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.

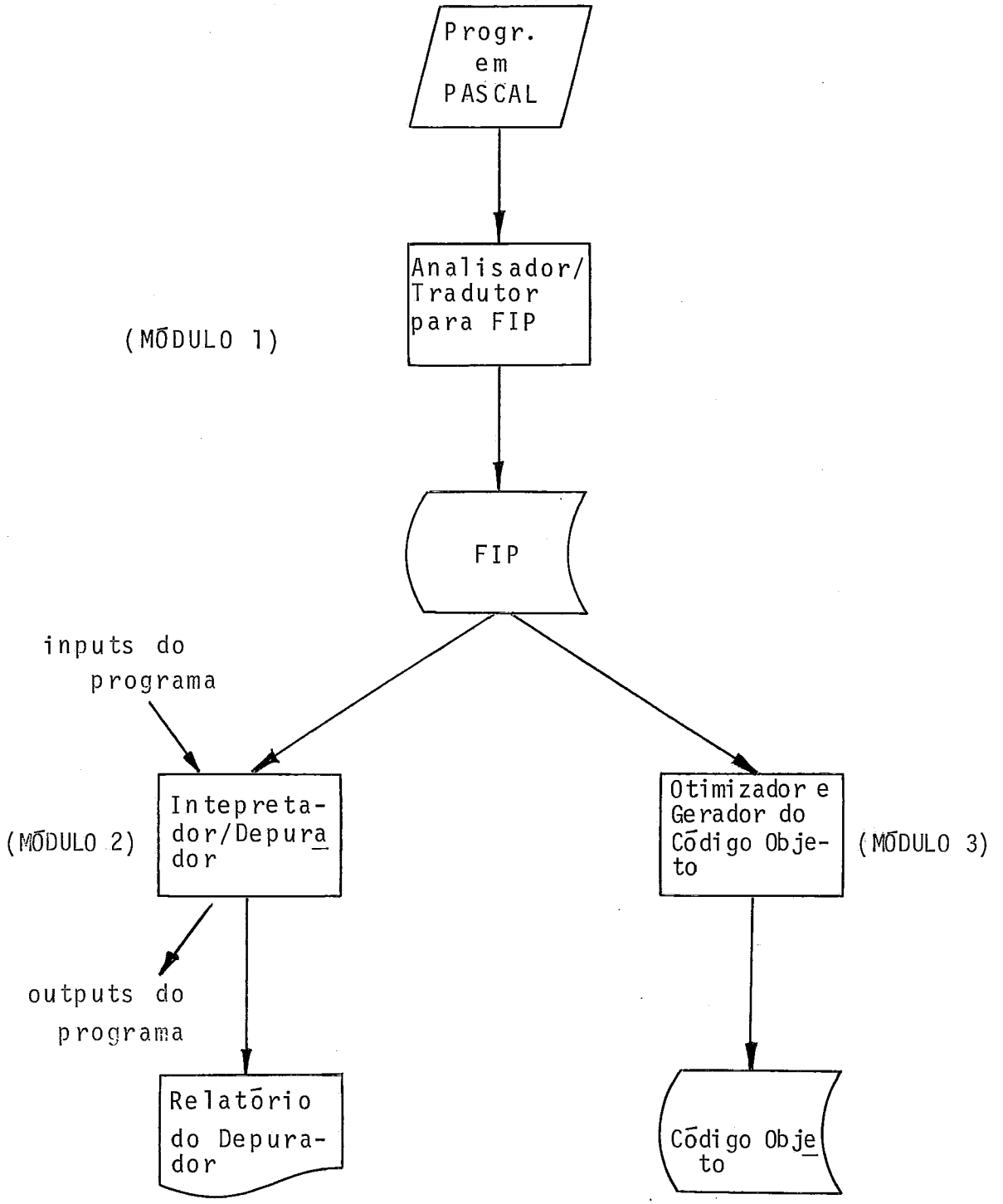

Progr.
em
PASCAL
Analisador/
Tradutor
para FIP
(MÓDULO 1)
FIP
inputs do
programa
Intepreta-
dor/Depura
dor
(MÓDULO 2)
Otimizador e
Gerador do
Código Obje-
to
(MÓDULO 3)
outputs do
programa
Relatório
do Depura-
dor
Código Obje
to

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.

