AMBIENTE DE PROGRAMAÇÃO PASCAL: ANALISADOR E TRADUTOR PARA UMA FORMA INTERMEDIÁRIA PASCAL

Eliane Camara Cerveira

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS (M.Sc.) EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. José Lucas Mourão Rangel Netto
Orientador

Profa. Sueli Mendes dos Santos

Prof. Estevam Gilberto de Simone

RIO DE JANEIRO, RJ - BRASIL
JANEIRO DE 1985

CERVEIRA, ELIANE CAMARA

Ambiente de Programação PASCAL: Analisador e Tradutor para Forma Intermediária (Rio de Janeiro) 1985.

v., 230 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1985).

Tese - Universidade Federal do Rio de Janeiro, COPPE.

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

Eliana Câmara Carneira

Aos meus pais.

AGRADECIMENTOS

A Estevam G. de Simone e José L.M. Rangel Netto pelo estímulo e dedicação na orientação desta tese;

A Lourdes pelo excelente trabalho de datilografia;

A Salvador pela colaboração na digitação da tese;

Ao meu companheiro José Carlos, pela paciência e pelo apoio dispensado durante esta fase;

Ao meu filho Raphael, razão da conclusão desta tese.

Muito Obrigado.

Resumo da Tese Apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

AMBIENTE DE PROGRAMAÇÃO PASCAL: ANALISADOR E TRADUTOR PARA UMA FORMA INTERMEDIÁRIA PASCAL

Eliane Camara Cerveira

Janeiro, 1985

Orientador: José Lucas Mourão Rangel Netto
Programa: Engenharia de Sistemas e Computação

RESUMO

Apresentamos aqui a especificação para implementação de um Analisador e Tradutor para uma Forma Intermediária PASCAL, que se pretende venha a fazer parte de um Ambiente de Programação PASCAL, projeto em andamento na COPPE/UFRJ. Tal projeto inclui também um Interpretador/Depurador e um Gerador/Otimizador de código.

Para esse fim, está especificada aqui uma proposta de Forma Intermediária PASCAL (árvore de sintaxe abstrata, com atributos coletados) e sua geração. Além disso estão descritas as ações de Análise Sintática e Semântica.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Master of Science (M.Sc.).

PASCAL PROGRAMMING ENVIRONMENT: TRANSLATOR

Eliane Camara Cerveira

January, 1985

Supervisor: José Lucas Mourão Rangel Netto
Department: Engenharia de Sistemas e Computação

ABSTRACT

We present here the specification for implementing of a Analyser and Translator to PASCAL, which is meant to be part of a PASCAL Programming Environment, presently being developped at COPPE/UFRJ. This project will also include a Interpreter/Debugger 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 collected atributes). We have also included a description of actions taken for semantics checking and parsing.

INDICE

p.

CAPÍTULO I - Ambiente de Programação PASCAL - Descrição do Projeto 1

CAPÍTULO II - Aspectos Gerais 6

CAPÍTULO III - Analisador Léxico 11

CAPÍTULO IV - Analisador Sintático 19

CAPÍTULO V - Tabela de Símbolos 37

CAPÍTULO VI - Analisador-Semântico/Tradutor..... 50

CAPÍTULO VII - Packages 184

CAPÍTULO VIII- Considerações Finais 196

BIBLIOGRAFIA..... 198

APÊNDICE I - Gramática PASCAL 200

APÊNDICE II - Gramática PASCAL RRP 208

APÊNDICE III - Autômatos Finitos 211

CAPÍTULO I

AMBIENTE DE PROGRAMAÇÃO PASCAL - DESCRIÇÃO DO PROJETO

Temos observado que nos últimos anos tem havido uma mudança das prioridades no desenvolvimento de sistemas de computação; isto se explica pelo barateamento do hardware. Atualmente a preocupação está voltada para o software, com o objetivo de viabilizar uma economia de tempo das tarefas de responsabilidade dos programadores e analistas de sistemas e, também, a de aumentar a confiabilidade do software gerado.

Com essa motivação, têm sido criadas linguagens de Alto-Nível que possuem, na sua própria especificação, os princípios de estruturação, além do conceito de tipo definido pelo programador, forçando assim uma disciplina sistemática na programação. Outro fato que podemos citar, e que também reflete a preocupação com o processo de produção de software, é a criação de ambientes de desenvolvimento que objetivam, justamente, aumentar a produtividade do pessoal envolvido na confecção de sistemas, provendo software como uma ferramenta capaz de simplificar o processo de produção.

A linguagem PASCAL, introduzida por Niklaus Wirth, foi a primeira a incluir esses conceitos e, devido a tais características, tem sido utilizada internacionalmente nos cursos de computação.

Outra linguagem, criada recentemente, que não podemos deixar de mencionar é a linguagem ADA, cuja definição baseia-se na da linguagem PASCAL, acrescida ainda de facilidades que se tornaram necessárias para uma linguagem de uso mais geral, permitindo processamento paralelo, distribuído, para sistemas de grande porte.

