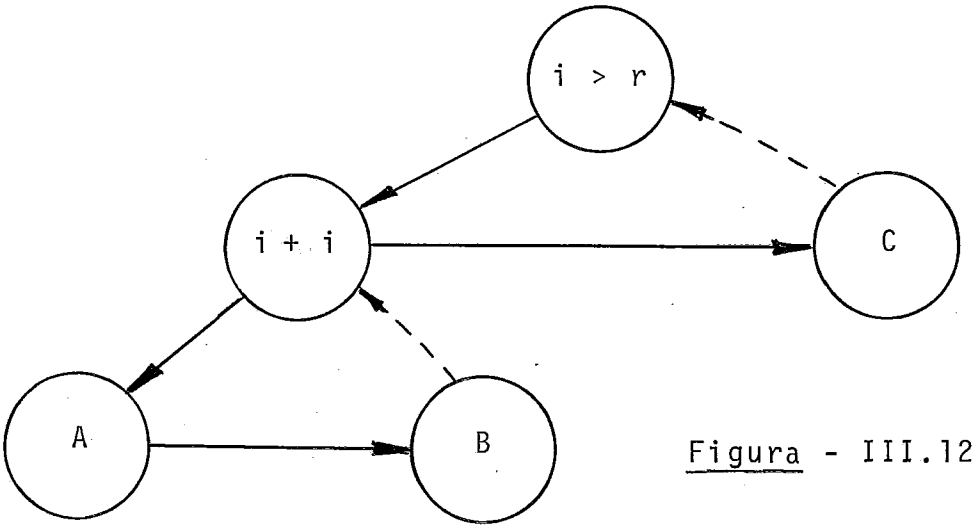


Figura - III.12

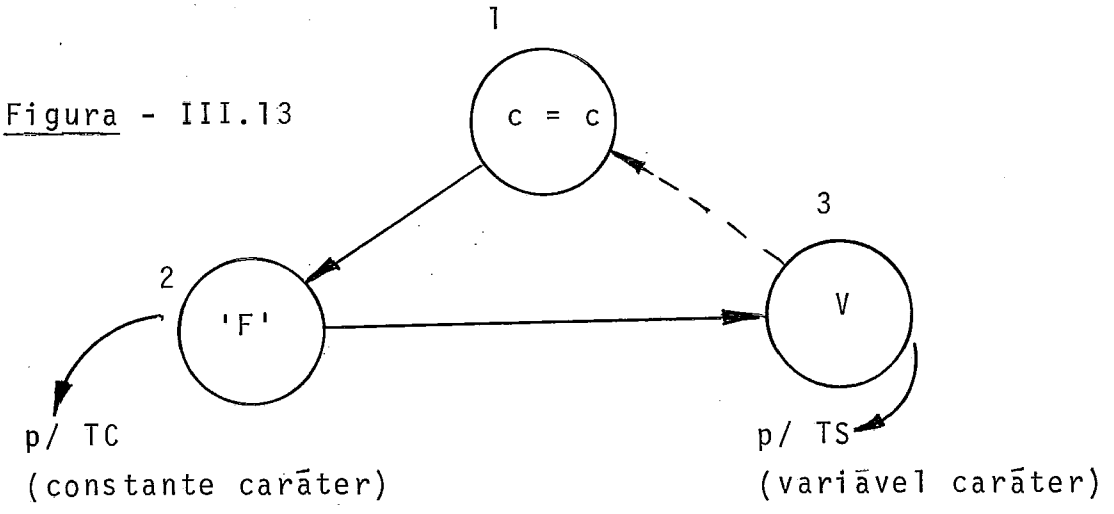


Figura - III.13

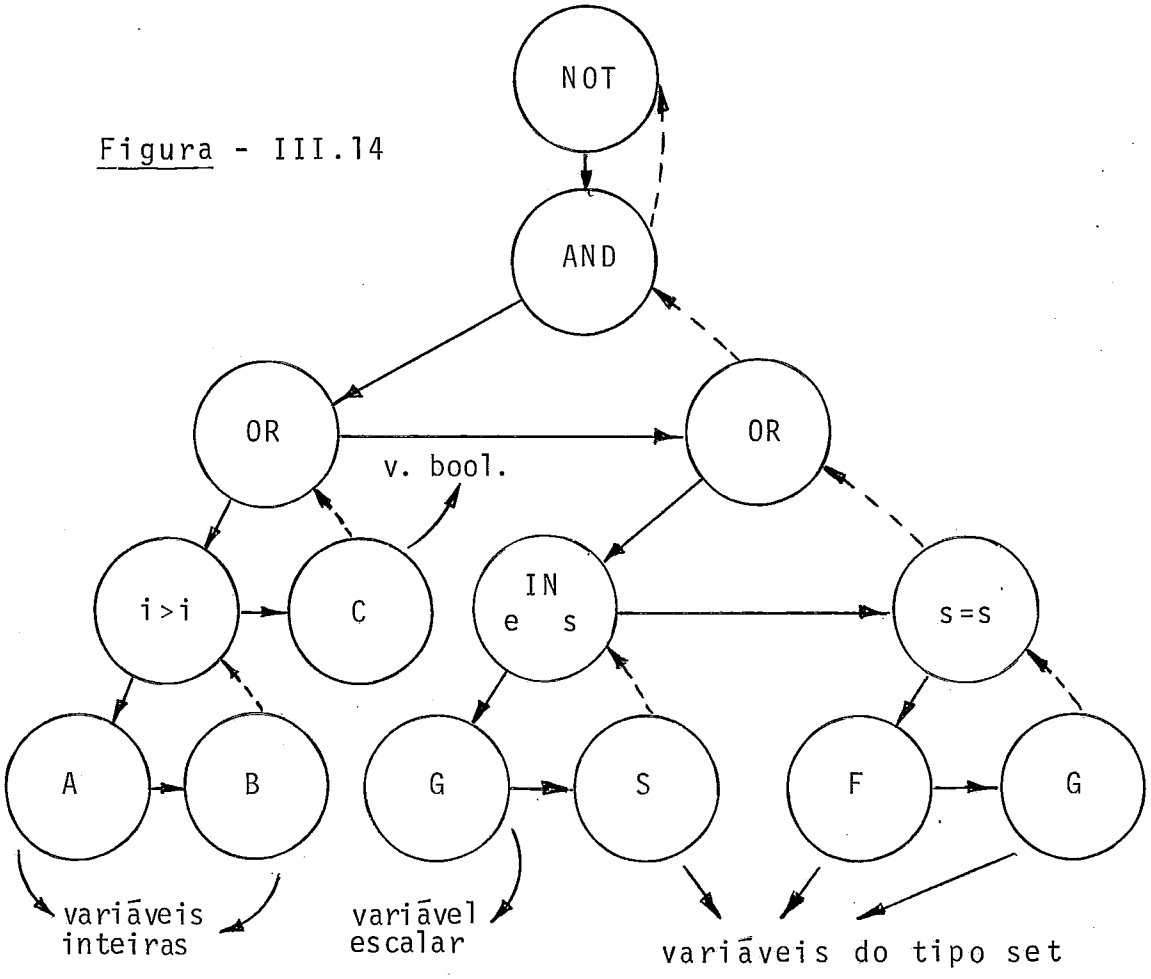
III.X. EXPRESSÕES BOOLEANAS

Discutimos, no tópico anterior, as expressões relacionais que, como dito, são um caso particular de expressões booleanas. As expressões booleanas mais simples são a própria expressão relacional, a variável booleana e a constante booleana. Outra classe de expressões booleanas simples envolve relações entre variáveis e/ou constantes dos tipos escalar e set ou exclusivamente do tipo set. São elas a de pertinencia entre escalar e set (e INs), a de igualdade entre sets (s = s), a de

desigualdade entre sets (s < > s) e as de inclusão entre sets (s > = s ou "contém" e s < = s ou "está contido"). Todas estas relações, como as expressões relacionais vistas, dão resultado booleano. Finalmente, as expressões booleanas mais gerais são formadas pela aplicação, às expressões booleanas mais simples, antes descritas, dos operadores AND, OR e NOT.

A figura (III.14) corresponde a um exemplo bastante geral de expressão booleana no qual aparecem uma expressão relacional, a referência a uma variável booleana, uma relação de pertinência entre escalar e set e uma relação de igualdade entre sets.

Figura - III.14



As operações de pertinência e de igualdade ali destacadas, bem como as demais já relacionadas para sets serão realizadas, na interpretação, através de operação primitiva da máquina, através de operações análogas na linguagem de implementação ou, em último caso, através de um procedimento de simulação. Quanto às operações NOT, AND e OR, cumpre ressaltar que a primeira é unária e as duas outras são binárias. Feitas tais observações, não julgamos necessária uma discussão mais aprofundada de expressões booleanas: as ações no percurso de uma árvore de expressão booleana são parecidas com as ações no percurso de uma árvore da expressão aritmética, ressaltada a evidente diferença quanto à natureza das operações.

III.XI. EXPRESSÕES SET

Sob esta denominação estamos englobando expressões que envolvem operações entre variáveis ou constantes set dando por resultado outro set. Tais operações são a união de sets (s + s), a interseção entre sets (s * s) e a diferença entre sets (s - s). Cabe aqui a observação anterior quanto às relações entre escalar e set e entre sets vista. A figura (III.15) corresponde a um exemplo de expressão set. As ações no percurso de uma árvore de expressão set são também análogas às dos percurso de uma árvore de expressão aritmética, com a mesma ressalva já feita quanto à evidente diferença na natureza das operações.

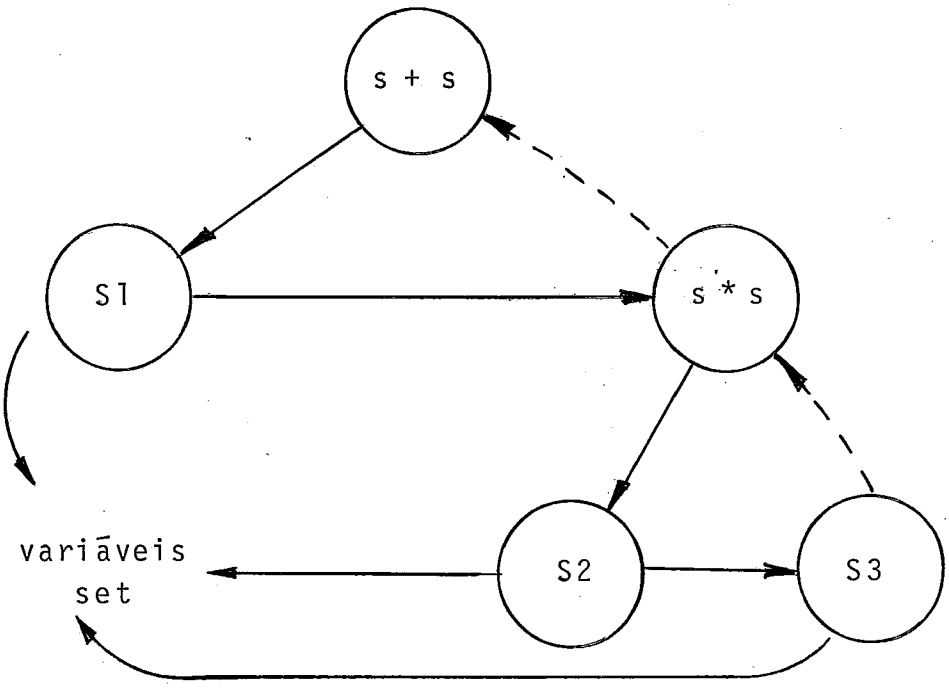


Figura - III.15

III.XII. FUNÇÕES E PROCEDIMENTOS PRÉ-DEFINIDOS

Uma série de funções e de procedimentos ditos pré-definidos devem estar disponíveis ao programador do Pascal. Várias funções, como o seno e a raiz quadrada poderão ser implementadas por chamadas a intrinsecos já disponíveis na máquina alvo. Outras, como por exemplo ORD e SUCC e procedures como READLN, específicas de Pascal, serão implementadas através de rotinas próprias.

Não julgamos importante, no corpo deste trabalho, desenvolver cada uma destas rotinas específicas. Até, porque, algumas delas, como a ORD e a CHR são óbvias. Já outras, como rotinas de entrada e saída e o procedimento NEW, para alocação de área dinâmica, pela importância, são descritas em capítulo próprio.

Neste capítulo, que trata particularmente de expressões, entendemos ser oportuno descrever, ainda que de forma geral, o que se passa no percurso da árvore de chamada de uma função, já que tais chamadas são parte de árvores de expressões.

Seja, pois, o exemplo da figura (III.16), em que estamos imaginando que a função será por nós escrita e, portanto, parte do interpretador.

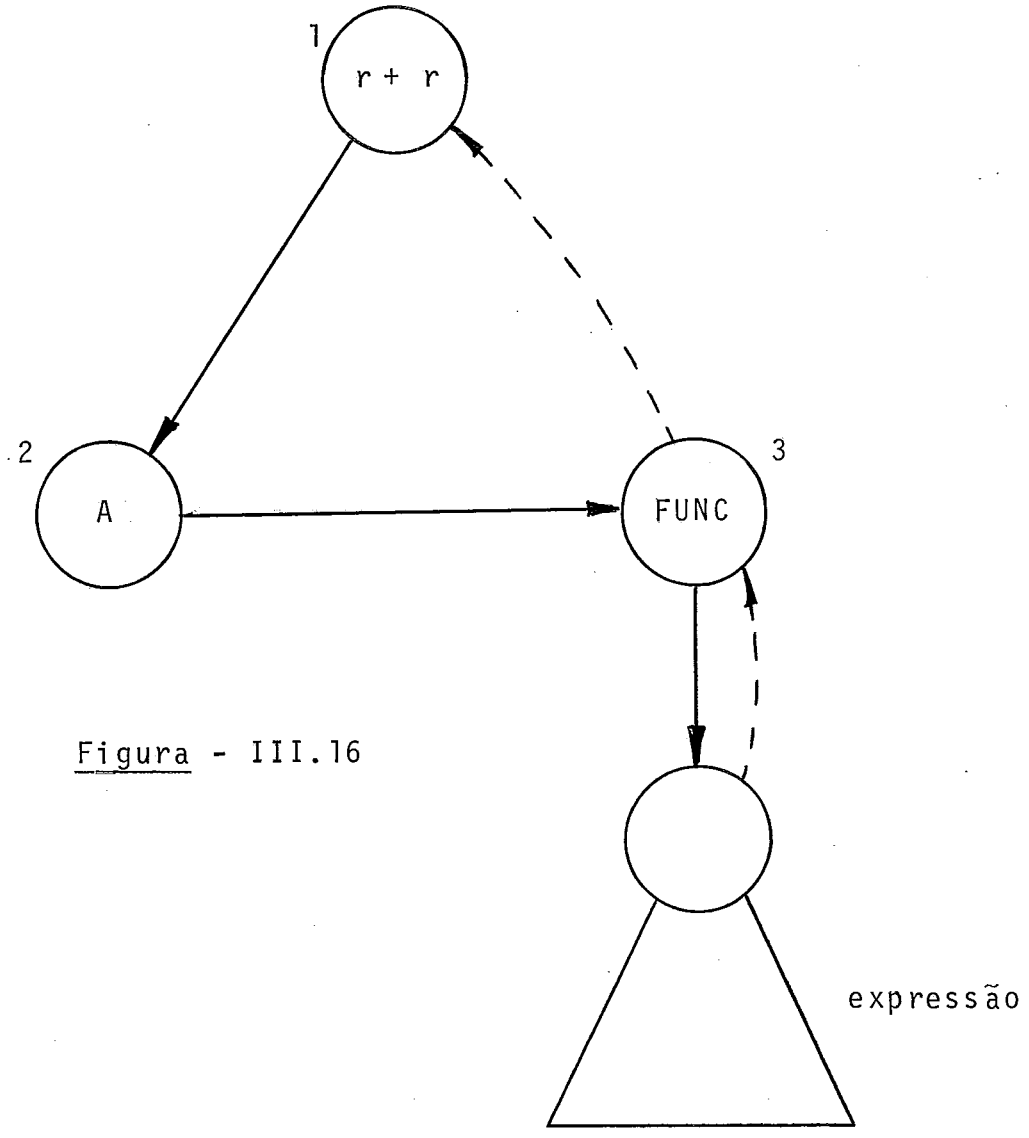


Figura - III.16

Ao descer no nó 3, nada é feito. Percorrida a árvore que tem por raiz o nó 4 resulta o valor da expressão logo a baixo do topo da PRA. Ao subir no nó 3, a rotina correspondente a FUNC, parte integrande do interpretador, é chamada. Esta busca o parâmetro abaixo do topo da PRA, avalia a função e substitui, na PRA, o parâmetro pelo valor calculado. Caso a função não fosse parte do interpretador, ao subir no nó 3, o interpretador buscaria o parâmetro abaixo do topo da PRA, chamaria o intrinseco correspondente, passando este. Recebido o resultado, o interpretador substituiria o parâmetro por ele, na PRA.

Quanto às procedures pré-definidas, por razão que será vista futuramente, em lugar de salvarmos os parâmetros para só depois executarmos o procedimento, iremos agindo à medida que passarmos pelos próprios parâmetros.

CAPÍTULO IV

IV. DADOS ESTRUTURADOS

Neste capítulo serão descritos o array, o record e suas combinações. Os casos abordados permitirão cobrir todas as possíveis combinações entre estes dois tipos.

IV.I. ARRAYS

A figura (IV.1) ilustra a árvore de uma referência a elemento de array: A [X + Y, Z]. O nó 2 será conhecido como nó de indexação: ao subir neste nó, no percurso da árvore, o

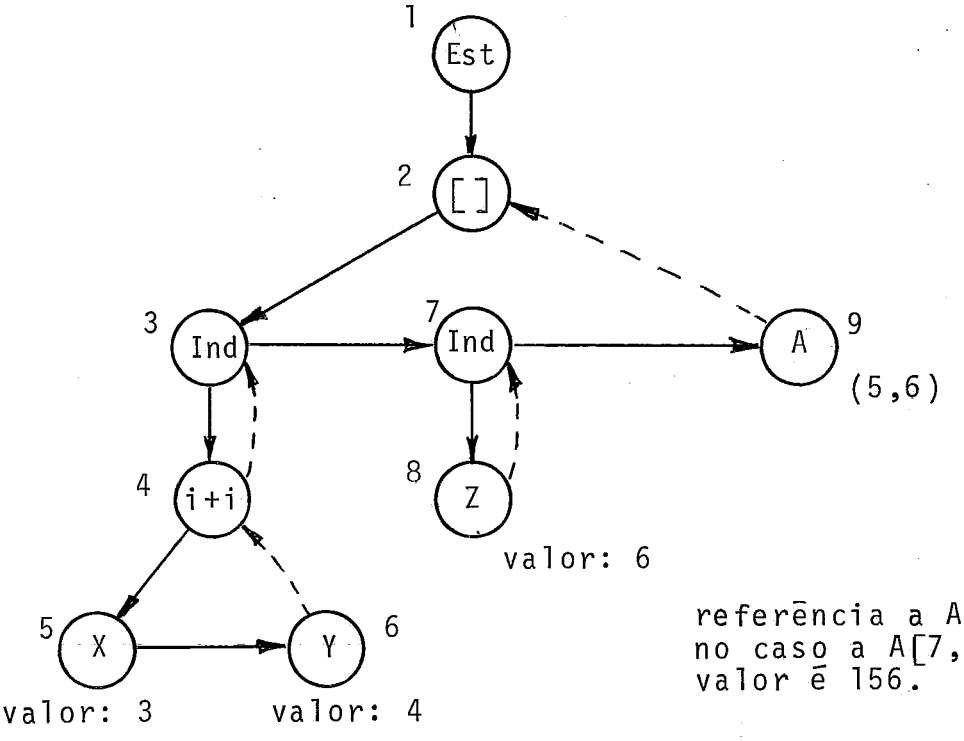


Figura - IV.1

endereço do elemento a referenciar é calculado; o nó 1 encabeça sempre referências a dados de origem estruturada; os nós 3 e 7 terão sua função entendida quando descrevermos as características de depuração do interpretador.

Vejamos, agora, as ações de interpretação ao longo do percurso da árvore-exemplo, considerando que esta referência não pertence a comando de depuração:

D1 - Ø (nenhuma ação)

D2 - Ø

D3 - Ø

D4, D5, D6, S4 - resulta a expressão avaliada, ficando o resultado imediatamente abaixo do topo da PRA.

S3 - Ø

D7 - Ø

D8 - resulta a expressão avaliada, etc.

S7 - Ø

D9 - como a variável é do tipo array (o tipo vem da TS): busca o endereço de A, linearizando-o na PRA. Empilha este endereço na PRA. Empilha, ainda na PRA, o endereço deste identificador A na TS (o endereço está no próprio nó). Armazena na variável auxiliar COMPRIMENTO o tamanho do elemento referenciado.

Ver figura (IV.2).

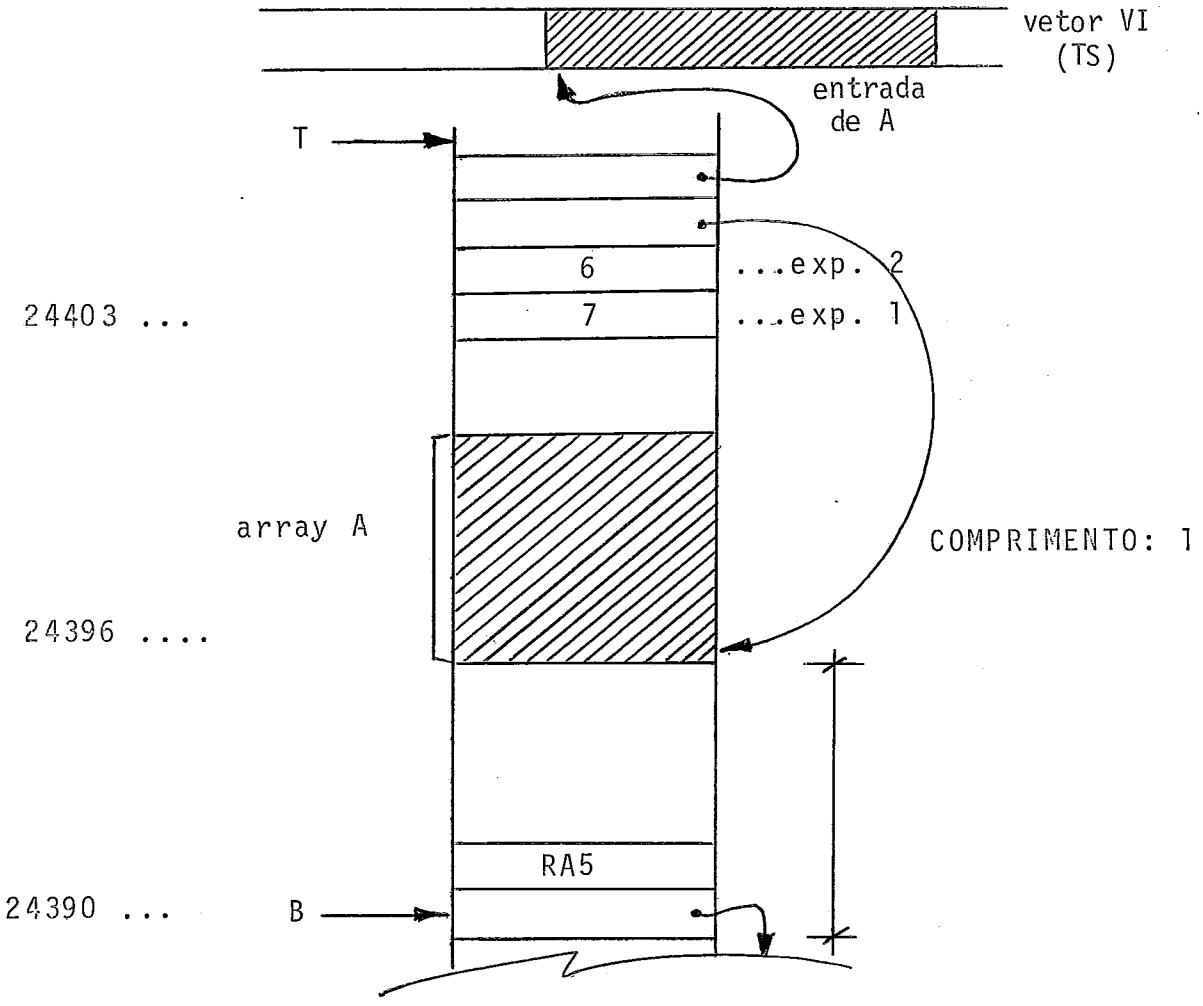


Figura - IV.2

S2 - Com base em informações da TS e em entradas armazenadas junto ao topo da PRA, calcula o endereço do elemento referenciado. Desempilha (n + 2) entradas - onde n é o número de índices - da PRA, empilhando, em seguida, na mesma, o endereço calculado.

Ver figura (IV.3).

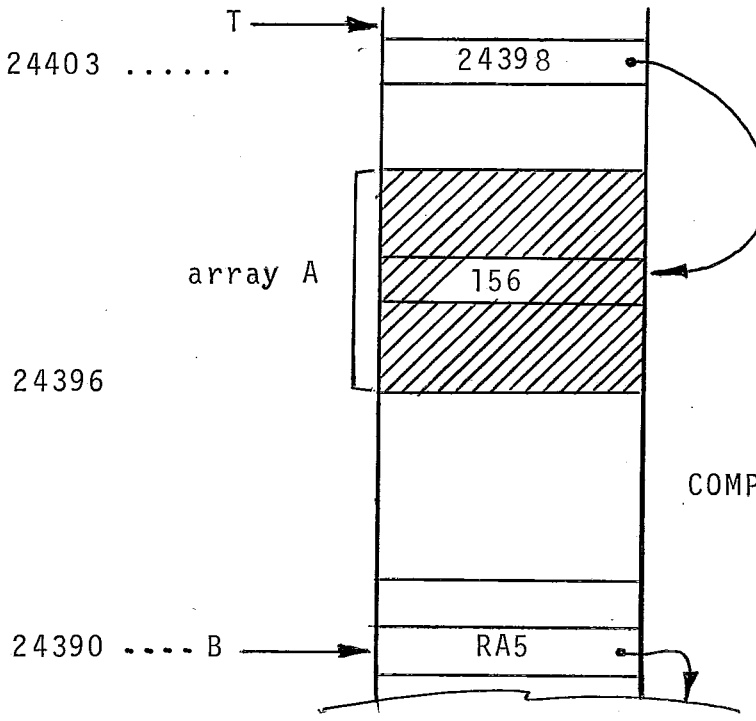


Figura - IV.3

S1 - Copia o elemento para a partir da entrada abaixo do topo da PRA (desempilha e empilha), tendo por base o endereço do elemento e seu tamanho.

Ver figura (IV.4).

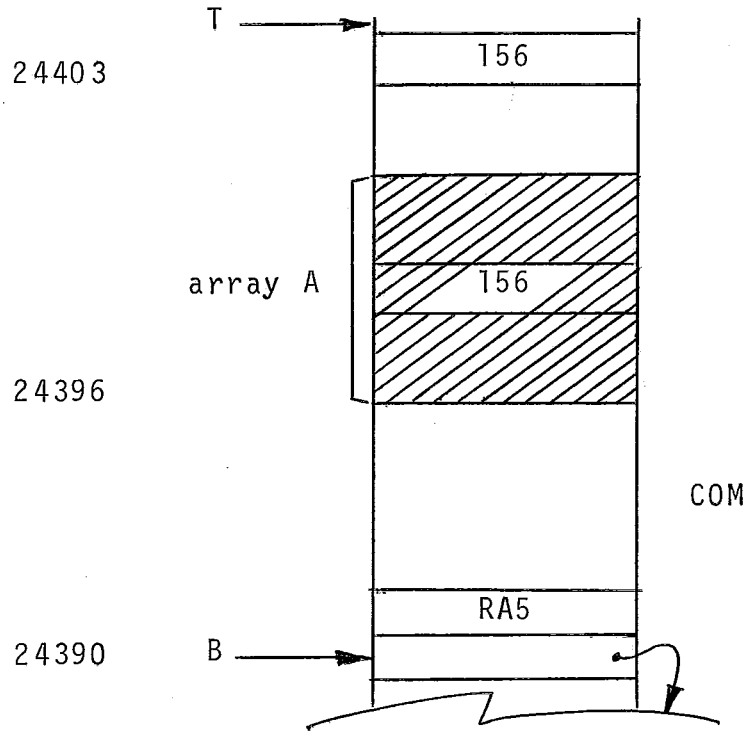


Figura - IV.4

Com a solução proposta ficam resolvidos casos em que um índice (ou mais) do array seja uma expressão envolvendo arrays: A [X + B [C], Y].

IV.II. RECORD

A figura (IV.5) ilustra a referência ao componente Y de um record X (referência a X.Y, sendo Y, por exemplo, inteiro).

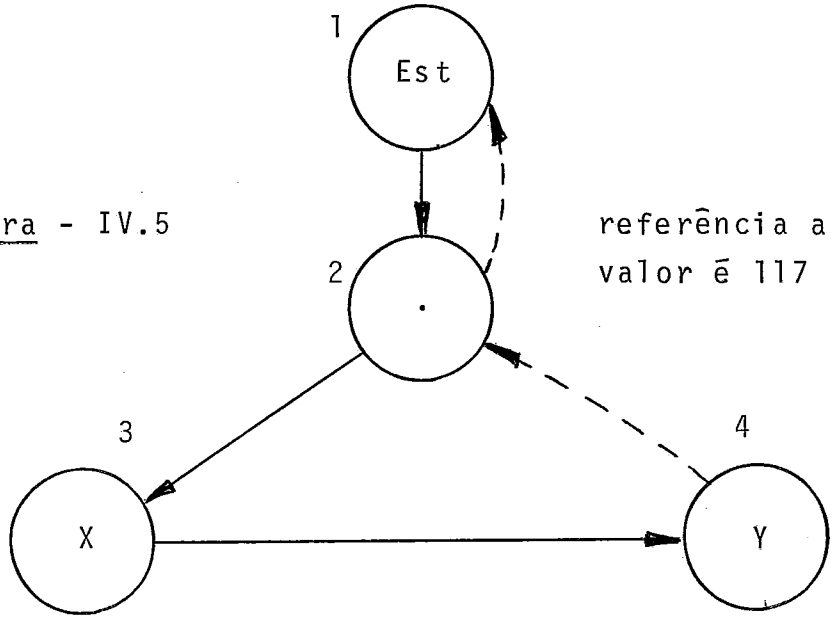


Figura - IV.5 referência a X.Y, cujo valor é 117

(3,6) deslocamento d=1

Vejamos as ações no percurso daquela árvore:

D1 - Ø

D2 - Ø

D3 - Como a variável X é do tipo record (o tipo vem da TS): Busca o endereço da variável X linearizando-o na PRA . Empilha tal endereço linearizado na PRA.

D4 - Como Y é componente de record: Busca o deslocamento (d = 1) de Y da TS, empilhando-o na PRA. Como este nó de componente possui alinhavo, salva na variável auxiliar COMPRIMENTO o tamanho da componente.

Ver figura (IV.6).

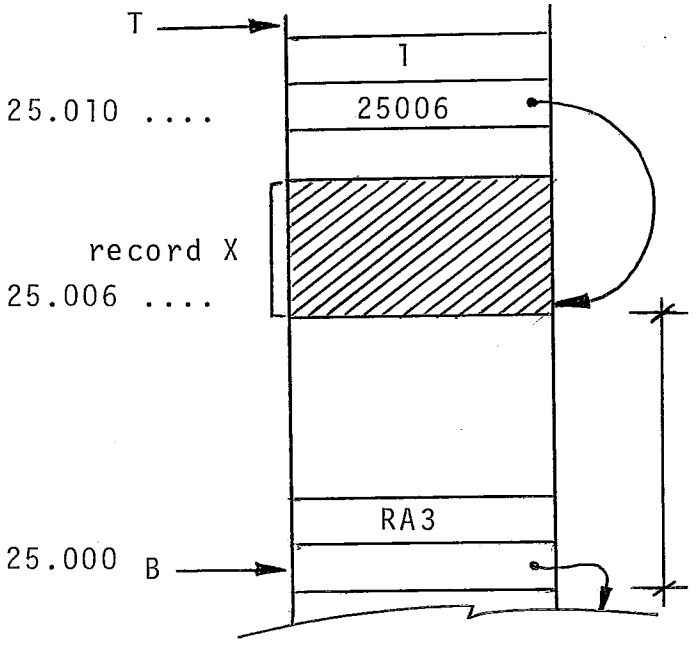


Figura - IV.6

S2 - Soma as duas entradas abaixo do topo da PRA substituindo-as por esta soma, que é o endereço da componente referenciada.

Ver figura (IV.7).

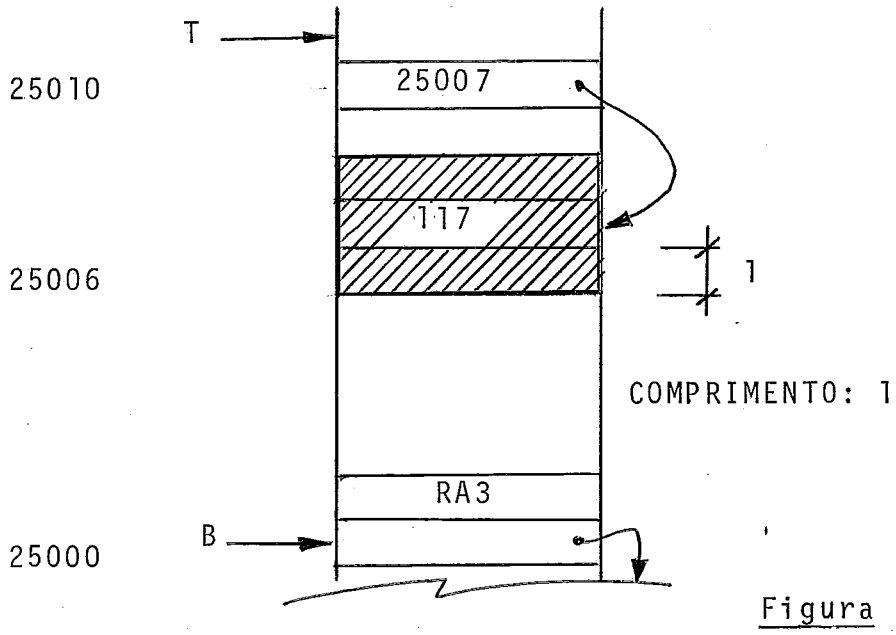


Figura - IV.7

S1 - Copia a componente para a partir da entrada abaixo do topo da PRA, tendo por base o endereço da componente e seu tamanho.

Ver figura (IV.8).

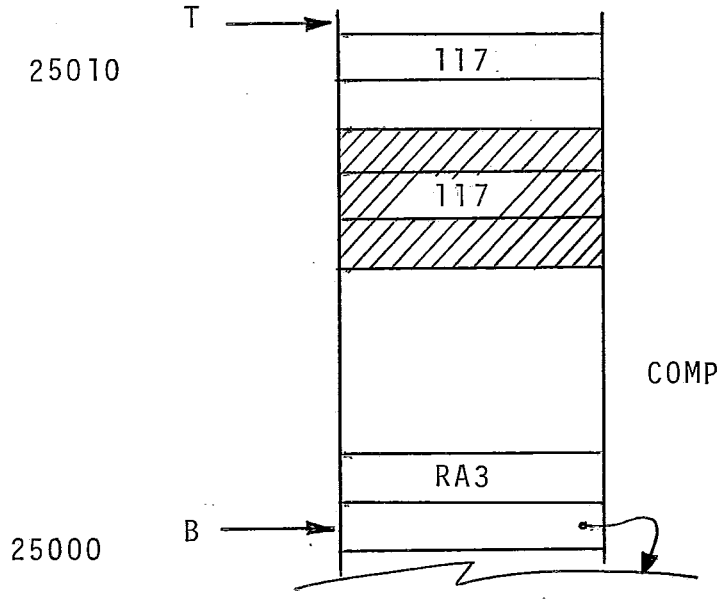


Figura - IV.8

Seja agora o caso em que a componente de um record seja outro record. Para isto, analisaremos a referência a X.Y.Z, onde X é uma variável do tipo record, Y é uma componente do tipo record e Z é uma componente simples (inteira, por exemplo).

A figura (IV.9) ilustra tal referência.

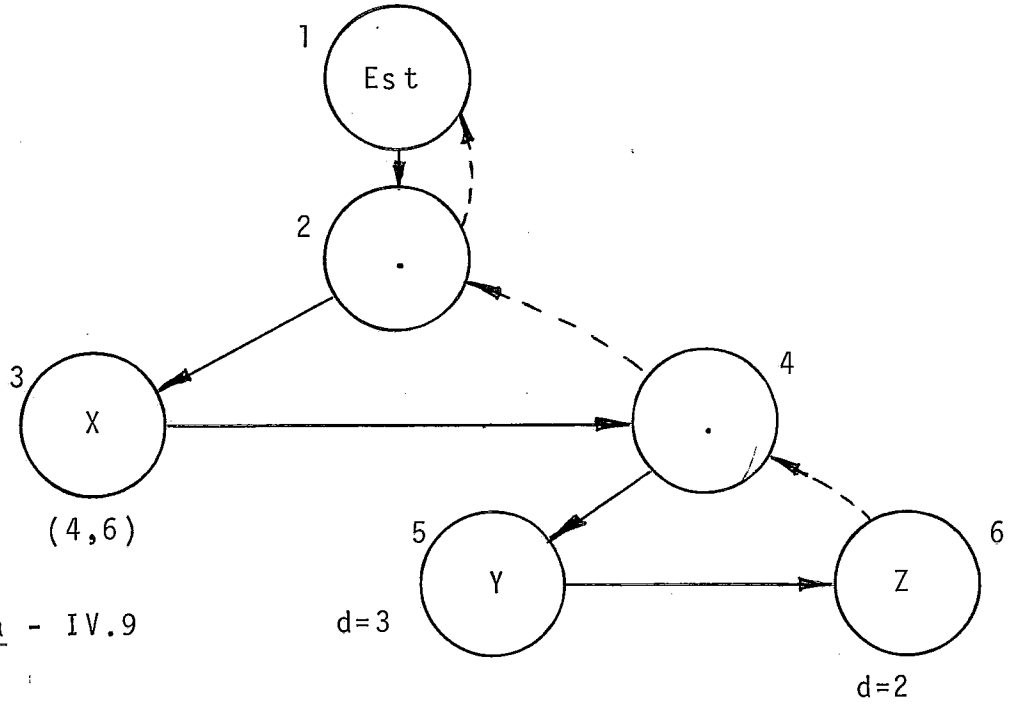


Figura - IV.9

Aqui, a novidade é o nó 5. A ação em D5, como Y é uma componente do tipo record, é análoga à ação ao descer em nó de componente simples: empilha-se o deslocamento de Y no record X na PRA.

A figura (IV.10) ilustra o estágio do processo de interpretação após D1, D2, D3, D4, D5 e D6 (a variável COMPRIMENTO teve valor a ela atribuido em D6, devido ao alinhavo).

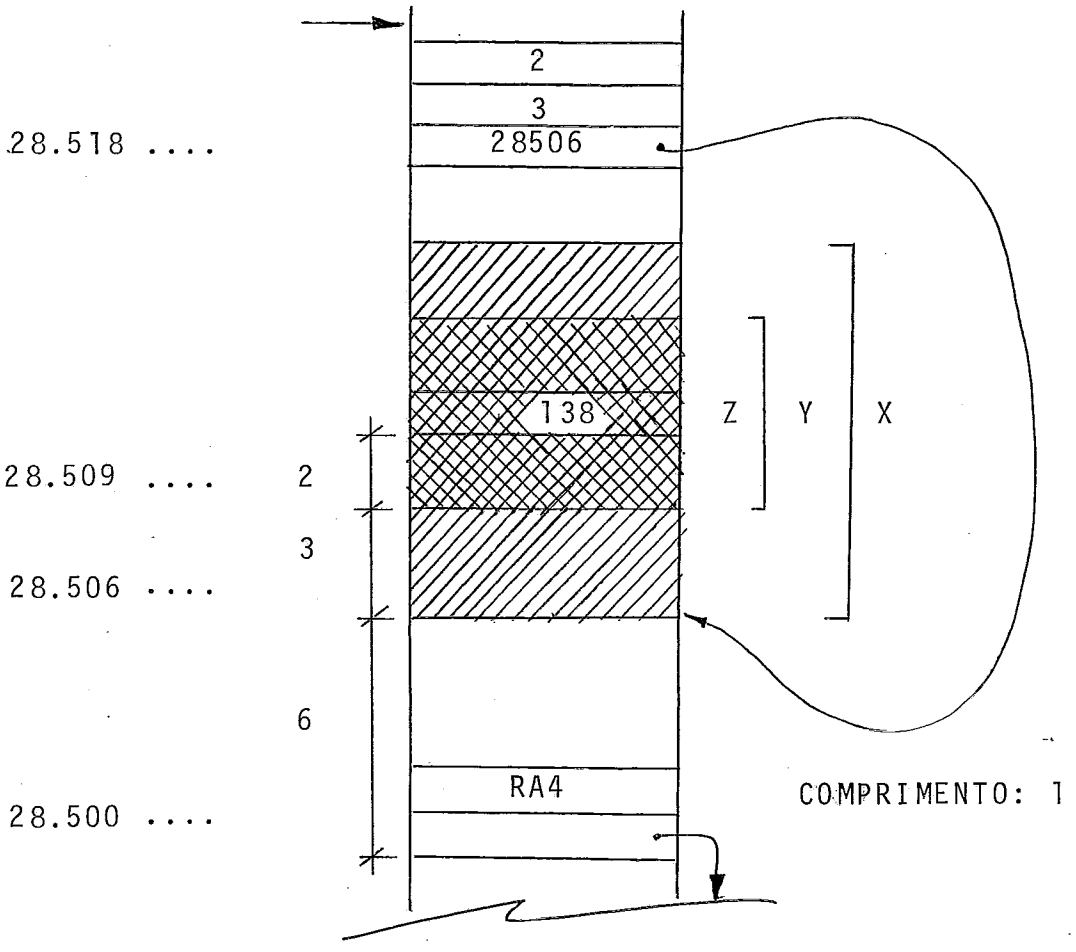


Figura - IV.10

Vejamos, agora, as ações seguintes:

S4 - Soma as duas entradas abaixo do topo da PRA, substituindo-as por tal soma (2 + 3 = 5).

S2 - Soma as duas entradas abaixo do topo da PRA, substituindo-as por tal soma (5 + 28506) = 28511).

Ver figura (IV.11).

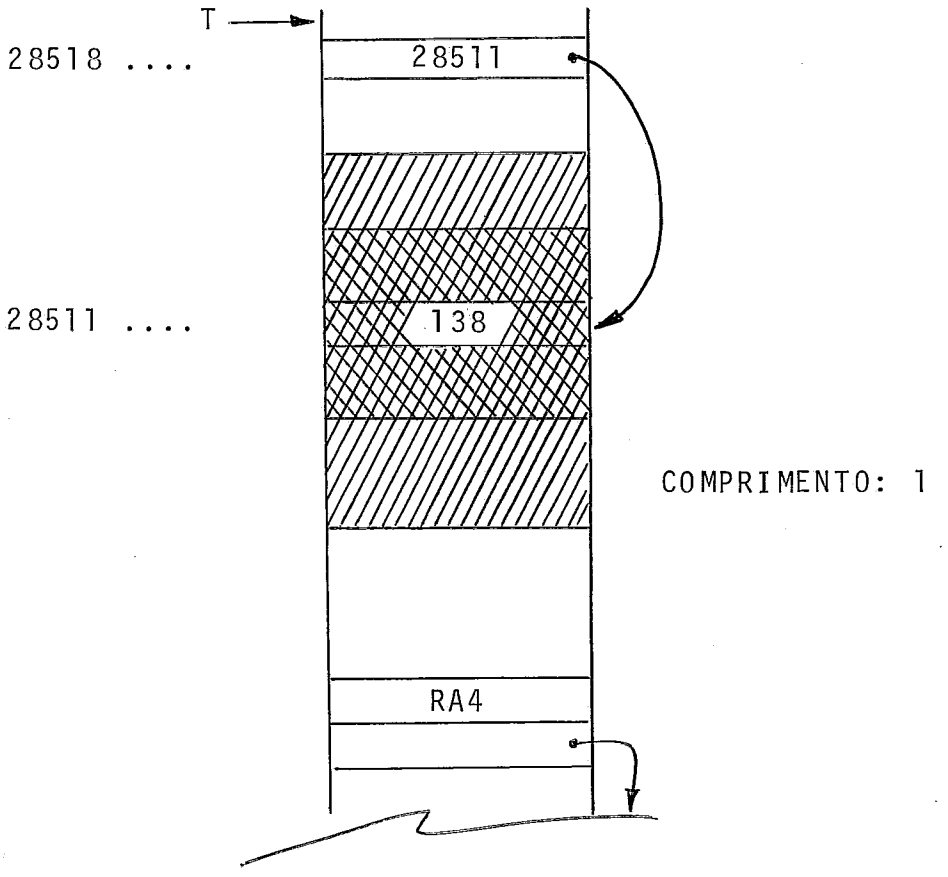


Figura - IV.11

SÌ - Copia a componente para a partir da entrada abaixo do topo da PRA, tendo por base o endereço da componente e seu tamanho.