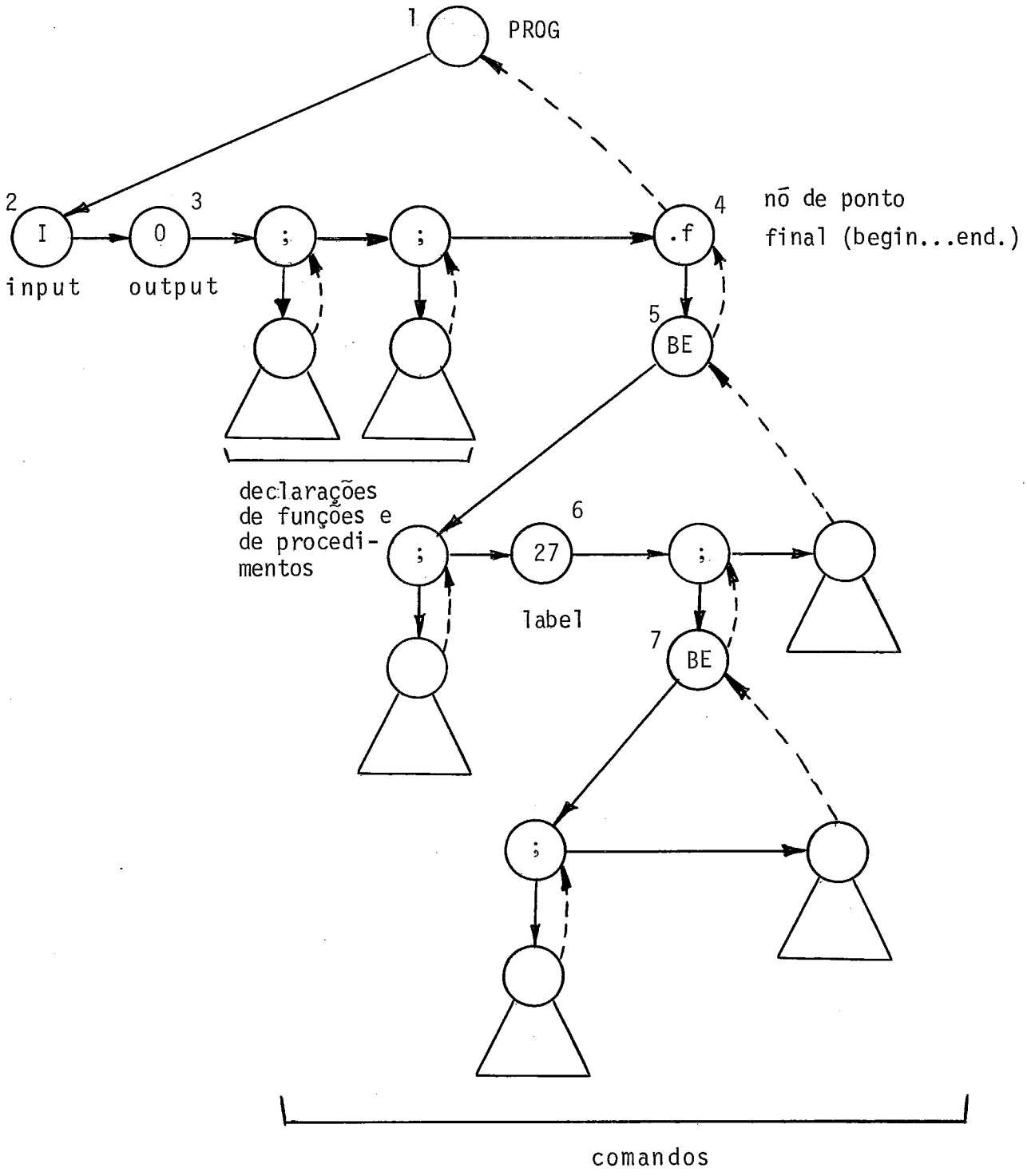

1
PROG
2
I
input
3
O
output
;
;
4
.f
nó de ponto
final (begin...end.)
5
BE
declarações
de funções e
de procedi-
mentos
6
27
label
;
;
7
BE
;
comandos