Em meados de 1980, um grupo de professores da área de Compiladores da COPPE/UFRJ, reuniu-se para desenvolver um projeto de um Ambiente de Desenvolvimento (baseado no artigo Automated Development Environments [14]), para computadores de fabricação nacional.

Os pontos principais deste projeto são:

1) Programação em módulos, compilados separadamente.

Com essa facilidade, uma tarefa de programação pode ser subdividida em várias tarefas menores, a serem executadas pelo mesmo programador ou por diversos programadores, em um trabalho de equipe.

2) Compilação de uma interface para módulos relacionados.

Para confecção de sistemas complexos, realizados geralmente por uma equipe que necessite seguir uma determinada especificação.

3) Possibilidade de restringir a consulta e o acesso a determinadas informações.

4) Análise Compreensiva.

A compilação de um módulo deve gerar mensagens capa

zes de ajudar o programador a corrigir os erros cometidos, de uma só vez.

5) Facilidades de depuração e execução controlada.

O sistema terá um Interpretador/Depurador capaz de permitir ao usuário a execução controlada de um Sistema, permitindo sua monitoração controlada, através de comandos específicos para esse fim. Tais comandos têm, sistematicamente, a forma de comentários especiais, que só serão levados em conta pelo Interpretador/Depurador e não o sendo pelo Gerador de código.

6) Facilidades para geração/otimização de código, compilado ou interpretado, para diversas máquinas.

A linguagem escolhida para o projeto foi o PASCAL, a mais adequada dentro do contexto do projeto, o ambiente universitário, e por ser possuidor de todas as qualidades necessárias a implementação deste projeto.

O projeto foi dividido em três módulos, que originaram três teses, as quais se propõem, numa fase inicial, a desenvolver apenas um Ambiente de Programação PASCAL, a ser utilizado como base para a realização deste projeto. Para esta primeira etapa do projeto, algumas simplificações tiveram que ser feita e serão facilmente constatáveis no corpo deste trabalho.

Na Figura (I.1) podemos ver o esquema geral do projeto inicial e também a divisão deste, nos 3 módulos, que têm como ponto comum a forma intermediária PASCAL.

O módulo 1 analisa o programa Fonte PASCAL e o tra-

duz para a Forma Intermediária PASCAL (FIP).

O módulo 2 recebe como entrada a FIP executando-a de 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 é responsável pela otimização da FIP e geração do código da máquina.

Esta tese corresponde ao módulo do Analisador/Tradutor para Forma Intermediária PASCAL.

A tese que corresponde ao módulo 2, o Interpretador/Depurador já foi concluída e pode ser encontrada na referência |11|.

Em linhas gerais podemos mencionar o fato de a FIP conter agora, os atributos dos identificadores, e os nós em vez de apontarem para uma tabela, apontam para o nó correspondente a sua declaração na própria FIP. Em relação a gramática, além de ser considerada toda a especificação padrão do PASCAL, acrescentou-se o conceito de Packages, que merecerá neste trabalho, um capítulo próprio.