Ver figura (IV.12).

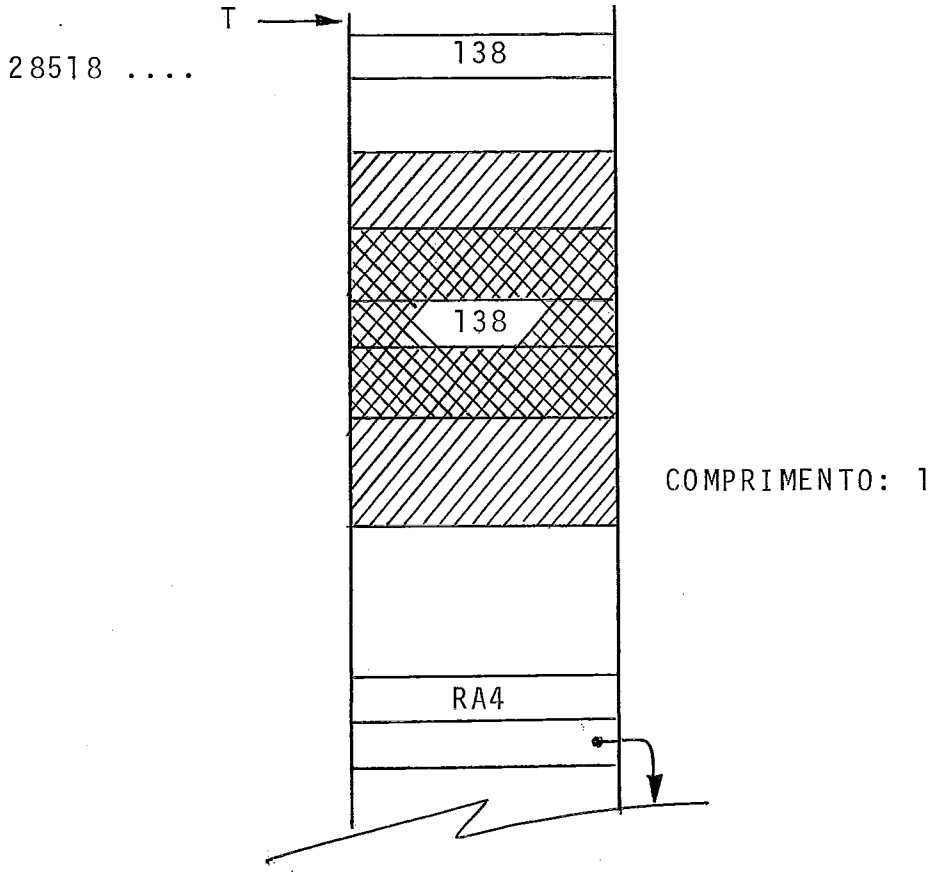


Figura - IV.12

Seja agora o caso de um record no qual uma componente seja do tipo array (de inteiros, por exemplo).

A figura (IV.13) apresenta um exemplo de referência do tipo mencionado: X.A [exp. 1 , exp.2]

Ao se descer no nó 9, vai-se buscar, da tabela de símbolos, o deslocamento do início do array A no record X: esta a ação a destacar no percurso da árvore do exemplo.

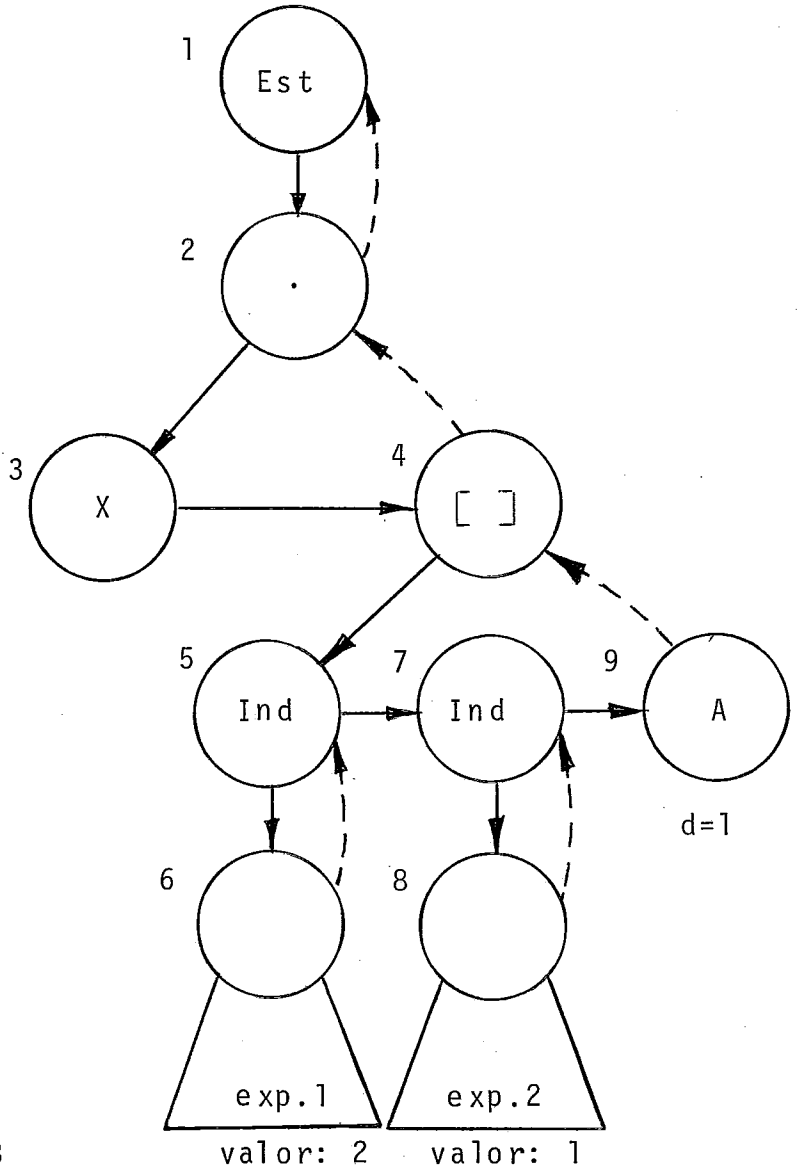


Figura - IV.13

Ações:

D1 - Ø

D2 - Ø

D3 - Como a variável X é do tipo record (o tipo vem da TS): Busca o endereço da variável X, linearizando-o na PRA. Empilha tal endereço linearizado na PRA.

D4 - Ø

D5 - Ø

D6 ... S6 - resulta a expressão avaliada, ficando o resultado imediatamente abaixo do topo da PRA.

S5 - Ø

D7 - Ø

D8 ... S8 - resulta a expressão avaliada, etc.

S7 - Ø

D9 - Como a componente é do tipo array (o tipo vem da TS): Busca o deslocamento de A no record X, empilhando-o na PRA. Empilha, ainda na PRA, o endereço deste identificador A na TS (o endereço está no próprio nó). Armazena na variável auxiliar COMPRIMENTO o tamanho do elemento referenciado.

Ver Figura (IV.14).

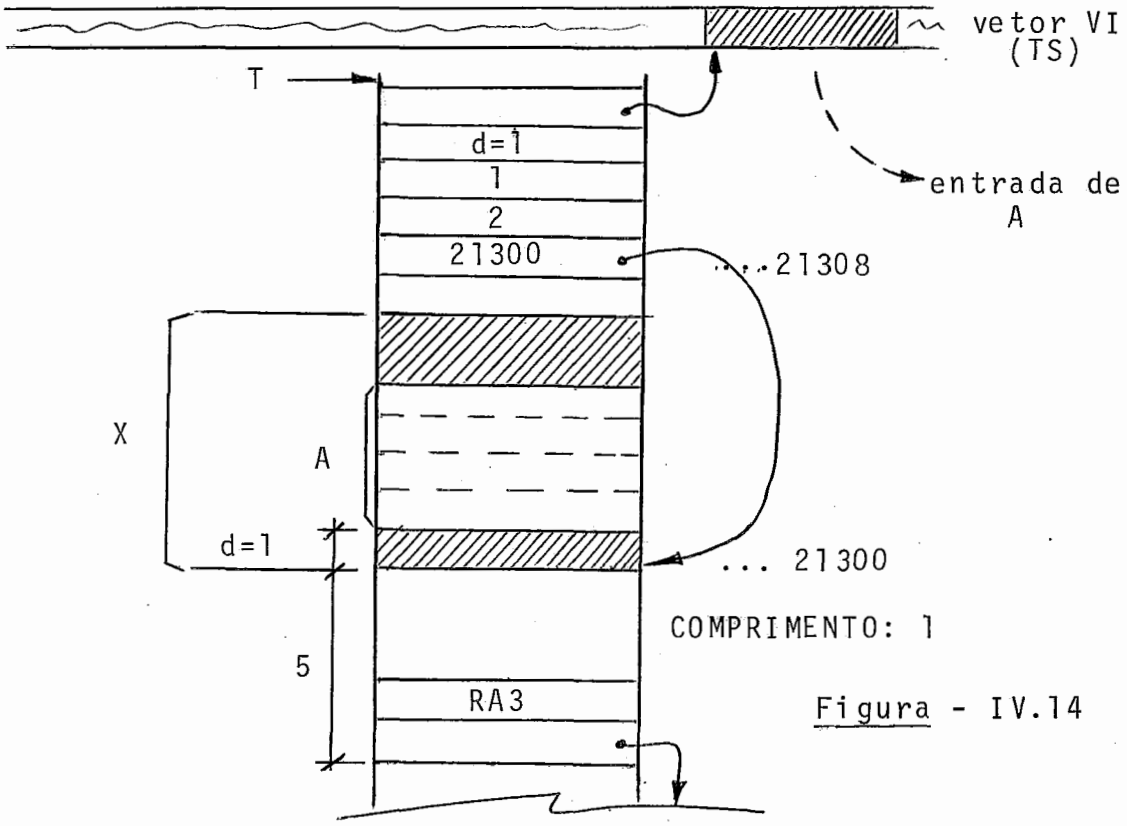


Figura - IV.14

S4 - Com base em informações da TS e em entradas armazenadas junto ao topo da PRA, calcula o deslocamento d' do elemento de A dentro do record X (este deslocamento já inclui o deslocamento básico, d = 1). Desempilha (n + 2) entradas - onde n é o número de índices - da PRA, empilhando, em seguida, naquela pilha, o deslocamento d' calculado.

Ver figura (IV.15).

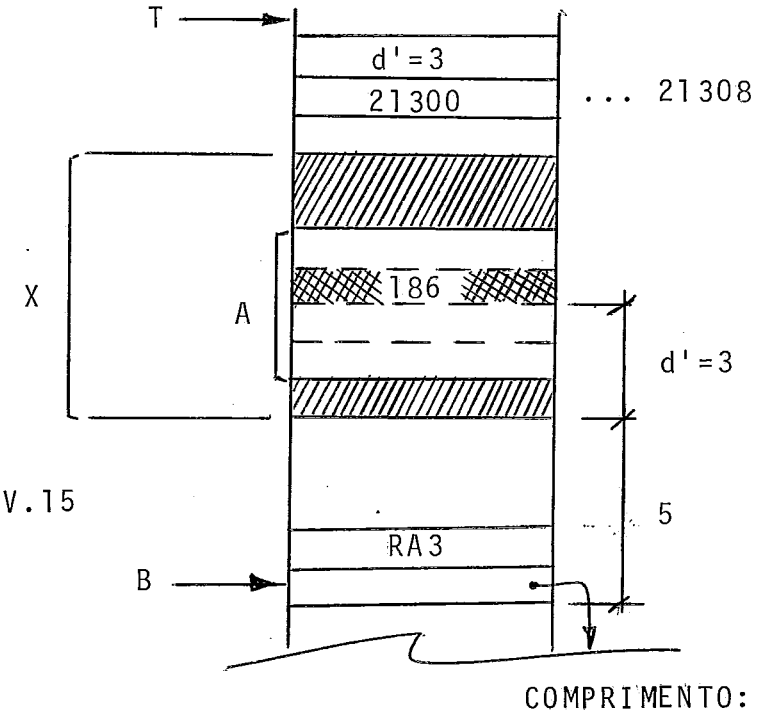


Figura - IV.15

S2 - Soma as duas entradas abaixo do topo da PRA, substituindo-as por esta soma, que é o endereço do elemento da componente A de X referenciada.

Ver figura (IV.16).

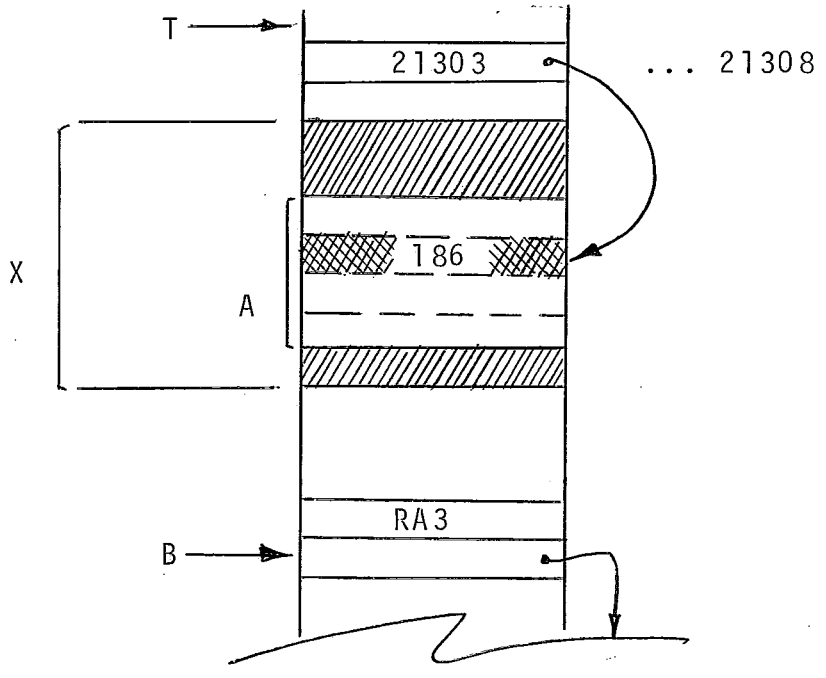


Figura - IV.16

S1 - Copia o elemento da componente A de X para a partir da entrada abaixo do topo de PRA, tendo por base o endereço do elemento e seu tamanho.

Ver figura (IV.17).

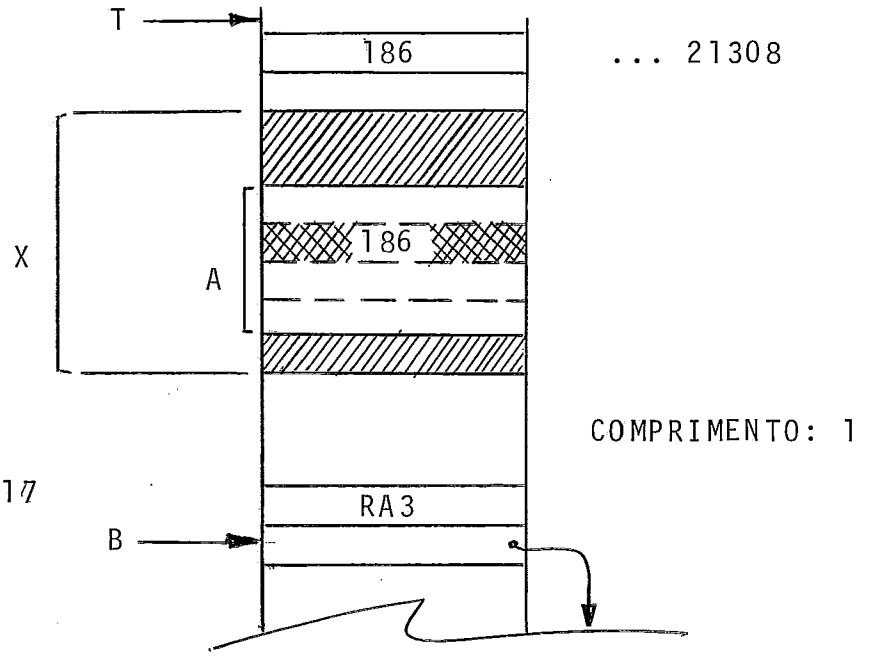


Figura - IV.17

Seja agora o caso de "array of records". A figura (IV.18) apresenta o exemplo de referência a X [exp.] .Y, onde Y é uma componente, por exemplo, do tipo inteiro.

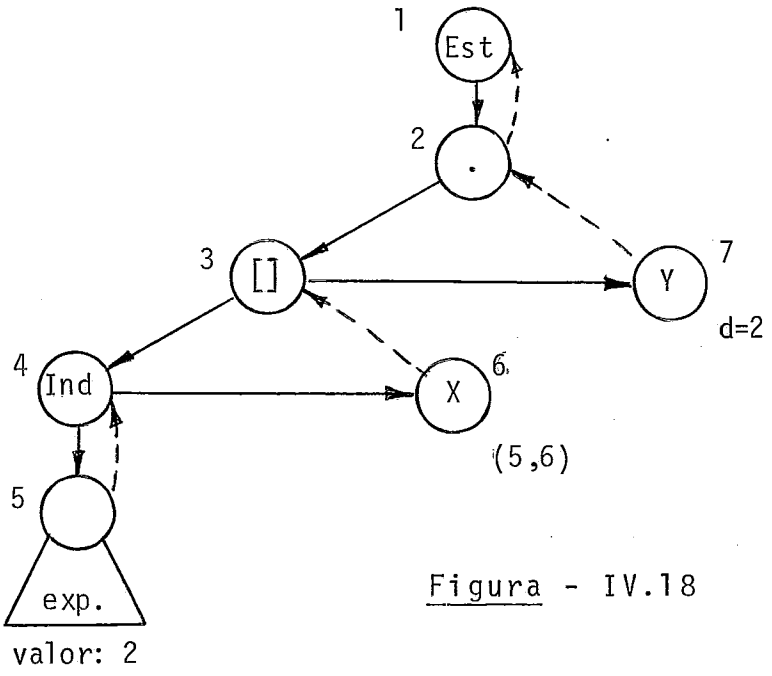


Figura - IV.18

A figura (IV.19) ilustra o estágio do processo de interpretação imediatamente após a ação S3.

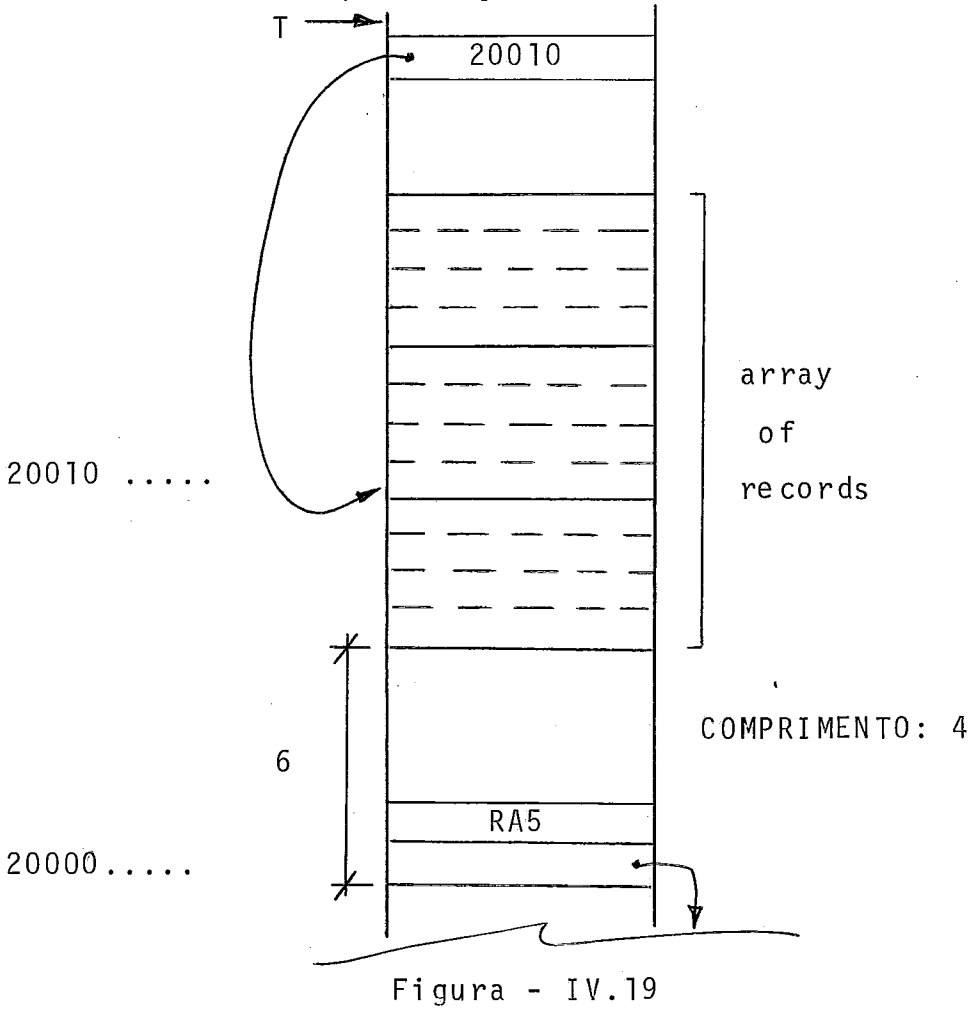


Figura - IV.19

Em seguida:

D7 - Como Y é componente de record: Busca o deslocamento (d = 2) de Y da TS, empilhando-o na PRA. Como este nó de componente possui alinhavo, salva na variável auxiliar COMPRIMENTO o tamanho da componente. (Observe-se que o comprimento 1, de componete inteira, vai recobrir o comprimento 4, de record, antes armazenado).

Ver figura (IV.20).

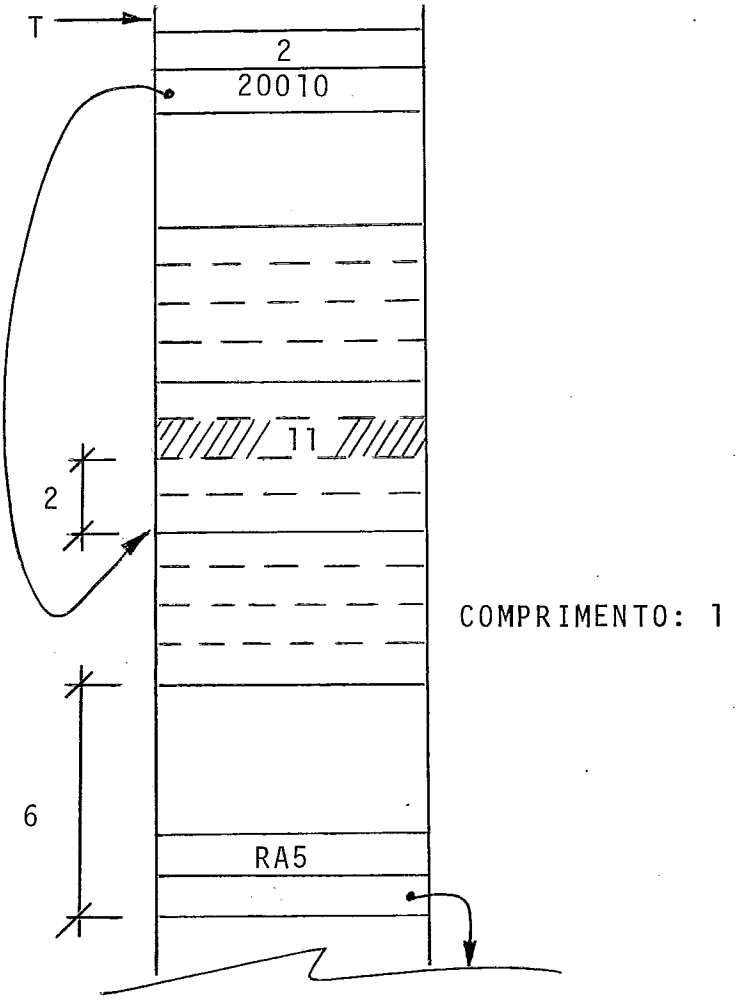


Figura - IV.20

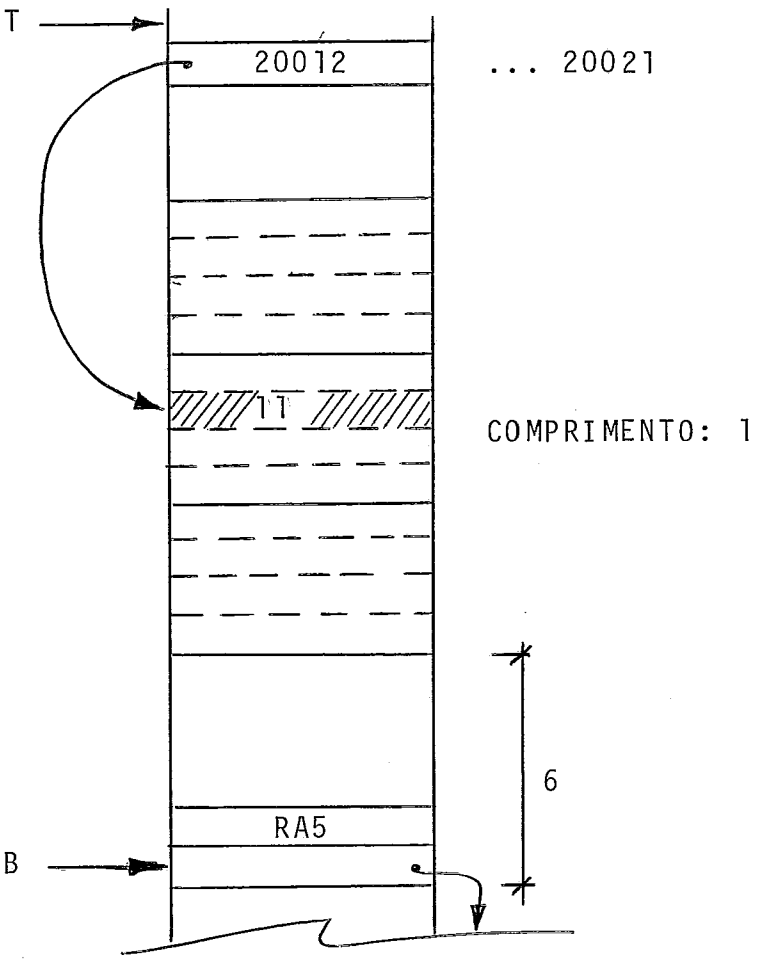


Figura - IV.21

S2 - Soma as duas entradas abaixo do topo da PRA, substituindo-as por esta soma.

Ver figura (IV.21).

S1 - Copia a componente do elemento record para a partir da entrada abaixo do topo da PRA, tendo por base o endereço da componente e seu tamanho.

Ver figura (IV.22).

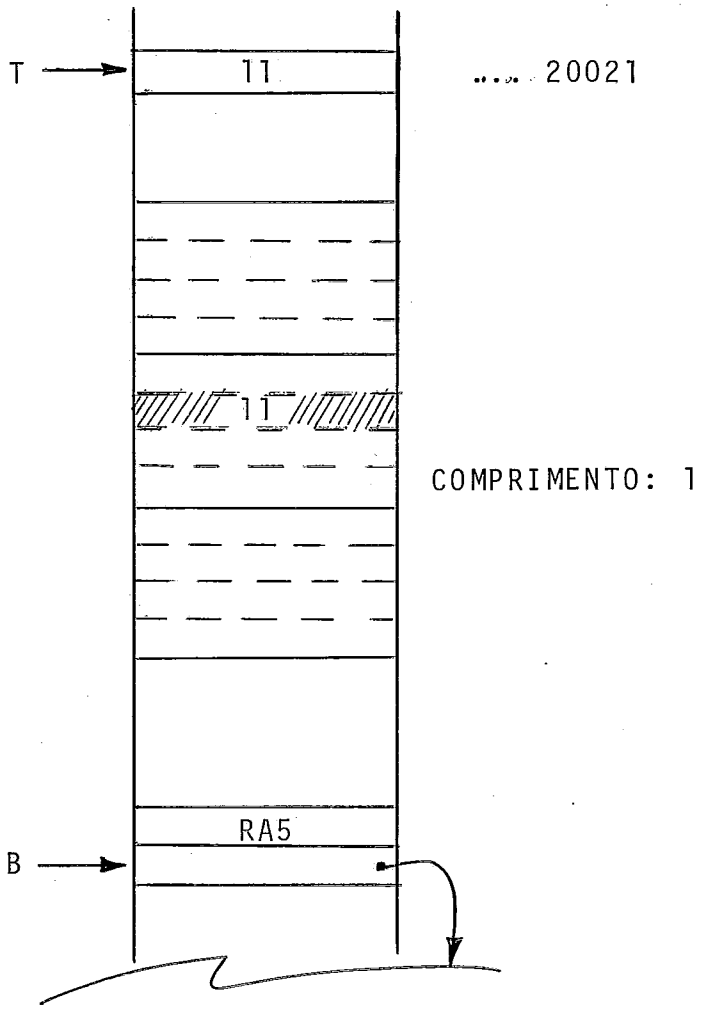


Figura - IV.22

CAPÍTULO V

V. ESTRUTURAS DINÂMICAS

A organização deste capítulo é semelhante à do anterior. Os diversos casos cobertos abrangerão os possíveis casos de referências a dados dinâmicos que propomos implementar.

Como já foi dito, dados dinâmicos são alocados numa área independente da PRA denominada Heap. Antes de estudarmos os dados dinâmicos, estudaremos um tipo simples que é o meio de acesso às estruturas apontadas: o pointer.

Cabe uma observação: os diversos tipos até aqui vistos e tratados como não-dinâmicos, quer simples (inteiro, real, etc), quer estruturados (record, array e combinações), passam na verdade a ser dinâmicos na medida que são apontados e, portanto, alocados no heap. Assim, podemos ter uma estrutura de records encadeados: os records são, neste caso, nós de uma estrutura dinâmica. O próprio pointer pode estar na PRA como meio de adentrar uma estrutura dinâmica, caso em que ele é considerado não dinâmico, ou pode ser, por exemplo, uma componente de um record de uma estrutura de records encadeados, quando passa a ser o meio de percorrê-lo, caso em que ele é parte de uma estrutura dinâmica.

Embora a maioria dos exemplos deste capítulo seja ilustrada com estruturas constituídas de records encadeados, na verdade as estruturas dinâmicas não estão restritas ao record. Poderíamos apontar para um inteiro ou para um array,

por exemplo.

V.I. O TIPO POINTER

O pointer é um tipo de dado simples, que permite adentrar estruturas dinâmicas ou percorrê-las. Para bem entendermos quero tipo pointer, quer as estruturas dinâmicas, procuraremos visualisar duas estruturas encadeadas, através da figura (V.1).

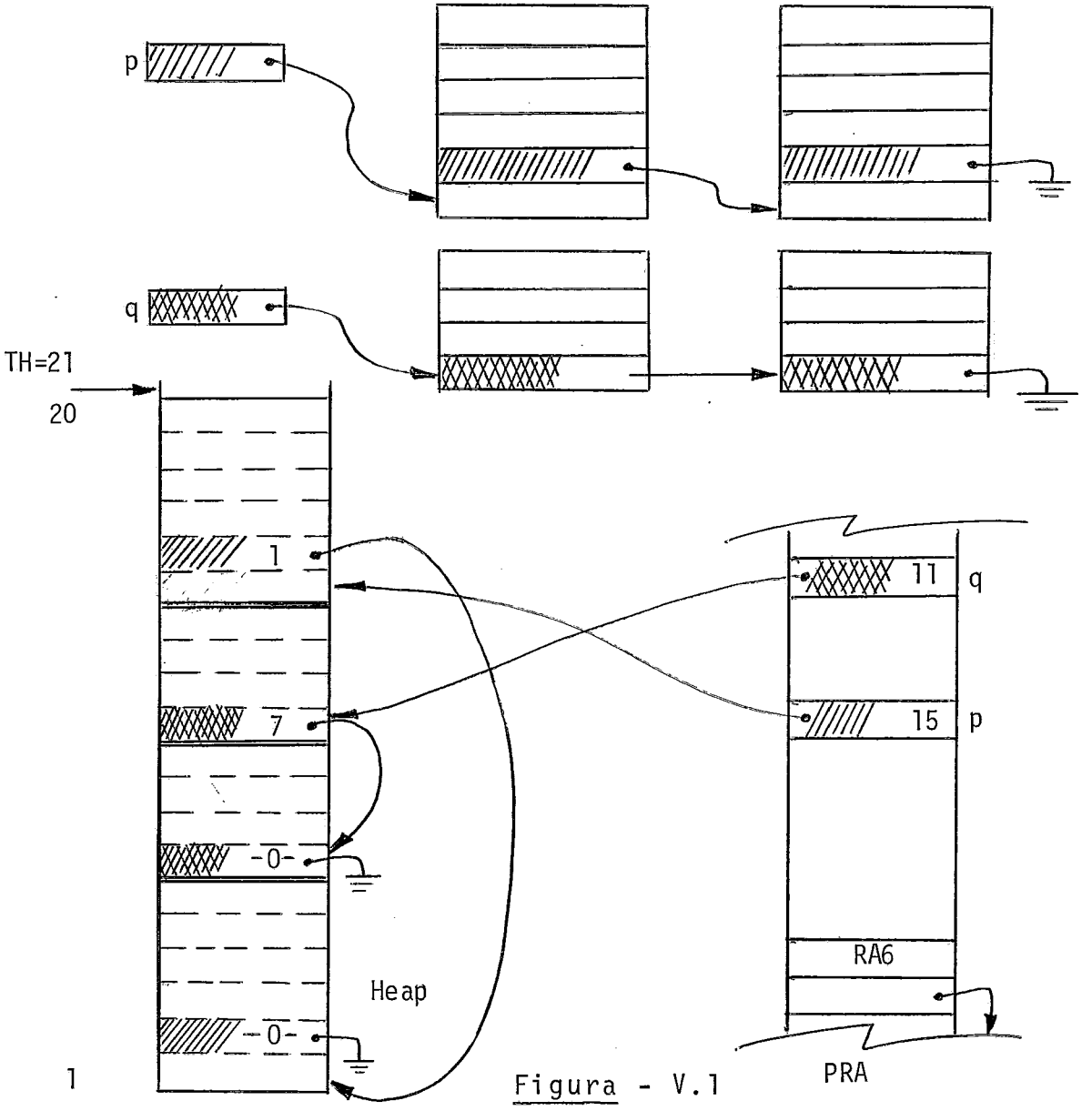


Figura - V.1

Ali se observa o que já foi mencionado: as variáveis p e q, do tipo pointer, são alocadas na PRA e permitem adentrar estruturas dinâmicas; por poutro lado, pointers são componentes dos records apontados e permitem percorrer as estruturas. No primeiro caso, os pointers são dados não-dinâmicos; no segundo, fazem parte de estruturas dinâmicas.

V.II. O PROCEDIMENTO NEW

O procedimento NEW (p) aloca, no topo do heap, o número de posições necessárias ao dado apontado por p, colocando ainda, na PRA, na localização de p, o endereço desta área alocada.

A figura (V.2) ilustra a árvore correspondente à chamada do procedimento NEW.

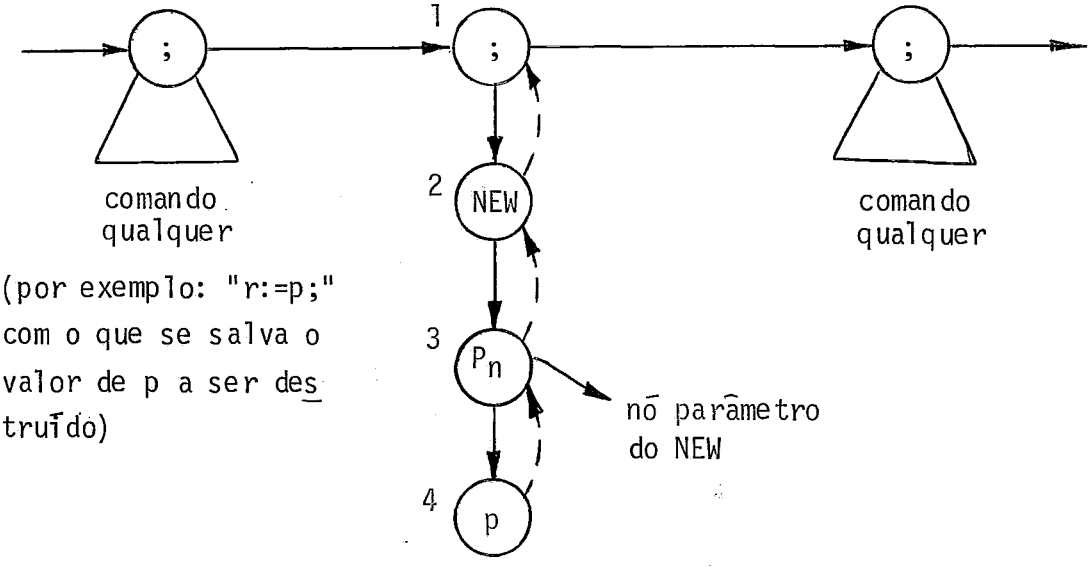


Figura - V.2

Ações:

D1 - Ø

D2 - Faz PARAM valer 1.

D3 - Ø

D4 - Como PARAM vale 1:

Busca o endereço de p, linearizando-o na PRA. Empilha este endereço na PRA. Copia ainda, da TS, o tamanho do apontado, na variável COMPRIMENTO.

S3 - Com base no endereço de p, que está abaixo do topo da PRA, atribui a p o valor de TH, topo do heap.

Ver figura (V.3).

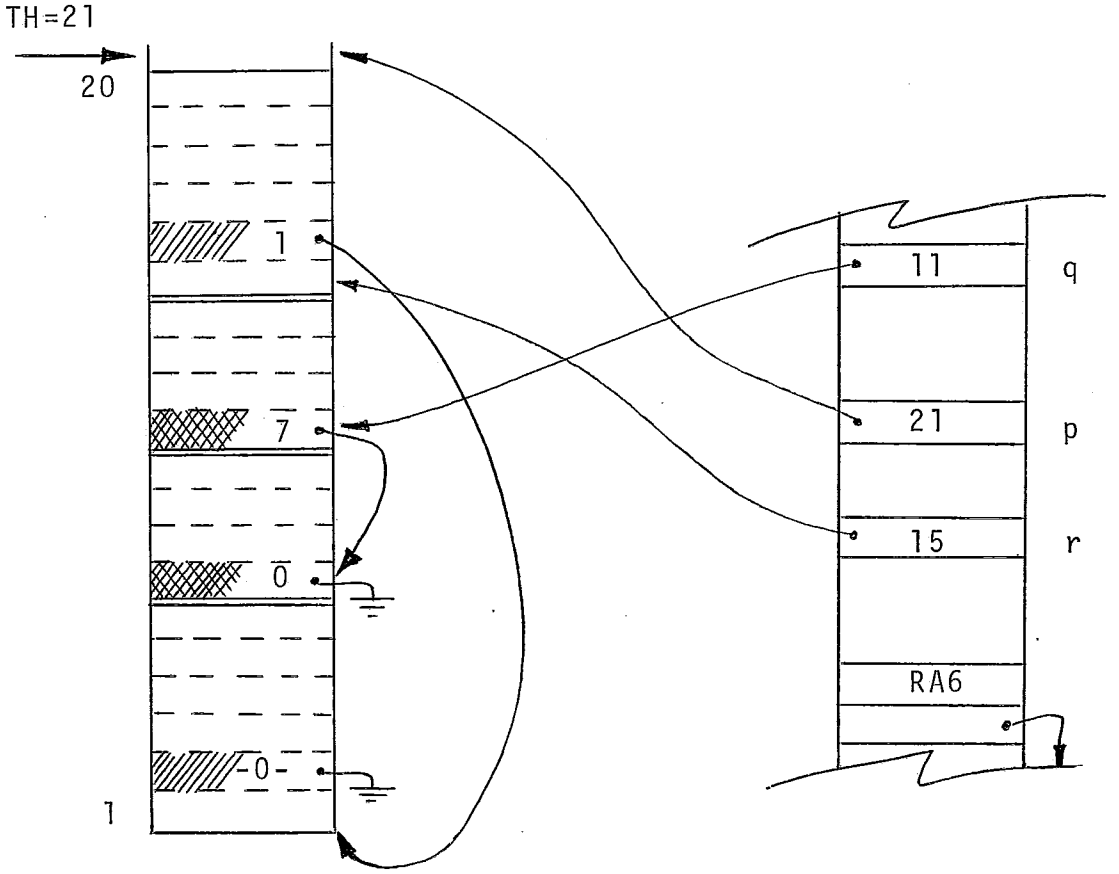


Figura - V.3

Com base na variável COMPRIMENTO, aloca espaço no heap, subindo seu topo.

Ver figura (V.4).

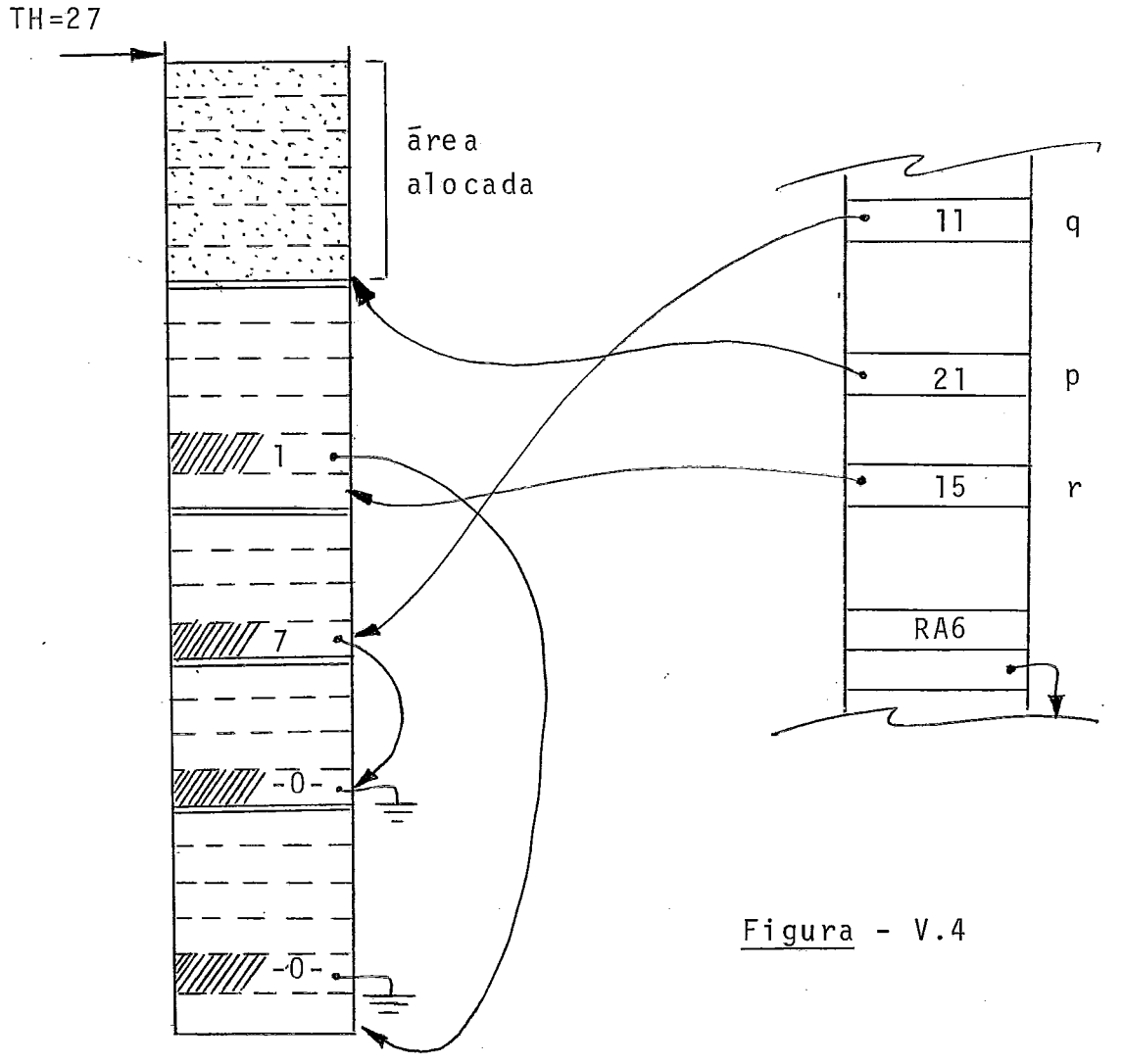


Figura - V.4

S2 - Faz PARAM valer zero.