Figura - II.1

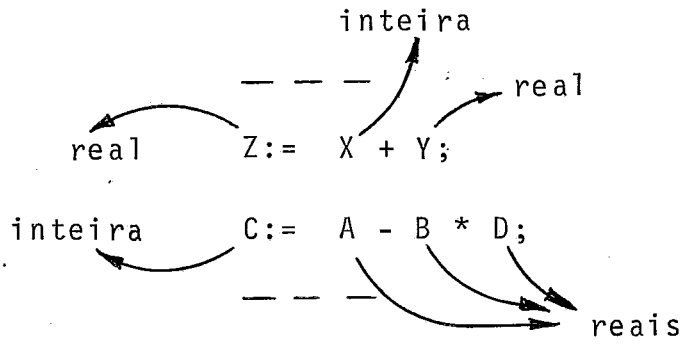


Figura - II.2

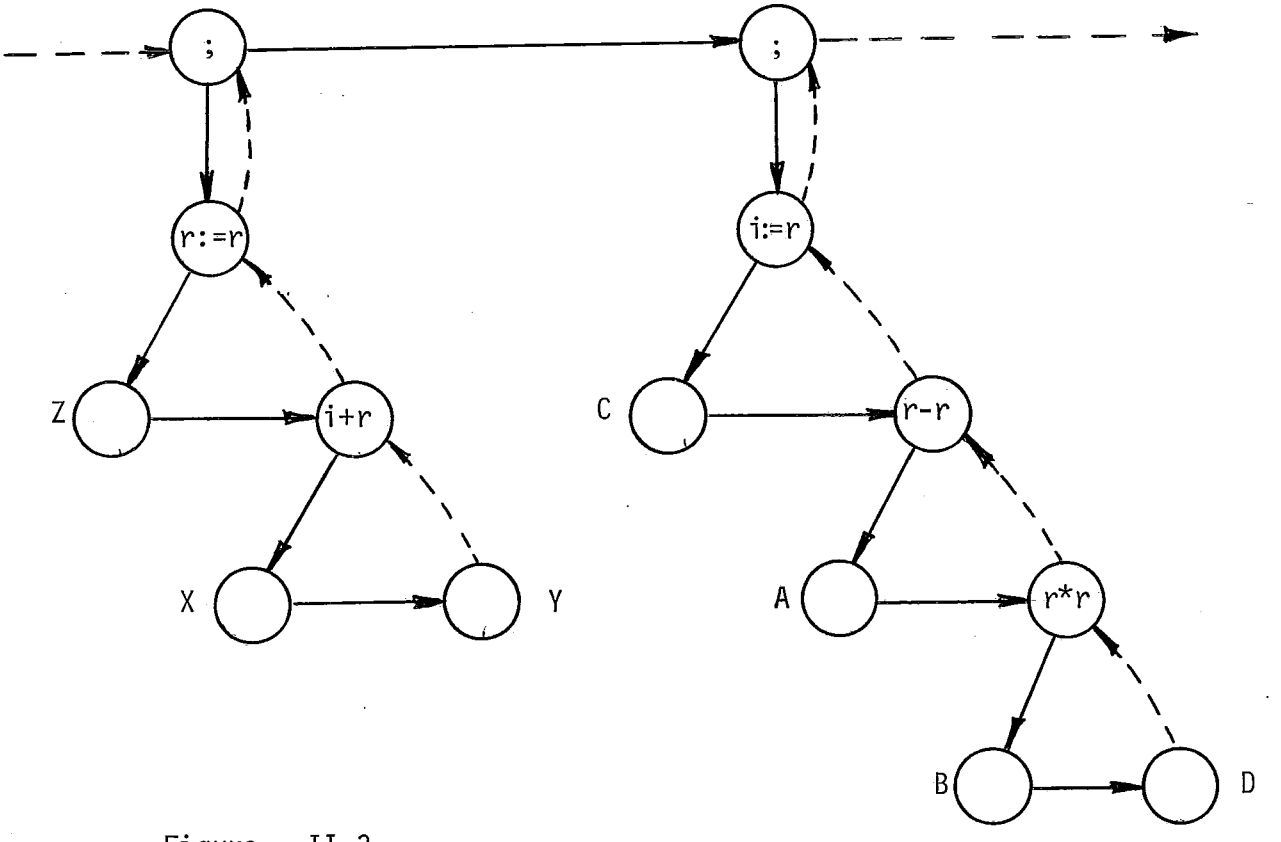


Figura - II.3

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).