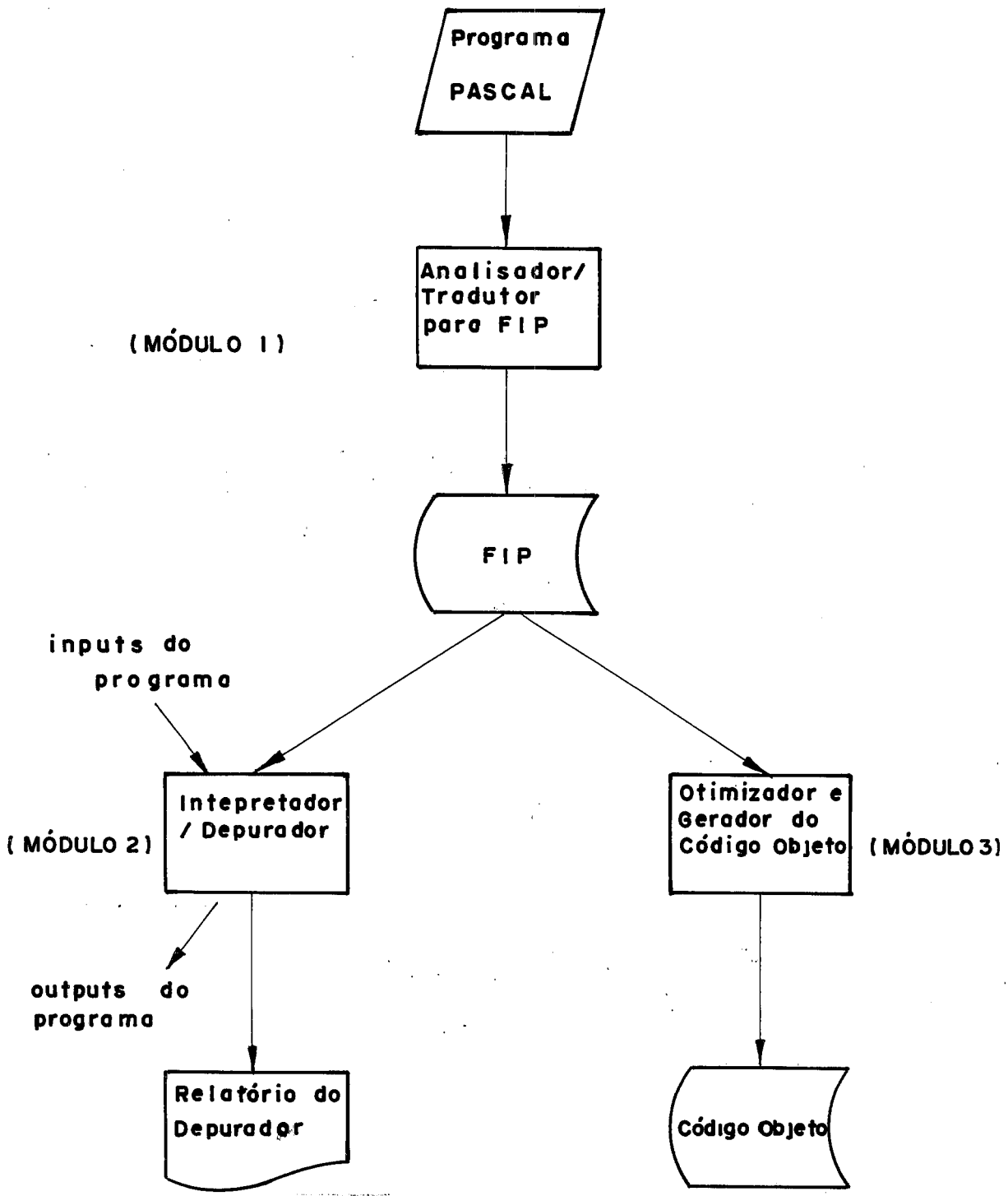

Programa
PASCAL
Analisador/
Tradutor
para FIP
(MÓDULO 1)
FIP
inputs do
programa
Intepretador
/ Depurador
(MÓDULO 2)
Otimizador e
Gerador do
Código Objeto
(MÓDULO 3)
outputs do
programa
Relatório do
Depurador
Código Objeto

Figura I.1

CAPÍTULO II

ASPECTOS GERAIS

Neste capítulo trataremos da estrutura funcional de nosso módulo. A Figura (II.1) apresenta um diagrama que será útil para sua visualização.