S1 - Ø

V.III. REFERÊNCIAS A APONTADO SIMPLES E A RECORD DINÂMICO

Referências do tipo p↑ são referências a apontado simples (inteiro, real, booleano, caráter, escalar declarado, subrange) ou tipo record (caso de atribuição de/a record dinâmico).

A figura (V.5) apresenta a árvore correspondente a uma referência a apontado, que pode ser simples ou record. O nó 1, cabeça de referência a dado dinâmico, irá encabeçar qualquer árvore de referência deste tipo.

Ações:

D1 - Ø

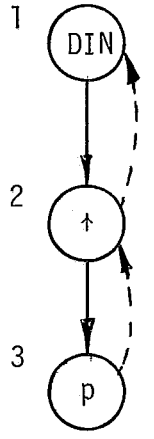


Figura - V.5

D2 - Ø

D3 - Busca o endereço de p, linearizando-o na PRA. Empilha o conteúdo deste endereço na PRA. Copia ainda, da tabela de símbolos, o tamanho do apontado , na variável COMPRIMENTO.

Ver figura (V.6).

S2 - Ø.

S1 - Copia o apontado para a partir da entrada abaixo do topo da PRA, tendo por base o endereço dele e seu tamanho.

Ver figura (V.7).

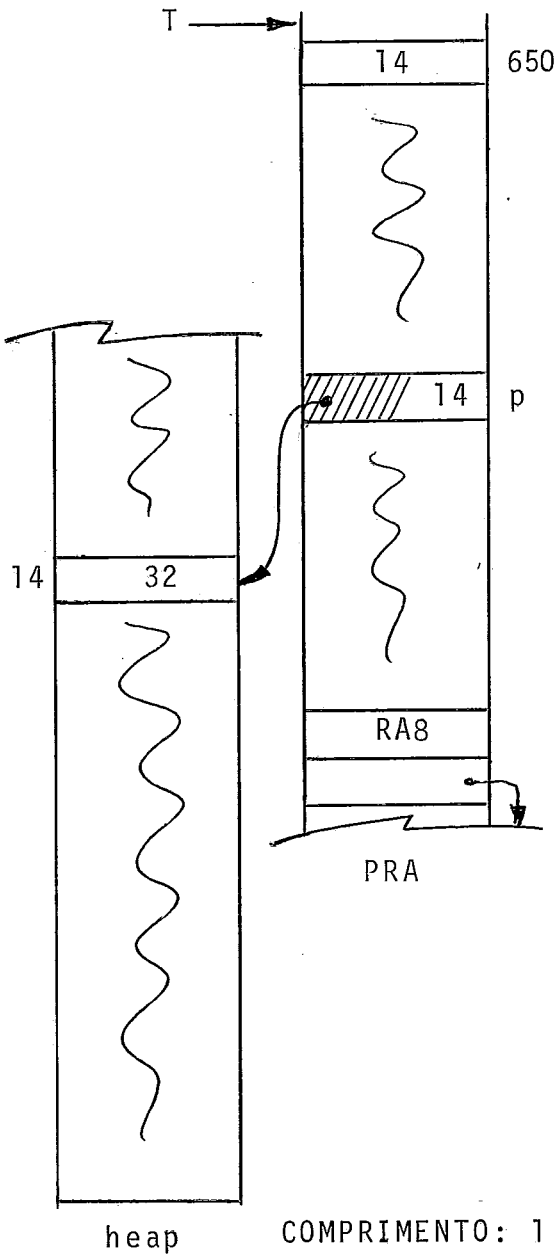


Figura - V.6

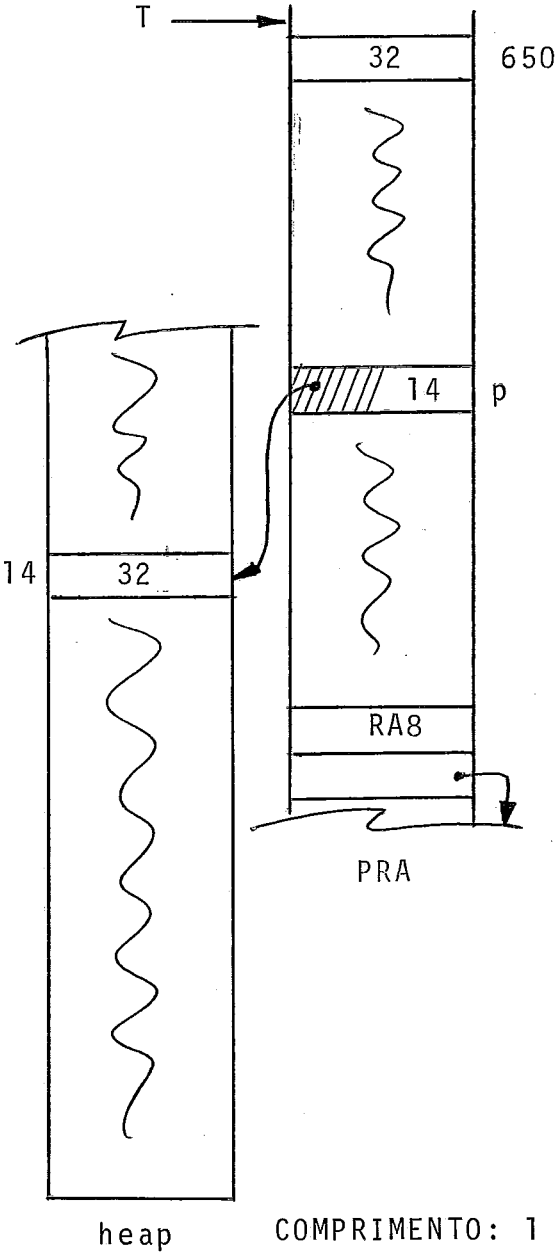


Figura - V.7

V.IV. REFERÊNCIA A APONTADO TIPO ARRAY

A figura (V.8) ilustra tal tipo de referência.

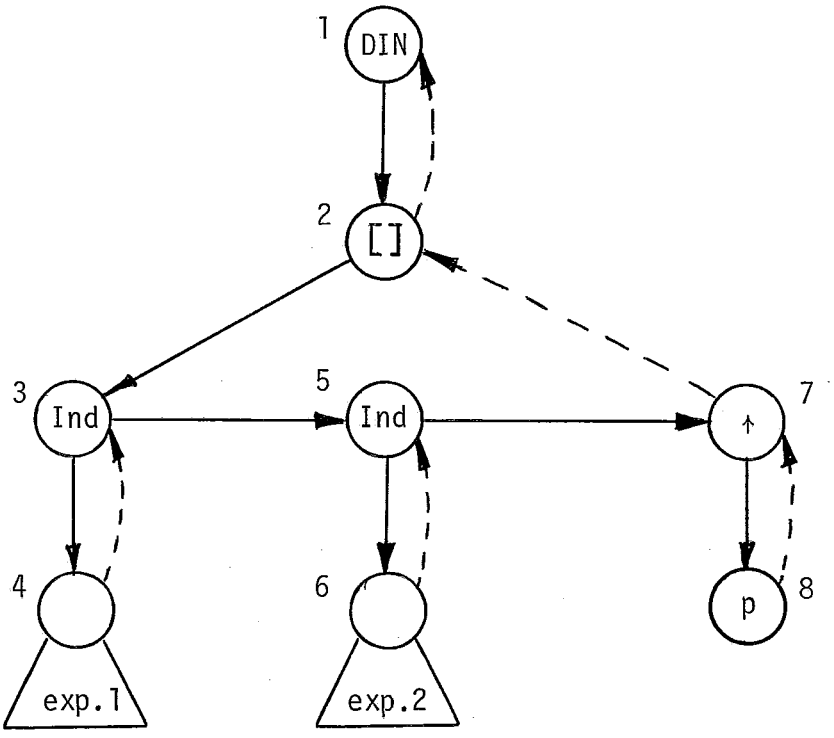


Figura - V.8

Ações:

D1 - Ø

D2 - Ø

D3 - Ø

D4 ... S4 - ações usuais para expressões aritméticas.

S3 - Ø

D5 - Ø

D6 ... S6 - ações usuais para expressões aritméticas.

S5 - Ø

D7 - Ø

D8 - Busca o endereço de p, linearizando-o na PRA. Empilha o conteúdo deste endereço na PRA. Como p aponta para array, salva na PRA o endereço na TS da descrição do array apontado e na variável COMPRIMENTO o tamanho do elemento.

S7 - Ø

Ver figura (V.9).

S2 - Com base em informações da TS e em entradas armazenadas junto ao topo da PRA, calcula o endereço do elemento referenciado. Desempilha (n + 2) entradas - onde n é o número de índices - da PRA, empilhando, em seguida, naquela pilha, o endereço calculado.

Ver figura (V.10).

Figura - V.9

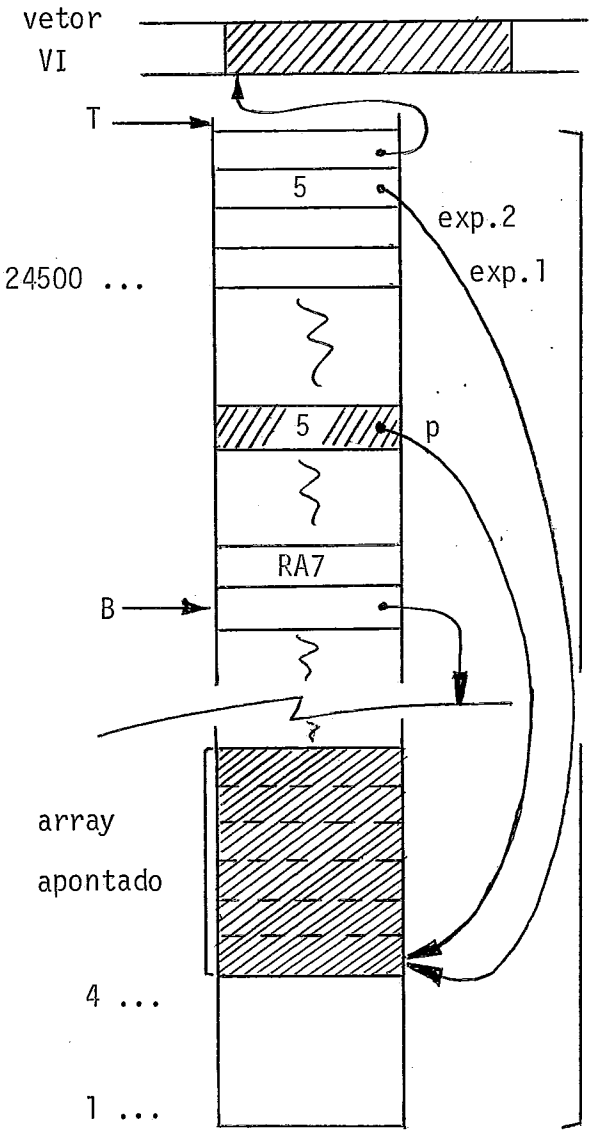
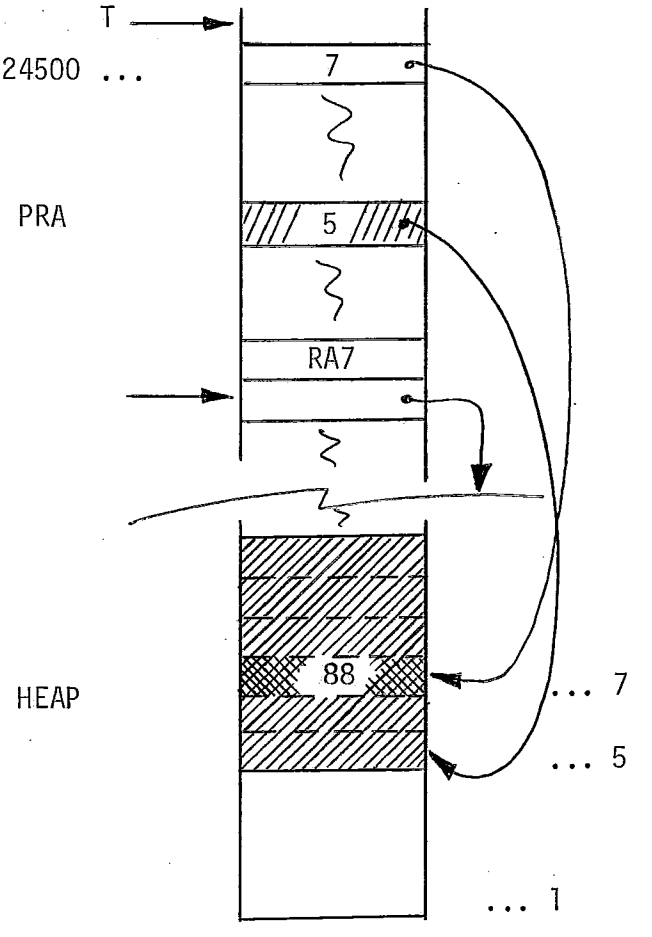


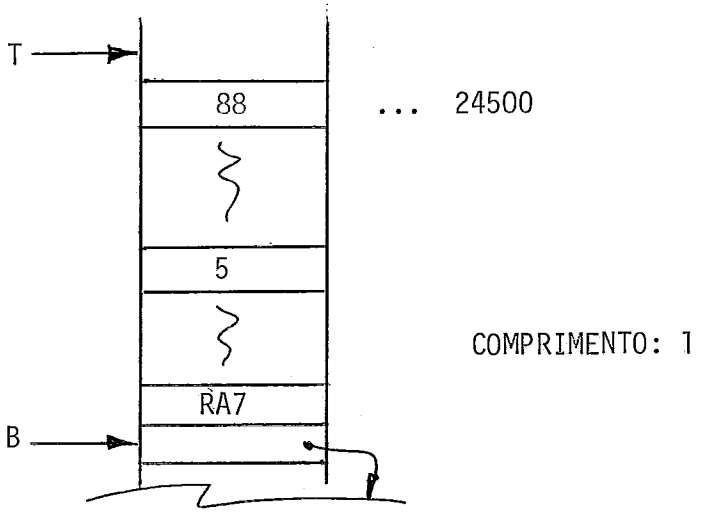
Figura - V.10



S1 - Copia o elemento para a partir da entrada abaixo do topo da PRA, tendo por base o endereço do elemento e seu tamanho.

Ver figura (V.11).

Figura - V.11



V.V. REFERÊNCIA A COMPONENTE DE RECORD APONTADO

Seja p↑.X . O significado de tal referência é o seguinte: p↑ corresponde a uma record do qual X é uma componente. Lembre-se que o record apontado não tem nome, não se constituindo numa variável explicitamente declarada. Assim, a referência ao record, como um todo, é feita por p↑, como já foi visto no item V.III. Quanto a X, é a componente de p↑ que foi referenciada.

A árvore para o exemplo visto é a da figura (V.12).

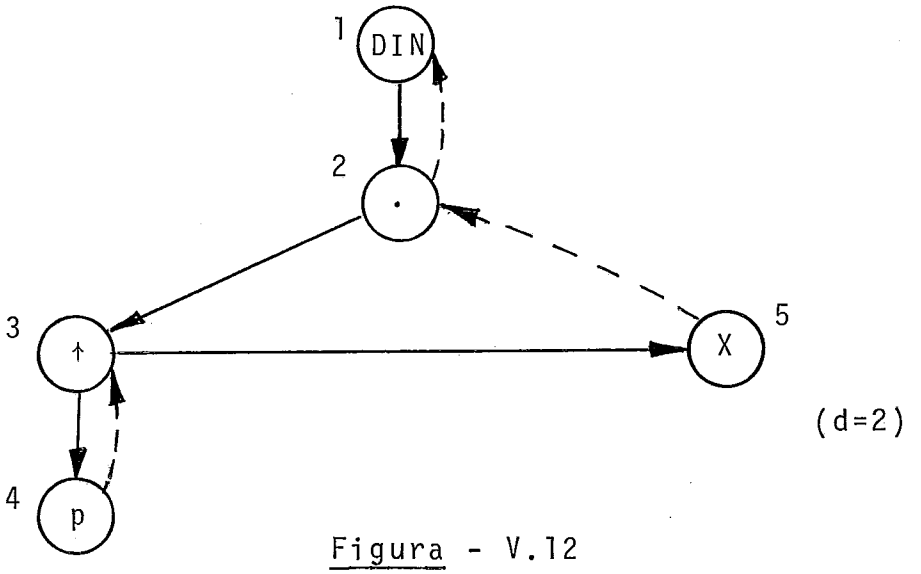


Figura - V.12

Ações:

D1 - Ø

D2 - Ø

D3 - Ø

D4 - Busca o endereço de p, linearizando-o na PRA. Empilha o conteúdo deste endereço na PRA (secundariamente, a variável COMPRIMENTO recebe o tamanho do apontado).

S3 - Ø

D5 - Como X é componente de record: Busca o deslocamento de X, da TS, empilhando-o na PRA. Como este nó de componente possui alinhavo, salva na variável COMPRIMENTO o tamanho da componente.

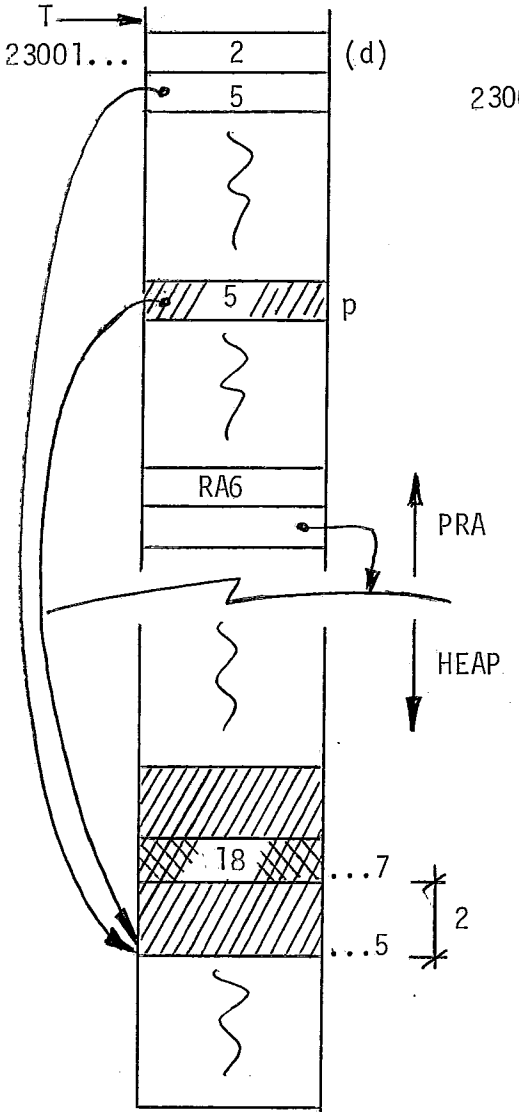
Ver figura (V.13).

S2 - Soma de duas entradas abaixo do topo da PRA, substituindo-as por esta soma, que é o endereço da componente referenciada.

Ver figura (V.14).

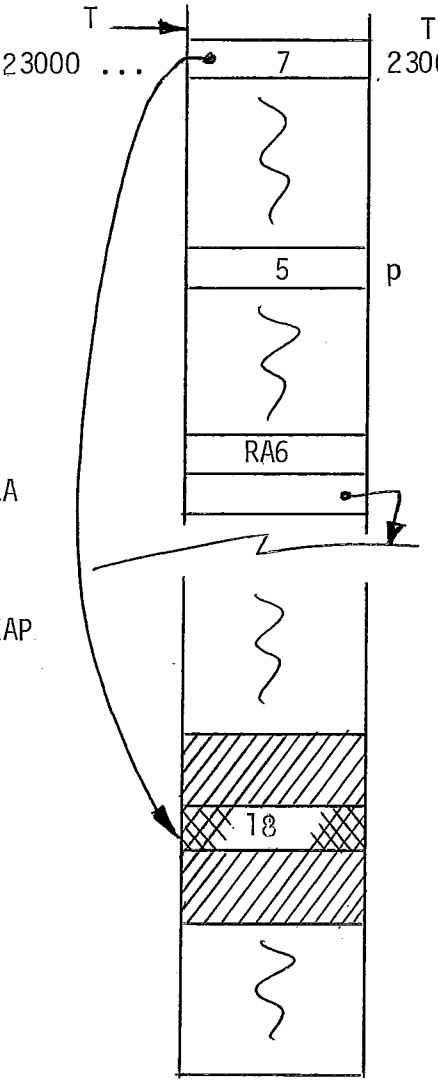
S1 - Copia a componente para a partir da entrada abaixo do topo da PRA, tendo por base o endereço da componente e seu tamanho.

Ver figura (V.15).



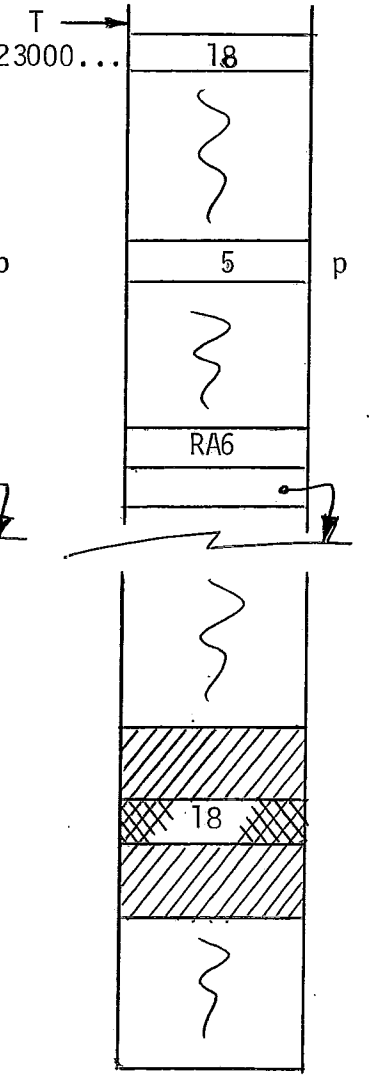
COMPRIMENTO: 1

Figura - V.13



COMPRIMENTO: 1

Figura - V.14



COMPRIMENTO: 1

Figura - V.15

Observe-se que o caso aqui analisado não restringe a componente referenciada a tipo simples. Ela poderia ser tipo record (para o caso da atribuição de/a record), quando a variável COMPRIMENTO, ao final, indicaria seu tamanho.

V.VI. CAMINHANDO NUMA ESTRUTURA LIGADA

Podemos, em PASCAL, fazer referência do tipo:

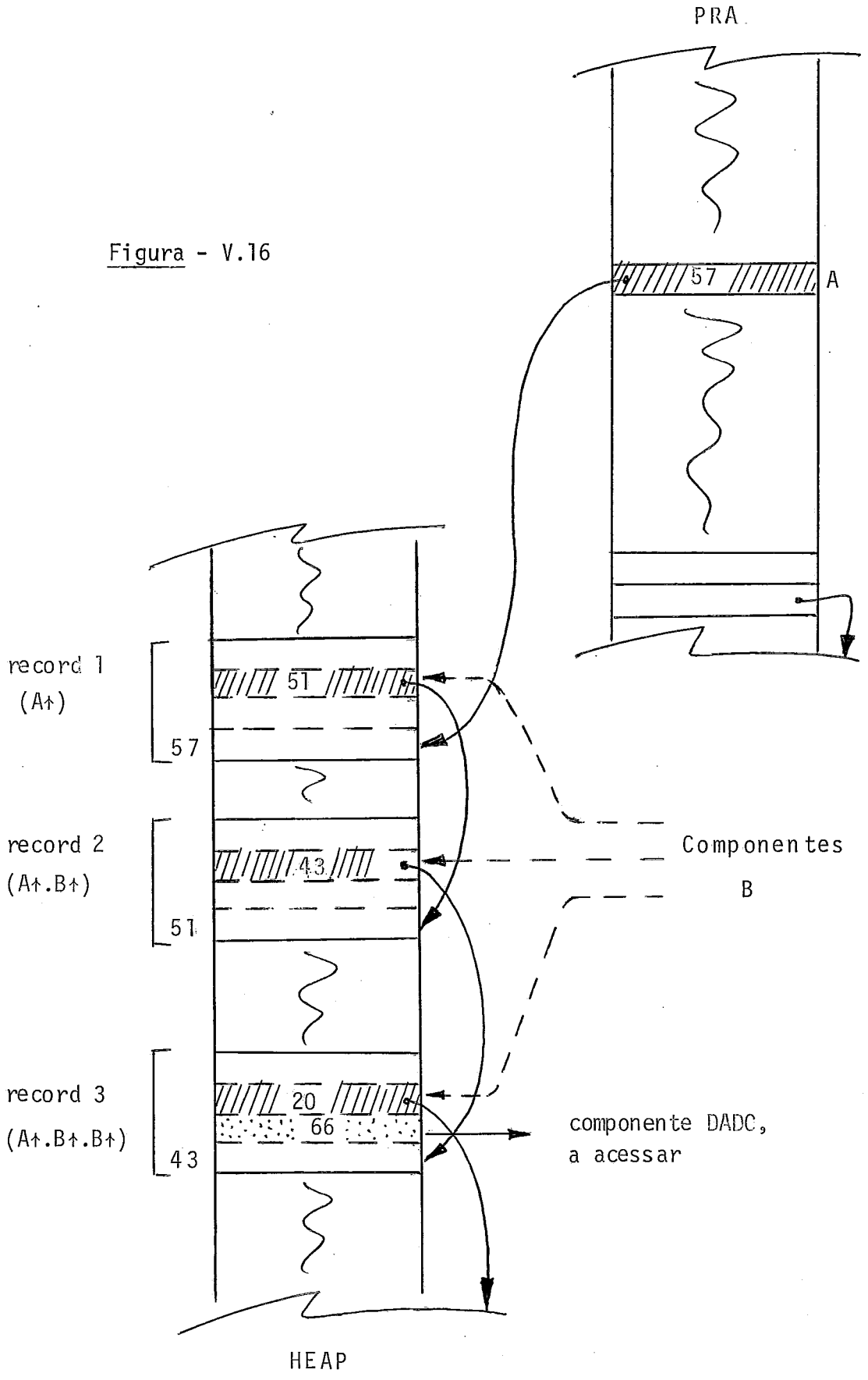
A↑ . B↑ . B↑ . DADO

Vejamos, na figura (V.16), o significado desta referência.

Observe-se que A↑ define um record (apontado); que A↑ . B refere-se à componente B do record apontado A↑; que A↑ . B↑ define outro record na cadeia; que A↑ . B↑ . B refere-se à componente B do record apontado A↑ . B↑; que A↑ . B↑ . B↑ define um terceiro record na cadeia; que A↑ . B↑ . B↑ . DADO refere-se à componente DADO do record apontado A↑ . B↑ . B↑ .

A árvore para A↑ . B↑ . B↑ . DADO é mostrada na figura (V.17).

Figura - V.16


PRA
57
A
record 1
(A↑)
51
57
record 2
(A↑.B↑)
43
51
Componentes
B
record 3
(A↑.B↑.B↑)
20
66
43
componente DADC,
a acessar
HEAP

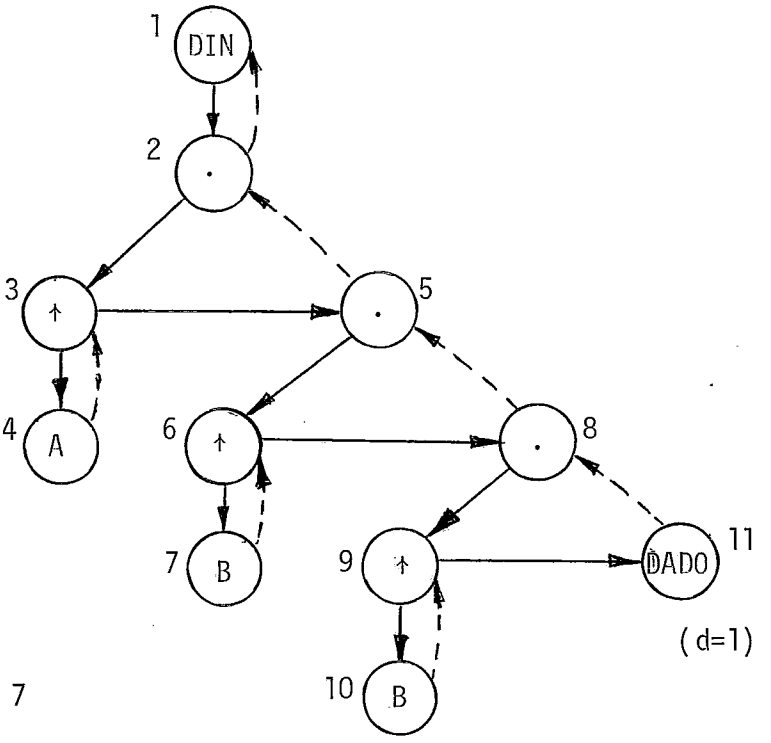


Figura - V.17

Ações:

D1 - Ø

D2 - Ø

D3 - Ø

D4 - Como A é um ponteiro externo à estrutura (como os ponteiros até aqui vistos): Busca o endereço de A, linearizando-o na PRA. Empilha o conteúdo deste endereço na PRA (secundariamente a variável COMPRIMENTO recebe o tamanho do apontado).

S3 - Ø

D5 - Ø

D6 - Ø

D7 - Como B é uma componente pointer de record apontado: Empilha,na PRA, o deslocamento de B no

record apontado (o deslocamento vem da TS).

Soma as duas entradas abaixo do topo da PRA, substituindo-as por uma entrada nula e depois tal soma (esta soma é a localização, no heap, da componente B do primeiro record apontado.

Substitui a soma pelo conteúdo da localização por ela endereçado.

Ver figura (V.18).

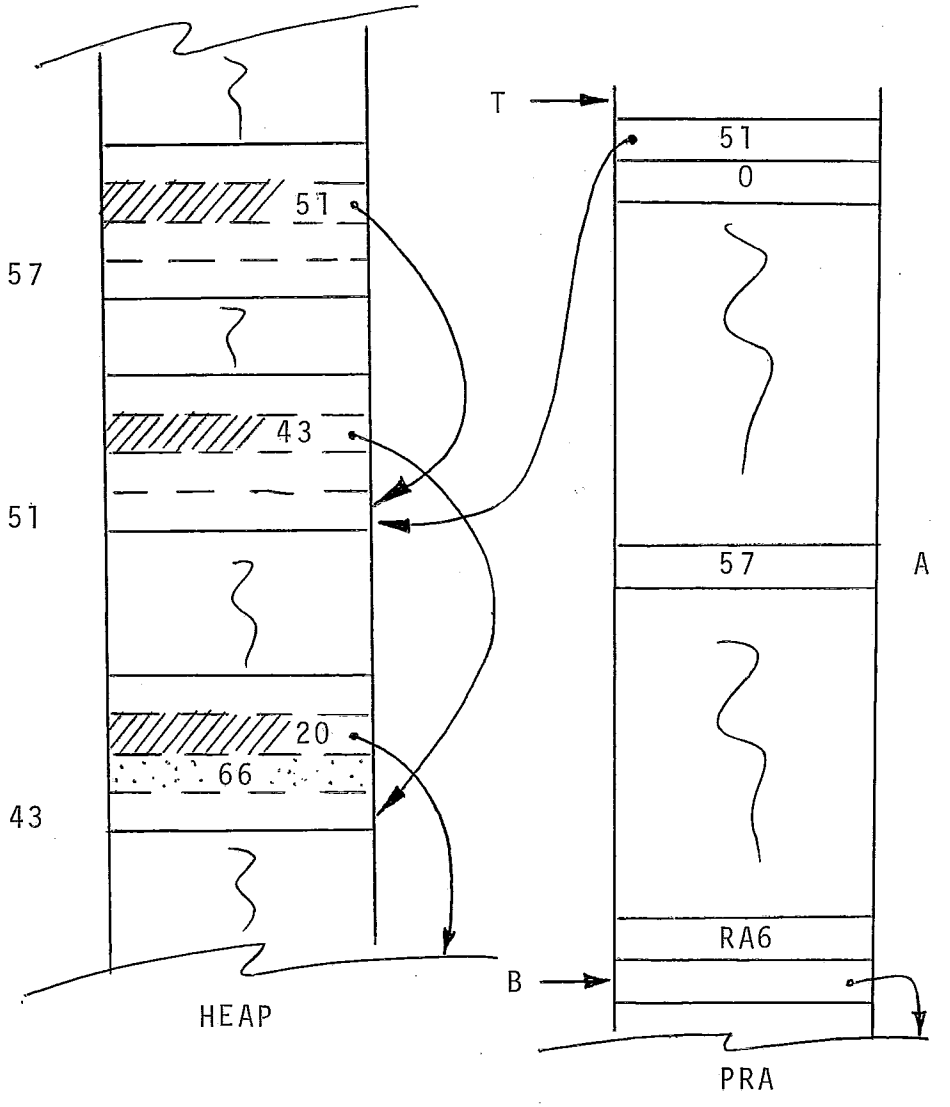


Figura - V.18

S6 - Ø

D8 - Ø

D9 - Ø

D10 - Análogo a D7.

Ver figura (V.19).

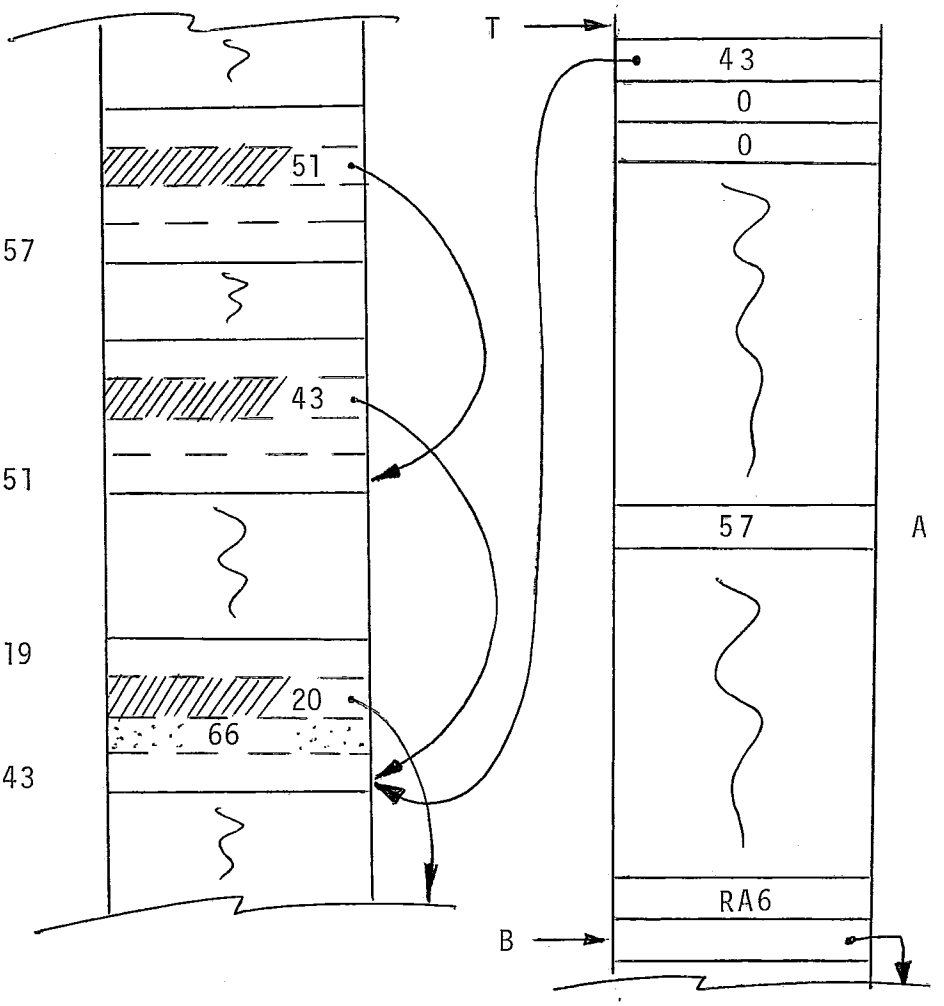


Figura - V.19

D11 - Como DADO é componente de record: Busca o deslocamento (d = 1) de DADO da TS, empilhando-o na PRA. Como este nó de componente possui alinhavo, salva na variável auxiliar COMPRIMENTO o tamanho da componente.

Ver figura (V.20).

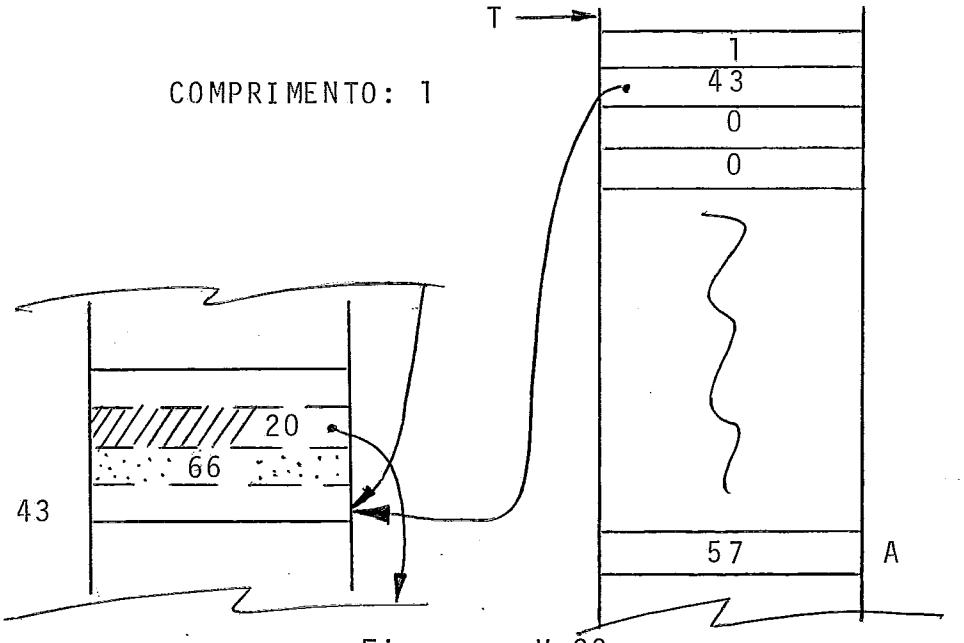


Figura - V.20

S8 - Soma as duas entradas abaixo do topo da PRA, substituindo-as por esta soma.

Ver figura (V.21).

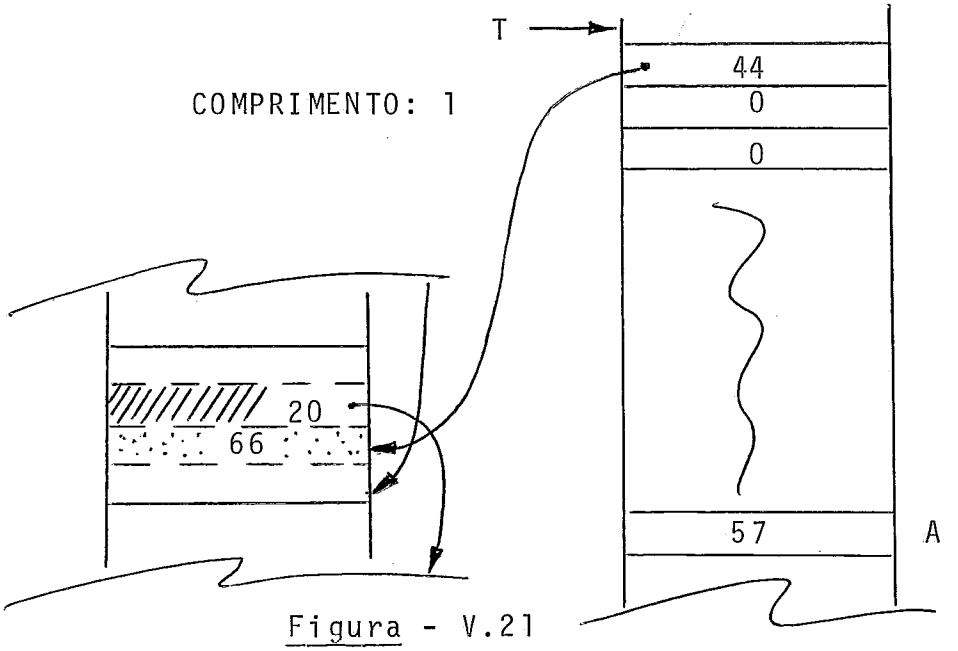


Figura - V.21

S5 - Análogo a S8.

Ver figura (V.22).

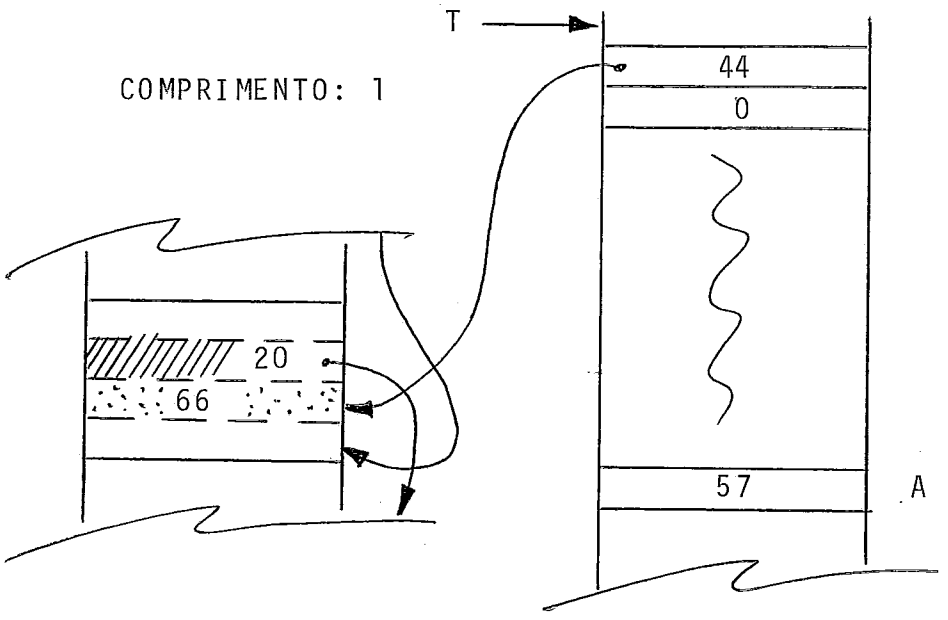


Figura - V.22

S2 - Análogo a S8.

Ver figura (V.23).