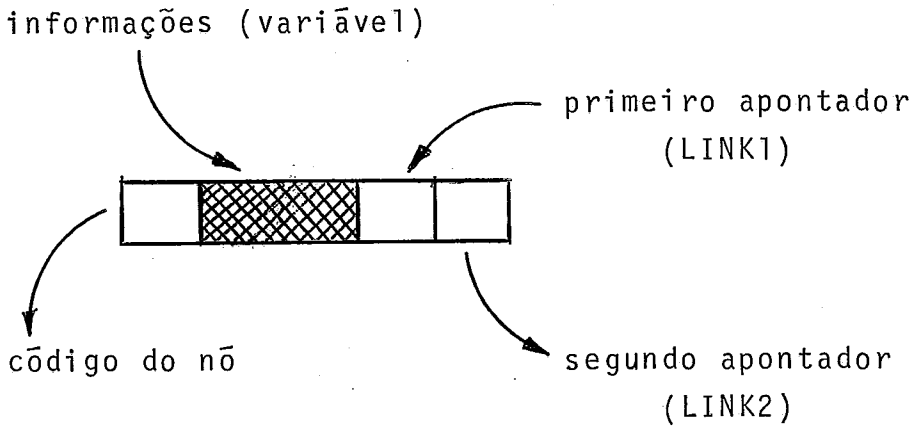


Figura - II.4

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).

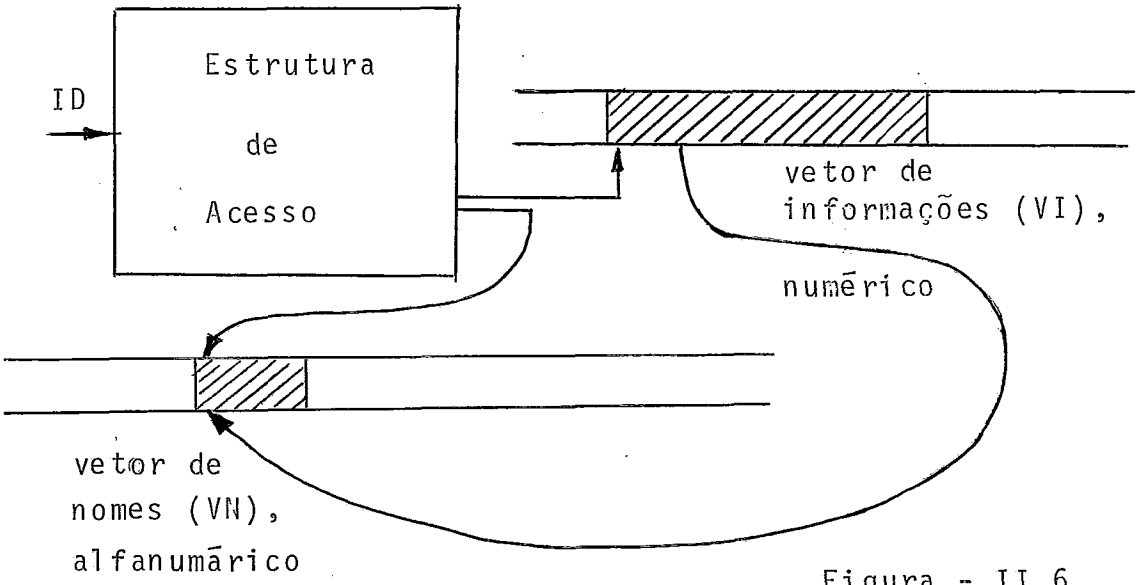


Figura - II.6

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.