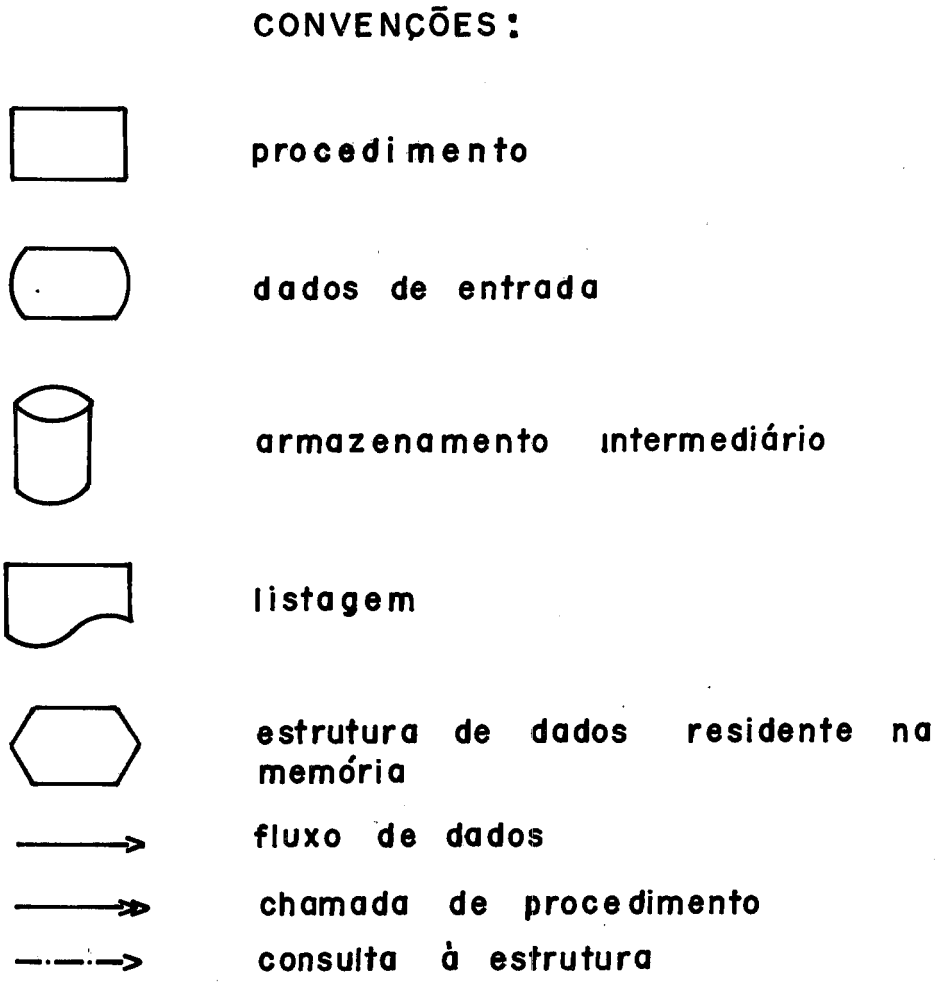


Figura II.1

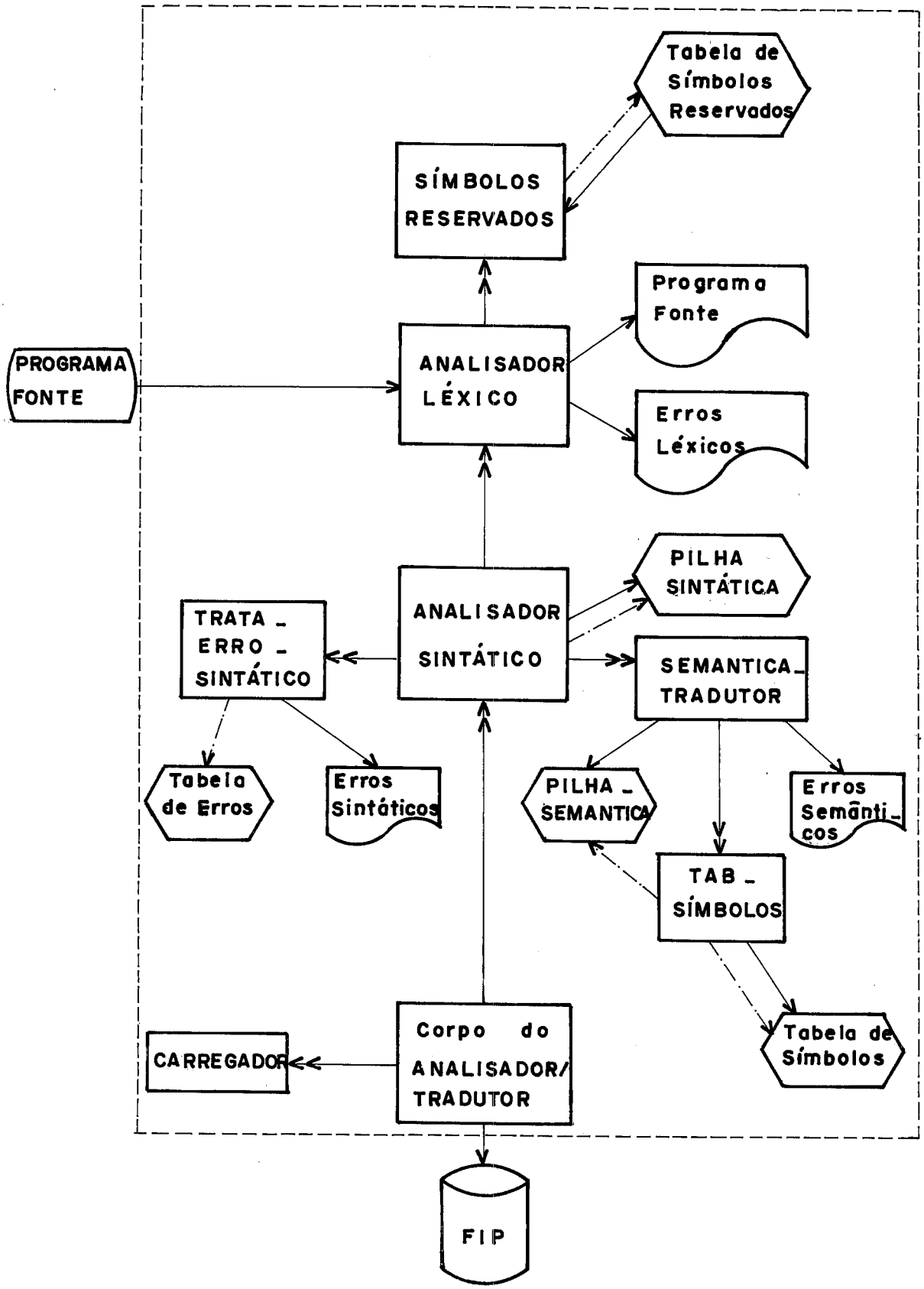


Figura II.1 (continuação)

Vejamos uma breve explanação desse diagrama.

A rotina central do ANALISADOR/TRADUTOR é, claramente, o ANALISADOR-SINTATICO (abreviado por A.S.). Este será chamado dentro do corpo deste módulo, e através da execução de um loop, só sairá quando o programa fonte for totalmente analisado, ou seja atingido um número limite de erros, ou devido a uma terminação anormal.