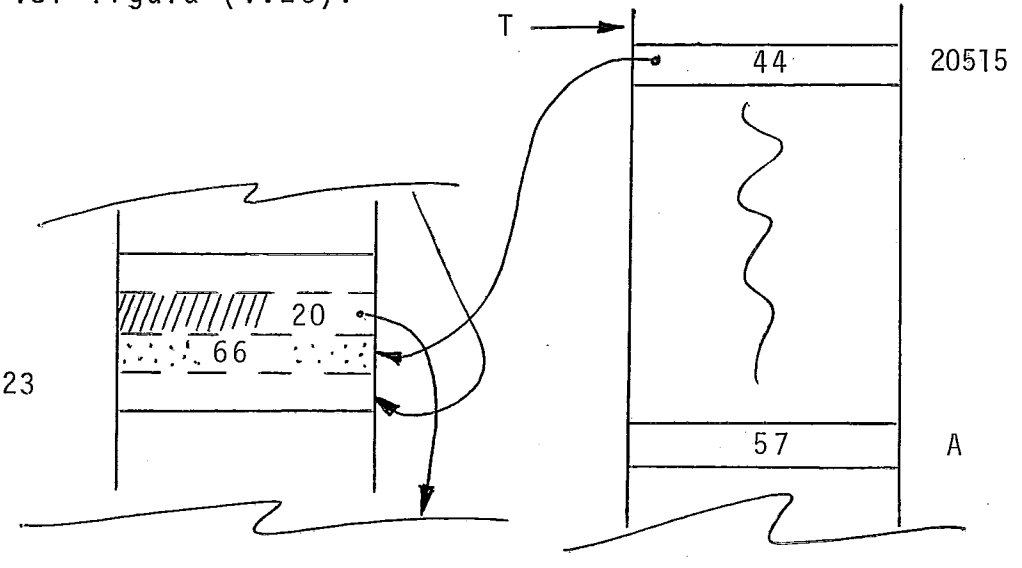
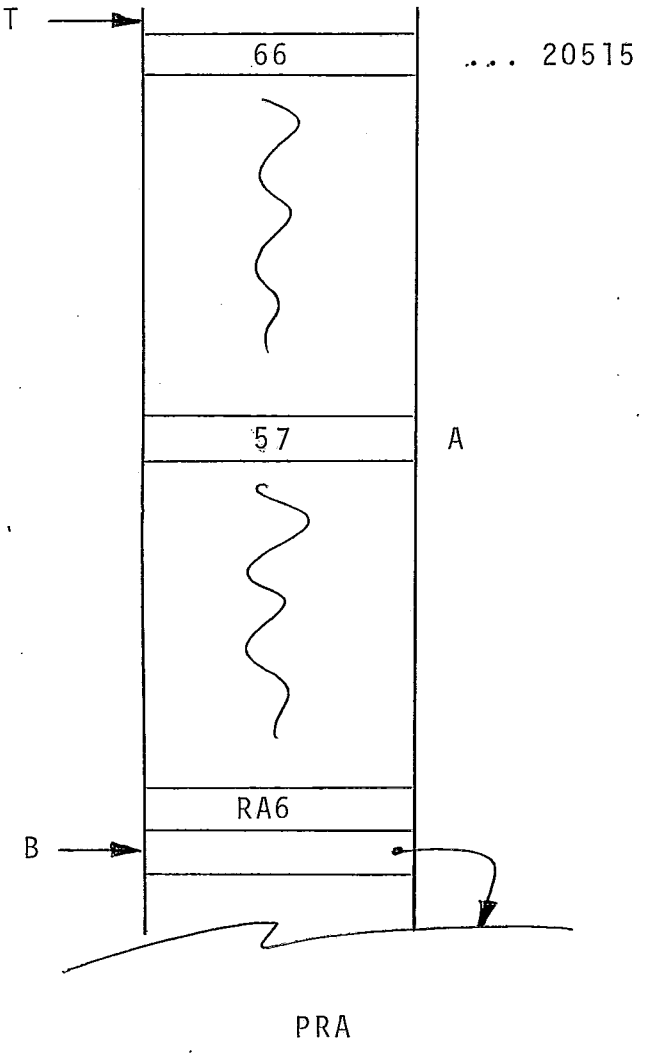


Figura - V.23

S1 - Copia a componente para a partir da entrada abaixo do topo da PRA, tendo por base o endereço da componente e seu tamanho.

Ver figura (V.24).



COMPRIMENTO: 1

Figura - V.24

CAPÍTULO VI

VI. OS COMANDOS PASCAL

Até este ponto, exploramos detidamente os diversos tipos de dados mantidos pela linguagem PASCAL, sua representação na FIP e soluções propostas para interpretação. Foi ainda focalizada em detalhes a resolução de expressões. Neste capítulo, abordaremos os diversos comandos providos pelo PASCAL, expondo a FIP para cada caso e mostrando os procedimentos de interpretação associados.

Neste ponto, sugerimos um retorno à figura (II.1) deste trabalho. Aquela figura nos dá a idéia geral de um programa representado por sua FIP. Consideremos, em particular, dois nós daquela figura: os nós 5 e 6. O nó 5, de comando composto, não tem ação específica no percurso da árvore-FIP, quer subindo, quer descendo. Já o nó 6, de label, poderá ter ação específica de depuração, a ser descrita em capítulo próprio, além da ação de desempilhamento de RA's a ser vista adiante, neste capítulo.

Ainda com relação à figura (II.1), destaquemos dois outros nós: o nó 1 é a raiz da FIP e não tem ação específica a ele associada. O nó 4 marca o final da interpretação: em S4, ela é interrompida.

Vamos, a seguir, analisar os diversos comandos PASCAL. E o faremos inicialmente por um comando extremamente relacionado com expressões, até aqui detalhadamente estudadas.

Começaremos pelo comando de atribuição. A figura (II.3) apresenta exemplos de tal comando.

VI.I. O COMANDO DE ATRIBUIÇÃO

Vimos até aqui, minuciosamente, os diversos tipos de dados em PASCAL que propomos implementar e seu uso em expressões. Quanto a estas, vimos que, após o percurso de suas árvores, o valor das mesmas ficará logo abaixo do topo da PRA. Vimos também que, para o caso de expressões que se resumem à referência a um record, como um todo, a variável COMPRIMENTO conterá seu tamanho. Ainda da figura (II.3) pode-se observar que o comando de atribuição é totalmente especificado. Assim, o código do nó dirá se se trata de uma atribuição de inteiro a real ou de real a inteiro, por exemplo. Desta forma, temos todas as informações de que necessitamos quanto ao lado direito das atribuições: sabemos de onde copiar o valor e seu tamanho, que estará em COMPRIMENTO se atribuição de records ou implicito no nó de comando de atribuição se atribuição de elemento simples. E quanto ao lado esquerdo? Naturalmente, não nos interessa, ao descermos em nó de referência a variável (ou ao caminharmos por conjunto de nós que formam uma referência) do lado esquerdo de uma atribuição, o valor da mesma. O que nos interessa é seu endereço. Daí a necessidade de "setarmos" um flag (a que denominaremos AT) ao descermos no nó de atribuição. Analisemos a figura (VI.1). Ali, em D1, D3 e D5 liga-se tal Flag. Por outro lado, em D2 e em D4, como estes nós são de variável simples, e o flag AT está ligado, busca-se o ende

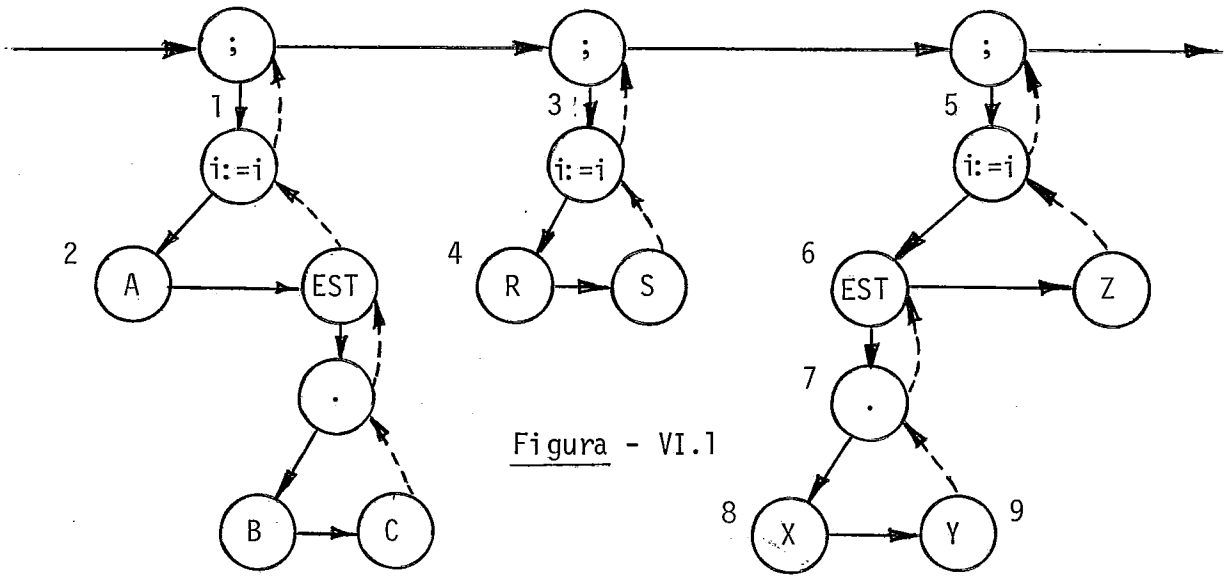


Figura - VI.1

reço da variável (A ou R), lineariza-se tal endereço e guarda se este endereço linearizado no topo da PRA. Após, desliga-se o flag AT. Já em D6, D7, D8, D9, S7 toma-se as ações usuais ainda que com o flag AT ligado. Mas, em S6, como AT vale 1, não se transfere o valor para abaixo do topo da PRA. O que lá fica é o endereço de X.Y. Após, faz-se AT valer zero.

Quando se tratar de atribuição a subrange, isto é percebido ao descer em nós como 2, 4 e 9 da figura (VI.1) (estando o flag AT ligado), por consulta à TS. Nestes casos, salva-se na variável SUBADRESS o endereço na tabela de símbolos da entrada que contém os limites: entradas de A e R nos 2 primeiros casos e de Y no terceiro. (Quando a variável a que se atribui valor é um elemento de array, verifica-se se é subrange ao descer no nó da variável array, estando AT ligada). Em qualquer caso, ao subir no nó de atribuição, sendo SUBADRESS diferente de zero, testa-se os limites: se o valor obtido estiver dentro destes, segue-se com a interpretação. Caso contrário, ela é interrompida. No caso de prosseguir a interpretação faz-se antes SUBADRESS valer zero.

A ação geral de atribuição (em S1, S3 e S5 da figura (VI.1)) consiste em copiar o valor que está logo abaixo do topo da PRA (com tamanho indicado no nó de atribuição se atribuição simples ou na variável COMPRIMENTO se atribuição de record) para o endereço guardado abaixo do valor. A localização do endereço na PRA é simples, obtida por [T - tamanho - 1]. A ação é complementada pela retirada da PRA do valor e do endereço que estão abaixo do topo da PRA. Ver figura (VI.2).

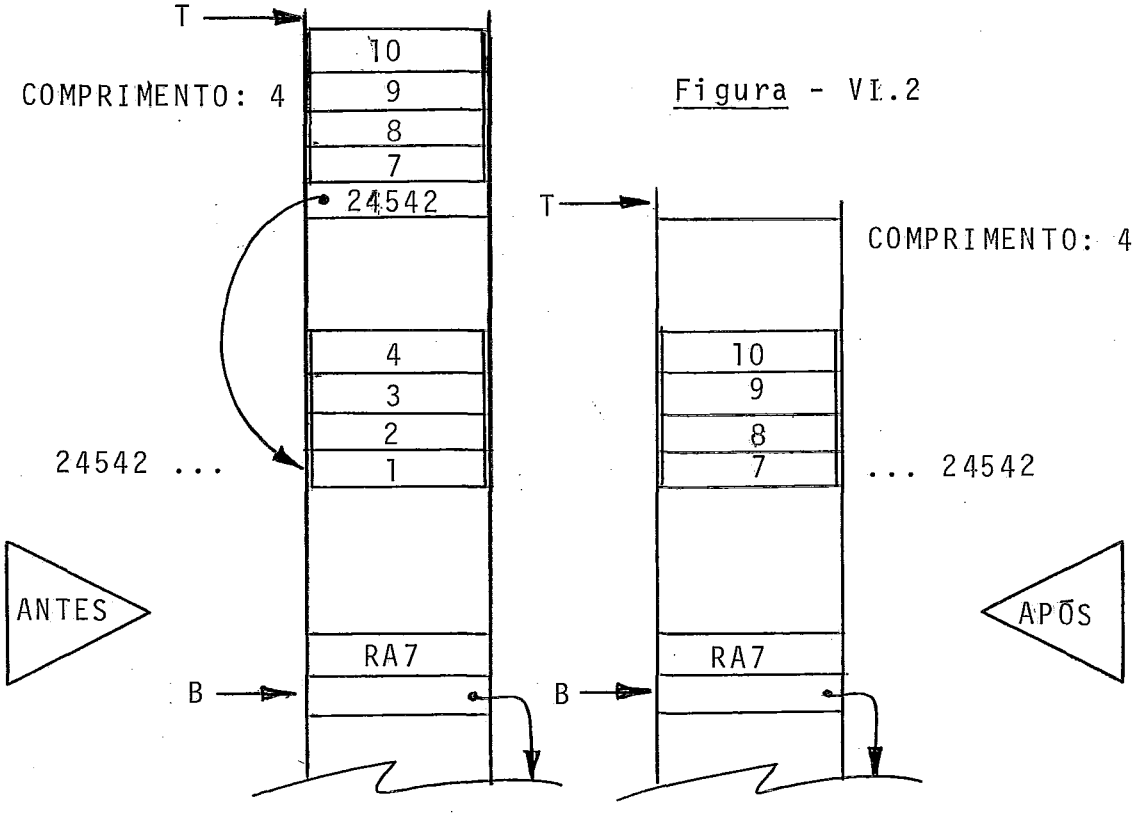


Figura - VI.2

Cumpre ressalvar que, quando se tratar de atribuição de records, existirá um nó especial (:= RC) para este fim.

OBSERVAÇÃO - A atribuição pode ter solução mais simples, dispensando flags, caso o analisador classifique nós como 2 e 6 da fig (VI.1), numa categoria especial (em atribuição).

VI.II. PSEUDO REGISTROS DE ATIVAÇÃO

Será visto adiante que o comando FOR vai requerer que se aloque, na PRA, para seu controle, espaço para três informações: o endereço da variável de controle, a constante com o valor limite e uma constante (0/1) que indicará se é um FOR-TO ou FOR-DOWNTO. Ora, é preciso que se crie um mecanismo para desalocar memória a cada desvio de um FOR para fora de seus domínios. Do contrário, se estes desvios forem numerosos, teremos a possibilidade de "estouro" da PRA. Naturalmente estamos prevendo que, ao final normal de execução de um FOR o espaço para ele alocado será devidamente desalocado. O mecanismo antes referido será por nós tratado como "pseudo-registros de ativação" ou "registros de ativação de comando FOR". Tentemos entender tal mecanismo. Quando se vai executar um comando FOR e, portanto, alocar espaço para ele, está-se executando ou o programa principal ou um procedimento ou uma função estando pois ativo o RA correspondente. O que se espera é que o espaço a ser alocado esteja nos domínios deste RA. Mas não é o que faremos. Ao adentrarmos um comando FOR, será montado um RA para ele e neste RA será alocado o espaço antes discutido. Acreditamos que um esquema visual seja a forma mais rápida de entendermos esta proposta, razão porque sugerimos consulta à figura (VI.3). Toda vez que, no processo de interpretação, descermos em nó cabeça de comando FOR, começará a montagem do pseudo RA cor

respondente: o número do pseudo RA vem do próprio nó cabeça. Quando acontecer um desvio (no meio da execução de comando FOR ou não) testa-se o número do RA (ou do pseudo RA) correspondente à definição do label (22: X:= Y;) com o número do RA (ou do pseudo RA), atualmente ativo. Se for o mesmo número nada é feito. Se o número for diferente, vai-se à tabe

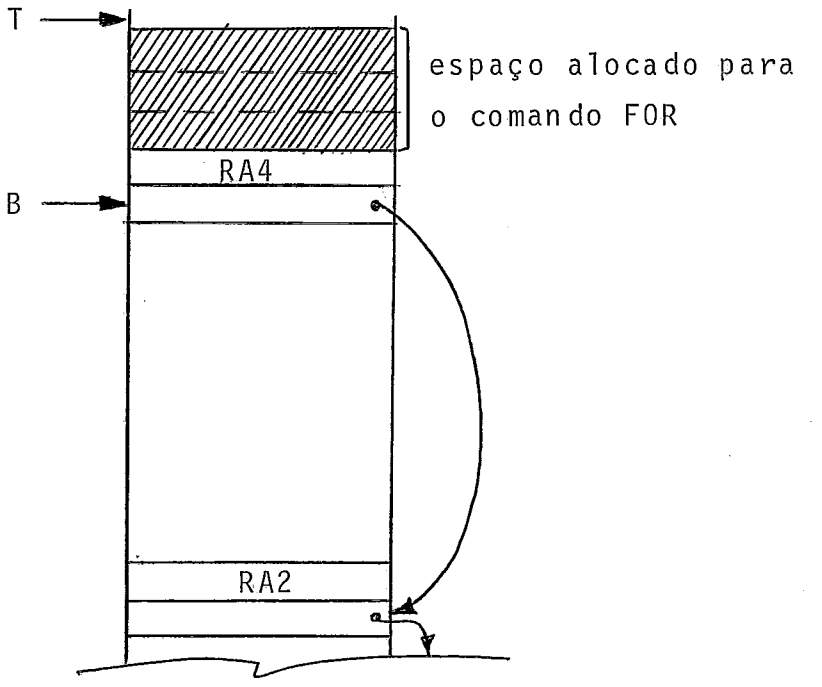


Figura - VI.3

la DISPLAY e busca-se o endereço do RA correspondente à definição do label. Então, ele é deixado ativo (os acima na PRA são desempilhados) e prossegue-se a interpretação. Este procedimento resolve, também, o desvio para fora de procedure / function. Observe-se, neste ponto, que a numeração dos RA's não é uma numeração de níveis. Ela é, na verdade, uma mera numeração sequencial, onde os números apenas identificam os RA's correspondentes. Ver figura (VI.4).

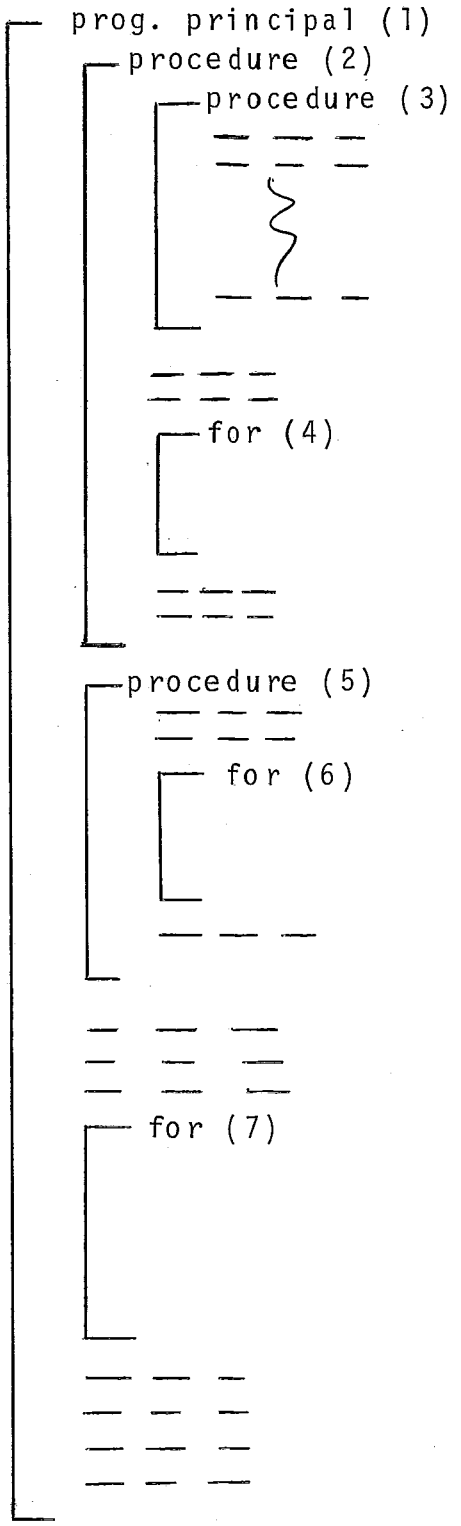

prog. principal (1)
procedure (2)
procedure (3)
for (4)
procedure (5)
for (6)
for (7)

Figura - VI.4

VI.III. O COMANDO FOR

A figura (VI.5) ilustra a FIP correspondente a um exemplo deste comando.

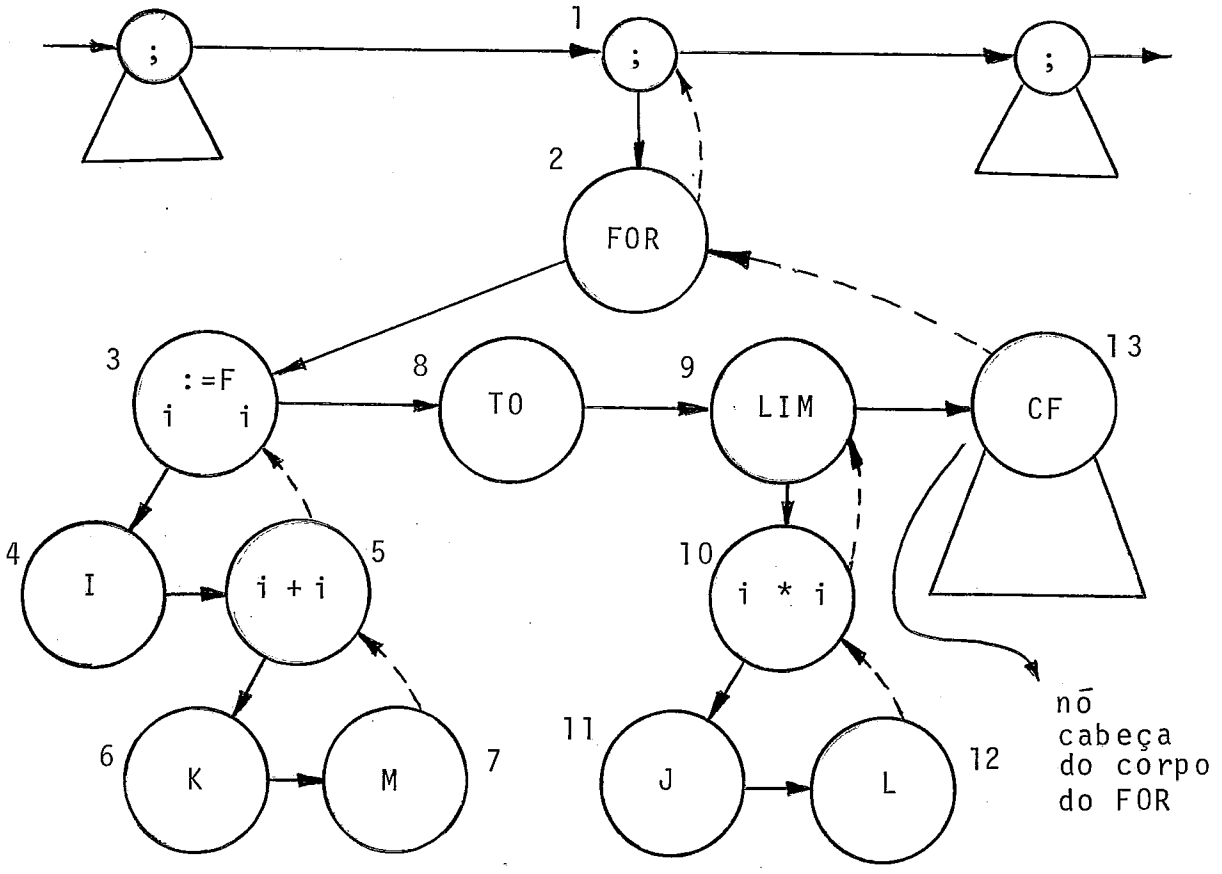


Figura - VI.5

Ações:

D1 - Ø

D2 - Começa a montagem do pseudo RA com duas entradas básicas: link e número do RA.

D3 - Faz AT valer 1.

D4 - Como AT vale 1: Resulta o endereço de I abaixo do topo da PRA. Após, faz AT valer zero.

D5, D6, D7, S5 - Resulta o valor da expressão abaixo do topo da PRA.

S3 - Esta ação é em tudo análoga à de atribuição vista, exceto que o endereço de I permanece, agora, abaixo do topo da PRA.

D8 - Empilha, na PRA, o valor 1 se TO ou o valor -1 se DOWNTO.

D9 - Ø

D10, D11, D12, S10 - Resulta o valor da expressão abaixo do topo da PRA.

Ver figura (VI.6).

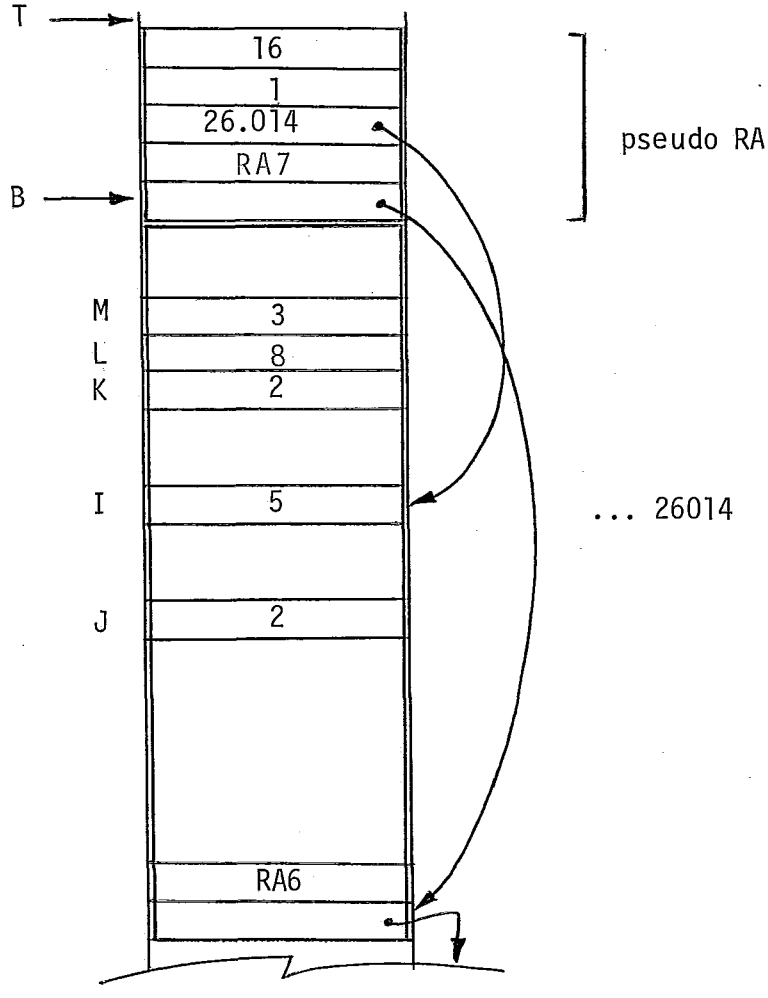


Figura - VI.6

S9 - Ø

D13 - Testa o valor inicial com o limite. Se o limite já for menor (caso de TO ou 1), segue via alinhavo (neste caso segue para ação S2). Senão, desce pela árvore cuja raiz é este nó indo executar o corpo do FOR.

S13 - Incrementa (se TO ou 1) a variável de controle I, apontada em (T - 3) da PRA. Testa a variável de controle com o limite: se a variável de controle for maior que o limite (caso de TO ou 1), então segue via alinhavo. Senão, desce pela árvore cuja raiz é este nó indo executar o corpo do FOR.

S2 - Baixa os ponteiros T e B da PRA, desempilhando assim o pseudo RA do comando FOR.

S1 - Ø

OBSERVAÇÃO - Não estamos propondo o uso de variável do tipo caráter para controle do FOR.. Se o fizéssemos, o incremento poderia ser obtido por aplicação da função SUCC, e o decremento por aplicação da função PRED, implicitamente. Ou da forma vista, transformando-se,no começo, os valores inicial e final da variável de controle, nos respectivos números de ordem.

VI.IV. O COMANDO REPEAT

A figura (VI.7) apresenta a FIP correspondente a este comando.

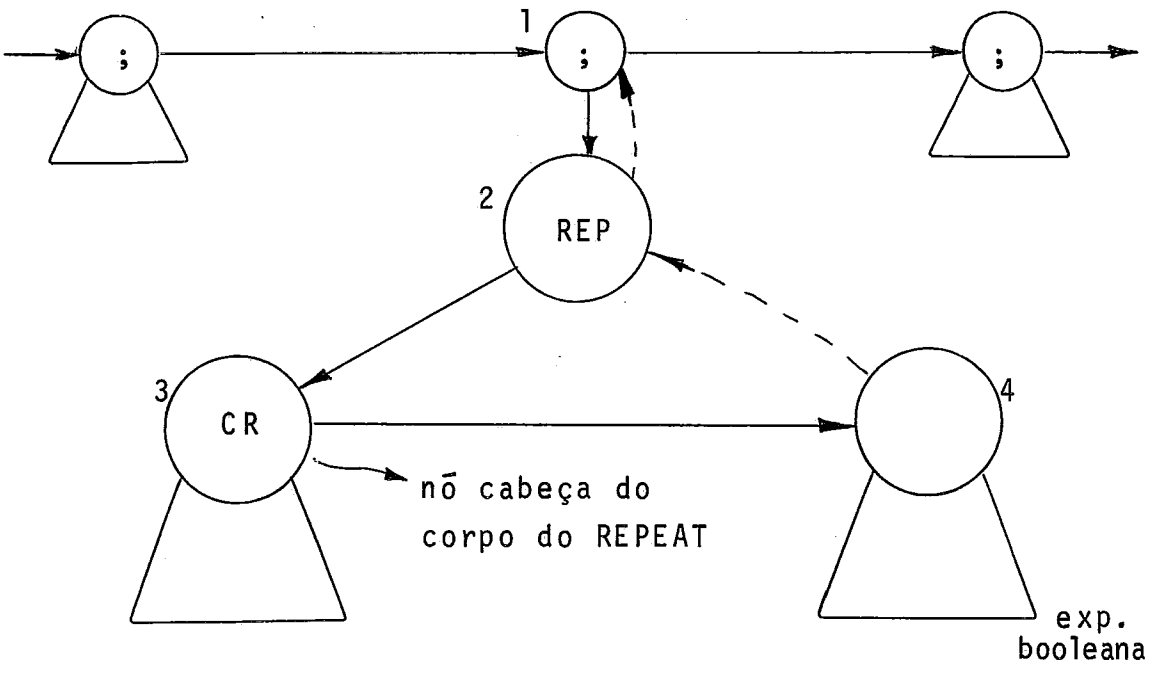


Figura - VI.7

Ações:

D1 - Ø

D2 - Ø

D3 ... S3 - executa o corpo do REPEAT (D3 e S3 : Ø).

D4 ... S4 - avalia a expressão booleana. Seu resultado (0 ou 1) ficará logo abaixo do topo da PRA.

S2 - Testa o resultado booleano em (T - 1) da PRA:

Se o valor for 1 (true), desempilha o resultado booleano e segue via alinhavo.

Se o valor for 0 (false), desempilha o resultado boolea

no e segue para o nõ filho.

S1 - Ø

VI.V. O COMANDO WHILE

A figura (VI.8) apresenta a FIP correspondente a este comando.

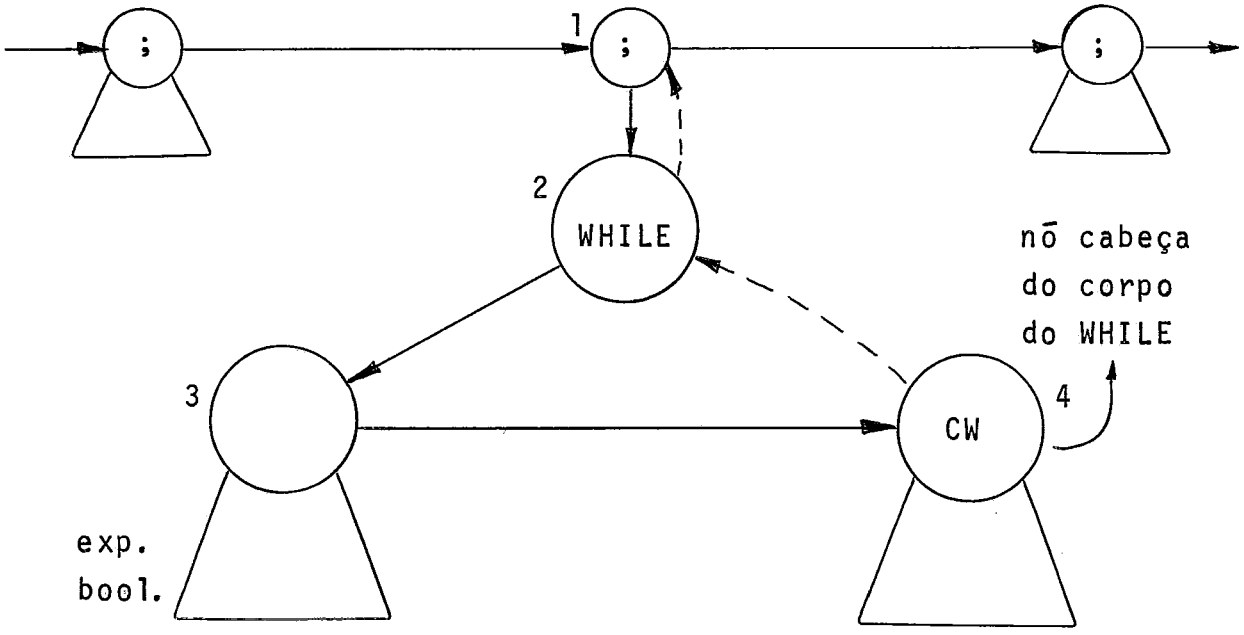


Figura - VI.8

Ações:

D1 - Ø

D2 - Ø

D3 ... S3 - Avalia a expressão booleana. Seu resultado (0 ou 1) ficará logo abaixo do topo da PRA.

D4 - Testa o resultado booleano em (T - 1) da PRA:

Se o resultado valer zero (false) segue Via alinhavo para o nó WHILE.

Se o resultado valer 1 (true), desempilha este resultado e desce para o nó filho, indo percorrer o Corpo do WHILE.

S4 - Empilha novamente o valor 1 na PRA e segue, via alinhavo, para o nó WHILE.

S2 - Testa o valor em (T - 1) da PRA:

Se for zero, desempilha-o e segue via alinhavo.

Se for 1, desempilha-o e desce para o nó filho.

S1 - Ø

VI.VI. O COMANDO CASE

A figura (VI.9) apresenta a FIP correspondente a este comando.

Observe-se que o nó OF aponta para uma tabela onde estarão os labels e ponteiros. Ao descer no nó OF, pesquisa-se a tabela apontada e, achando-se o label, desvia-se para o nó indicado. Não se achando o label, segue-se via alinhavo do nó OF.

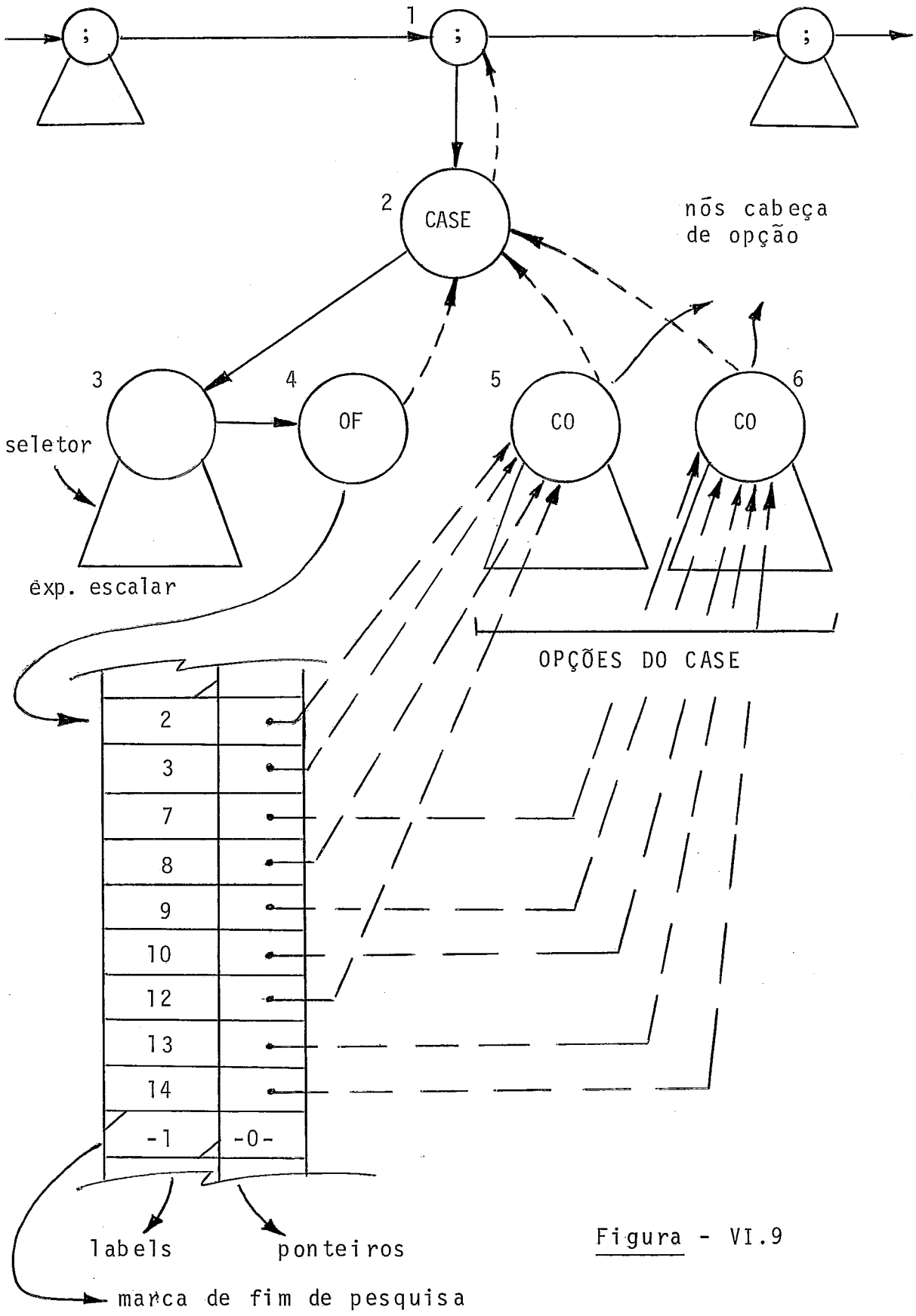

;
1
;
;
2
CASE
nós cabeça
de opção
3
4
OF
5
CO
6
CO
seletor
exp. escalar
OPÇÕES DO CASE
2
3
7
8
9
10
12
13
14
-1
-0-
labels
ponteiros
marca de fim de pesquisa

Figura - VI.9

Ações:

D1 - Ø

D2 - Ø

D3 ... S3 - Avalia a expressão escalar. Seu resultado ficará em (T - 1) da PRA.

D4 - Desvia para a tabela de CASE's e de lá para a árvore de opção correspondente (ou sobe de volta, via alinhavo, para o nó 2, cabeça do CASE). Em qualquer caso, desempilha antes o resultado escalar.

<<Percorre a árvore selecionada como opção do CASE, sendo as ações ao descer e subir no nó cabeça de opção vazias; após, segue via alinhavo para o nó cabeça do CASE.>>

S2 - Ø

S1 - Ø

VI.VII. O COMANDO IF - THEN

A figura (V.10) apresenta a FIP correspondente a este comando.

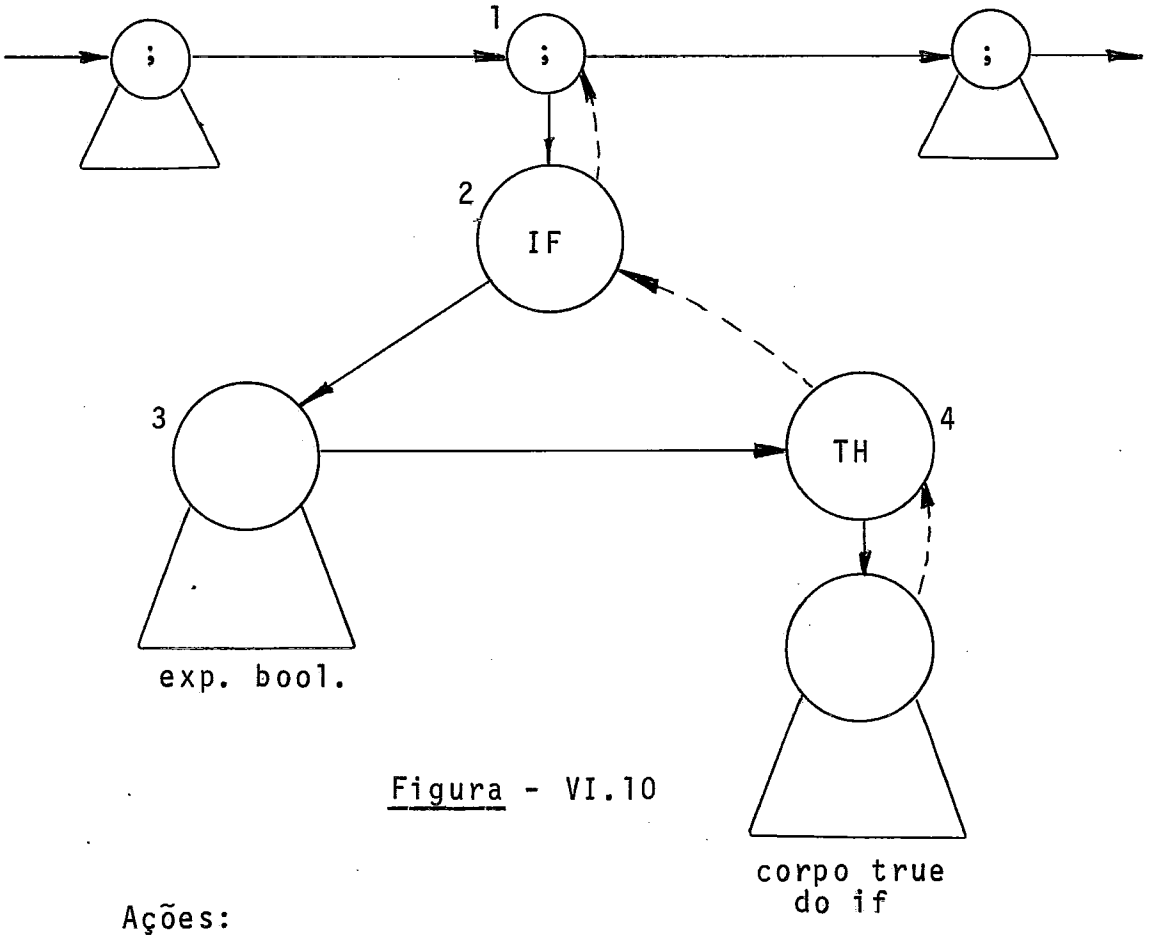


Figura - VI.10

Ações:

D1 - Ø

D2 - Ø

D3 ... S3 - Avalia a expressão booleana. Seu resultado ficará em (T - 1) da PRA.

D4 - Testa o resultado booleano em (T - 1) da PRA:

Se o valor for 1 (true), segue para nó filho, indo executar o Corpo True do IF, antes desempilhando aquele valor da PRA.

Se o valor for zero (false) segue via LINK2 (no caso, alinhavo), antes desempilhando aquele valor da PRA.

S4 - Segue via LINK2 (no caso, alinhavo).

S2 - Ø

S1 - Ø

VI.VIII. O COMANDO IF - THEN - ELSE

A figura (VI.11) apresenta a FIP correspondente a este comando.