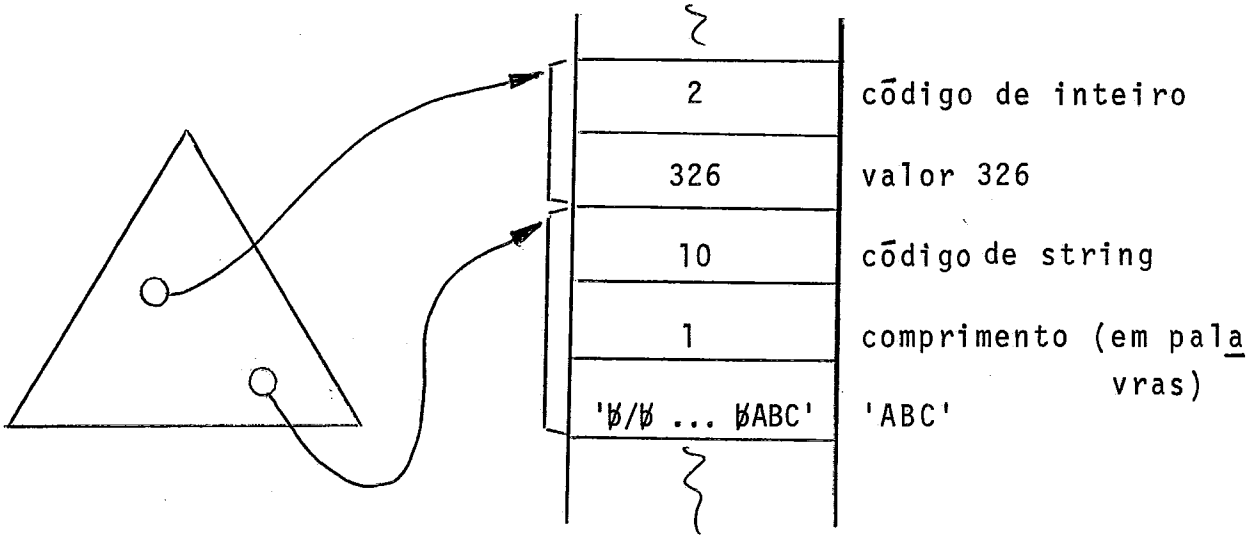


Figura - II.7

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.