O ANALISADOR-LÉXICO (abreviado por A.L.) será chamado pelo A.S. cada vez que este necessitar um ítem léxico. Para isso o A.L. lê o programa fonte e constrói o TOKEN, parâmetro de saída, com informações sobre o ítem lido. Os itens léxicos formados pelos identificadores necessitam que sejam diferenciados em identificadores definidos pelo programador e palavras reservadas. Para isso, o A.L. consultará uma tabela onde encontrará todos os identificadores reservados da linguagem adotada; a inicialização desta tabela será realizada no corpo principal deste módulo.

O A.S., a ser utilizado neste trabalho, utiliza o método RRPLL(1); sua tabela de controle será gerada pelo Gerador de Analisadores Sintáticos RRPLL(1) do projeto NHAONHAO, da COPPE/UFRJ [13]. Esta tabela encontra-se em formato de lista e também será inicializada no corpo principal deste módulo. Mensagens especificando os erros sintáticos detectados pelo A.S. serão emitidas pelo Recuperador de Erros, que se utiliza, basicamente de inserções para se recuperar de um erro ocorrido.

A saída deste módulo se constitui na Forma Intermediária PASCAL (FIP) que será a entrada do Interpretador e do Gerador de Código. Esta árvore deve retratar,ao máximo, as características do programa fonte, tendo o aspecto de uma árvore sintática com atributos coletados. Esta árvore será gerada durante a análise do programa fonte, através de ligações de subárvores que serão construídas para cada produção a ser analisada.As rotinas responsáveis pela geração das subárvores encontram-se na rotina SEMANTICA-TRADUTOR onde estão também os procedimentos que as utilizam.

A chamada da rotina SEMANTICA-TRADUTOR será feita pelo AS, que possui, para cada estado de sua tabela, uma indicação da necessidade ou não, da realização de procedimentos para construção da subárvore referente a produção que está sendo analisada, tal qual o mecanismo de construção da árvore sintática.

A rotina SEMANTICA-TRADUTOR utiliza para seu funcionamento uma estrutura de dados em forma de pilha. Isto se deve ao fato de que, durante a geração da subárvore de uma produção, é necessário parar o processo para que seja gerada uma subárvore correspondente a um Não-Terminal que pertence a forma sentencial da produção que está sendo trabalhada. Então deve ser empilhada na pilha as informações da última geração efetuada.

Ao final da construção da subárvore para uma produção, será feita uma análise semântica em cima dessa subárvore. O conjunto de ações relacionados com as produções encontram- se também na rotina SEMANTICA-TRADUTOR.

Neste trabalho, ao invés de utilizarmos uma tabela com todos os atributos de um símbolo, para que seja efetuada a análise semântica, e também para os passos seguintes, optamos por deixar estas informações na própria FIP. Então, na forma final da FIP, os nós de uso de identificadores e rótulos apontarão para os nós de definição destes.

Introduzimos neste trabalho, o conceito de Packages, que provê uma poderosa ferramenta para realização de complexos sistemas que necessitem de vários programadores, visto permitir que seja especificado um conjunto de declarações e heading de rotinas, que podem ser compartilhados pelos programas que constituem o sistema, criando assim uma interface comum. Outro aspecto importante é a capacidade de se delinear informações que ficarão acessíveis aos usuários da Package. Com isto cria-se uma relativa proteção contra o uso deliberado de certas informações e estruturas.

A sintaxe da linguagem PASCAL a ser suportada por este projeto pode ser encontrada no Apêndice I,

As fases deste módulo foram codificadas na linguagem PASCAL. O programa está autodocumentado, uma vez que, praticamente, para cada linha de comando, está associado um comentário explicando sua necessidade.

O fonte deste trabalho pode ser encontrado no Laboratório de Sistemas da COPPE/UFRJ.

CAPÍTULO III

ANALISADOR LÉXICO

Neste capítulo iremos descrever o Analisador Léxico. Não entraremos em muitos detalhes quanto ao procedimento desta fase porque foi utilizada a metodologia usual, ou seja, Automatos Finitos Determinísticos e, além disso, encontra-se no programa fonte, a rotina ANALISADOR-LEXICO, onde comentários inseridos entre os comandos da rotina facilitam muito o entendimento do funcionamento deste módulo.

III.1. REPRESENTAÇÃO DO ANALISADOR LEXICO COMO AUTOMATO FINITO

Na Figura (III.1) apresentamos o Automato Finito,no qual foi baseado o Analisador Léxico, que reconhece um único ítem léxico, o qual será executado cada vez que o Analisador Sintático necessitar de um novo ítem léxico.

Os tipos dos símbolos que serão reconhecidos pelo AF são: identificadores, símbolos simples, símbolos duplos, constantes inteiras, constantes reais e cadeia.

Em algumas transições do A.F. da figura, estão indicados, entre parênteses, o que podemos chamar de ações "semânticas" do Analisador Léxico denotadas por a_i que serão explicados no ítem (III.2).