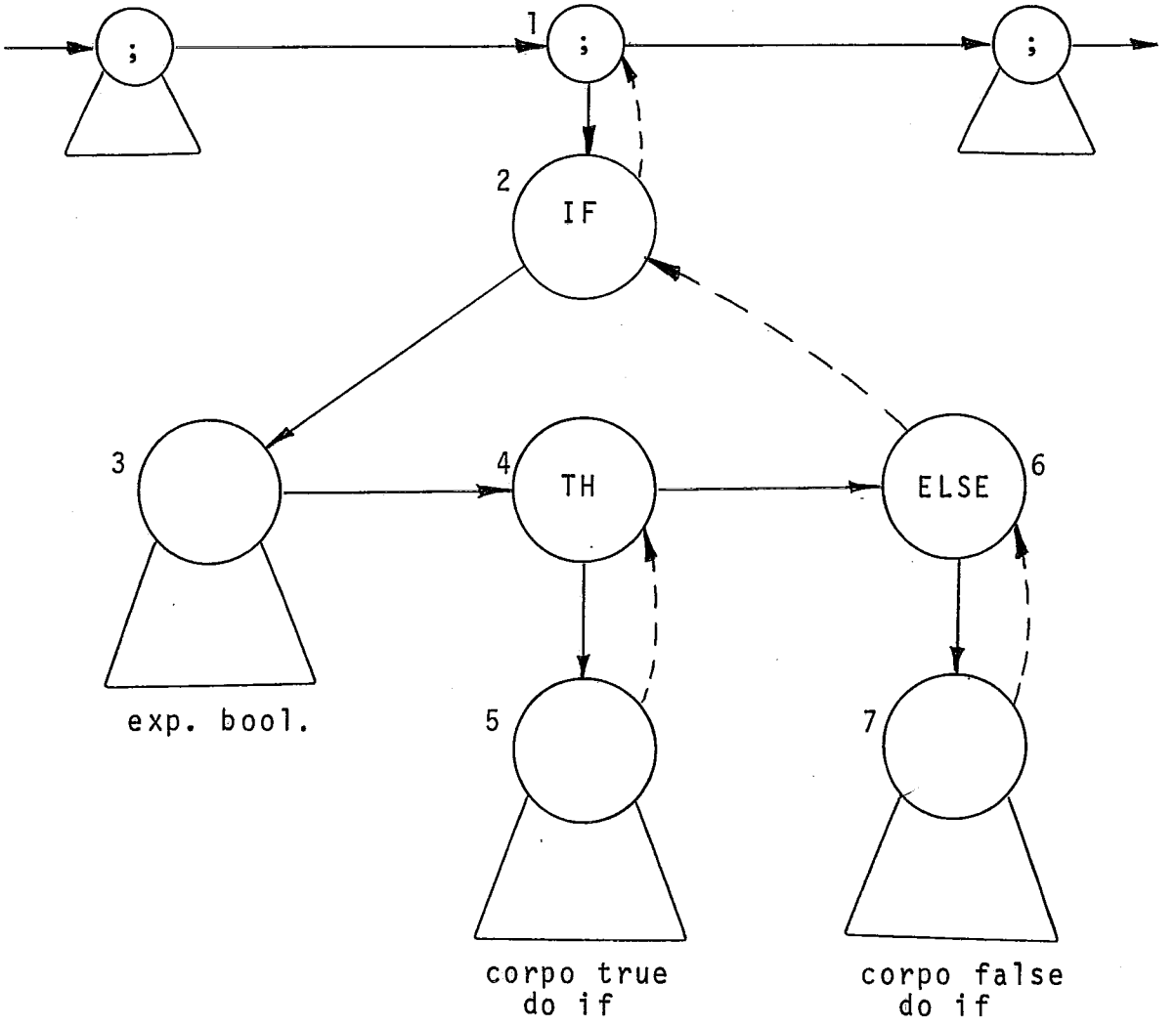


Figura - VI.11

Ações:

D1 - Ø

D2 - Ø

D3 ... S3 - Avalia a expressão booleana. Seu resultado ficará em (T - 1) da PRA.

D4 - Testa o resultado booleano em (T - 1) da PRA:
Se o valor for 1(true), segue para o nó filho, indo executar o Corpo True do IF, antes desempilhando aquele valor da PRA.
Se o valor for zero (false), segue para o nó irmão.

D5 ... S5 - executa o Corpo True do IF, se for o caso.

S4 - Empilha, novamente, o valor 1 na PRA.

D6 - Testa o resultado booleano em (T - 1) da PRA:
Se o valor for 1 (true), segue, via alinhavo, para o nó IF, antes desempilhando aquele valor da PRA.
Se o valor for zero (false), desempilha-o da PRA e desce para o nó filho, indo executar o Corpo False do IF.

D7 ... S7 - executa o Corpo False do IF, se for o caso.

S6 - Segue, via alinhavo, para o nó IF.

S2 - Ø

S1 - Ø

VI.IX. O COMANDO GOTO

A figura (VI.12) apresenta a FIP correspondente a este comando.

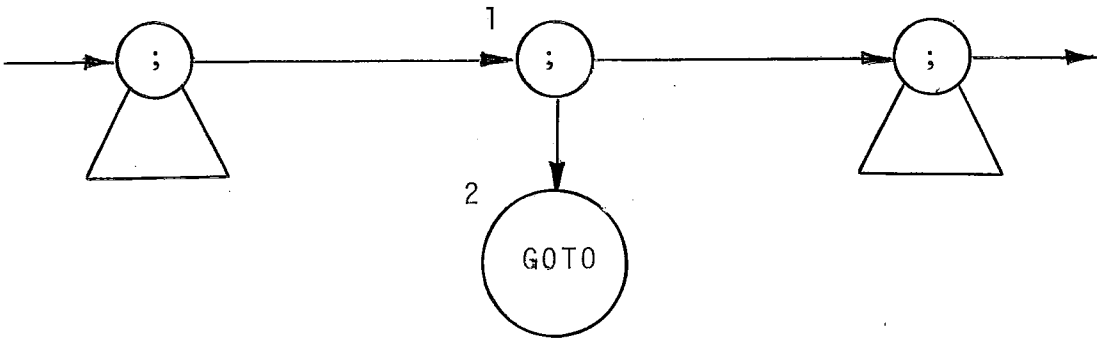


Figura - VI.12

O nó do comando GOTO conterá, como informação, o endereço do label de desvio. A ação ao descer neste nó resume-se a uma simples ramificação no percurso da árvore.

Ações:

D1 - Ø

D2 - Desvia.

VI.X. OS COMANDOS DE ENTRADA E SAÍDA

Temos constatado ser comum a primeira versão de compiladores PASCAL restringir sua entrada/saída ao tipo textfile. Na verdade, não chega a ser esta opção uma restrição grande, ainda mais se considerarmos a possibilidade de conversão automática de tipo, permitida. Assim, julgamos oportuno con

siderar, para implementação da versão inicial de nosso compilador, exclusivamente este tipo. No âmbito deste trabalho, trataremos em particular dos textfiles input e output, suficientes para a confecção de programas simples.

Um textfile é, basicamente, um "file of char" , ou seja, um arquivo de caracteres, subestruturado em linhas através de caracteres especiais denominados "line markers". Estes são gravados em arquivos por uma procedure denominada WRITELN. A figura (VI.13) apresenta a estruturação de um textfile em fita magnética gerado por um programa PASCAL. Ali se observa a existência de uma área auxiliar que é para onde cada registro é lido antes de ser transferido, caráter a caráter, para as variáveis correspondentes. Caso o arquivo fosse de saída existiria também uma área auxiliar na qual a linha é montada, caráter a caráter, antes de sua gravação. Na mesma figura se observa, também, a existência da chamada "janela", que nos possibilita "ver" ou "passar" um caráter de cada vez. Embora os registros tenham tamanho fixo, o final de cada conjunto de caracteres nos registros é assinalado

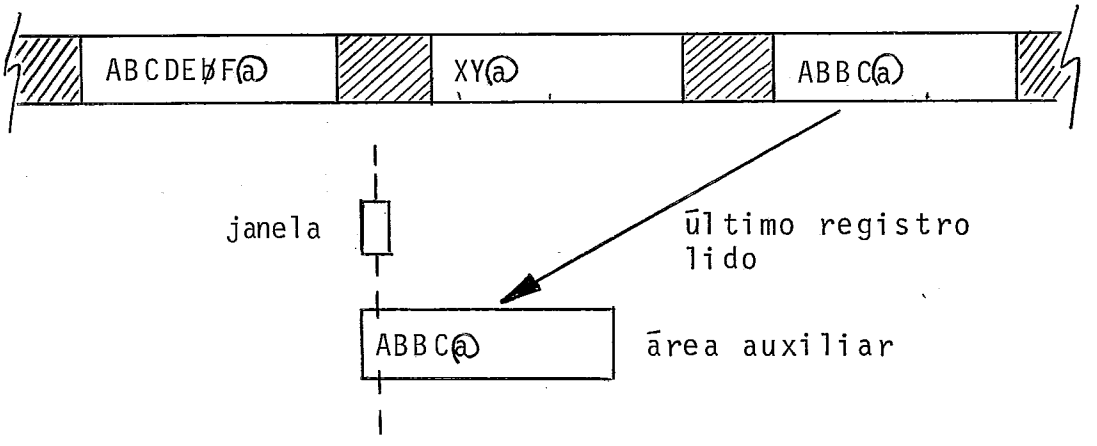
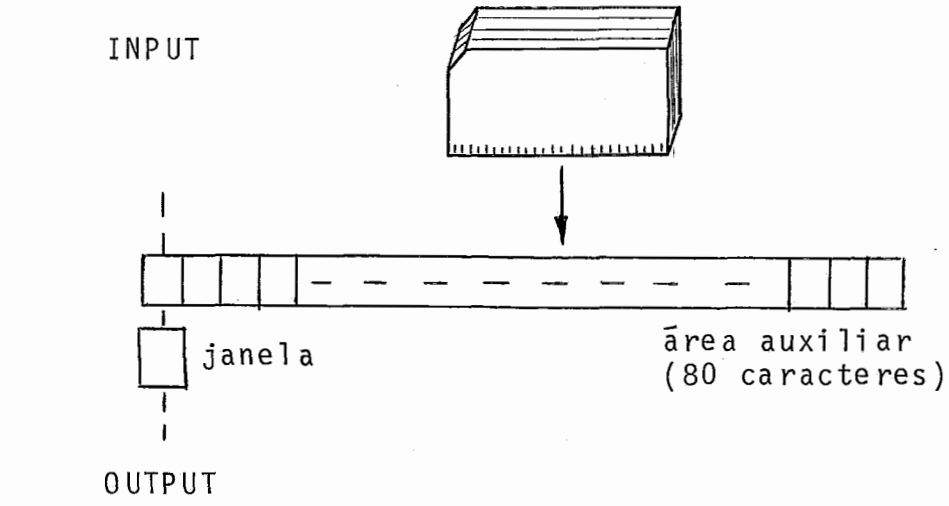


Figura - VI.13

por um "line marker", representado no exemplo por "@". O line marker otimiza a pesquisa na área auxiliar: atingido o caráter separador de linhas, após o comando READ (este comando não lê o registro físico, mas transfere o caráter da área auxiliar para sua variável) a função EOLN passa a retornar true. Um próximo comando READLN tornará EOLN false, alimentará um novo registro na área auxiliar e deixará a janela ligada ao primeiro caráter desta área. Quanto à função EOF, retornará true após uma tentativa de leitura (READLN) em que for detetado fim de arquivo.

A figura (VI.14) ilustra a estruturação dos arquivos input e output. Em nosso trabalho, estamos associando o arquivo input à leitora de cartões e o arquivo output à impressora. Adotaremos, para este caso, um line marker implícito: após ser transferido o 80º caráter, no READ, EOLN passará a retornar a true, não havendo marca no cartão; ao comando WRITELN será impressa uma linha, mas não será obviamente impressa qualquer marca. Da figura (VI.14) pode-se observar a existência de uma área auxiliar para o arquivo input, com capacidade para 80 caracteres e de outra para o arquivo output, com capacidade para 132 caracteres.



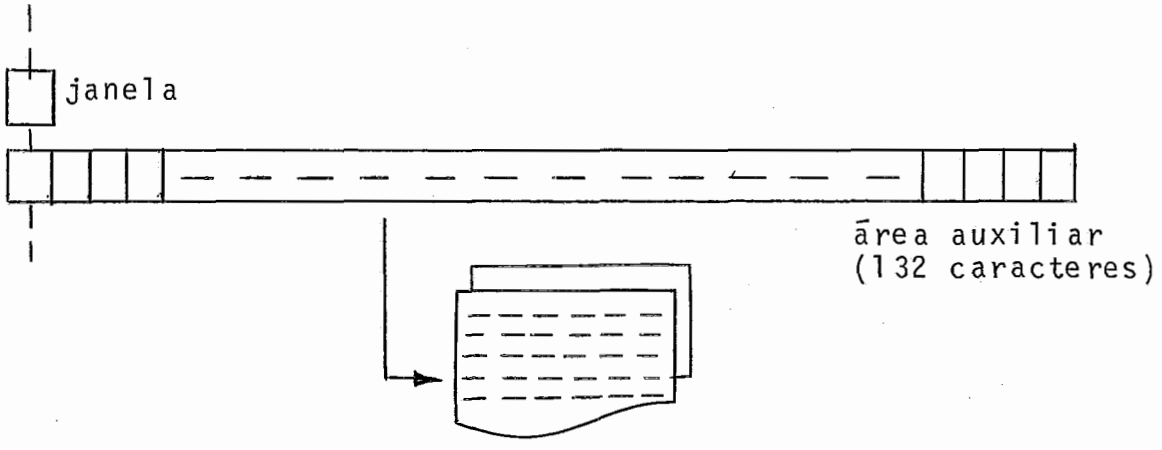


Figura - VI.14

Será adotado o seguinte conjunto de procedures e functions (sendo CH uma variável do tipo caráter):

(1) WRITE (CH) que é equivalente a WRITE (OUTPUT, CH)

(2) READ (CH) que é equivalente a READ (INPUT, CH)

(3) WRITELN que é equivalente a WRITELN (OUTPUT)

(4) READLN que é equivalente a READLN (INPUT)

(5) EOF que é equivalente a EOF (INPUT)

(6) EOLN que é equivalente a EOLN (INPUT)

. A primeira permite transferir um caráter da variável CH para a janela e desta para uma posição da área auxiliar.

. A segunda permite transferir para a variável CH, da janela, um caráter, avançar a janela para o próximo caráter da área auxiliar, atribuindo este caráter à janela. Se não existe tal caráter (a janela está na posição de um separador de linhas, mas conterá branco) então EOLN passa a devolver true.

. A terceira permite imprimir uma linha (132 caracteres) armazenada na área auxiliar, limpando esta área depois.

. A quarta permite a leitura de um novo cartão para a área auxiliar.

. A quinta é tornada true após uma tentativa de leitura (READLN) em que for detetado fim de arquivo.

. A sexta é tornada true após ser transferido o 80º caráter da área auxiliar, através do comando READ.

OBSERVAÇÃO 1

A presença do nó INPUT, na FIP, provocará uma primeira leitura de registro (cartão) para a área auxiliar, com capacidade para 80 caracteres, posicionando a janela no primeiro caráter. Depois, através do comando READ, são os caracteres transferidos da área auxiliar para as variáveis correspondentes. Finalmente, através do comando READLN, um novo registro (cartão) é lido, posicionando-se à janela no primeiro caráter.

OBSERVAÇÃO 2

Como não estamos propondo referências do tipo f↑, onde f é uma variável file, a janela não precisa ser implementada. Assim, por exemplo, na descrição do comando WRITE (CH) poderíamos dizer que o caráter em CH é transferido direto para a área auxiliar.

A seguir, apresentaremos exemplos dos comandos e, após, serão mostradas as árvores correspondentes. Opcionalmente, a linguagem PASCAL permite, também, além da leitura/gravação de caracteres, a conversão automática de tipo, para arquivos textfile. No caso de comando de gravação (WRITE) pode-se usar ou não formatação. Os exemplos que se seguem consideram, também, a conversão automática de tipos. Ressalve-se ainda que os comandos READ e WRITE podem referenciar diversas variáveis, como uma facilidade, embora a idéia básica do READ seja a leitura de um caráter.

EXEMPLO 1

READ (X,Y,C) X é inteira
Y é real
C é caráter

Será buscada da área auxiliar uma sequência de caracteres que forme um número inteiro, outra que forme um número real e outra constituida de um só caráter, sendo os valores transferidos para as variáveis X, Y e C (X e Y convertidos). Brancos são saltados e, por definição, também o são separadores de linhas, o que no caso corresponde a ler novo cartão se parte das va

riáveis não estiver no primeiro (quando a janela corresponder à posição após a 80ª da área auxiliar e ainda houver o que ler).

EXEMPLO 2

```
WRITE (X, Y, C, B, 'ABC');   X é inteira
                             Y é real
                             C é caráter
                             B é booleana
                             'ABC' é um string de caracteres.
```

Neste caso, as variáveis X, Y, C e B terão seus valores (X,Y e B convertidos) armazenados na área auxiliar, seguidos do string, com comprimentos padronizados. Sugestão: 8 para inteiro, 16 para real, 8 para booleano, 1 para caráter e 16 para strings).

EXEMPLO 3

```
WRITE (X:4, Y:10:2, C:6, B:8, 'ABC');
```

Este comando transfere para a área auxiliar de saída, com capacidade para 132 caracteres, obedecendo à formatação especificada, os valores de X, Y, C e B (convertidos) e o string 'ABC'(com conversão para caracteres):

. X é inteira e terá seu valor convertido armazenado com 4 caracteres, com brancos à esquerda.

. Y é real e terá seu valor convertido armazenado com 10 caracteres, sendo duas casas decimais, com brancos à esquerda.

. C é caráter e terá seu valor armazenado com 5 brancos à esquerda.

. B é booleano e terá seu valor convertido (TRUE ou FALSE) ar

mazenado como ØØØØTRUE ou ØØØFALSE.

. 'ABC' é um string e será impresso com 13 brancos à esquerda.

EXEMPLO 4

READLN

Um novo cartão é alimentado.

EXEMPLO 5

WRITELN

Uma nova linha é impressa.

Feitas estas considerações, apresentaremos agora a proposta de FIP para cada caso.

READLN

A figura (VI.15) apresenta a árvore correspondente.

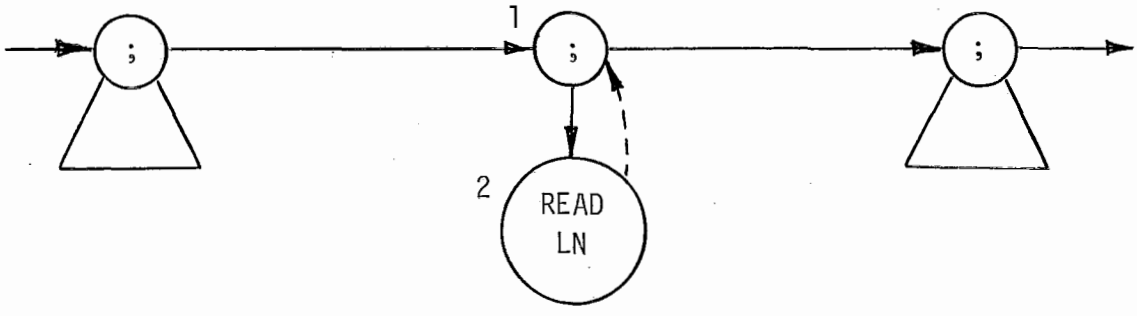


Figura - VI.15

Destaque-se que em D2 simplesmente lê-se um novo cartão para a área auxiliar de leitura.

WRITELN

A figura (VI.16) apresenta a árvore correspondente.

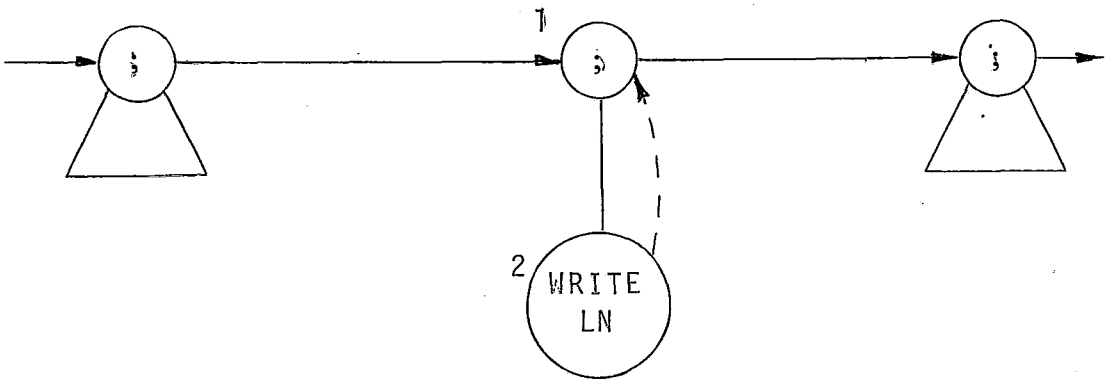


Figura - VI.16

Destaque-se que em D2 simplesmente imprime-se o conteúdo da área auxiliar de saída.

READ

A figura (VI.17) apresenta a árvore correspondente a um exemplo. Os nós P_r (read parameter) assinalam os tipos dos valores a ler (os tipos das variáveis).

Antes de descrevermos as ações no percurso da árvore exemplo, cabe uma observação. Mencionamos anteriormente que as procedures pré-definidas em PASCAL serão executadas à medida que se vai percorrendo os nós de parâmetros. A vantagem desta escolha pode ser vista na procedure READ. Tal procedure exige conversão de valores e, portanto, que se conheça seus tipos. Ora: se agimos à medida que subimos nos nós P_r, os ti

pos são conhecidos passo a passo. Mas, se alternativamente, escolhessemos apenas agir ao subir no nó que designa a procedure, seria necessário que se guardasse, por parâmetro, seu tipo para uso posterior. Esta segunda opção nos parece mais elegante, mas a primeira, que propomos, é mais eficiente.

Ações:

D1 - Ø

D2 - Faz PARAM valer 2.

D3 - Ø

D4 - Como PARAM vale 2, busca e lineariza o endereço de X, empilhando-o na PRA (não transfere o valor).

S3 - Com base no endereço de X, armazenado no passo anterior, e no seu tipo (inteiro), que está neste nó, percorre a área auxiliar à procura de uma sequência de dígitos precedida ou não de sinal, calcula o número inteiro correspondente, armazenando-o naquele endereço.

D5 - Ø

D6 ... S6 - Como PARAM vale 2:

Resulta o endereço de p↑.A [I+J] abaixo do topo da PRA.

S5 - Com base no endereço de p↑.A [I+J] , armazenado no passo anterior, e no seu tipo (caráter), que está neste nó, busca da área auxiliar o próximo caráter, armazenando-o naquele endereço.

S2 - Faz PARAM valer zero.

S1 - Ø

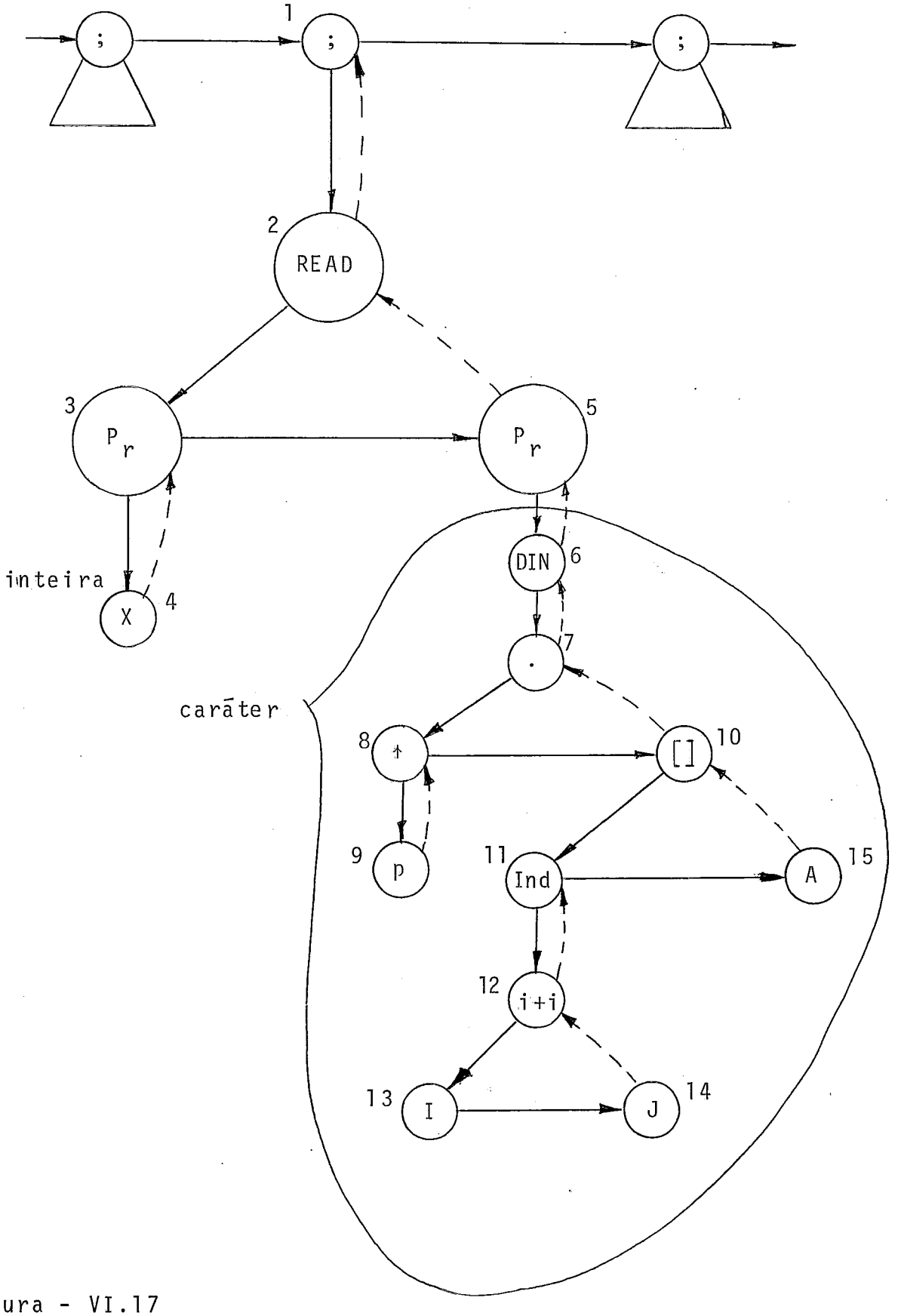

;
1
;
;
2
READ
3
Pr
5
Pr
inteira
X
4
DIN
6
.
7
caráter
8
↑
[]
10
9
p
11
Ind
A
15
12
i+i
13
I
J
14

Figura - VI.17

OBSERVAÇÃO 1

Se, ao subir em nó P_r, o final da área auxiliar da entrada for atingido (genericamente por deteção de line-marker, particularmente por "estouro" das 80 posições), automaticamente um novo registro será lido (READLN implícito) e a pesquisa continuará.

OBSERVAÇÃO 2

A procedure READLN pode ter dupla função: ela pode permitir a transferência de valores da área auxiliar para as respectivas variáveis e, ao final, o avanço de um registro. A árvore para este caso é análoga à do READ. Ao subir no nó READLN, um novo cartão é lido para a área auxiliar. No exemplo da figura (VI.15) lemos um novo cartão ao descer no nó READLN porque este não possuia filho.

WRITE

A figura (VI.18) apresenta a árvore correspondente ao exemplo:

WRITE (X + Y, C, E:14:3);

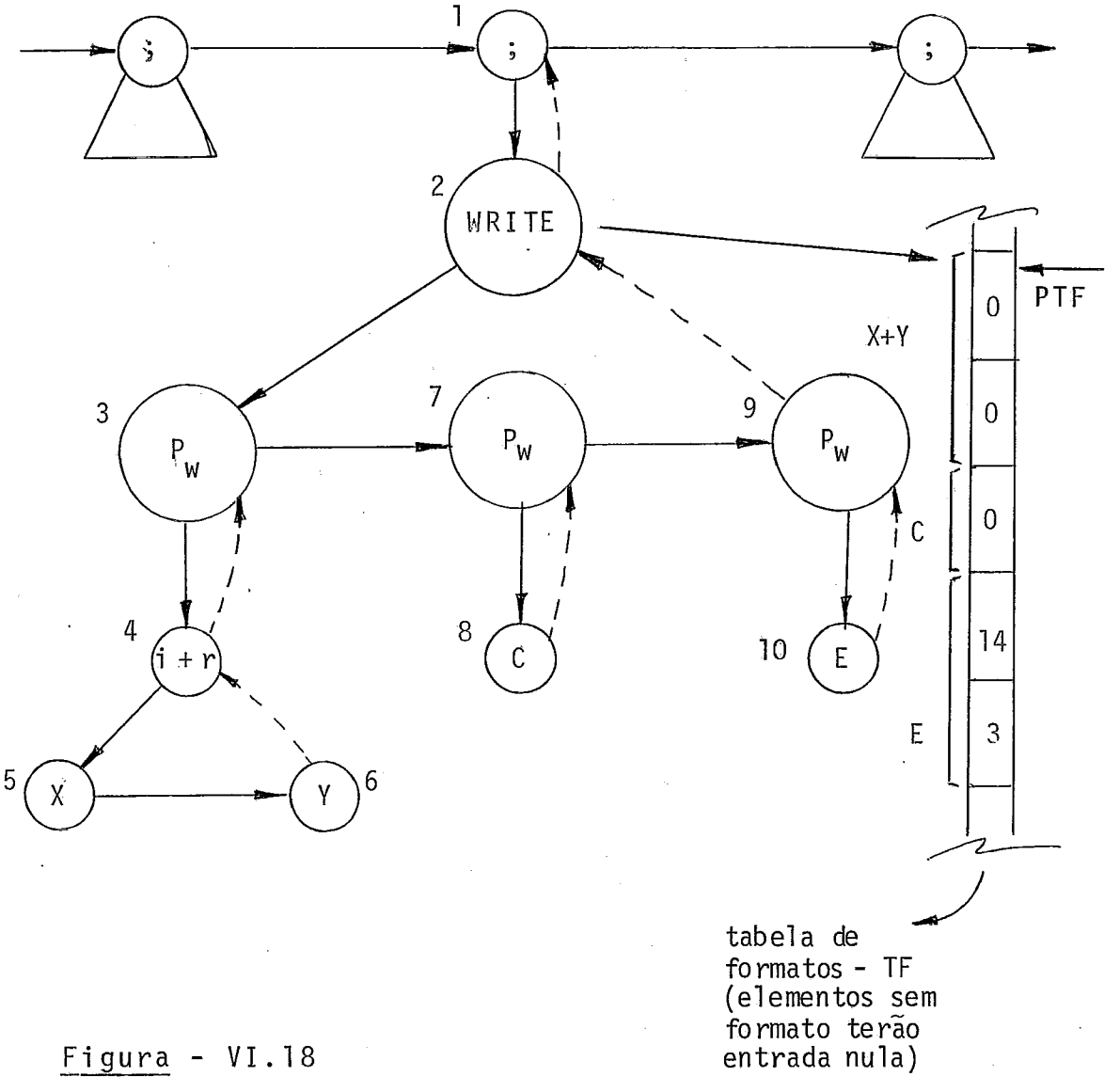


Figura - VI.18

Os nós 3, 7 e 9 (write parameters) assinalam os tipos das respectivas expressões.

Ações:

D1 - Ø

D2 - Faz PTF apontar para a Tabela de Formatos.

D3 - Ø

D4, D5, D6, S4 - Avalia a expressão, deixando o resultado (real) abaixo do topo da PRA.

S3 - Converte para a forma de caracteres o valor da expressão em função do tipo assinalado neste nó (real, no caso) e transfere o valor convertido para a área auxiliar (ocupando, neste caso, por ser real, sem formato, 16 caracteres, com brancos à esquerda). Após, incrementa PTF.

D7 - Ø

D8 - Resulta o caráter em questão abaixo do topo da PRA.

S7 - Transfere o caráter para a área auxiliar (onde ocupará uma posição por não haver formato). Após, incrementa PTF.

D9 - Ø

D10 - Resulta o valor de E abaixo do topo da PRA.

S9 - Análogo a S3, com o valor ocupando 14 posições, com 3 casas decimais, devido ao formato.

S2 - Ø

S1 - Ø

OBSERVAÇÃO 1

Para o caso de impressão de strings, ao descer no nó do string, que aponta para a tabela de constantes, o string é transferido para abaixo do topo da PRA, sendo seu tamanho, que está também naquela tabela, armazenado na variável COMPRIMENTO.

Ao subir no nó P_W, é feita a transferência do string para a área auxiliar, caráter a caráter, em função do tipo (string) e do tamanho.

OBSERVAÇÃO 2

Se, ao subir em nó P_W, esgotar-se a área auxiliar, uma linha será impressa (WRITELN implícito), limpar-se-á a área auxiliar, recomeçando-se a preenchê-la. Um valor nunca será "quebrado", sendo colocado no início da linha seguinte.

OBSERVAÇÃO 3

Também com o WRITELN pode-se transferir valores. Seria o caso de, na árvore da figura (VI.18), o nó 2 ser substituido pelo nó WRITELN. Neste caso, as ações seriam as mesmas, exceto que, em S2, seria impressa uma linha.

VI.XI. CHAMADA DE PROCEDURE DO USUÁRIO

A figura (VI.21) apresenta a FIP correspondente a uma chamada de procedure.

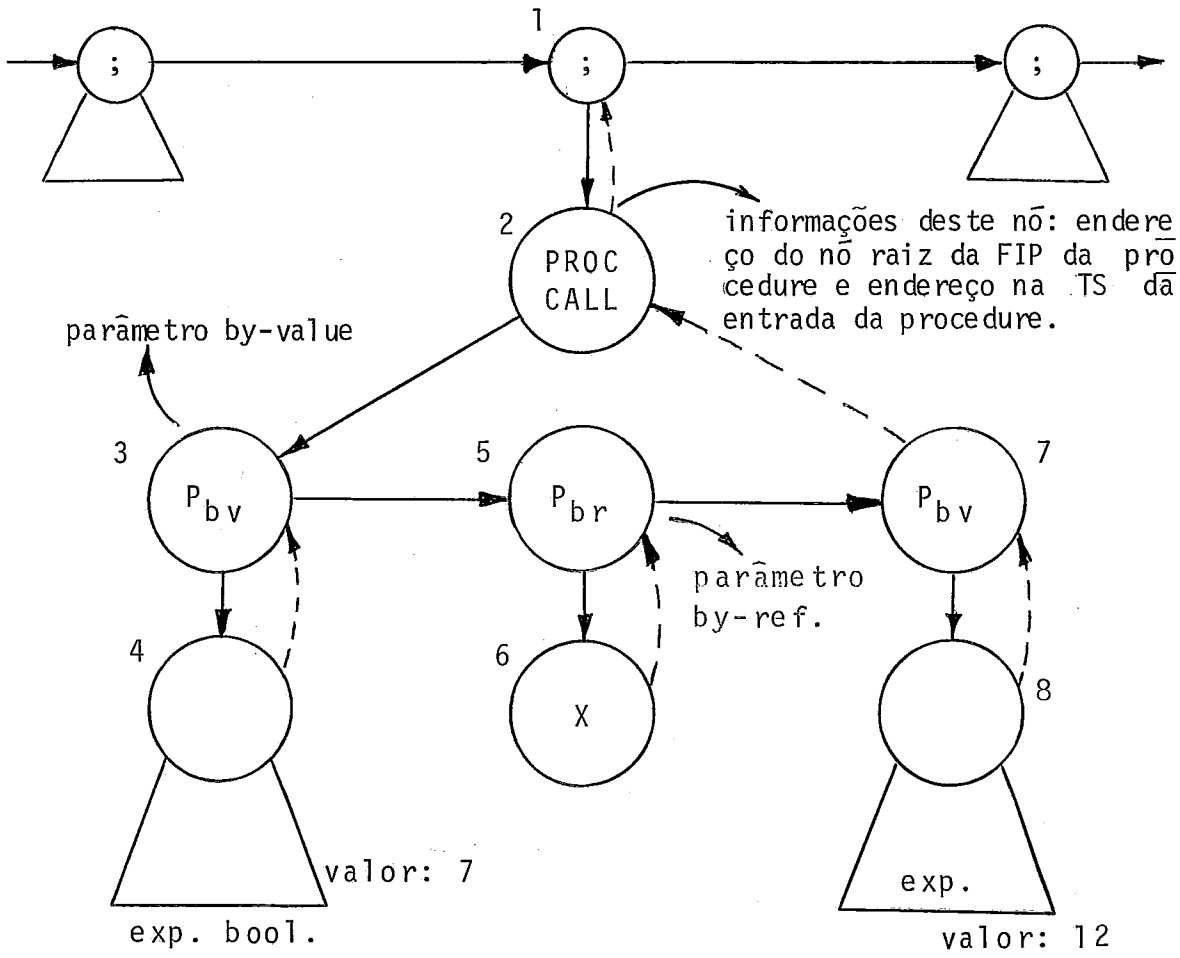


Figura - VI.21

Ações:

D1 - Ø

D2 - Começa a montagem do RA da procedure, colocando o ponteiro para o RA anterior e o número deste RA (2, por exemplo), buscado da TS. Atualiza ainda a tabela DISPLAY (marca a última entrada deste RA).

Ver figura (VI.22).

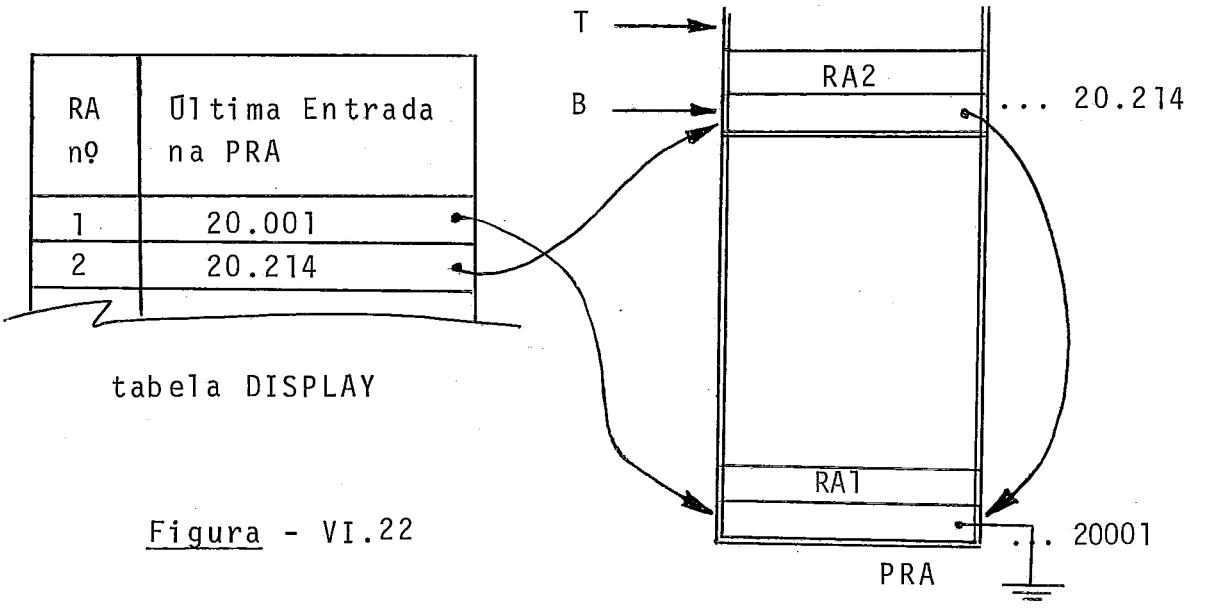


Figura - VI.22

Em seguida, coloca na próxima entrada disponível no RA que está sendo montado o endereço de retorno que é o conteúdo do segundo apontador deste nó (LINK2).

Ver figura (VI.23).

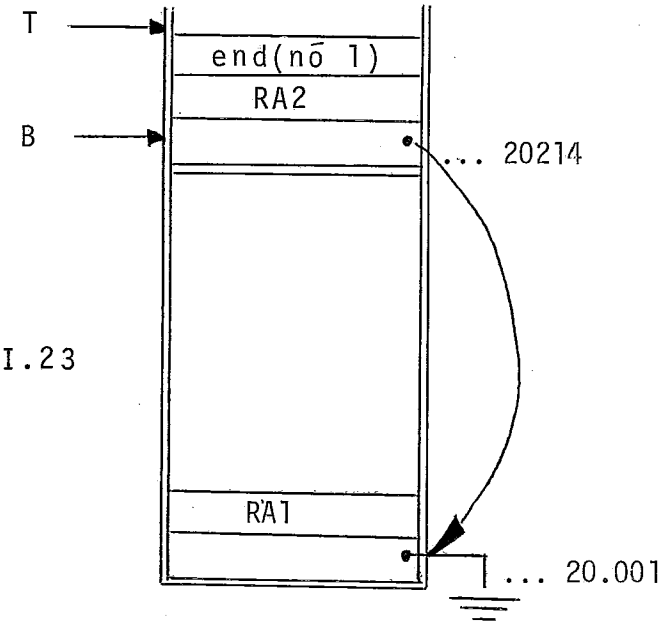


Figura - VI.23

D3 - Ø

D4 ... S4 - A expressão é avaliada, ficando seu resultado logo abaixo do topo da PRA.
Ou seja: a transferência by-value foi feita automaticamente.

Ver figura (VI.24).

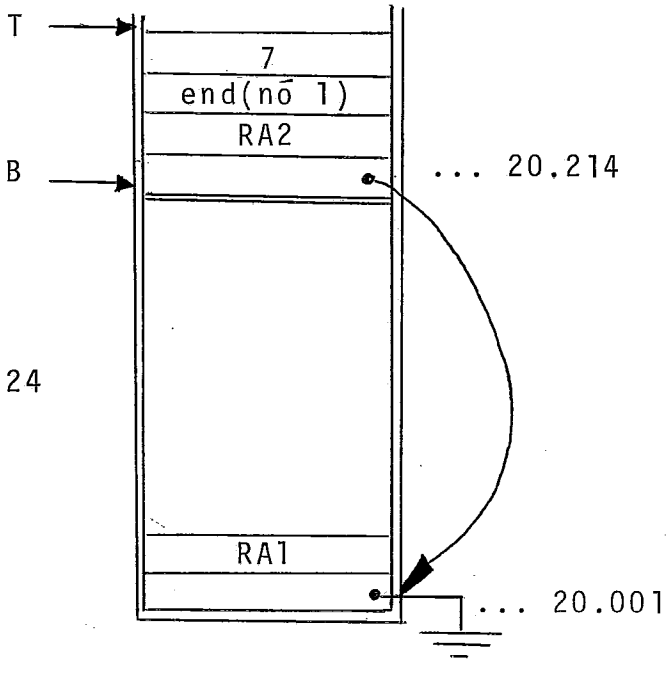


Figura - VI.24

S3 - Ø

D5 - Faz PREF valer 1.

D6 - Como PREF vale 1:

Busca o endereço de X na TS, lineariza-o na PRA, guardando-o na próxima entrada livre do RA que está sendo montado.

S5 - Faz PREF valer zero.

A figura (VI.25) mostra a situação neste ponto. Observe-se que, no corpo da procedure, referências à variável correspondente ao parâmetro X passado aqui do programa principal, terão endereço indireto. Ou seja: encontrado, no percurso da árvore da procedure chamada, um nó de variável-parâmetro by-reference, lineariza-se o endereço na PRA e neste busca-se o endereço efetivo do dado.