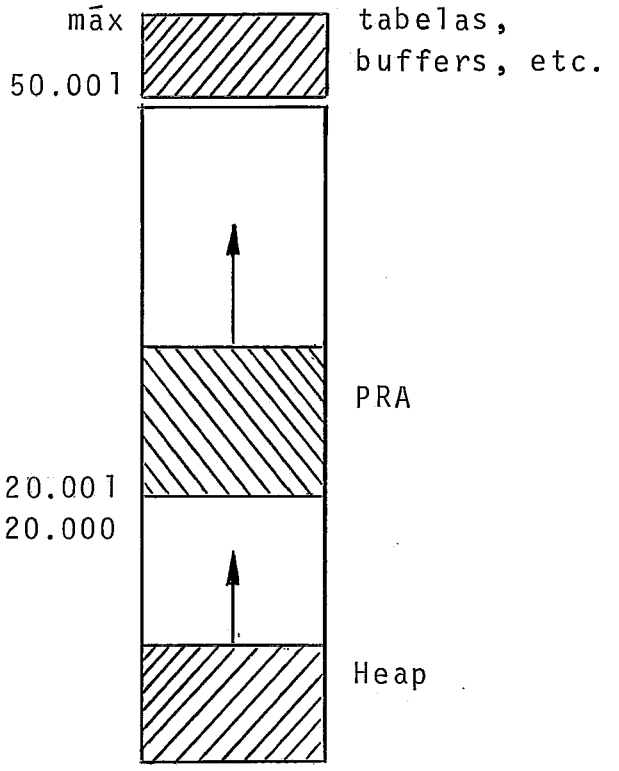


Figura - II.8

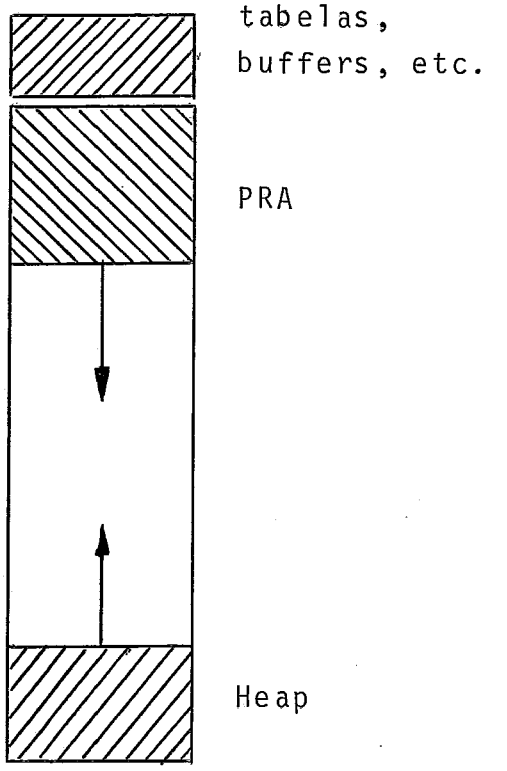


Figura - II.9

II.VII. QUANTO À GERÊNCIA DA PILHA DE RAs

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

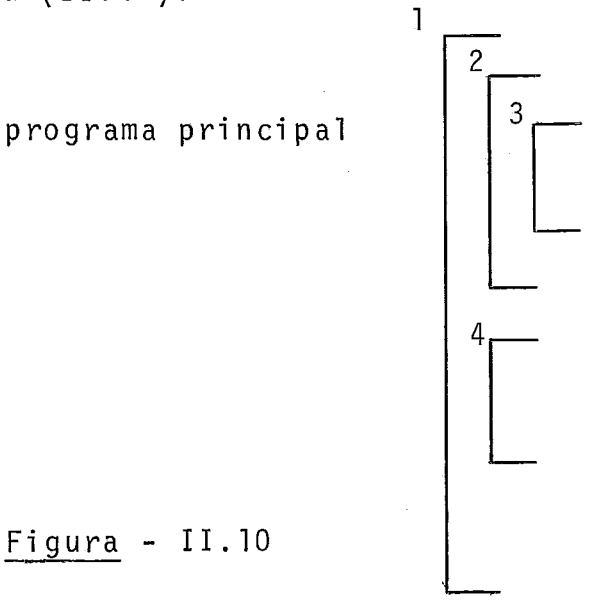


Figura - II.10

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).

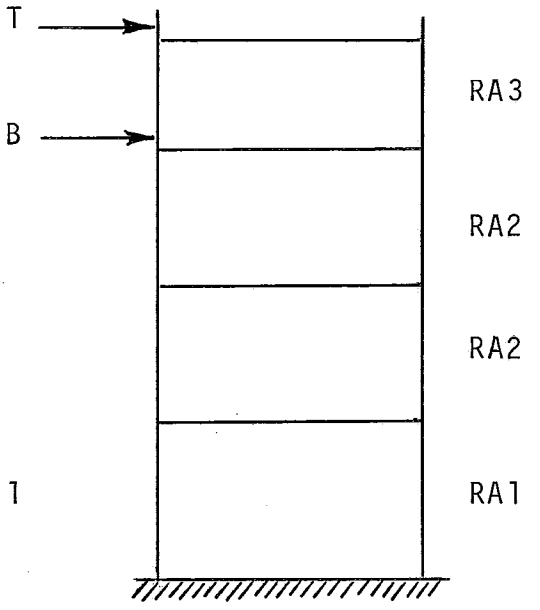


Figura - II.11

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.