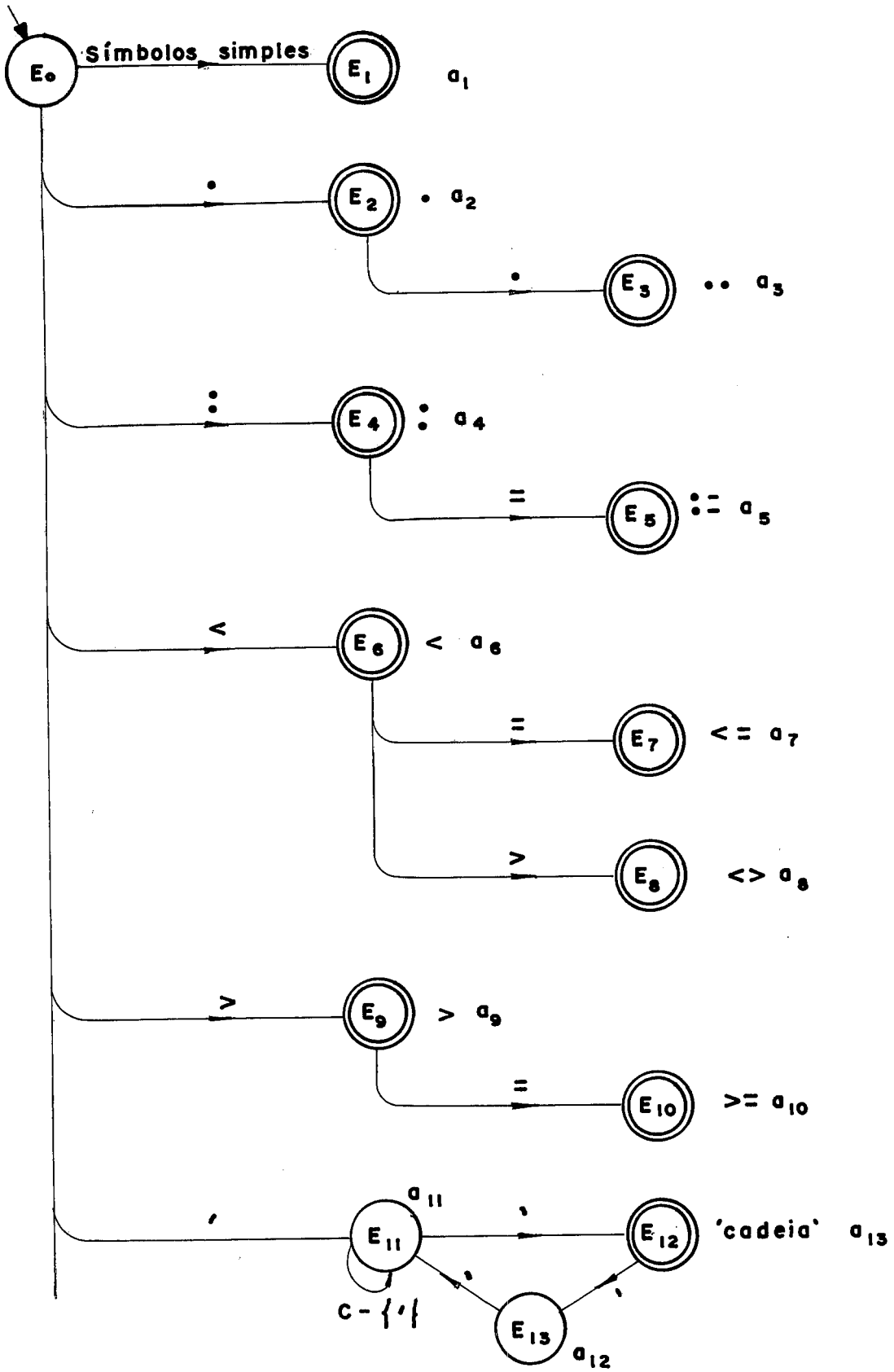
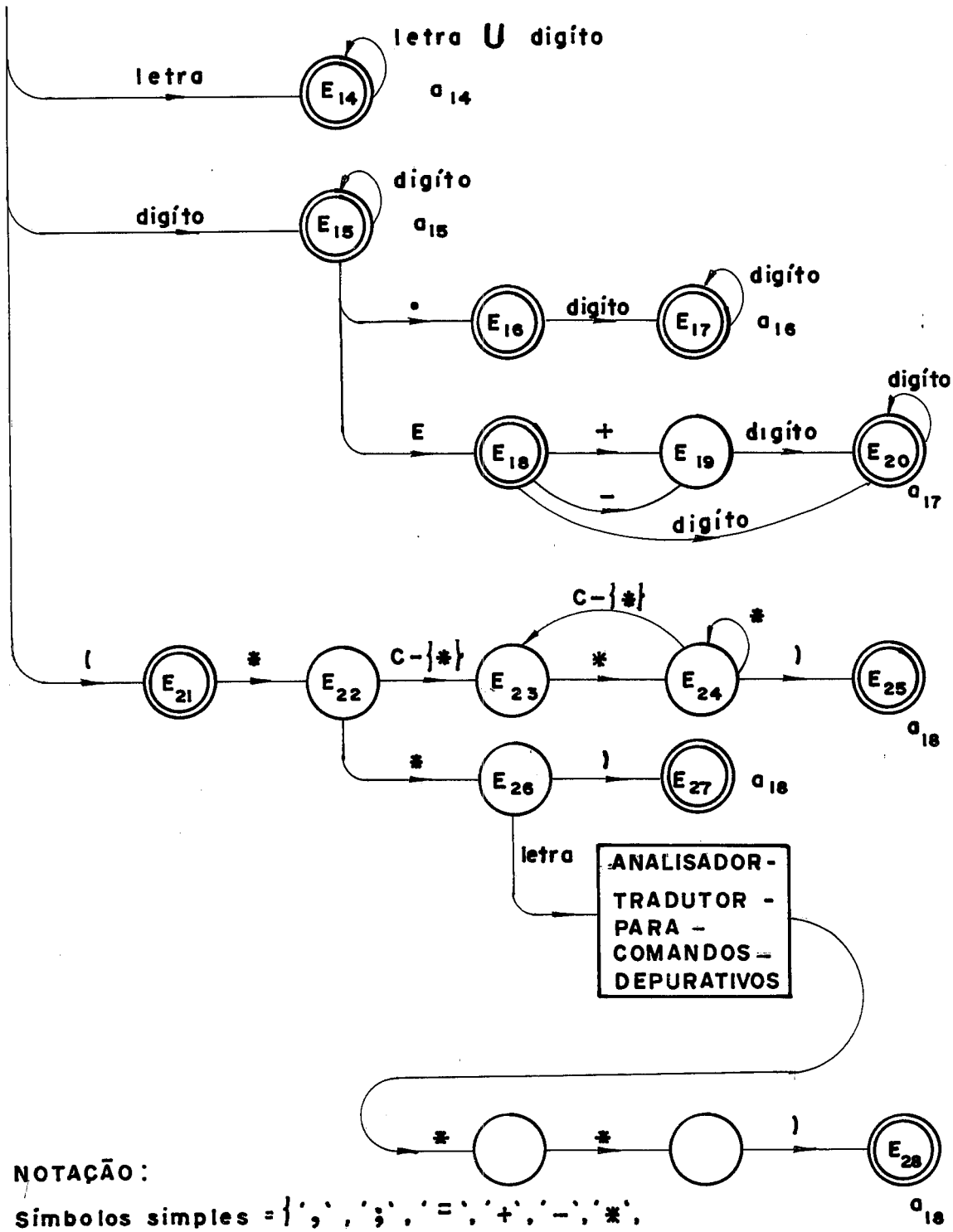