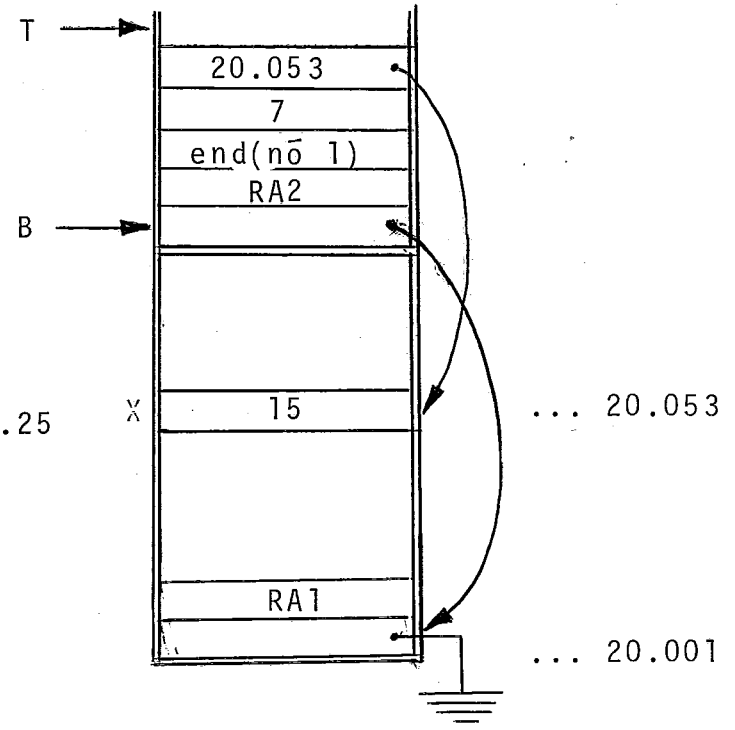


Figura - VI.25

D7, D8 ... S8, S7 - Análogo a D3, D4 ... S4, S3.

S2 - Vái à TS, na entrada da procedure, e de lá busca o tamanho do RA da mesma. Com base neste tamanho, sobe o topo da PRA.

Ver figura (VI.26).

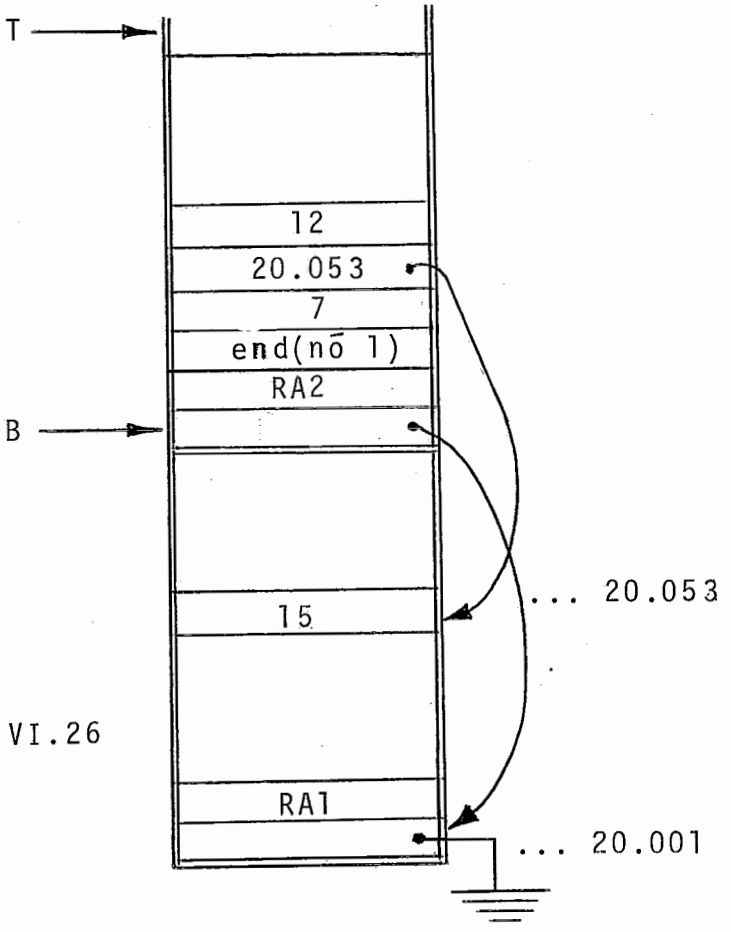


Figura - VI.26

Finalmente, desvia, descendo, para o nó raiz da subárvore da procedure, cujo endereço está registrado neste nó 2, antes fazendo CALLSUB valer 1 (a razão deste procedimento será explicada adiante).

OBSERVAÇÃO

Especial atenção deve ser tomada na passagem de parâmetros by-reference. Assim, pode acontecer de uma procedure chamar outra passando, by-reference, uma variável-parâmetro que já fora passada do programa principal, por exemplo, by-reference. Naturalmente, não se vão gerar vários níveis de indireção o que seria deficiente. O problema deve ser resolvido na chamada da procedure. Assim, feito PREF valer 1, se, após, descermos em nó de variável-parâmetro by-reference, busca-se o endereço efetivo do seu valor e é este o endereço que é armazenado no topo da PRA.

VI.XII. EXECUÇÃO DE PROCEDURE DO USUÁRIO

A figura (VI.27) esquematiza a árvore de uma procedure (neste ponto julgamos oportuno que se reveja a figura (II.1) deste trabalho, para que se visualize onde esta árvore de declaração de procedure se situa na árvore geral do programa).

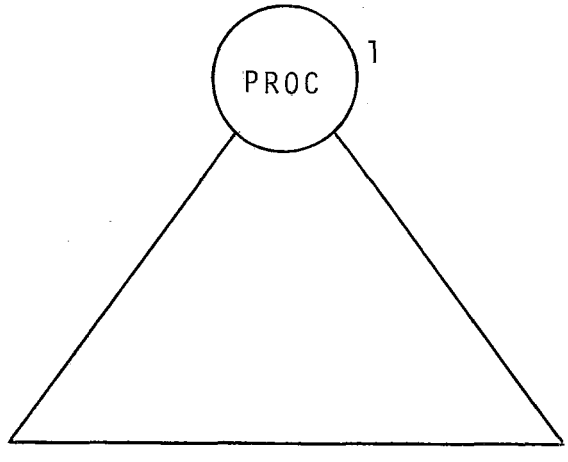


Figura - VI.27

O percurso da árvore de uma procedure só deve ocorrer quando de sua chamada. Quem faz esta seleção é o flag CALLSUB. Assim, em D1 da figura (VI.27), se CALLSUB valor zero, segue-se para o nó indicado pelo segundo apontador (LINK2) deste nó. Mas, se CALLSUB valer 1, faz-se CALLSUB valer zero e segue-se para o nó filho (endereçado por LINK1), indo percorrer a árvore da procedure. Desta forma evita-se, por exemplo, que ao início do programa sáia-se percorrendo árvores de procedures.

Como já foi visto, haverá necessidade de se distinguir duas categorias de nós de variáveis:

. aquelas cujo endereço é direto;
. aquelas cujo endereço é indireto (a variável é um parâmetro formal by-reference).

A distinção será feita pelos próprios códigos de operação dos nós.

No percurso da árvore, o que há a destacar é o cuidado na distinção entre estas duas classes de nós.

Ao final, baixam-se os ponteiros T e B da PRA e segue-se, subindo, para o nó cujo endereço está em (T + 2) da PRA.

VI.XIII. CHAMADA DE FUNCTION DO USUÁRIO

A chamada de uma function ocorre dentro de uma expressão. O que se espera é que, ao voltar da function, o resultado da mesma esteja em (T-1) da PRA. Espera-se ainda que o controle volte para o nó indicado pelo segundo apontador (LINK2) do nó de chamada da function.

A figura (VI.28) ilustra a FIP de uma expressão na qual foi feita a chamada de uma function.

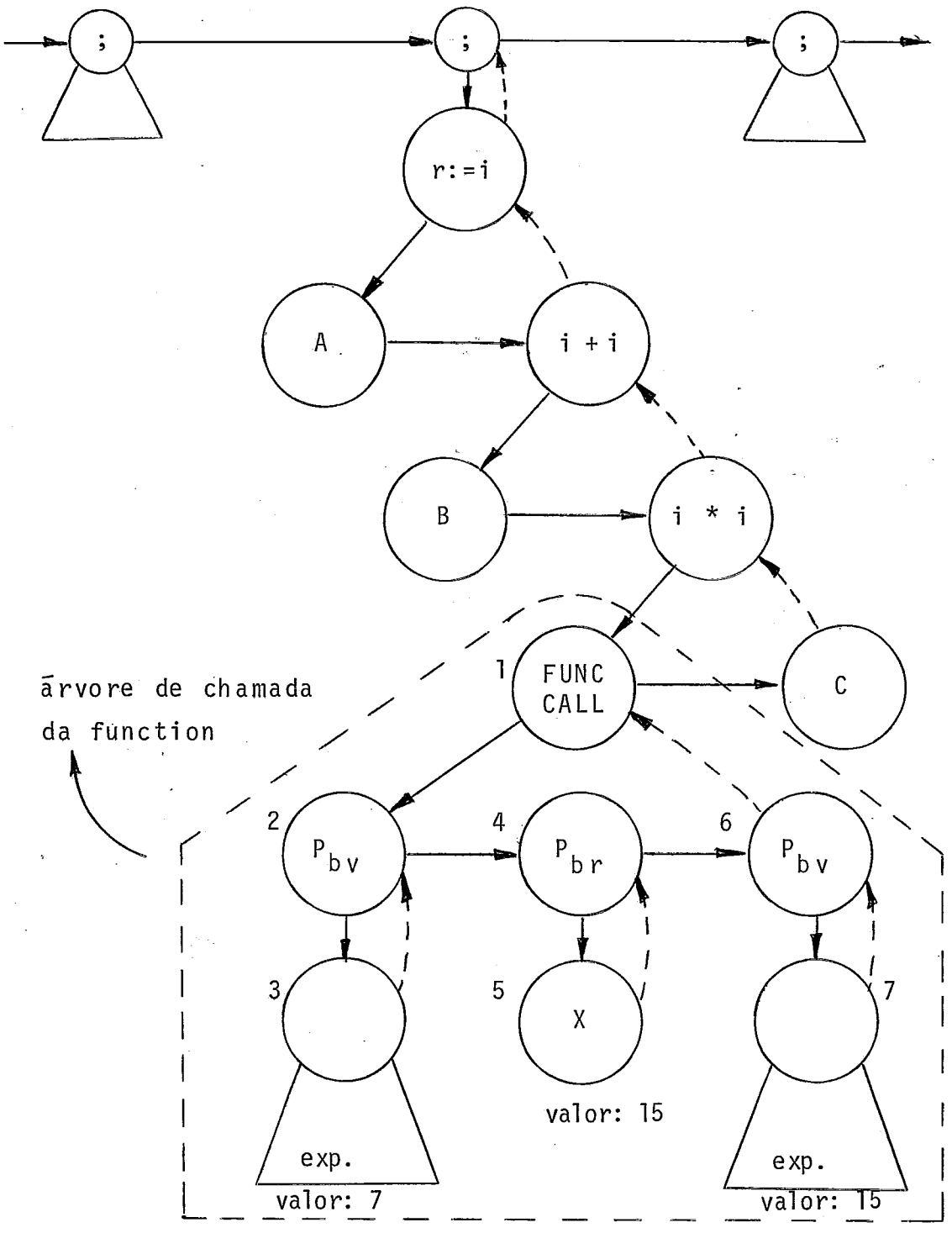


Figura - VI.28

O percurso da árvore que tem por raiz o nó 1 é análogo ao percurso da árvore de chamada de procedure, visto. A diferença está na ação ao descer no nó FUNCCALL, em que, além dos procedimentos da ação ao descer no nó PROCCALL, visto, reserva-se a quarta entrada do RA que está sendo montado para conter o resultado.

A figura (VI.29) mostra como fica a PRA ao final do percurso da árvore da figura (VI.28) cuja raiz é o nó 1.

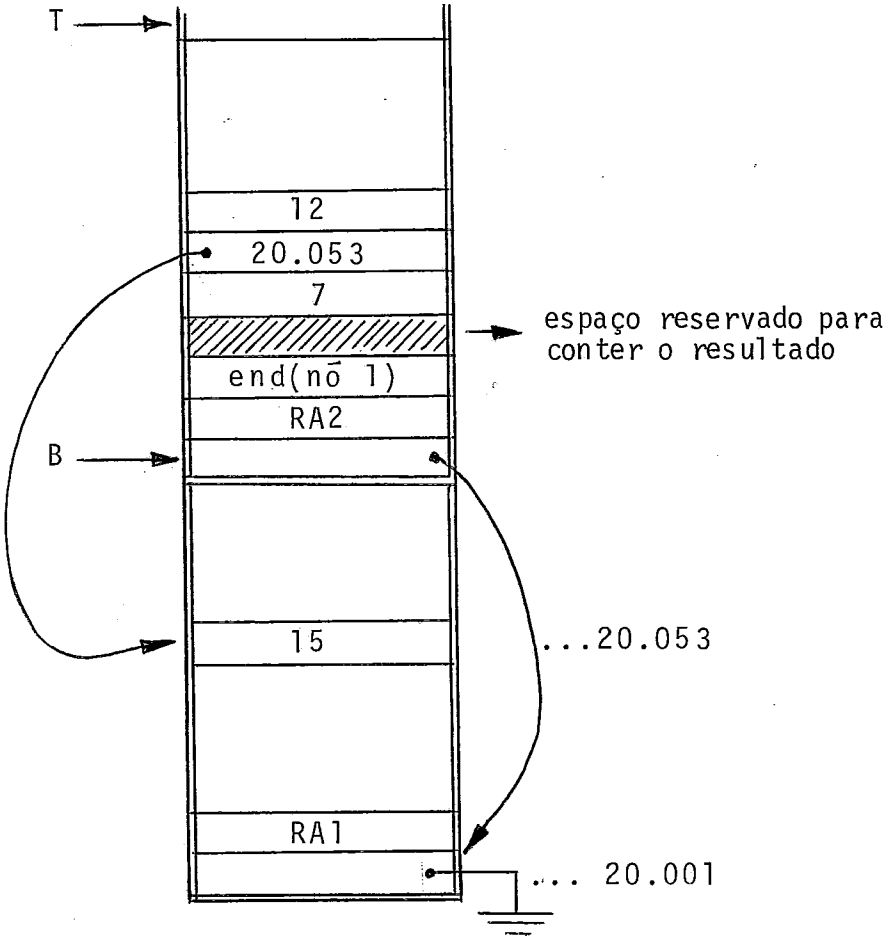


Figura - VI.29

VI.XIV. EXECUÇÃO DE FUNCTION DO USUÁRIO

A figura (VI.30) esquematiza a árvore de uma function.

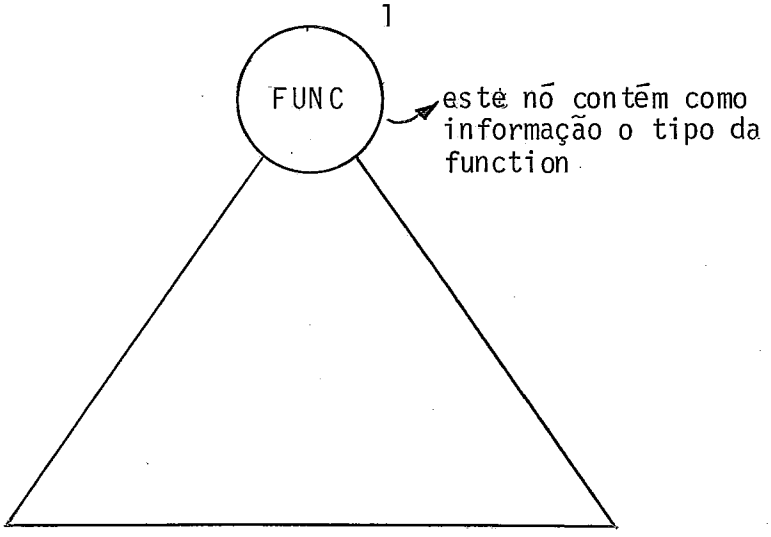


Figura - VI.30

Cabem aqui as mesmas observações já feitas para a árvore de procedure.

Quanto a ação S1, há diferenças:

. Baixa-se os ponteiro T e B;

. Salva-se o valor em (T + 2) da PRA;

. Transfere-se o valor em (T + 3) para T da PRA e incrementa-se T levando em conta o comprimento, que é função do tipo de function (o tipo de function vem indicado neste nó).

. Segue-se para o nó de retorno:

- Descendo, se o valor salvo for positivo, sendo este valor o endereço de ramificação.
- Subindo, se o valor salvo for negativo, sendo o módulo deste valor o endereço de ramificação.

VI.XV. PASSAGEM DE PARÂMETROS ESTRUTURADOS

A passagem de um record by value se dá naturalmente. Basta que se lembre os procedimentos já vistos em expressões, em que se menciona a existência de uma variável de nome COMPRIMENTO, com base na qual records inteiros eram transferidos para o topo da PRA. (Com base nisto, faz-se a atribuição de records).

Vejamos, agora, o caso de arrays. A figura (VI.31) ilustra um exemplo. O nó 5 tem código diferente dos nós de array até aqui vistos. Estes dizem respeito a referências a elementos; aquele diz respeito a uma referência ao array, como um todo. Em D5, apenas salva-se em COMPRIMENTO o tamanho do array. Em S4, ele é normalmente transferido para o topo da PRA, em função de COMPRIMENTO. Com isto, a passagem do array by value se dará naturalmente.

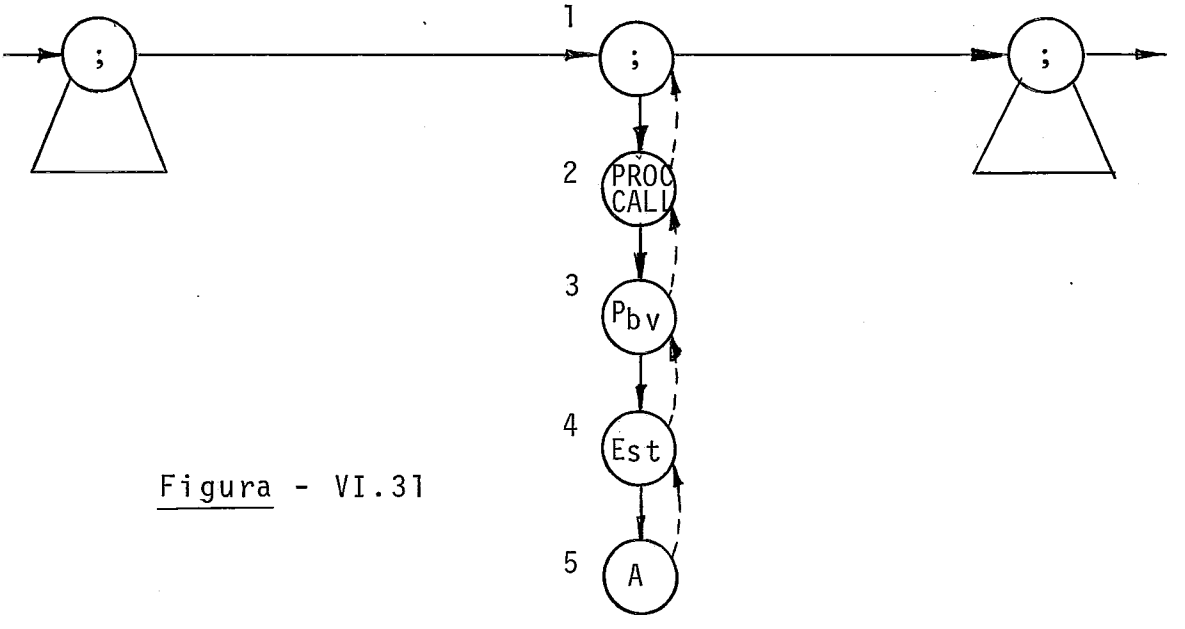


Figura - VI.31

CAPÍTULO VII

VII. RECURSOS DE DEPURAÇÃO

Estarão disponíveis no módulo interpretador/depurador PASCAL quatro classes de recursos de depuração: DUMP, TRACE, TRACEBACK e STATISTICS. As três primeiras podem ser solicitadas de forma condicional.

Haverá um relatório próprio para depuração. Ou seja, informações para depuração não sairão intercaladas com relatório do programa.

Através do DUMP, poderá o usuário solicitar a impressão de dados elementares (estáticos ou dinâmicos) na forma [nome, valor] em pontos específicos do programa. Um recurso alternativo, a que denominamos DUMPHEAP, permitirá a impressão de todos os dados dinâmicos.

Através do TRACE, poderá o usuário acompanhar o comportamento de certos elementos do programa, como, por exemplo, o de uma determinada variável, observando suas mudanças de valor.

Através do TRACEBACK, poderá o usuário solicitar a história de elementos escolhidos do programa.

Todos os recursos mencionados podem ser requeridos de forma condicional ou não. Tais recursos poderão ser extendidos pelo uso de um comando FOR, com auxílio do qual poderão ser solicitadas tais facilidades para estruturas completas ou parciais.

Finalmente, através de STATISTICS, algumas apurações, quanto à execução do programa, poderão ser solicitadas.

A forma de especificação dos comandos de depuração é simples e foi projetada de modo a permitir portabilidade para o programa fonte. Assim, os comandos de depuração são escritos nos moldes de comentário especial, começado por "(**" e terminado por "**)". Deste modo, o programa escrito para "rodar" no Ambiente de Programação Pascal poderá ser traduzido por outro compilador Pascal, quando os comandos de depuração serão ignorados, considerados como comentários.

VII.I. AS QUATRO CLASSES DE RECURSOS DE DEPURAÇÃO DISPONÍVEIS

A seguir discutiremos, ilustradas por exemplos simples, as quatro classes de recursos de depuração disponíveis.

VII.I.I. DUMP

A figura (VII.1) apresenta o esquema de um módulo no qual foram inseridos comandos de depuração da classe 1 (entenda-se por módulo uma procedure, uma function ou o programa principal).

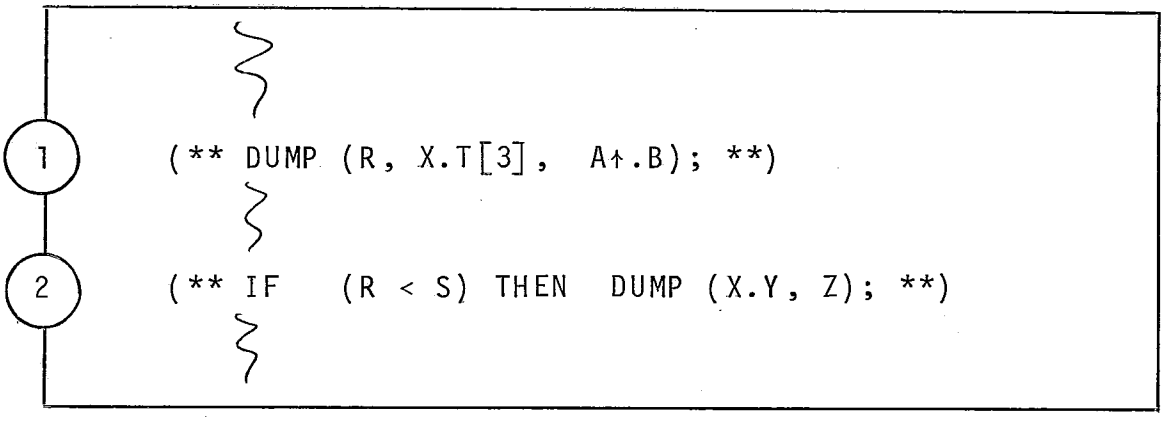


Figura - VII.1

Na linha 1 é solicitado um Dump incondicional de R, de X.T[3] e de A↑.B. Ou seja: sempre que o programa passar por este ponto serão impressos os valores de R, de X.T[3] e de A↑.B, acompanhados dos respectivos nomes.

Na linha 2 é solicitado um Dump condicional. Ele só acontecerá se, quando o programa passar por aquele ponto, a condição testada (R < S) resultar verdadeira.

VII.I.II. CLASSE 2: TRACE

A figura (VII.2) apresenta o esquema de um módulo no qual foram inseridos comandos de depuração da classe 2, para rastreio incondicional de variáveis.

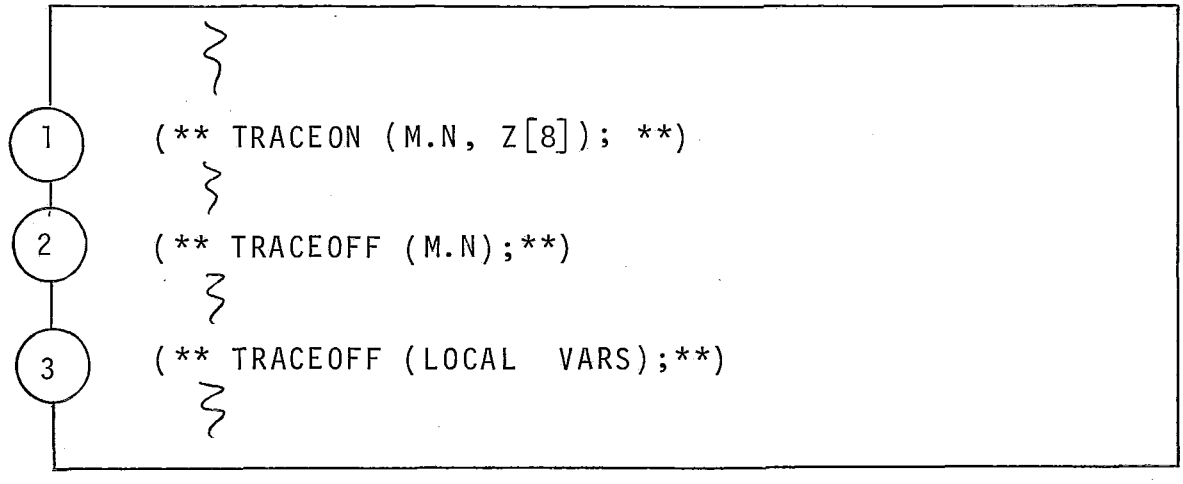


Figura - VII.2

A partir do instante em que o programa passar pela linha 1 estará ligado o trace para M.N e Z[8]. Isto significa que, sempre que elas receberem um valor por atribuição ou leitura, terão seus nomes e novos valores impressos. O trace de M.N será desligado quando o programa passar pela linha 2, vindo por qualquer caminho; na linha 3 é desligado o trace de todas as variáveis locais, o que desligará o trace de Z[8], caso Z seja local. Fica, assim, entendido ser possível solicitar o trace (o mesmo vale para o dump) para variáveis não-locais.

A figura (VII.3) apresenta o esquema de um módulo no qual foram inseridos comandos de depuração de classe 2 para rastreio de variáveis sujeito a condição IF.

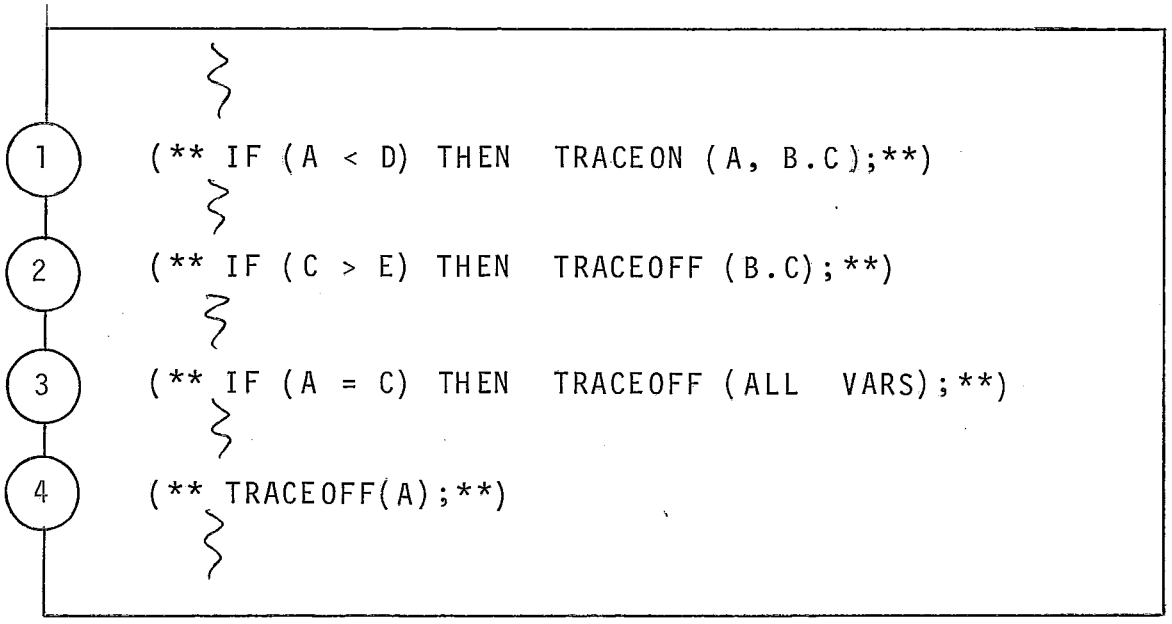


Figura - VII.3

A partir do instante em que o programa passar pela linha 1, A e B.C estarão sob trace, caso a condição(A < D) seja verdadeira naquele ponto. A partir do instante em que o programa passar pela linha 2, B.C terá seu trace suspenso, caso a condição(C > E)seja verdadeira naquele ponto. No instante em que o programa passar pela linha 3, estará suspenso o trace de todas as variáveis, caso a condição(A = C)seja verdadeira naquele ponto. A partir do instante em que o programa passar pela linha 4 estará suspenso, incondicionalmente, o trace da variável A.

A figura (VII.4) apresenta o esquema de um módulo em que foi inserido um comando de depuração da classe 2 para rastreio de variáveis sujeiro a condição WHEN.

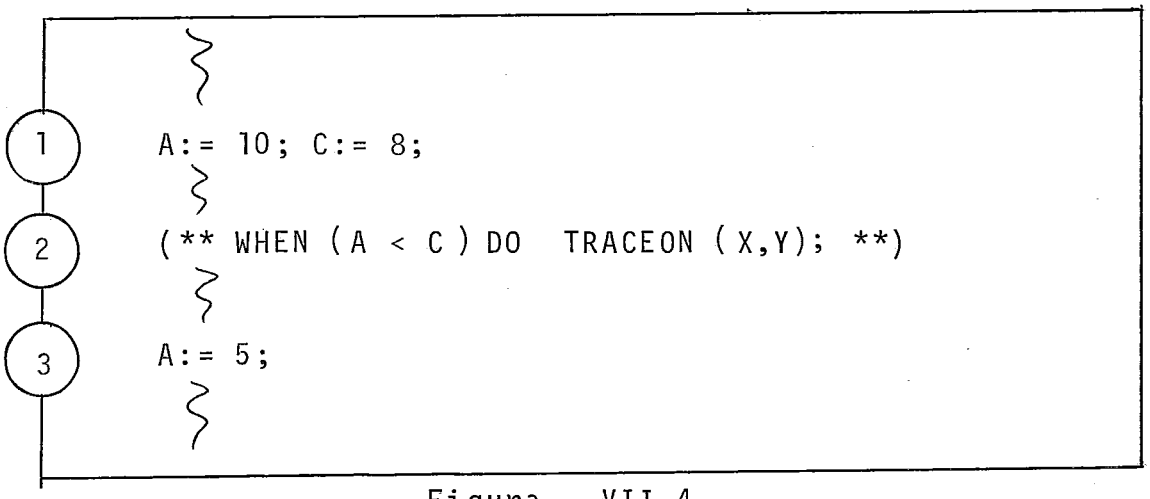


Figura - VII.4

Na linha 3 do programa, sendo até então a condição (A < C) falsa, as variáveis X e Y serão colocadas sob trace. Em outro ponto do programa, este trace poderá ser desligado, normalmente, através do comando TRACEOFF.

Estamos propondo, ainda, o trace para os seguintes objetos: labels, procedures e functions.

Através de "(** TRACEON(ALL LABELS);**)" que pode ser escrito em qualquer região de comandos, de qualquer módulo, é ligado o trace para todos os labels. Ou seja: a partir daí, sempre que o programa passar por algum label, este será impresso no relatório do depurador.

Através de "(** TRACEOFF(ALL LABELS);**)" o trace de todos os labels é desligado.

Através de "(** TRACEON(LOCAL LABELS);**)" o trace apenas de labels locais a uma procedure/function é ligado.

Através de "(** TRACEOFF(LOCAL LABELS);**)" o trace de labels locais àquela procedure/function é desligado.

Os comandos de trace para labels podem ser escritos como saída verdade de um IF de depuração, como em:

"(** IF(D > B) THEN TRACEON (LOCAL LABELS);**)"

e em

"(** IF(X = Y) THEN TRACEOFF (ALL LABELS);**)"

ou como cláusula DO de um WHEN , como em:

"(** WHEN (X = Z) DO TRACEON (ALL LABELS);**)"

A figura (VII.5) apresenta o esquema de uma procedure em cujo início foi pedido o trace de todos os labels e em cujo fim foi este trace desligado. Com tal procedimento, o que se obterá, no relatório do depurador, é a sequência da labels percorridos, desde o início da execução de P1, até o término desta, incluidos os labels das procedures processadas em cadeia (exemplo: P1 → P2 → P3).

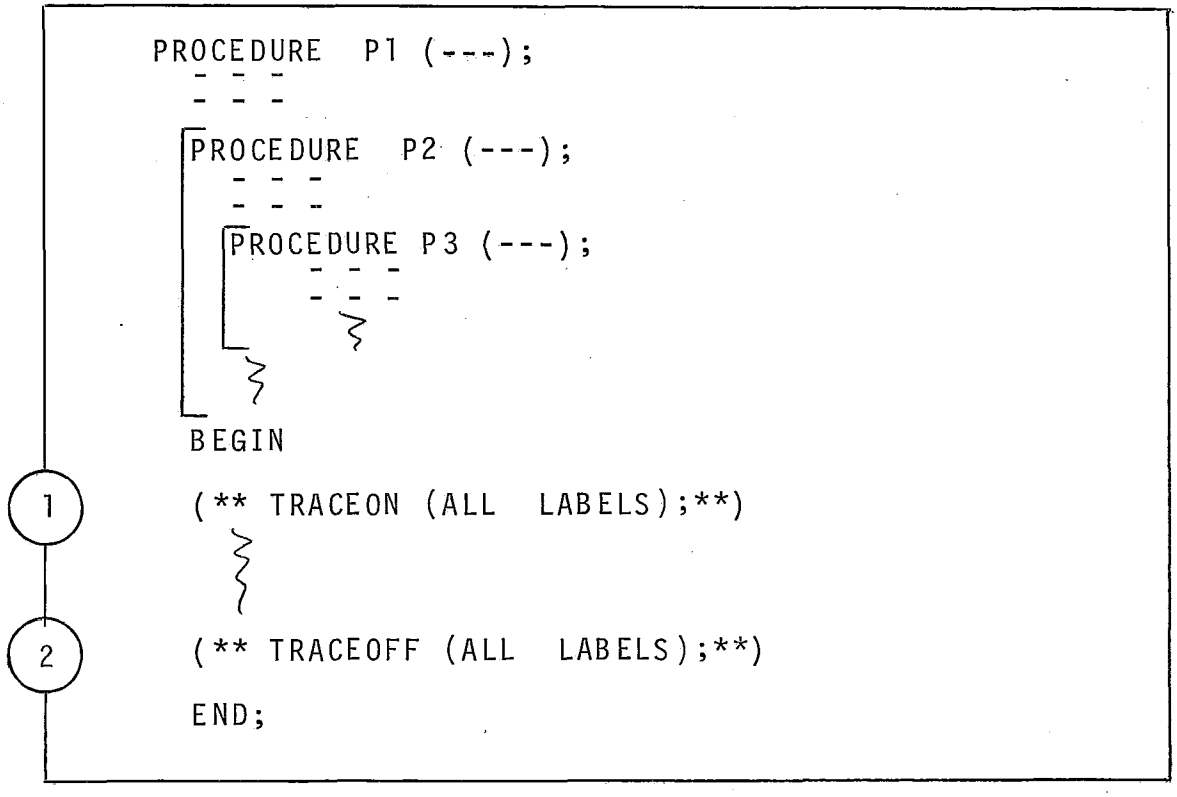


Figura - VII.5

Para procedures e functions existirão as opções "(** TRACEON(ALL SUBPROGRAMS);**)" e "(** TRACEOFF(ALL SUBPROGRAMS);**)", que podem também ser escritas como opção verdade de IF do depurador, como por exemplo, em "(** IF(X < Y) THEN TRACEON(ALL SUBPROGRAMS);**)" e em "(** IF(X = Y)THEN TRACEOFF (ALL SUBPROGRAMS); **)" ou como cláusula DO de um WHEN como em "(** WHEN(Z = T)DO TRACEON(ALL SUBPROGRAMS); **)".

As opções de trace para labels, procedures, functions e mesmo para variáveis podem ser conjugadas num único comando. Assim, a linha 1 do exemplo da figura (VII.5) poderia ser reescrita como "(** TRACEON(ALL SUBPROGRAMS, ALL LABELS, X);**)". O efeito que se obteria, seria a impressão, no relatório do depurador, da sequência de subprogramas chamados, de seus labels percorridos e de valores de X desde o início até o final da execução de P1.

Uma forma mais criteriosa de se ligar/desligar o trace para procedures, functions e labels seria ligar o trace imediatamente antes da chamada do primeiro subprograma a rastrear e desligá-lo imediatamente após. Com tal procedimento, obtém-se sucesso no rastreamento para o caso de encadeamento de chamadas com recursividade direta ou indireta. A figura (VII.6) apresenta duas opções, onde a primeira fará o rastreamento completo, mas a segunda o interromperá ao término da execução da última ativação de P.

```
PROGRAM   TESTE  (INPUT, OUTPUT);
  - - -
  PROCEDURE  P (---);
    - - -
    BEGIN
     ≷
    P (---);
     ≷
    END;
  BEGIN
   ≷
  (** TRACEON(ALL SUBPROGRAMS, ALL LABELS);**)
  P (---);
  (** TRACEOFF(ALL SUBPROGRAMAS, ALL LABELS);**)
   ≷
  END .
```

```
PROGRAM TESTE (INPUT, OUTPUT);
  - - -
   PROCEDURE P (---);
     - - -
    BEGIN
    (** TRACEON(ALL SUBPROGRAMS, ALL LABELS);**)
      ≷
    P (---);
      ≷
    (** TRACEOFF(ALL SUBPROGRAMS, ALL LABELS);**)
    END;
   BEGIN
    ≷
   P (---);
    ≷
   END .
```

Figura - VII.6

VII.I.III. CLASSE 3: TRACEBACK

O traceback, como já foi dito, permite ao usuário solicitar a história de elementos escolhidos do programa. Através de comandos SAVEON são ligados "pseudo-traces", pelos quais as informações de rastreio dos elementos selecionados vão sendo gravadas num arquivo circular. Através de comandos SAVEOFF desliga-se os "pseudo-traces" para os elementos neles referidos. Através do comando TRACEBACK o arquivo circular é descarregado. Estes recursos podem também ser empregados de forma condicional.

A figura (VII.7) apresenta o esquema de um módulo em que é utilizado o recursos de TRACEBACK.

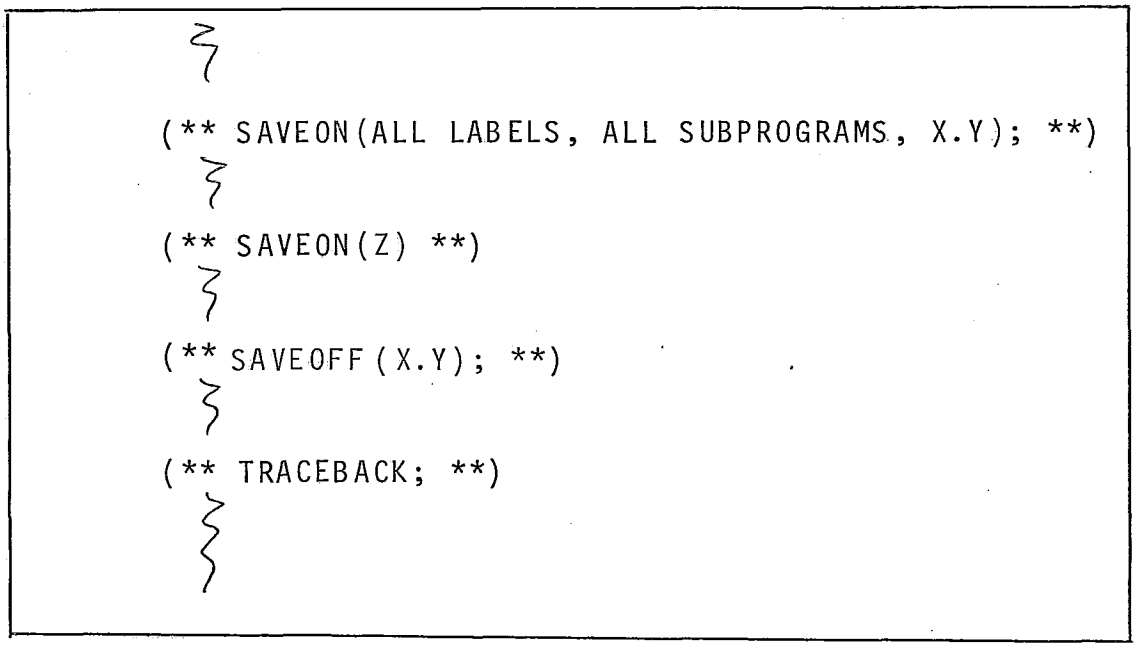


Figura - VII.7

VII.I.IV. CLASSE 4: STATISTICS

Nesta classe é permitido ao usuário solicitar, basicamente, dois tipos de estatísticas quanto à execução do programa:

- estatística de variáveis não utilizadas;
- estatística de frequência de passagem por labels, procedures e functions.

Para solicitar tais estatísticas, basta escrever, logo após o cabeçalho do programa (PROGRAM ...): "(** STATISTICS REQUIRED; **)".

VII.I.V. EXEMPLO

A figura (VII.8) apresenta um exemplo de relatório de depurador. Observe-se que valores do tipo caráter são impressos entre aspas, enquanto que valores do tipo escalar declarado são impressos entre asteriscos.

```
<<<<< RELATÓRIO DO DEPURADOR - APP >>>>>
<<<< PROGRAMA: CONTABØØ1 >>>>

<< ROTULO: 27
<< PROC/E: P1
<< PROC/E.: P1
<< ROTULO: 32
<< VAR: Z.T.R[7]
        NOVO VALOR:*SABADO*
<< ROTULO: 45
<< PROC/S: P1
<< VAR: S.M
        VALOR: 32
<< ROTULO: 13
<< PROC/S: P1

<< ESTATISTICA FINAL:

   ** VARIAVEIS NAO USADAS **
      1.NO PROG CONTAB ØØ1:
        VX,VY
      2.NO PROC. P1:
        VI,VF
   ** FREQUENCIAS **
      1.DE CHAMADAS DE PROC/FUNC:
        PROC. P1 - 2
        PROC. P2 - Ø
      2.DE PASSAGEM POR ROTULOS:
        NO PROG. CONTABØØ1 (ROTULO/FREQ):
           20/Ø , 27/1
        NO PROC. P1 (ROTULO/FREQ):
           13/1, 32/1, 45/1, 86/Ø


<<<<< FIM - APP >>>>>
```

Figura - VII.8

A sequência é a seguinte:

. O programa CONTABØØ1 passou pelo label 27.
. O programa CONTABØØ1 chamou a procedure P1.
. A procedure P1 chamou P1 recursivamente.
. A procedure P1 (2ª chamada) passou pelo label 32.
. A variável Z.T.R[K], sendo K = 7, recebeu valor por atribuição ou leitura: *SÁBADO*(trace).
. A procedure P1 (2ª chamada) passou pelo label 45.
. Terminou P1 (2ª chamada), voltando-se a P1 (1ª chamada).
. A variável S.M tem valor 32 neste ponto (dump).
. A procedure P1 (1ª chamada) passou pelo label 13.
. Terminou P1 (1ª chamada), voltando-se ao programa principal, CONTABØØ1.

A estatística final é auto-explicativa.

VII.I.VI. OUTROS RECURSOS

Neste ponto, um novo comando será apresentado, como recurso de depuração: o FOR. Ele é absolutamente análogo ao comando FOR normal, distinguido-se pelo fato de ser escrito na fórma de comentário especial e por algumas restrições. Através deste FOR, poderá o programador ampliar os recursos de Dump e de Trace para estruturas completas ou parte. Suponhamos ser a estrutura A um array de records onde cada componente B é um array de escalar. Ver figura (VII.9).