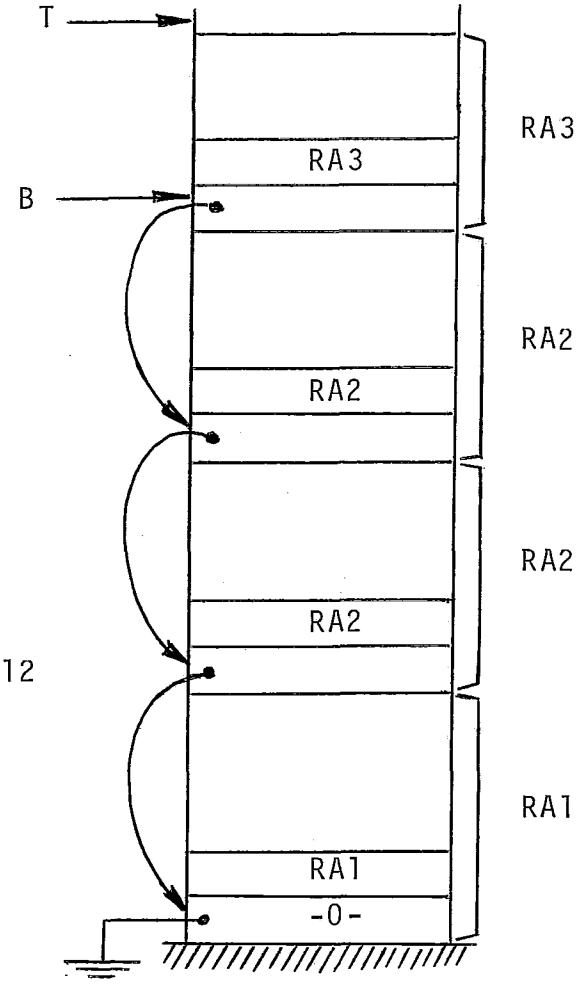
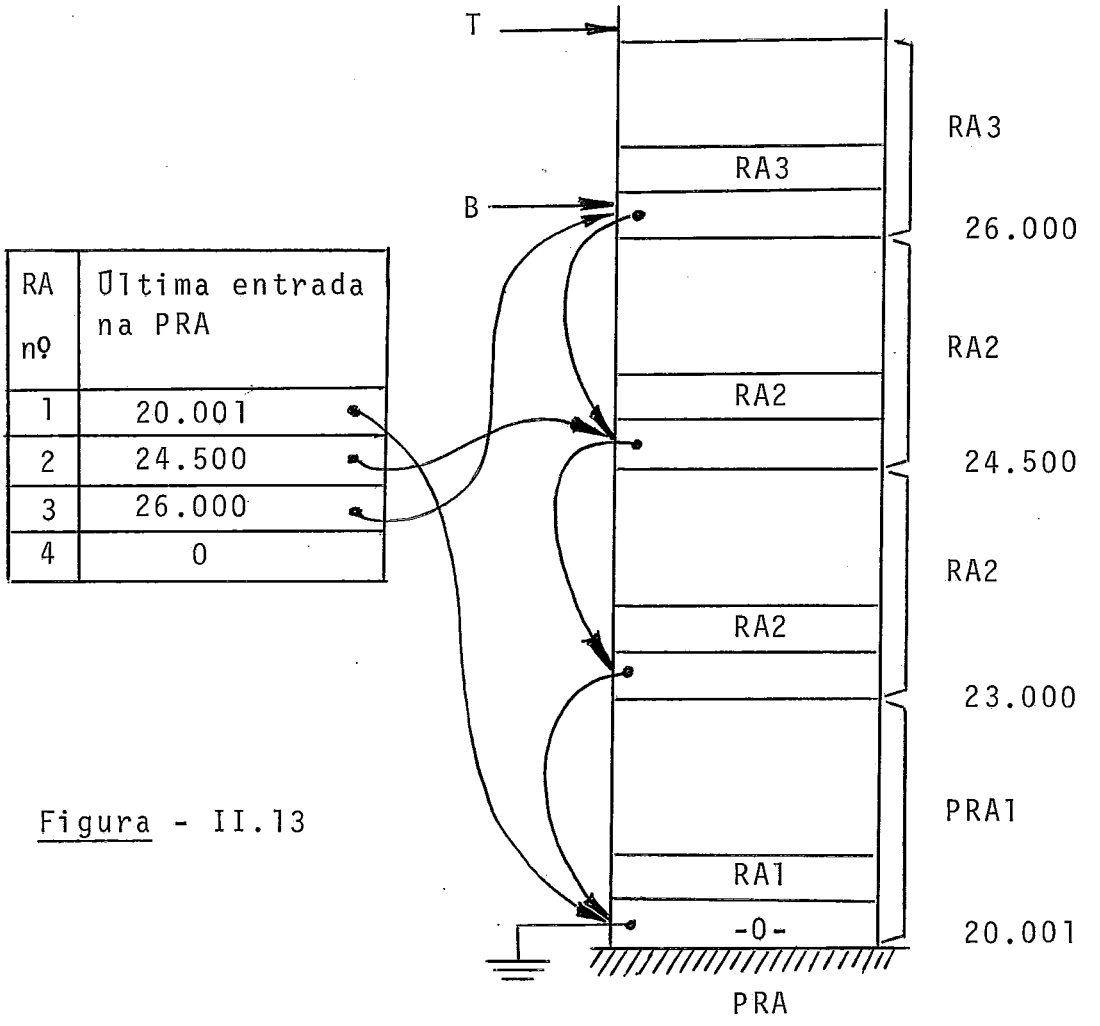


Figura - II.12

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).