E0
Símbolos simples
E1
a1
.
E2
. a2
.
E3
.. a3
:
E4
: a4
=
E5
:= a5
<
E6
< a6
=
E7
<= a7
>
E8
<> a8
>
E9
> a9
=
E10
>= a10
'
a11
E11
'
E12
'cadeia'
a13
C - {'}
E13
a12

Figura III.1



NOTAÇÃO:

Símbolos simples = {',', ';', '=', '+', '-', '*', '/', ')', 'C', 'Ǝ', 'Λ'}

letra = {'A', 'B', ... ,'Z'}

digíto = {'0', '1', ... ,'9'}

C = conjunto de todos os caracteres

Figura III.1 (continuação)

Note que o estado E1 corresponde ao reconhecimento de um símbolo simples e, do estado E2 ao estado E10, temos o reconhecimento dos símbolos duplos. Em E12 forma-se a cadeia, onde para se utilizar um plique temos que colocá-lo entre pliques (E13).

No estado E14 reconhece-se um identificador, que pode ser um símbolo definido pelo programador ou uma palavra reservada. No próximo ítem veremos como será feita esta distinção.

Uma constante inteira é reconhecida no estado E15, em E17 temos uma constante real com ponto flutuante, e, em E20, constante real com notação científica; nos estados E25 e E27 identifica-se final de comentário (qualquer seqüência de caracteres entre (* e *)) que será ignorado, não causando portanto a construção de um ítem léxico.

O leitor poderá estranhar o aparecimento de uma "caixa", no estado E26 do Automato da Figura (III.1) contendo a inscrição ANALISADOR-TRADUTOR-PARA-COMANDOS-DEPURATIVOS. Já foi mencionado, no Capítulo I, que esta tese é um dos módulos de um projeto onde também consta o módulo Interpretador/Depurador que, para realizar o processo de depuração, deverá interpretar os comandos de depuração, os quais devem estar inseridos nos pontos correspondentes aos especificados pelo usuário, na FIP. Esses comandos devem ser analisados e gerados numa F.I., por um outro Analisador/Tradutor, pois fazem parte de uma outra gramática. Então este Analisador/Tradutor deverá ser o res

ponsável, da fase léxica à geração da Forma Intermediária dos comandos depurativos, inclusive lugando a subárvore resultante, na FIP. Ao se sair deste Analisador/Tradutor, um outro ítem léxico deverá ser buscado, pois, para o Analisador/Tradutor da FIP, os comandos de depuração não passam de comentários, sendo então ignorados.