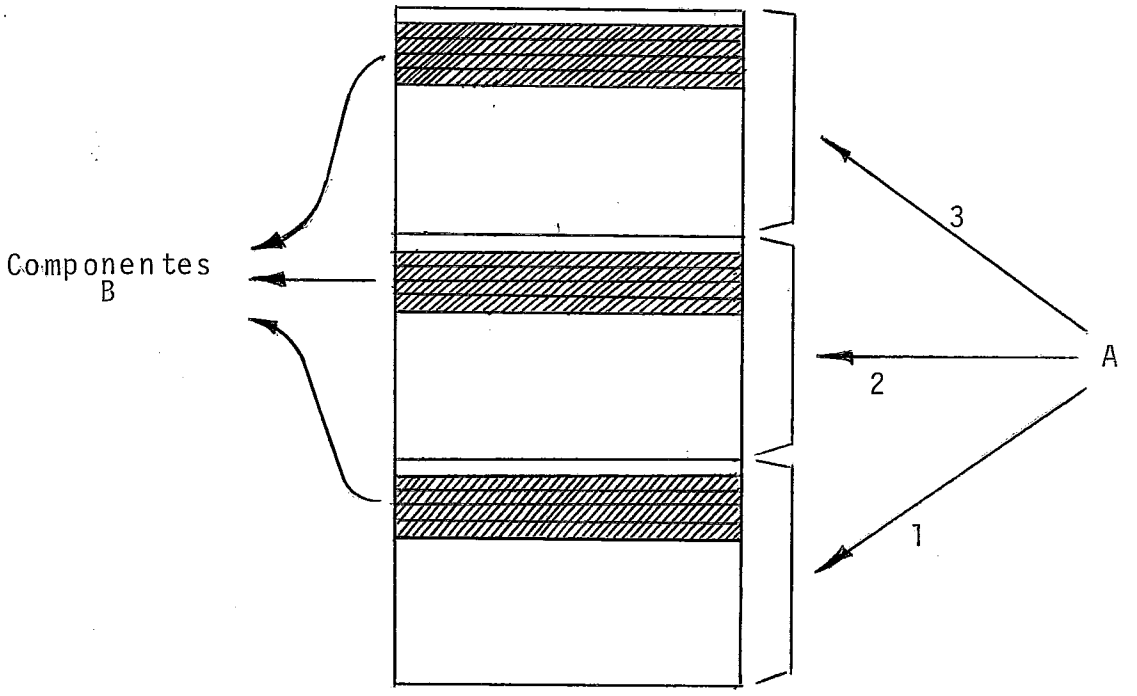


Figura - VII.9

Suponhamos, agora, que por algum motivo o programador deseja rastrear aqueles 12 elementos escalares. Ele poderá fazer isto escrevendo:

```
"(** FOR  K:= 1 TO 3 DO
        FOR  L:= 1  TO4  DO
          TRACEON  A[K].B[L]; **)".
```

Naturalmente, para que os recursos de depuração não interfiram no programa, é preciso que se permitam declarações de variáveis para controle dos FOR's.

No caso:

```
"(** VAR K: 1..3;
          L: 1..4; **)".
```

Na implementação, poder-se-á exigir que estas variáveis tenham nome especial e padronizado. Por exemplo: $01, $02, etc.

Estamos supondo que o programador irá usar com parcimônia o comando FOR em depuração. Do contrário, uma enorme quantidade de "nomes de variáveis" deverá ser guardada, o que causará "estouro" de memória ou exigirá armazenamento em disco, de consulta mais lenta. Partimos do princípio que um programador normal poderá suspeitar de algumas variáveis simples ou de alguns elementos de estruturas de seu programa, mas não usualmente de estruturas completas e até com dados de características diferentes (exemplo: nome, salário, estado civil num array de records). Certamente o programador que não usar com critério o recurso do FOR para depuração estará transferindo para o Ambiente de Programação Pascal seus vícios de depuração por tentativa e erro!

Há, evidentemente, situações de exceção. Suponhamos que o programador simulou uma pilha de inteiros num array B e que o ponteiro de topo se chame T. O programador poderá pedir o trace da variável simples T e de todo o array B com o que acompanhará todos os "movimentos" da pilha.

VII.II. FORMA INTERMEDIÁRIA PASCAL PARA OS RECURSOS DE DEPURAÇÃO E AÇÕES CORRESPONDENTES DO INTERPRETADOR/DEPURADOR

A seguir, através de novos exemplos, analisaremos as ações do interpretador/depurador para FIPs de recursos de depuração.

VII.II.I. DUMP

A figura (VII.10) apresenta um exemplo de pedido de DUMP e a figura (VII.11) a FIP correspondente.

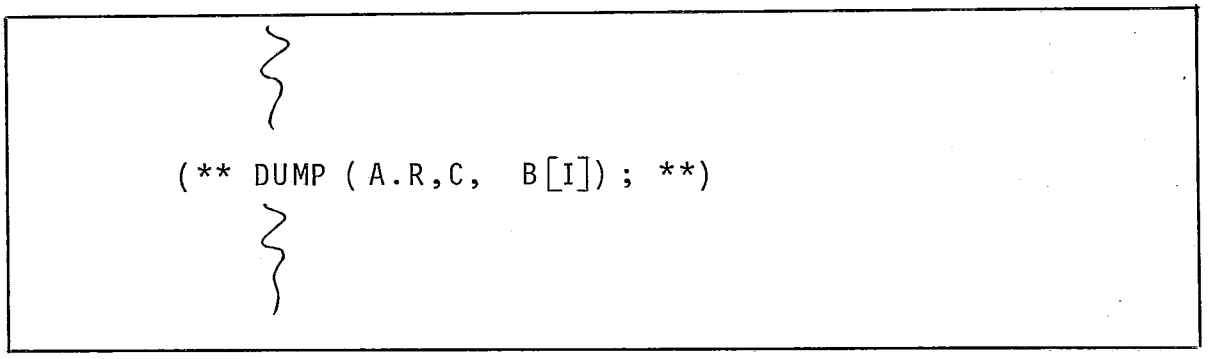


Figura - VII.10

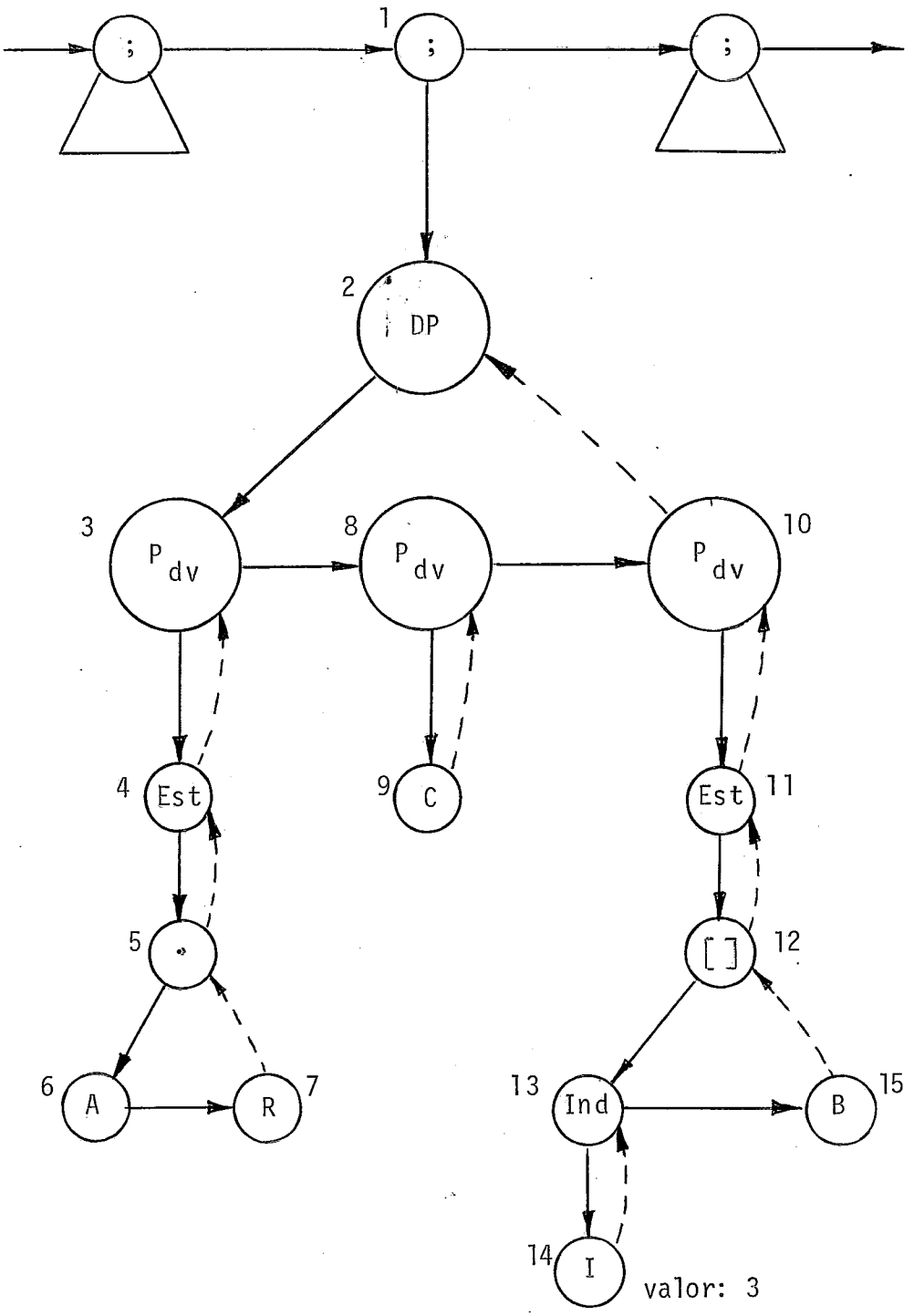


Figura - VII.11

Ações:

D1 - Ø

D2 - Faz PARAM valer 3.
(Este valor para PARAM possibilita a recuperação de nomes de variáveis para posterior impressão).

D3 - Ø

D4, D5, D6, D7, S5, S4 - ações normais de percurso de árvore de referência a componente de record. Como PARAM vale 3, resulta ainda o nome "A.R" recuperado (a ser tratado adiante).

S3 - Imprime, no relatório do depurador, "<<VAR: A.R" e "VALOR:" e, em seguida, o valor de A.R, convertido em função do tipo da variável, indicado no campo de informações deste nó. (1 se booleano, 2 se inteiro, 3 se real, 4 se caráter, 5 se pointer, ou endereço na TS se escalar declarado).

D8 - Ø

D9 - Ação normal ao descer em nó de variável simples. Como PARAM vale 3, resulta ainda o nome "C" recuperado.

S8 - Imprime, no relatório do depurador, "<<VAR: C" e "VALOR:" e, em seguida, o valor de C, convertido em função do tipo de variável, indicado no campo de informações deste nó.

D10 - Ø

D11, D12, D13, D14, S13, D15, S12, S11 - ações normais de percurso de árvore de referência a elemento de array. Como PARAM vale 3, resulta ainda o no

me "B[3]" recuperado, supondo que I valha 3.

S10 - Imprime, no relatório do depurador; "<<VAR: B[3]" e "VALOR:" e, em seguida, o valor de B[3], convertido em função do tipo de variável, indicado no campo de informações deste nó.

S2 - Faz PARAM valer zero.

S1 - Ø

VII.II.II. TRACE/TRACEBACK

A figura (VII.12) apresenta um exemplo de pedido para ligar e desligar a opção trace e as figuras (VII.13) e (VII.14) as FIP's correspondentes.

```
        ≀

(** TRACEON (X.Y,Z, ALL SUBPROGRAMS, LOCAL LABELS); **)
        ≀

(** TRACEOFF (X.Y,Z, ALL SUBPROGRAMS, LOCAL LABELS); **)
        ≀
```

Figura - VII.12

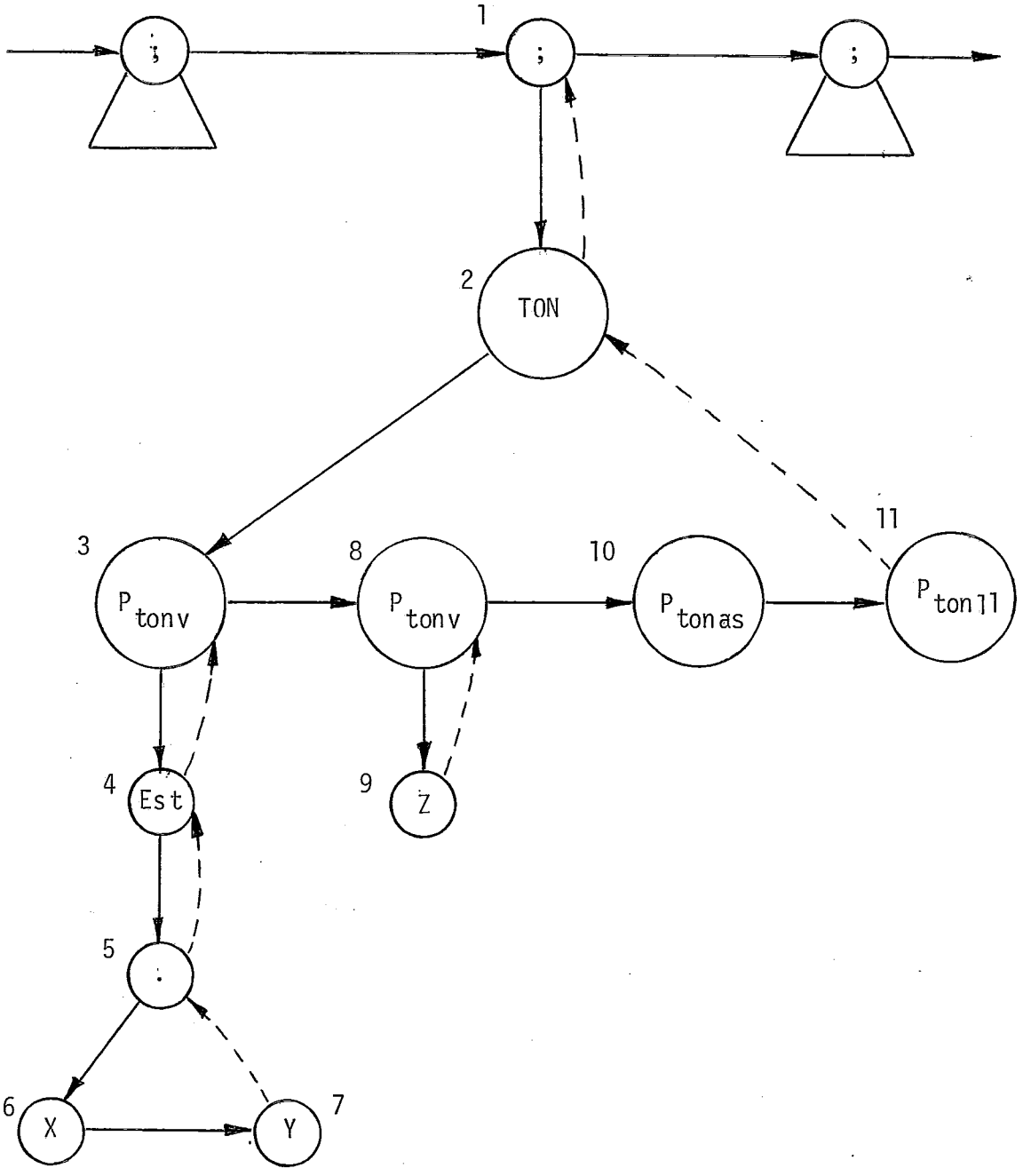


Figura - VII.13

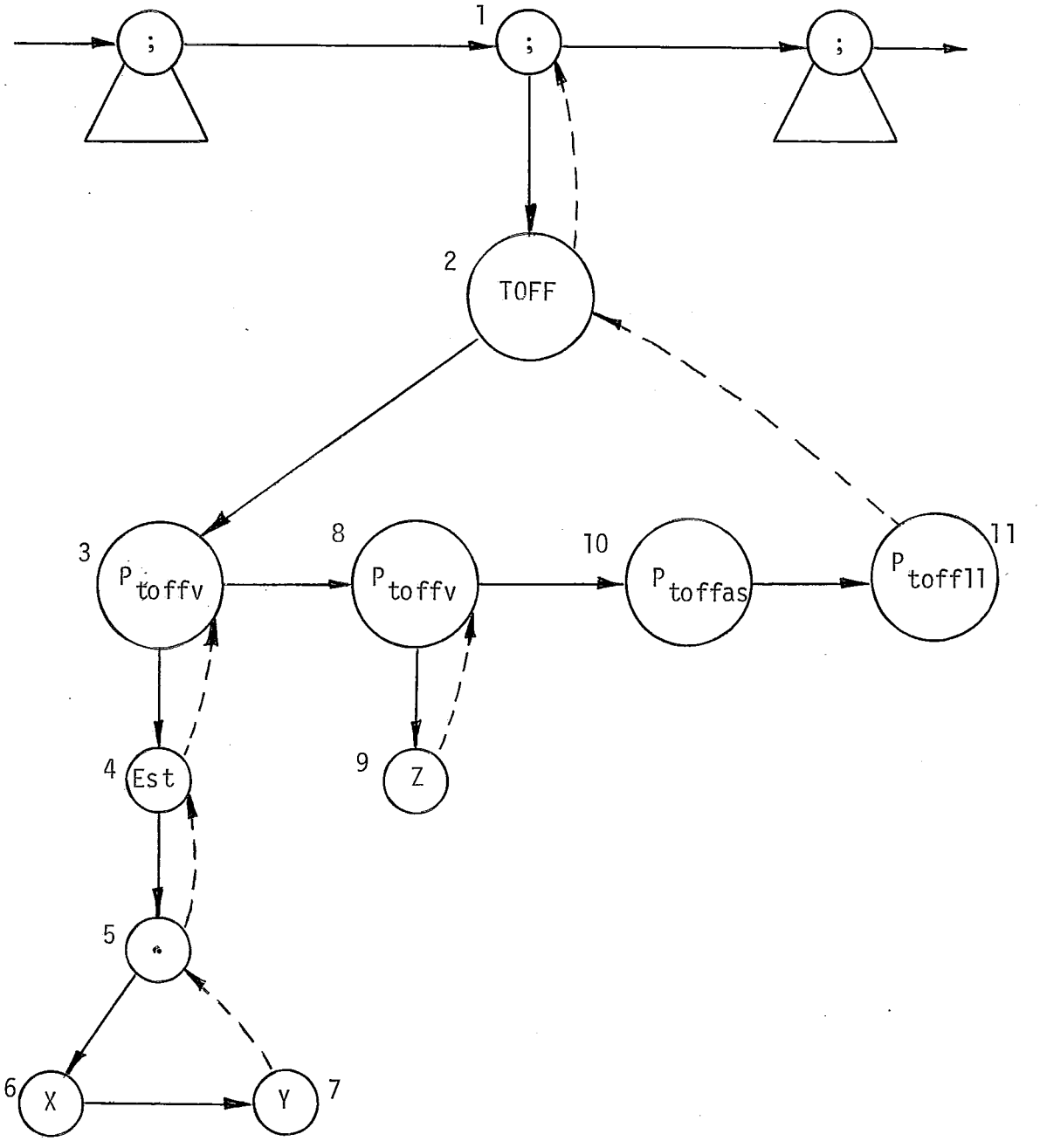


Figura - VII.14

Um exemplo análogo poderia ser criado para o recurso de traceback. A figura (VII.15) apresenta o trecho de programa correspondente. As FIP's correspondentes aos comandos SAVEON e SAVEOFF diferem das FIP's das figuras (VII.13) e (VII.14) apenas quanto aos nós de número 2, que passam a ser, respectivamente, SON e SOFF. A FIP correspondente ao comando TRACEBACK é apresentada na figura (VII.16).

```
(** SAVEON(X.Y,Z,ALL SUBPROGRAMS, LOCAL LABELS); **)

(** SAVEOFF(X.Y,Z, ALL SUBPROGRAMS, LOCAL LABELS); **)

(** TRACEBACK; **)
```

Figura - VII.15

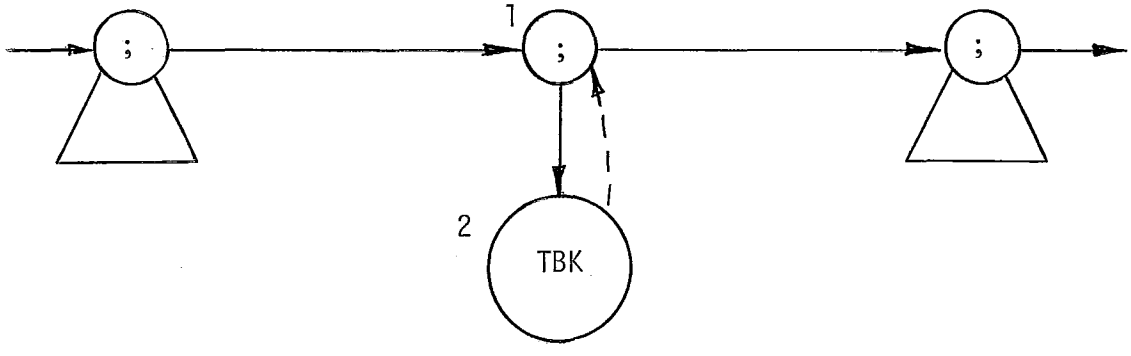


Figura - VII.16

A diferença básica entre os recursos de trace e traceback é que ocorrências de traceback são apenas registradas em um arquivo circular enquanto ocorrências de trace são imediatamente impressas no relatório do depurador. O comando TRACEBACK se encarrega de descarregar o conteúdo do arquivo circular. Este arquivo circular terá registros de tamanho fixo, com capacidade equivalente a uma linha de impressão.

Como já foi visto, os comandos TRACEON E SAVEON podem estar associados a uma condição booleana, especificada através de um WHEN. A figura (VII.17) ilustra uma árvore envolvendo esta situação.

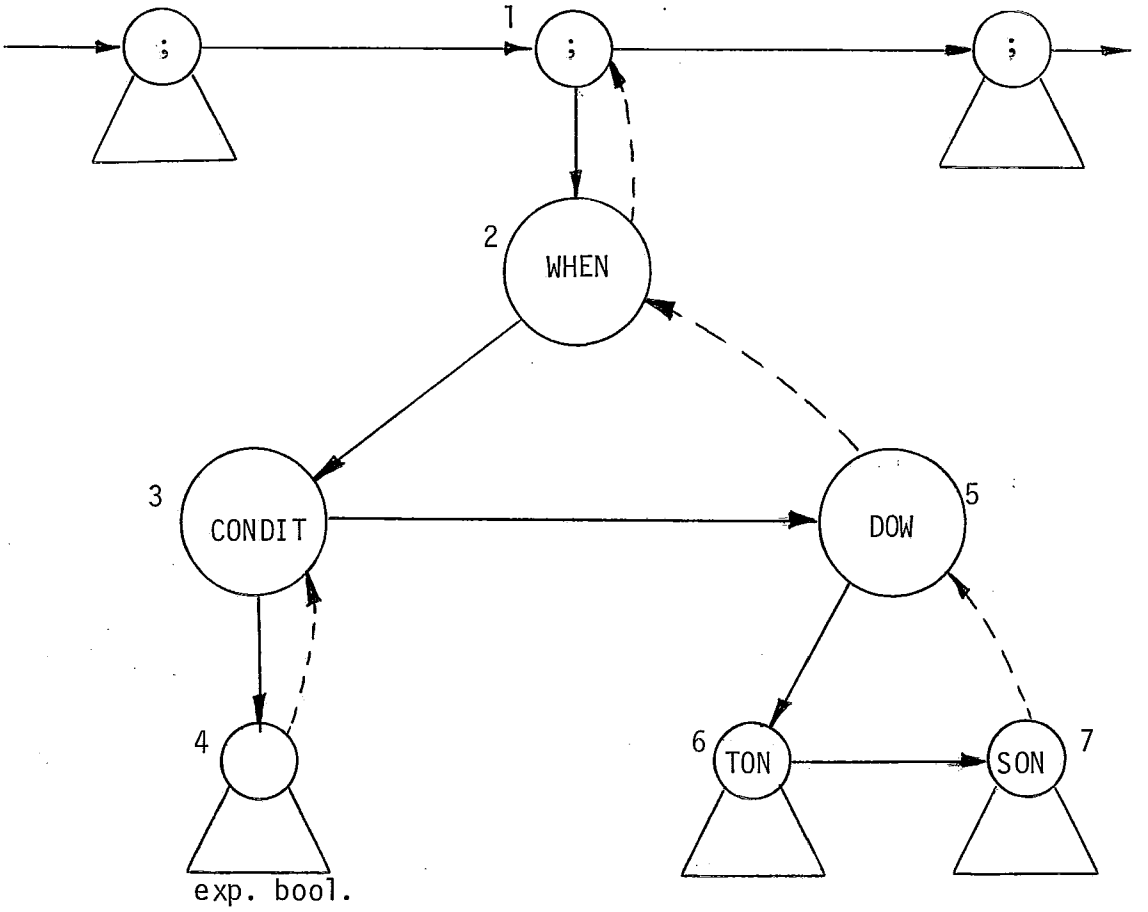


Figura - VII.17

Como se vê daquela figura, a árvore da expressão booleana é encabeçada por um nó especial, de nome CONDIT. Tal nó possui um campo de informação que pode valer 0,1 ou 2. Seu objetivo é registrar três possíveis situações para a expressão booleana, durante o ciclo daquele WHEN, ao longo da execução do programa (entenda-se por ciclo do WHEN o intervalo entre duas execuções do mesmo). Seu valor zero indica que a expressão booleana, desde o WHEN até este instante na execução do programa, sempre teve valor FALSE; seu valor 1 indica que desde o WHEN até este instante na execução do programa, a expressão booleana, em algum momento, tornou-se TRUE, ainda que neste instante seja FALSE; o objetivo do valor 2 será explicado adiante.

No percurso da árvore da figura (VII.17), em D2 o interpretador/depurador faz WHENFLAG valer 1; em D3, como WHENFLAG vale 1, salva o endereço deste nó e desce, indo percorrer a árvore da expressão booleana, ainda que o campo de informação valha 1 (feito igual a 1 no ciclo anterior deste WHEN); percorrida a árvore da expressão booleana, em S3, como WHENFLAG vale 1, faz o campo de informação deste nó igual a 0 ou 1, dependendo do valor da expressão booleana; em D5 e S5 nada faz e em S2 faz WHENFLAG valer zero. No percurso da árvore da expressão booleana, com WHENFLAG valendo 1, é preciso registrar, por variável envolvida, o endereço do nó CONDIT, que encabeça a árvore da expressão booleana. A razão disto é permitir que, a cada novo valor de uma destas variáveis, a expressão booleana seja reavaliada, ficando este valor registrado no respectivo nó CONDIT. Observe-se que este percurso da

árvore da expressão booleana para reavaliação é feito com WHEN FLAG valendo 2. Em S3, ao invés de seguir para o nó DOW, o interpretador/depurador volta para o ponto da árvore do programa do qual saíra para reavaliar a expressão booleana. Observe-se também que tal desvio para reavaliação pode ser imediatamente interrompido em D3, com retorno ao ponto de onde se veio do programa, caso o campo de informação deste nó valha 1 (do contrário, a expressão pode ter sido verdadeira e agora estar voltando a ser falsa, condição nova que não se quer registrar). Finalmente, observe-se que o percurso das árvores da figura (VII.17), que têm por raízes os nós 6 e 7, é afetado pelo fato de WHENFLAG valer 1. Assim, para as variáveis envolvidas nos comandos TRACEÓN e SAVEON, é preciso registrar o endereço do nó CONDIT da correspondente expressão booleana.

Cada vez que uma variável muda de valor, é preciso verificar duas coisas. Primeiro, se ela faz parte de expressão booleana associada a WHEN. Se fizer, a expressão deve ser reavaliada, como descrito. Segundo, se ela está sob trace/save. Se está, dá-se o trace, se incondicional, ou desvia-se para o correspondente nó CONDIT, se sob condição WHEN. Neste caso, WHENFLAG é feita igual a 4 (se trace) ou 5 (se save) e, se o nó CONDIT registrar 1, o trace/save é tornado incondicional (onde se guardara o endereço do nó CONDIT, para registro do trace /save condicional, coloca-se o valor 1) e o recurso de depuração é executado. Observe-se que visitou-se, então, apenas o nó CONDIT, não se caminhando pela árvore da expressão booleana, o que só é feito na avaliação inicial e nas posteriores reavaliações.

Note-se ainda que, encerrada a execução de uma procedure (ou function), verifica-se, para as variáveis locais a ela, envolvidas em expressões booleanas de WHEN, se os correspondentes nós CONDIT registram 0 ou 1. Se registram 1 a reavaliação da expressão booleana já está bloqueada e nada é feito; se registram zero, faz-se os nós CONDIT registrarem 2, o que proibe novas reavaliações (o que se permitido seria um erro, já que as variáveis locais a esta procedure teriam, daí para a frente, valor indefinido). Naturalmente, foi preciso desviar, descendo, para o nó CONDIT, antes salvando o ponto de desvio, e fazendo WHENFLAG valer 3.

Neste ponto, cumpre expor, com maior detalhamento, as técnicas adotadas para registro das condições de trace e de save.

Quatro variáveis, de nomes TALLLABS, TALLSUBS, SALLLABS e SALLSUBS serão os mecanismos responsáveis pelo trace e pelo save de todos os labels e de todos os subprogramas. Cada uma destas variáveis poderá valer zero, indicando que o trace/save correspondente está desligado; ou 1, indicando que está incondicionalmente ligado; ou o endereço do nó CONDIT que encabeça a árvore da expressão booleana correspondente ao comando WHEN ao qual está associado um comando TRACEON ou SAVEON (o valor 1 jamais poderá corresponder ao endereço do nó raiz de expressão booleana).

Assim, passando o interpretador pelo nó de definição de um label (<label>:...), são consultadas as variáveis TALLLABS e SALLLABS. Para aquela que valer zero, nada é feito. Para aquela que valer 1, é tomada a ação de depuração correspon-

dente. Para aquela que tiver valor maior que 1, salva-se o endereço deste nó e desvia-se, descendo, para o nó CONDIT apontado, antes fazendo WHENFLAG valer 6 (se TALLLABS > 1) ou 7 (se SALLLABS > 1). Na ação ao descer no nó CONDIT, se seu campo de informação valer zero, apenas volta-se ao nó salvo; se valer 2, faz-se TALLLABS/SALLLABS (a que originou o desvio) valer zero; se valer 1, substitui-se em TALLLABS/SALLLABS (a que originou o desvio) o seu valor por 1 e toma-se a ação de depuração correspondente.

Observe-se que se for pedido um trace ou save sob condição WHEN, mas o trace ou save incondicional já estiver ligado, aquele será ignorado; e que se a condição WHEN estiver registrada para trace ou save e um ou outro for ligado incondicionalmente, a condição WHEN correspondente perde seu registro (exemplo: TALLLABS vale 983 e é encontrado um pedido de trace para todos os labels incondicional ou sob condição IF verdadeira,o que o torna a partir daí incondicional. Neste caso, TALLLABS passa a valer 1).

Raciocínio análogo vale para as variáveis TALLSUBS e SALLSUBS. As ações são tomadas ao descer em nó cabeça de procedure/function com CALLSUB valendo 1 e ao subir em nó cabeça de procedure/function.

Cabem as mesmas observações anteriores que dão precedência ao trace e ao save incondicionais sobre os mesmos sob condição WHEN.

Por outro lado, será criada uma pilha paralela à PRA, a que denominaremos AUXSTACK, que:

. das entradas correspondentes às das variáveis na PRA apontará para uma tabela, de nome TTABLE, onde estão registros para variáveis sob trace/save e para variáveis participantes de expressão(ões) booleana(s) associada(s) a WHEN(S).

. nas duas entradas correspondentes à entradas básicas do RA na PRA registrará, respectivamente, as condições de trace e de save para labels locais.

A figura (VII.18) esquematiza o conjunto descrito.

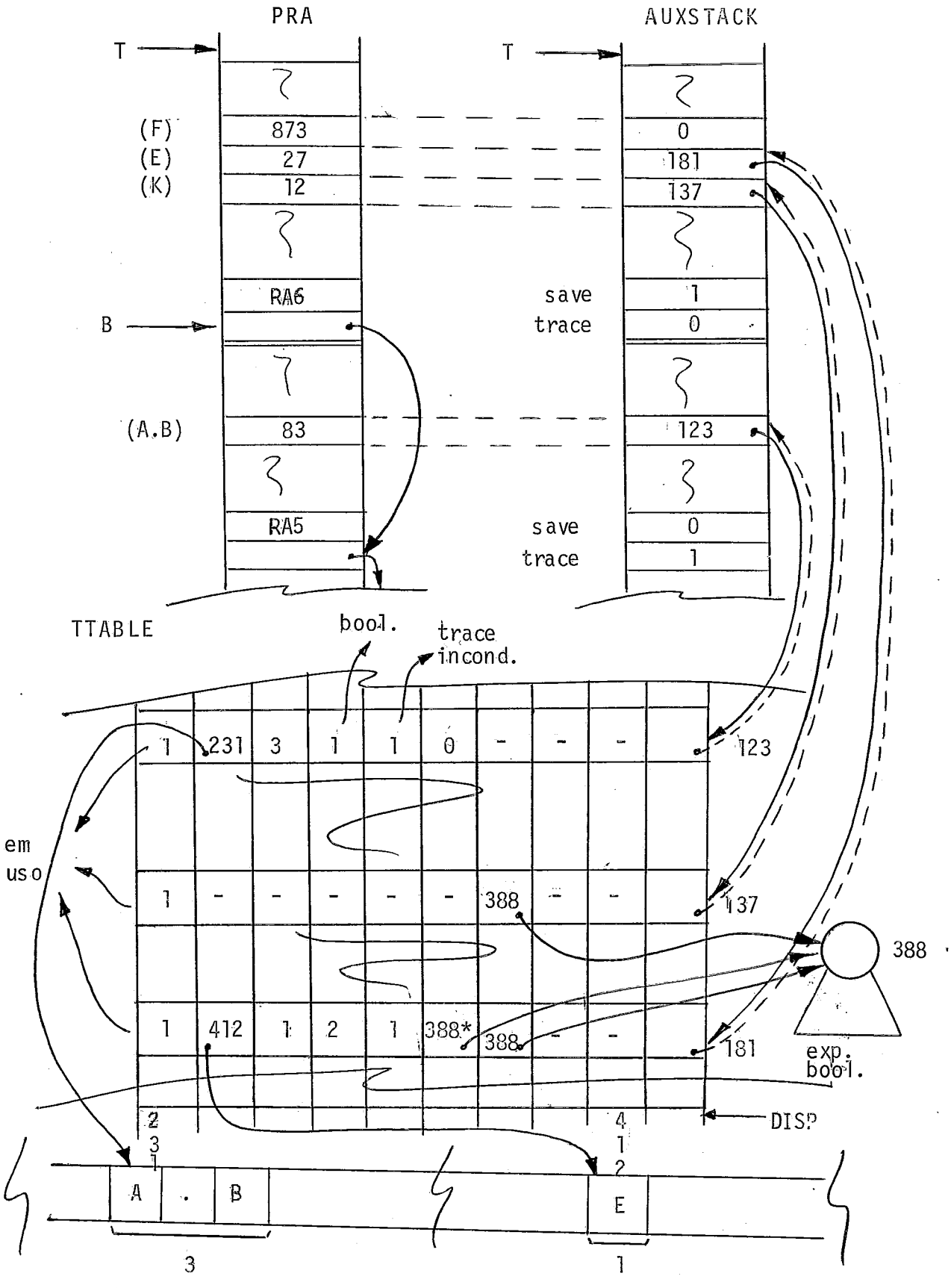


Figura - VII.18

Retomemos o mecanismo criado para marcar como em trace e save os labels locais a uma procedure ou function. Tal mecanismo consiste, como visto, nas duas entradas em AUXSTACK correspondentes às duas entradas básicas de cada RA. A primeira marca como em trace incondicional todos os labels da ativação da procedure/function correspondentes ao RA em questão, ao valer 1; ou guarda o endereço do nó CONDIT que encabeça a árvore da expressão booleana correspondente ao comando WHEN ao qual está associado um comando TRACEON para aqueles labels; ou vale zero, indicando que tal trace está desligado. A segunda tem função análoga, sendo reservada para o save.

A ação completa do interpretador/depurador ao passar por um nó de definição de label (<label>:...) consiste na ação já descrita, envolvendo as variáveis TALLLABS e SALLLABS seguida de ação análoga, envolvendo, das duas entradas em AUXSTACK, correspondentes às entradas básicas do RA em questão, aquela(s) relativa(s) ao recurso não executado no primeiro passo (trace ou save).

Para vermos com detalhe os mecanismos para marcar como em trace/save as variáveis do programa e para registrar informações sobre variáveis participantes de expressão(ões) booleana(s) associada(s) a WHEN(S), cumpre descrevermos a tabela TTABLE, já apresentada na figura (VII.18). Ali se observa que cada linha da TTABLE possui 10 entradas. A primeira é um código que avisa se tal entrada está em uso (1) ou não (0), possibilitando, em conjunto com a entrada 10, uma rearrumação da tabela em caso de "overflow", como será visto adiante. As entradas de 2 a 6 são usadas se esta linha registrar informa

ções de variável sob trace/save. Assim, a entrada 2 aponta para o início do nome da variável no vetor NOMES e a entrada 3 indica seu comprimento. A entrada 4 registra o tipo de variável (1 se booleana, 2 se inteira, 3 se real, 4 se caráter, 5 se pointer e endereço da TS se escalar declarado) para conversão. As entradas 5 e 6 indicam o tipo de rastreio a que a variável está sujeita: se a quinta valer 1 é trace incondicional, se a sexta valer 1 é save incondicional. Estas duas entradas podem também registrar trace e save sob condição WHEN. Neste caso, tais entradas conterão o endereço do nó CONDIT correspondente. As entradas 7, 8 e 9 são para variáveis que fazem parte de expressão(ões) booleana(s) associada(s) a WHEN(S) e apontam para os respectivos nós CONDIT. Observe-se que admitimos até 3 participações em tais expressões por uma mesma variável (o que poderia ser facilmente expandido, podendo-se também optar por capacidade ilimitada, pela criação de uma linha de continuação na TTABLE). Observe-se ainda que uma variável pode simultaneamente ter registro de trace/save e de participação naquelas expressões. A entrada 10 aponta de volta para a AUXSTACK. Esta entrada, em conjunto com a primeira, permite a rearrumação da TTABLE em caso de "overflow", como dito. Assim, quando se termina a execução de uma procedure/function, são pesquisadas as entradas de AUXSTACK correspondentes às das variáveis na PRA. Para aquelas que apontarem para a TTABLE, o interpretador/depurador consulta aquela tabela. Se um ou mais dentre os campos 7, 8 e 9 apontar para nó(s) CONDIT, marca o campo de informação de tal (tais) nó(s) com 2, como visto. De qualquer modo, faz o primeiro campo desta li

nha da TTABLE valer zero. Deste modo, quando ocorrer "overflow", é feita uma rearrumação da TTABLE: todas as linhas cuja primeira entrada valer 1, vão sendo agrupadas no início da TTABLE, alterando-se os correspondentes ponteiros em AUXSTACK, com auxílio da décima entrada.

Um esquema alternativo de gerenciamento da TTABLE consistiria na criação de uma lista de espaço disponível. Inicialmente a TTABLE seria uma lista apenas de espaços livres. Cada vez que uma linha (um nó) fosse requerida, seria buscada daquela lista.

Cumpre agora detalhar o que acontece quando, numa atribuição ou leitura, uma variável recebe novo valor (situação detetada ao subir em nó de atribuição ou de parâmetro de leitura). Neste ponto, dispõe o interpretador/depurador do endereço de tal variável linearizado na PRA. Vai, então, à entrada oorrespondente em AUXSTACK, e verifica se ela é diferente de zero. Se for, segue para a linha da TTABLE apontada. Ali, verifica inicialmente por consulta às entradas 7, 8 e 9, se tal variável faz parte de expressão booleana de WHEN. Para cada caso afirmativo desvia para o nó CONDIT, reavaliando a expressão booleana, se for o caso, como visto. No retorno de cada reavaliação, se o nó CONDIT da respectiva expressão booleana não registrar zero, deve-se zerar a entrada da TTABLE (7, 8 ou 9) que permitiu o acesso para reavaliação.

Depois, são consultadas as entradas 5 (trace) e 6 (save) de TTABLE. Caso a entrada 5 valha zero, nada é feito; caso a entrada 5 valha 1 é tomada a ação de depuração correspondente; caso a entrada 5 tenha valor maior que 1, salva-se o ende

reço de retorno e desvia-se, descendo, para o nõ CONDIT apontado, antes fazendo WHENFLAG valer 4. Na ação ao descer no nõ CONDIT, se seu campo de informação valer zero, apenas volta-se ao nõ salvo; se valer 2, faz-se a quinta entrada da linha da TTABLE cujo endereço fora salvo, valer zero; se valer 1, substitui-se o valor da quinta entrada por 1 e toma-se a ação de depuração (análogo para a entrada 6).

Além disto, caso, ao final, as entradas 5, 6, 7, 8 e 9 da linha da TTABLE valham zero , zera também a sua primeira entrada, bem como a entrada de AUXSTACK que deu acesso àquela tabela.

Observe-se que o trace e o save incondicionais para variáveis têm também procedência sobre o trace e o save para variáveis sob condição WHEN. Assim, se for pedido um trace ou um save para uma variável sob condição WHEN, mas o trace ou save incondicional para a mesma já estiver ligado, aquele será ignorado. Por outro lado, se a quinta (sexta) entrada da linha de uma variável na TTABLE estiver apontando para nõ CONDIT, mas o trace (save) for ligado incondicionalmente, a quinta (sexta) entrada será feita igual a 1.

Vejamos, agora, as ações de trace/save para variáveis: busca-se o nome da variável, via TTABLE, do vetor NOMES; converte-se o valor (o tipo, para conversão, está na TTABLE) e imprime-se (se trace) ou grava-se no arquivo circular (se save) o nome seguido do valor.

Note-se que uma mesma variável pode estar sob trace e save onde ambos podem estar sujeitos à mesma condição WHEN ou a condições diversas, ou um pode estar sujeito a condição

WHEN e o outro não, ou ambos podem estar livres de condição WHEN.

Vejamos, agora, de que maneira as diversas entradas da figura (VII.18) vão sendo preenchidas. Para isto, acompanhemos o percurso da árvore da figura (VII.13).

Ações:

D1 - Ø

D2 - Faz PARAM valer 4.

D3 - Ø

D4, D5, D6, D7, S5 - Ações normais de percurso de árvore de referência a componente de record, resultando assim o endereço de X.Y abaixo do topo da PRA.

Como PARAM vale 4, resulta ainda o nome "X.Y" recuperado.

S4 - Como PARAM vale 4, o valor de X.Y não é transferido para o topo da PRA.

S3 - Com base no endereço de X.Y que está abaixo do topo da PRA, vái à entrada correspondente em AUXSTACK.

Se esta for nula, fá-la igual a DISP, ponteiro da TTABLE, incrementando-o.

Vai à linha apontada na TTABLE. Lá, caso a quinta entrada valha 1, nada é feito; caso a quinta entrada tenha valor maior que 1, substitui tal valor por 1; caso a quinta entrada tenha valor zero, faz esta entrada igual a 1, e, se a sexta valer ze

ro, transfere o nome recuperado para o vetor NOMES fazendo a segunda entrada apontar para seu início, a terceira conter seu comprimento e a quarta conter seu tipo, que está registrado neste próprio nó.

D8 - Ø

D9 - Como PARAM vale 4, apenas busca o endereço de Z da TS e o lineariza na PRA, empilhando-o, em seguida, na mesma (não empilha o valor).
Resulta ainda o nome "Z" recuperado.

S8 - Análogo a S3.

D10 - Faz TALLSUBS valer 1.

D11 - Faz a entrada de AUXSTACK responsável por marcar labels locais à última ativação de procedure/function como em trace igual a 1.

S2 - Faz PARAM valer zero.

S1 - Ø

OBSERVAÇÃO

As ações no percurso da árvore correspondente a esta, relativa ao comando SAVEON, são semelhantes. O nó 2 seria substituido pelo nó SON e os nós 3, 10 e 11 pelos nós Psonv, Psonas e Psonll. Em D2, a variável PARAM seria feita igual a 5 e este valor guiaria as demais ações.

Vejamos, agora, as ações no percurso da árvore da figura (VII.14), lembrando que passos semelhantes seriam seguidos no percurso de uma árvore correspondente, relativa ao comando SAVEOFF. Em tal árvore, o nó 2 seria substituido pelo nó SOFF e os nós 3, 10 e 11 pelos nós Psoffv, Psoffas e Psoffll. Por outro lado, em D2, PARAM valeria 7.

Ações:

D1 - Ø

D2 - Faz PARAM valer 6.

D3 - Ø

D4, D5, D6, D7, S5 - Ações normais de percurso de árvore de referência a componente de record.

S4 - Como PARAM vale 6, mantém o endereço de X.Y abaixo do topo da PRA.

S3 - Com base no endereço de X.Y, vái à AUXSTACK e daí à TTABLE. Nesta, faz quinta entrada valer zero. Se a sexta entrada valer zero, e a sétima, a oitava e a nona também, zera a primeira entrada e volta à entrada de AUXSTACK correspondente

a X.Y, zerando-a.

D8 - Ø

D9 - Como PARAM vale 6, apenas busca o endereço de Z da TS e o lineariza na PRA, empilhando-o, em seguida, na mesma (não empilha o valor).

S8 - Análogo a S3.

D10 - Faz TALLSUBS valer zero.

D11 - Vái a entrada de AUXSTACK responsável por marcar labels locais à última ativação de procedure/function como em trace, zerando-a.

S2 - Faz PARAM valer zero.

S1 - Ø.

Vejamos, agora, o percurso bastante simples da árvore da figura (VII.16).

Ações:

D1 - Ø

D2 - Descarrega o arquivo circular de traceback no relatório do depurador, limpando-o.

S1 - Ø.

OBSERVAÇÕES

. Ao longo deste capítulo, e do anterior, tem sido frequentemente utilizada uma variável, de nome PARAM, que dirige ações no percurso de subárvores que têm por raiz nós de parâmetros. Cumpre, neste ponto, apresentar, organizadamente, uma tabela de valores de PARAM e respectivas funções:

0 - desligado
1 - usado no NEW
2 - usado no READ
3 - usado no DUMP
4 - usado no TRACEON
5 - usado no SAVEON
6 - usado no TRACEOFF
7 - usado no SAVEOFF

Além das funções específicas aos comandos, dirigidas pelos valores de 1 a 7 de PARAM, os valores 3, 4 e 5 possibilitam ações de recuperação de nomes de variáveis, a serem futuramente descritas.

. Foi vista, nas figuras (VII.2) e (VII.3), a possibilidade de desligar trace (o mesmo vale para save) para todas as variáveis locais a uma procedure/function e para todas as variáveis do programa. Haverá, naturalmente, quatro nós específicos para as quatro alternativas levantadas (Ptofflv, Ptoffav, Psofflv e Psoffav).

A ação ao descer num destes nós consiste em percorrer as entradas de AUXSTACK correspondentes a todas as variáveis ou apenas às locais ao RA que está no topo da PRA e dái, para cada uma que apontar para a TTABLE, desviar para aquela tabela, desligando o trace ou o save, conforme o caso.

VII.II.III. COMANDOS FOR E IF

Os comandos FOR e IF, em depuração, são análogos aos comandos de mesmos nomes para processamento normal. O que os distingue é apenas o conjunto de comandos permitidos nos chamados "corpo do For" e "corpo do If", controle este que é função do módulo 1 do projeto. Por esta analogia, não descreveremos aqui as FIP's e as ações de interpretação para estes comandos.

VII.II.IV. STATISTICS

Duas estatísticas especiais são ainda permitidas pelo interpretador/depurador:

. Estatística de Uso de Variáveis.
  Tal recurso permite conhecer, ao final da execução do programa, quais as variáveis que não receberam valor. Para tal, um campo é acrescido na tabela de símbolos, especificamente para variáveis, valendo "zero" se não foram submetidas a atribuição ou leitura, e 1 caso contrário.

Ao final da execução do programa, o interpretador/depurador percorre a tabela de símbolos, fornecendo tal estatística.

Esta idéia poderá ser explorada, caso se queira, futuramente, implementar um mecanismo de controle de erros de execução por tentatica de uso (em expressão) de variáveis com valor indefinido.

. Estatística de Frequência de Passagem por Labels, Procedures e Functions.

Na tabela de símbolos, um campo é criado por label, procedure e function com o objetivo de contar tais ocorrências. Toda vez que o interpretador/depurador passar por um nó de definição de label (<label>: ...) ou de chamada de procedure/function, deverá somar 1 ao campo correspondente.

Ao final da execução do programa, o interpretador/depurador percorre a tabela de símbolos, fornecendo a estatística solicitada.

VII.II.V. DUMPHEAP

Um outro comando de depuração que resta apresentar é o DUMPHEAP. Através dele, são listadas as diversas variáveis do tipo pointer (que são os meios de se adentrar as estruturas) bem como o estado do HEAP. Com isto, poderá o programa

dor acompanhar suas estruturas apontadas, sem maior dificuldade. O comando DUMPHEAP também poderá ser pedido sob condição IF.

Naturalmente, as entradas das diversas variáveis tipo pointer deverão estar interligadas, na TS. A busca dos nomes dos pointers é imediata, direto da própria TS.

VII.III. RECUPERAÇÃO DE NOMES

Foi visto que a maioria dos recursos de depuração requerem a impressão dos nomes das variáveis envolvidas. Neste tópico estudaremos os procedimentos do interpretador/depurador para recuperar ou reconstruir tais nomes. A grande flexibilidade que o PASCAL possibilita na construção de tipos nos leva a estudar caso a caso, incluídas as combinações básicas. Com elas, ficam cobertas as possíveis e infinitas combinações, para as quais os procedimentos são os mesmos.

Será discutida a recuperação de nomes para variáveis, já que para os demais objetos (labels, procedures e functions) a recuperação de nomes consiste numa simples busca na TS.

Cumpre observar, neste ponto, o que entendemos por nome recuperado: serão reconstruídos nomes de variáveis simples, elementos simples de arrays, componentes simples de records e combinações entre arrays e records cuja referência final seja um objeto simples. Os nomes serão reconstituídos como escritos no programa, sendo que, para o caso de arrays, serão considerados os valores de seus índices e não os nomes de variáveis ou expressões que os compõem.

Assim, se o programador escrever

A.B[R,L*K].C

e naquele momento R (escalar declarado) vale QUARTA e a expressão inteira L*K tem valor 6, será reconstituído o nome

"A.B[*QUARTA*, 6].C"

Estudaremos agora cada caso de recuperação de nomes. As ações a serem descritas dirão respeito, apenas, a este aspecto. Finalmente, cumpre lembrar que quem aciona a recuperação de nomes é a variável PARAM ao valer 3, 4 ou 5.

Para todos os casos, o nome será montado num vetor alfanumérico, de nome MONTANOME.

CASO 1 - VARIÁVEL SIMPLES

Este caso é elementar. Reportêmo-nos à figura (VII.11). Como visto, ao descer no nó 9, o nome "C" é recuperado. Antes de tratarmos da recuperação deste nome, de variável simples, observemos da mesma figura a subárvore que tem por raiz o nó 11. Trata-se de uma referência a elemento de array, cujo índice I é uma variável simples que não deverá ter seu nome recuperado. No caso, o que interessa é seu valor, como já visto. O nó 13, todavia, possibilita uma ação (uma variável de nome IND é incrementada) que assegura que a subárvore a ele subordinada é de expressão índice e portanto os nomes de variáveis, simples ou não, não serão recuperados. Feita esta consideração, voltemos ao nó 9 e à recuperação de seu nome. Descreveremos apenas a parte da ação específica para este fim:

D9 - Como IND = 0:

Busca o nome ("C") da tabela de símbolos e o guarda no vetor MONTANOME

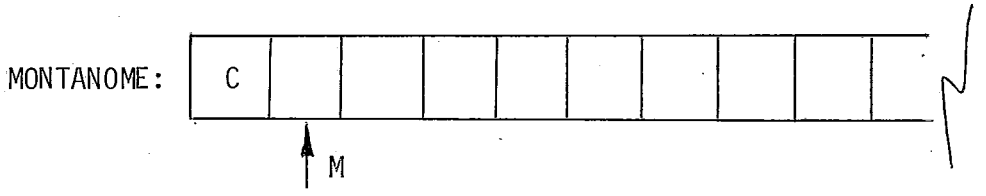


Figura - VII.19

OBSERVAÇÃO

No caso, tratando-se de DUMP, o nome recuperado é transferido para o buffer de impressão, e depois impresso com as demais informações.

No caso de TRACEON ou de SAVEON, o nome é transferido para o vetor NOMES, sendo guardados na TTABLE o endereço de seu início e seu comprimento.

CASO 2 - ARRAY

Como já foi visto, a recuperação de nome para uma referência a elemento de array consiste no nome deste seguido dos valores de seus índices, entre brackets.

Exemplos: "A[3,*SABADO*]"

"B[*DOMINGO*,3, "C"]"

A figura (VII.20) apresenta um exemplo de referência a array: A[V[Z[X,E],T], C+D,L]. O nome a ser recuperado será, por exem-

plo:"A[*DOMINGO*,4, *TERCEIRO*]".

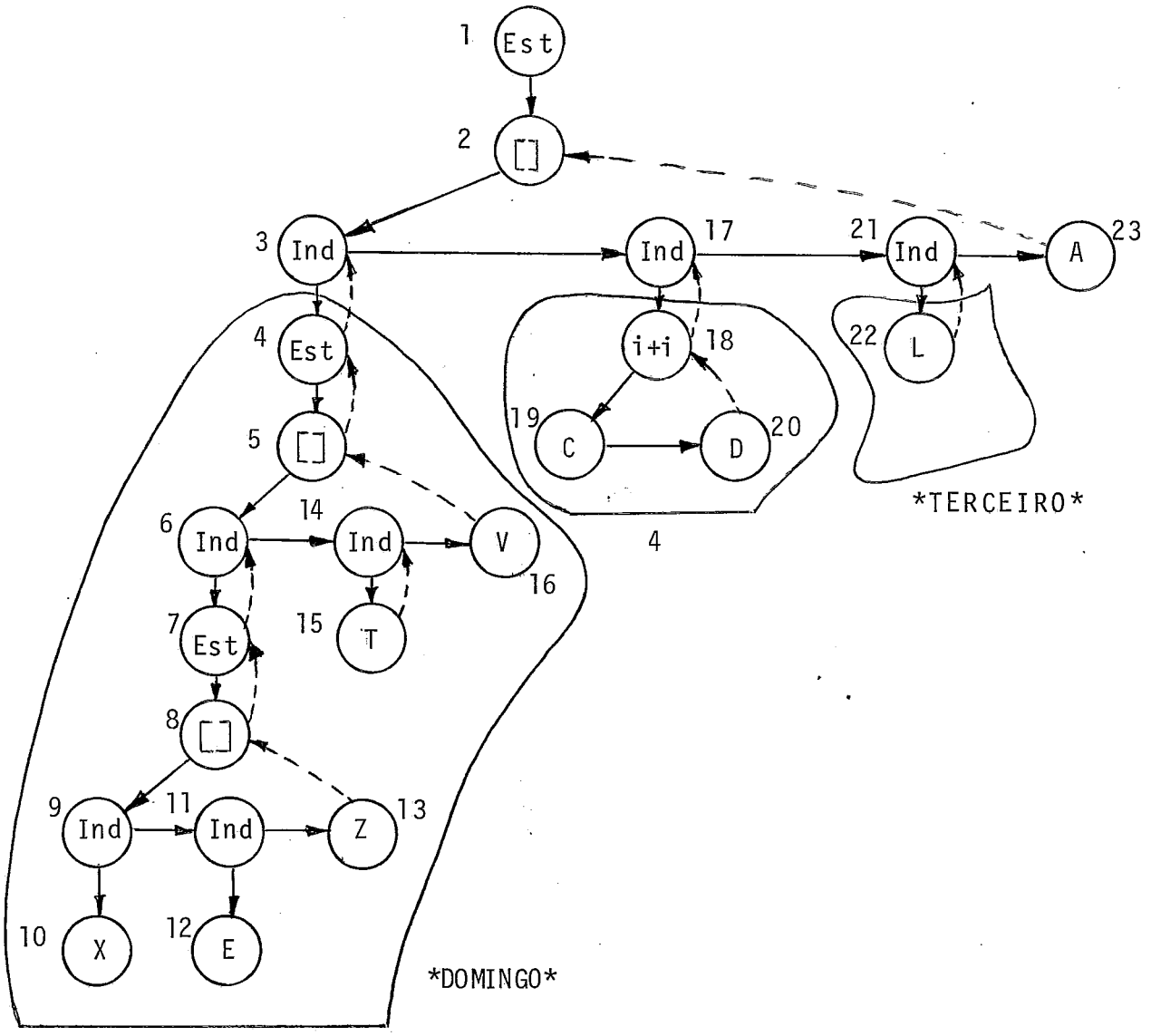


Figura - VII.20

Os nós do tipo Ind contêm, como informação, os tipos das expressões índices (se escalar declarado, o endereço na TS). Por outro lado, é mantido um contador de níveis de índices: ao descer em nó do tipo Ind, faz-se IND:= IND+1 e ao subir, faz-se IND:=IND-1. Subindo em nó do tipo Ind e tendo IND o valor 1, busca-se o valor da expressão índice na PRA, desempilhando-o, após, da mesma, converte-se este valor em função do tipo da expressão índice (envolvendo-o por asteriscos se escalar declarado ou aspas se caráter), faz-se IND:= IND-1, e guarda-se o valor convertido do índice, seguido de vírgula, no vetor MONTANOME. Ao subir em nó do tipo Ind, com IND > 1, apenas faz-se IND:= IND-1. Temos, assim, um mecanismo que permite estabelecer a diferença entre a expressão índice cujo valor convertido interessa para recuperação de nome e aquela cujo valor convertido não interessa.

Destaquemos, agora, as principais ações no percurso da árvore da figura (VII.20).

D1 - Ø

D2 - Como IND vale zero, guarda "[" em MONTANOME.

S3 - Como IND vale 1, toma-se as ações antes descritas para subida em nó Ind com a variável IND valendo 1.

Ver figura (VII.21).

Figura - VII.21

S17 - Análogo a S3.

S21 - Análogo a S3.

Ver figura (VII.22).

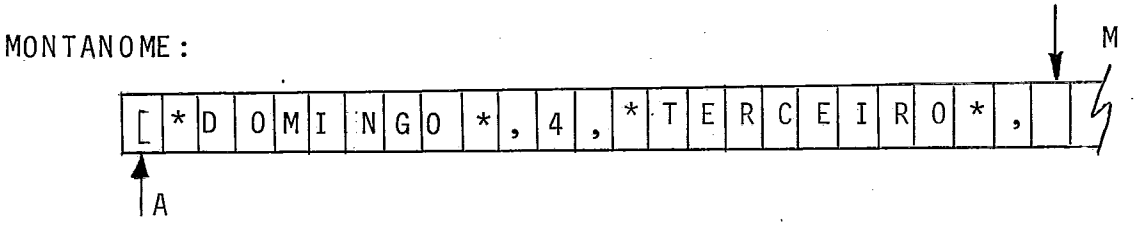


Figura - VII.22

D23 - Como IND vale zero, o conjunto já montado MONTANOME é "shiftado" para a direita do número de posições necessárias para conter o nome do array e este é inserido.

S2 - Como IND vale zero, a última vírgula é superposta pelo símbolo "]"

Ver figura (VII.23).

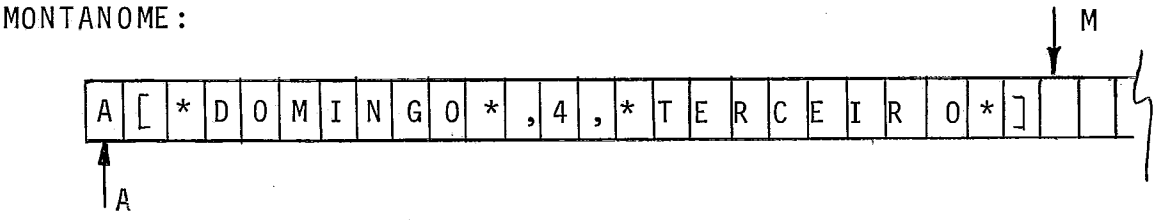


Figura - VII.23

CASO 3 - RECORD

Seja o exemplo da figura (VII.24). A idéia aqui é guardar-se um

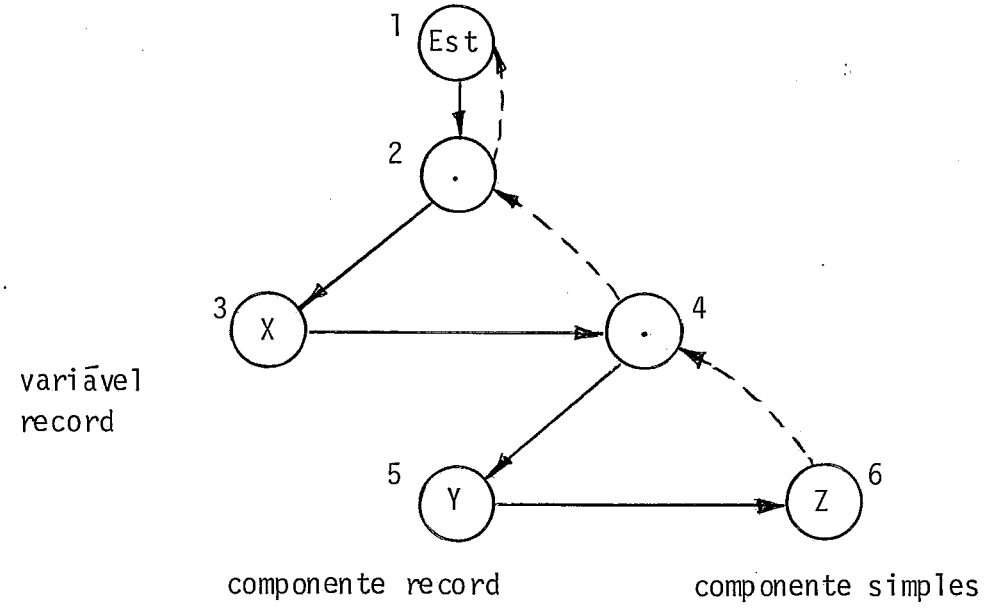


Figura - VII.24

ponto sempre à frente de nomes de componentes, no vetor MONTANOME. Mas não à frente do próprio nome da variável record. Vejamos as ações, apenas quanto a recuperação de nomes:

D1 - Ø

D2 - Ø

D3 - Busca o nome "X" através da TS, guardando-o em MONTANOME.

D4 - Ø

D5 - Como este nó é de componente record:

Guarda "." em MONTANOME e, após, busca o nome "Y" através da TS, guardando-o em MONTANOME.

D6 - Análogo a D5.

S4, S2, S1 - Ø

O resultado final é mostrado na figura (VII.25).

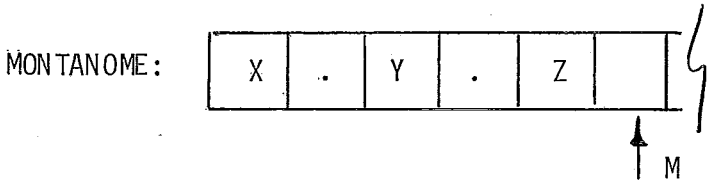


Figura - VII.25

Naturalmente, tais procedimentos foram assim tomados porque IND valia zero, ou seja, o record X não fazia parte de expressão índice de array. Do contrário, a recuperação de seu nome ficaria inibida.

Imaginemos agora que o nó 5, no lugar de ser de componente record, fosse a raiz de uma componente array de records (por exemplo: L[U+V, T[J]]). Neste caso, IND valeria zero no início do percurso de tal subárvore, e o nome seria recuperado analogamente ao do exemplo da figura (VII.20). Por exemplo:"L[3,*QUARTA*]", ficando o nome todo como"X.L[3,*QUARTA*].Z". Há aqui, porém, alguns aspectos a considerar. O ponto que antecede o nome do array (L) é inserido junto e imediatamente antes deste, na ação ao descer no nó de L,porque tal nó não é de array mas de componente array. Por outro lado, a composição do nome da componente array ("L[3,*QUARTA*]"), com o deslocamento necessário, já visto, é feita no próprio vetor MONTANOME. Isto explica o ponteiro A .

Ele aponta para o início do trecho do nome do array em montagem. A figura (VII.26) mostra alguns passos que julgamos autoexplicativos. No último passo, naturalmente, o apontador A já não terá mais função.

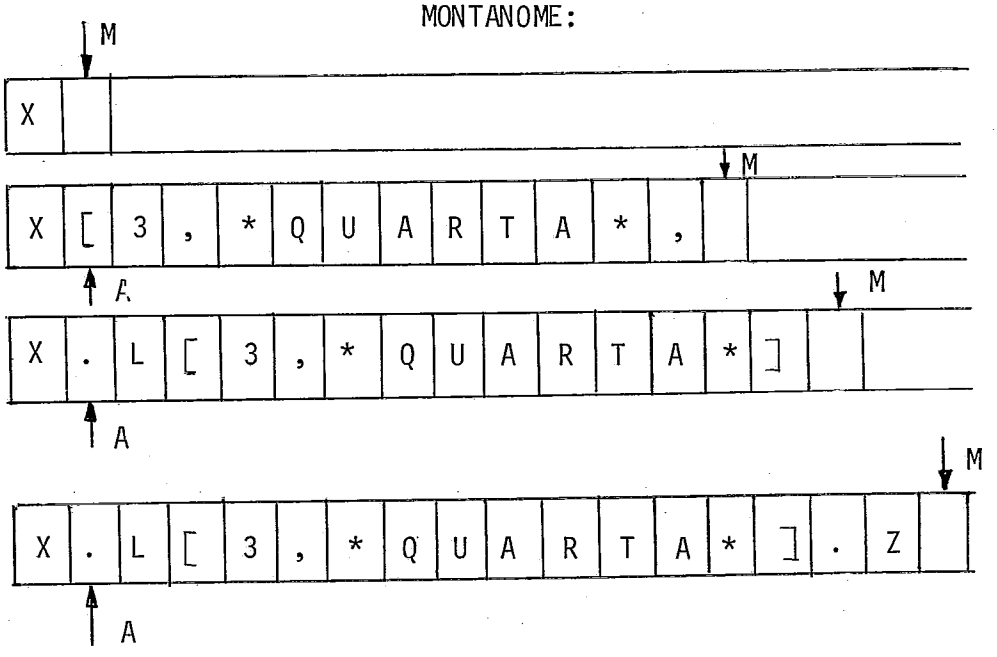


Figura - VII.26

Se tivéssemos records com várias componentes array, nada mudaria. Apenas o ponteiro A iria ser deslocado para outra região de MONTANOME, onde seria construído o nome da outra componente array. Isto é feito ao descer no nó "[ ]" com IND=0. O mesmo pode ser dito para o caso de a componente final do record referenciado ser um elemento simples de array.

Vejamos agora um outro caso, ilustrado pela figura (VII.27).

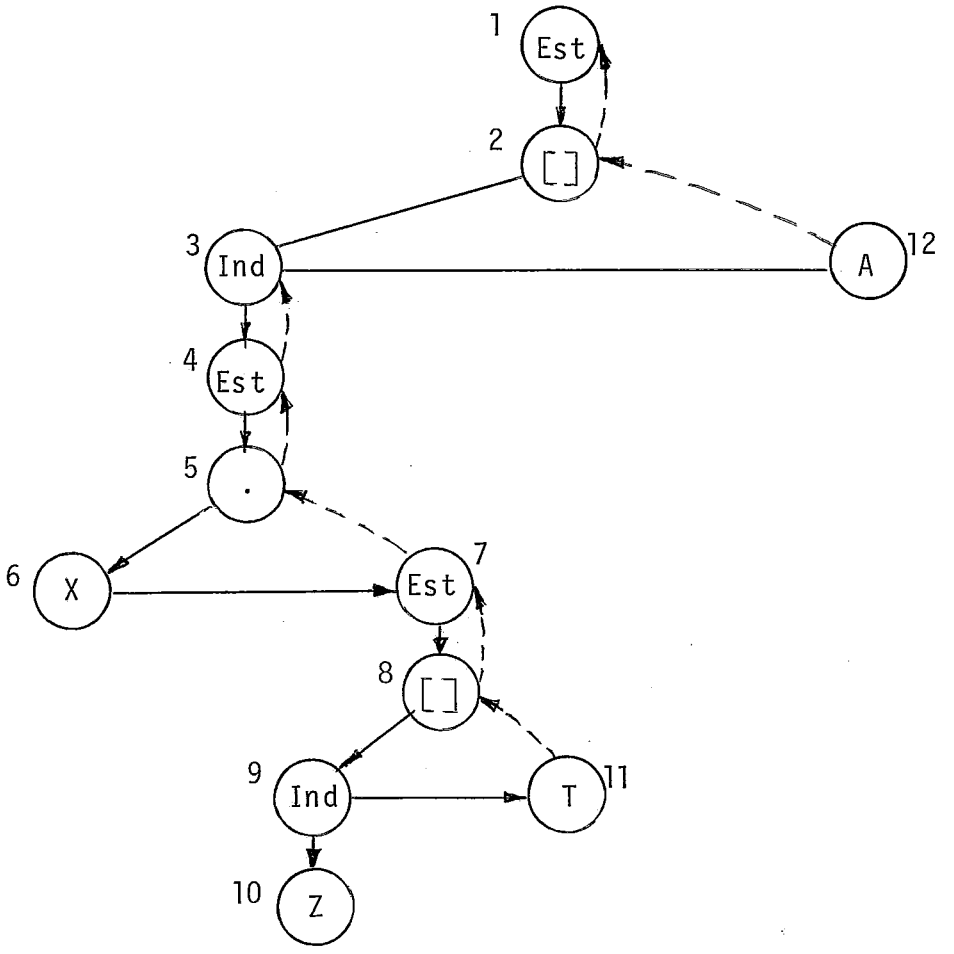


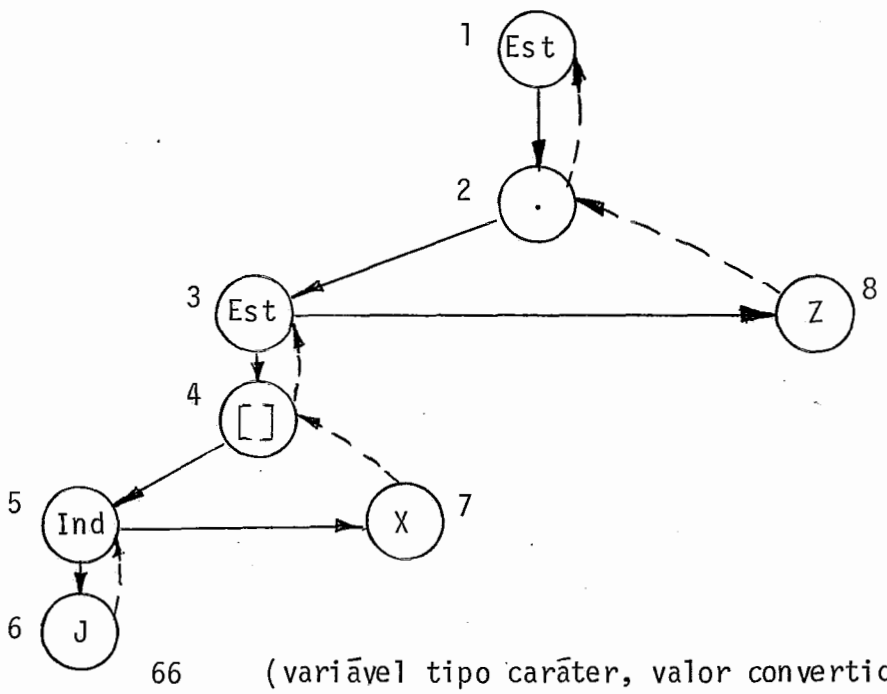
Figura - VII.27

Trata-se de uma referência a A[X.T[Z]] onde nos interessa recuperar, por exemplo, o nome "A[*DOMINGO*]".

Ora: este nome será recuperado naturalmente, já que IND valendo 1 inibirá a recuperação do nome do índice.

Analisemos, agora, o caso do array de records. Este caso não oferece novidade em relação aos demais. Vejamos o exemplo da figura (VII.28). Percorrida a subárvore que tem por raiz o nó 3 resultará em MONTANOME:"X["B"]". Aqui, não foi colocado "."

à frente do nome X porque o nó 7 é de variável array de records e não de componente array de records. Em seguida, em D8, sendo IND=0, interessa o nome Z, que é buscado via TS e agregado ao do array de records, antecedido de ".", resultando: "X["B"].Z".



66 (variável tipo caráter, valor convertido: "B")

Figura - VII.28

CASO 4 - TIPOS DINÂMICOS

Analisemos inicialmente a recuperação de nome para p↑. A figura (VII.29) ilustra a árvore correspondente.

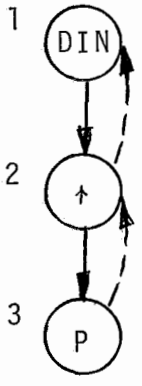


Figura - VII.29

Ações:

D1 - Ø

D2 - Ø

D3 - Busca da TS o nome do ponteiro ("p") e o guarda em MONTANOME.

S2 - Guarda em MONTANOME o símbolo "↑".

S1 - Ø

Analisemos agora o caso em que o apontado seja um array. A figura (VII.30) ilustra a árvore correspondente a p↑[X].

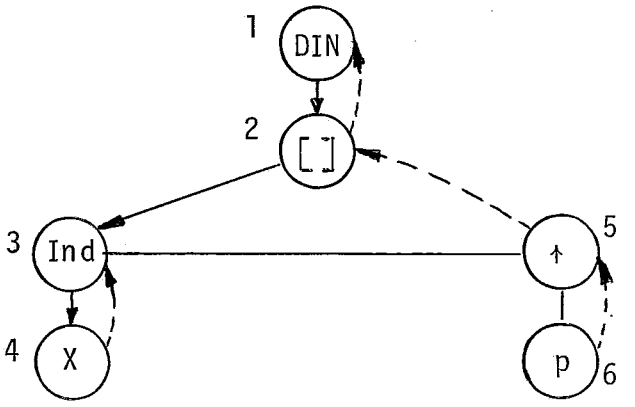


Figura - VII.30

Tudo se passa de forma análoga às referências a elementos simples de array não dinâmico, já vistas. A diferença está no nome do array, que aqui é "p↑". Assim, em D6 salva-se o nome "p", em S5 o símbolo "↑" e em S2 recobre-se a vírgula com o símbolo "]".

Caso p fosse uma componente pointer, seu nome seria precedido de ".".

Seja agora uma referência do tipo p↑.q↑.r. A figura (VII.31) ilustra a árvore correspondente.

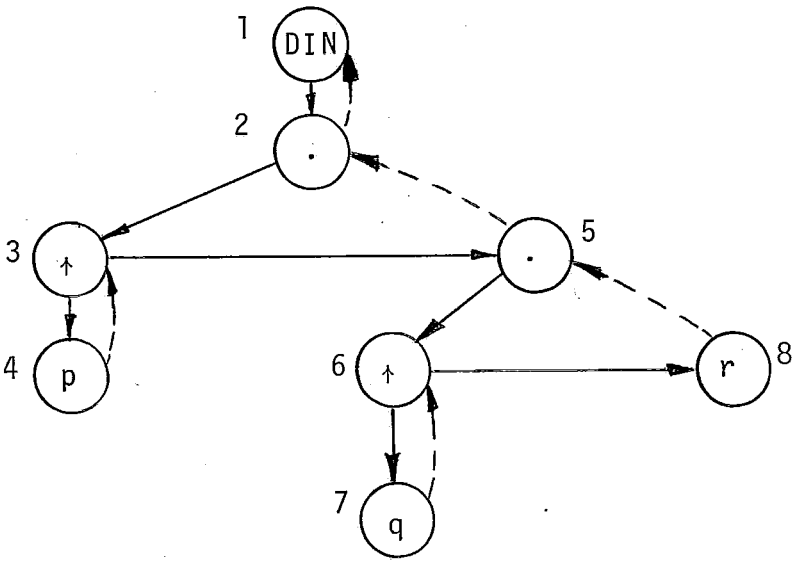


Figura - VII.31

Aqui, em D4 salva-se o nome"p",e em S3 o símbolo "↑"; em D7, como este nó é de componente (pointer em estrutura dinâmica), salva-se "." seguido do nome"q"e em S6 o símbolo "↑"; finalmente, em D8, como tal nó é de componente (simples) salva-se "." seguido do nome r.

Seja agora o caso em que um apontado é um array de pointers: p↑.q[ℓ]↑.r. A figura (VII.32) ilustra a árvore correspondente.

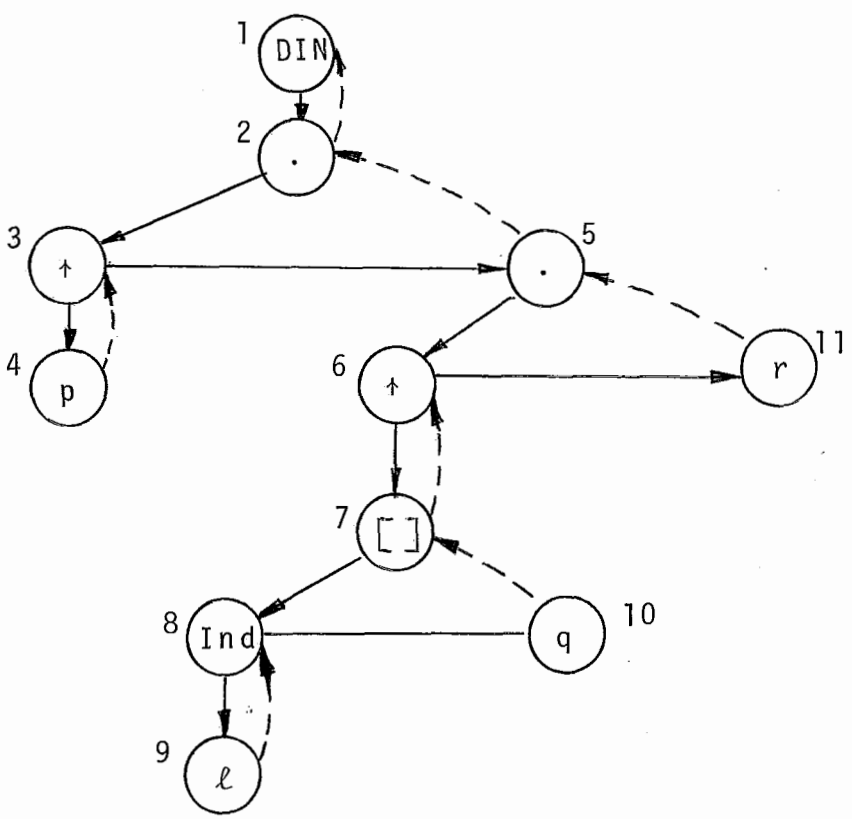


Figura - VII.32

Aqui, em D10 (sendo o nõ de componente array de pointers em estrutura dinâmica), insere-se o nome do array ("q"), antecedido do sĩmbolo ".", "shiftando" a parte dos índices. Em S7 cobre-se "," com "]". Em S6 é inserido em MONTANOME o sĩmbolo "↑". No mais, nada a acrescentar.

Finalmente, cumpre observar que a expressão índice de um array pode conter referência dinâmica. Neste caso, o mecanismo constituido do contador IND inibe a recuperação do nome.

CAPÍTULO VIII

VIII. CONSIDERAÇÕES FINAIS

Foram discutidas ao longo dos sucessivos capítulos desta Tese, quer as características da Forma Intermediária que projetamos, quer as ações do interpretador/depurador no percurso de árvores-FIP. Procuramos, assim, explicitar soluções viáveis para as mais diversas situações, mas, dada a diversidade de propostas com que nos deparávamos, tínhamos que, forçosamente, optar por uma, que nos parecia indicada quanto a algum aspecto (clareza, padronização de tratamento, eficiência de execução, economia de memória, etc), em detrimento de outra, possivelmente superior quanto a alguma outra característica. Este dilema certamente consumiu ponderável parcela do tempo por nós dispendido, mas procuramos ser o mais conscienciosos nas escolhas feitas.

Por outro lado, além de todo critério adotado na elaboração das rotinas, tivemos por objetivo fundamental documentar, para a comunidade interessada, uma moderna concepção de Forma Intermediária e a mecânica de funcionamento de um interpretador/depurador. Preocupamo-nos em que as propostas apresentadas, afora a eficiência, fossem as mais claras possíveis, mesmo que, em alguns casos, sacrificássemos um pouco o primeiro aspecto (constante uso de flags, por exemplo). Não resta dúvida de que na implementação pequenas otimizações poderão ser feitas.

O interpretador/depurador, aqui proposto, não é, certamente, a melhor opção para o processamento quotidiano! Para es

te, está reservado o terceiro módulo do projeto, que se pretende venha a ser bastante eficiente. Todavia, o programador que o utilizar criteriosamente estará, sem dúvida, diante de um poderoso ferramental de auxílio à depuração de suma importância para o bom andamento dos trabalhos de um Centro de Desenvolvimento de Sistemas.

Algumas características da linguagem não foram propostas e pensamos em expandir, futuramente, os recursos de depuração.

Quanto à linguagem, basicamente não propusemos:

. O uso do WITH. Todavia qualquer programa em PASCAL poderá ser escrito sem este recurso, sem detrimento à sua qualidade final. Apenas, um pequeno esforço adicional do programador será requerido.

. Records variáveis. Esta é, sem dúvida, uma restrição, a ser eliminada em versão posterior do APP. Sugerimos que, para sua implementação, sejam reservados, na área do record, espaços consecutivos para as diversas alternativas. A figura (VIII.1) ilustra um record com duas componentes mutuamente exclusivas.

. Packed Array. Esta é outra característica, útil, a ser implementada na primeira versão do APP, embora não discutida no âmbito deste trabalho, no qual apresentamos apenas a representação de strings como packed arrays implícitos.

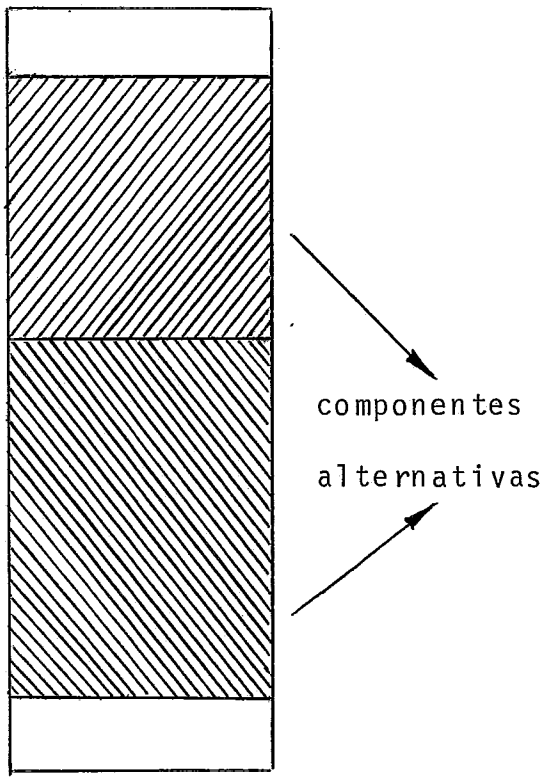


Figura - VIII.1

. O tipo File. Esta é uma restrição comum em versões primeiras de compiladores PASCAL. Como já foi dito anteriormente, optamos por implementar, nesta versão inicial, exclusivamente o tipo textfile . No texto reservado à descrição da entrada e saída, nos dedicamos aos textfiles input e output, com os quais se poderão "rodar" programas com entrada / saída elementar, comprovando a eficácia do interpretador/depurador. Entendemos,todavia, que já na versão inicial deste módulo, a implementar, a generalização de textfiles para outros arquivos deverá ser feita, tarefa que não oferece maior desafio , mas para cuja definição completa requererá o conhecimento da máquina "alvo", já que muitas especificações de entrada e saída (nome de arquivo, blocagem, tamanho de registro, etc) são feitas no corpo do programa de forma padronizada.

Por outro lado, a não implementação do tipo File, criou também uma restrição quanto ao uso de uma estrutura de dados, já que nada impede que se tenha variáveis File internas.

. Passagem de Nomes de Procedures/Functions como parâmetros. Este é um recurso que deliberadamente deixamos para outra versão deste APP, por ser seu uso mais restrito.

. Procedure Dispose. Este é um recurso útil, mas que também constatamos não ser comumente implementado em versões iniciais de compiladores PASCAL. Para uma segunda versão deverá, sem dúvida, ser incorporado.

Quanto aos recursos de depuração, pensamos em, futuramente, partir para uma segunda etapa mais ambiciosa.

A nível mais elementar, poderemos incorporar declarações de objetos para os quais se poderá pedir dump e ligar e desligar trace/save; expandir o recurso de traceback, pela criação de vários arquivos circulares, independentes, com capacidades definidas pelo programador; permitir a solicitação de trace/save para labels, procedures e functions individualmente escolhidos; criar um comando alternativo ao WHEN, a que poderíamos denominar WHILE, que manteria o trace ou o save ligado nos trechos em que a expressão booleana associada fosse verdadeira; etc.

A nível mais profundo, poderemos pensar num rastreio do programa com impressão do seu texto, passo a passo; numa iteração entre programador e programa, via terminal, com possibilidade de alteração do texto fonte; em rotinas de

Sistema Operacional que introduzam facilidades ao Ambiente; na criação de ferramentas gerenciais para controle da produtividade do projeto; etc.

Como se vê, temos um longo caminho pela frente na busca do verdadeiro Ambiente de Desenvolvimento. Mas a semente está lançada. Esperamos que frutifique!

BIBLIOGRAFIA

|1| AHO, V.A. e ULLMAN, J.D. - The Theory of Parsing, Translation and Compiling, volumes 1 e 2, Prentice Hall, 1973.

|2| AHO, V.A. e ULLMAN, J.D. - Principles of Compiler Design, Addison-Wesley, 1978.

|3| BAUER, F.L e Outros - Compiler Construction, An Advanced Course, Springer-Verlag, 1976.

|4| GRIES, D. - Compiler Construction For Digital Computers, Wiley International Edition, 1971.

|5| FAIRLEY, R.E. - ADA Debugging and Testing Support Environments, Sigplan Notices, Volume 15, Número 11, nov. 1980.

|6| BARRON, D.W. (Chairman) e outros - PASCAL - The Language and its Implementation, Proceedings of the Simposium held at the University of Southampton, 1977.

|7| CHUNG, K e YUEN, H. - A "Tiny" Pascal Compiler, Part 1: The P-Code Interpreter, Byte, setembro de 1978.

|8| CHUNG, K. e YUEN, H. - A "Tiny" Pascal Compiler, Part 2: The P-Compiler, Byte, outubro de 1978.

|9| CHUNG, K. e YUEN, H. - A "Tiny" Pascal Compiler, Part 3: P-Code to 8080 Conversion, Byte, novembro de 1978.

|10| WASSERMAN, A.I. - Automated Development Environments, Computer, Volume 14, Número 4, IEEE Computer Society, 1981.

|11| WIRTH, N - The Design of a Pascal Compiler, Software - Practice and Experience, Volume 1, 309-333, 1971.

|12| DE SIMONE, E. e TELES A.A.S. - Gerador de Analisadores Sintáticos RRP LL(1), Anais do 8º SEMISH, 1981.

|13| JENSEN, K. e WIRTH, N. - PASCAL, User Manual and Report, Springer-Verlag, 1978.

|14| GROGONO, P. - Programming in PASCAL, Addison-Wesley, 1978.

|15| WELSH, J., SNEERINGER, W.J. e HOARE, C.A. - Ambiguities and Insecurities in Pascal, Software-Practice and Experience, Volume 7, 685-696, 1977.