Apesar desses comandos deputarivos fazerem parte do projeto, a análise destes não merecerá qualquer outra referência ao largo deste trabalho. Tomamos esta decisão, porque, ao nosso ver, seria melhor canalizarmos nossos esforços para a Análise e Tradução da FIP, utilizando o conceito de Packages que traria contribuição maior para área de Ambientes de Programação.

III.2. AÇÕES 'SEMÂNTICAS" DO ANALISADOR LÉXICO

Vimos no Autômato Finito da Figura (III.1) a indicação de ações "semânticas", nos estados finais do autômato. Essas ações podem ser entendidas como procedimentos que completam a atuação do Analisador Léxico, ou seja, a construção dos TOKENS e a verificação da ocorrência de erros léxicos. Na Figura (III.2), podemos ver representado graficamente o TOKEN, que foi implementado como um registro onde a componente CÓDIGO conterá o código do símbolo e, em COLUNA-OCORRÊNCIA, teremos a coluna em que se inicia o símbolo. Na Figura (III.3), temos a representação de CADEIA-CHAR, uma estrutura de packed-array, criada para conter os caracteres de identificadores e cadeias.

TOKEN

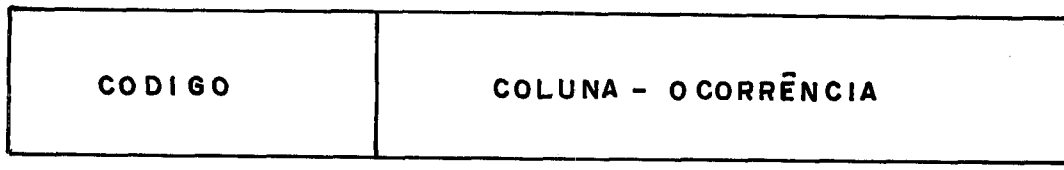


Figura III.2

CADEIA-CHAR

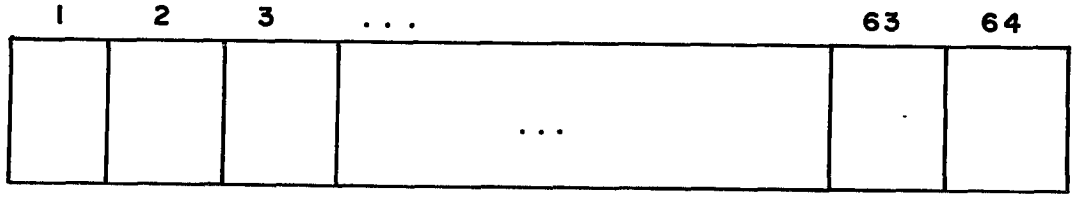


Figura III.3

AÇÕES:

a_1 - Verifica qual é o símbolo simples, e completa o TOKEN com o código do símbolo e coluna em que ocorreu.

a_2 até a_{10} - Análoga a anterior, sendo que para símbolos duplos.

a_{11} - Vai acumulando os caracteres da cadeia em CADEIA-CHAR.

a_{12} - Se o uso do plique dentro da cadeia não estiver correto, será dada uma mensagem de erro.

a_{13} - Verificar se a cadeia ultrapassou o tamanho máximo permitido, então dar mensagem de advertência e ignorar caracteres restantes. Preencher o TOKEN com o código do símbolo e a coluna em que iniciou a cadeia.

a_{14} - Verificar se o tamanho do nome do identificador ultrapassou o permitido. Caso isto aconteça serão ignorados os caracteres restantes e será dada mensagem de advertência. Verificar se o identificador é uma palavra reservada ou um identificador definido pelo programador. Para isto, será chamada a Função CLASS-IDENTIFICADOR, que dará como resultado o código da palavra reservada ou código de Identificador (identificador definido pelo programador).

a_{15} - Verificar se o tamanho da constante inteira ultrapassou o tamanho máximo permitido para número inteiro sem sinal. Caso isto aconteça, será dada uma mensagem de erro. Cabe aqui comentarmos que as constantes numéricas não serão convertidas para representação interna, pois este módulo deve trabalhar apenas a nível simbólico, de maneira a ficar portátil.

a_{16} - Verificar se o tamanho da constante real ultrapassou o tamanho máximo permitido. Caso isto aconteça, uma mensagem de erro será impressa.

a_{17} - Verificar se o expoente ultrapassou a 3 dígitos. Será dada uma mensagem de erro, caso isto aconteça.

a_{18} - Todos os caracteres lidos até o símbolo) serão ignorados. Deverá ser lido um outro ítem léxico.

Observar que, mesmo quando detectarmos um erro léxico, como por exemplo, um overflow, o TOKEN será construído para que possamos anotar o maior número possível de erros. Quando for o caso de identificarmos um caracter inválido, este será ignorado, uma mensagem de erro será impressa e um novo ítem léxico será lido.

CAPÍTULO IV

ANALISADOR SINTÁTICO

Neste capítulo abordaremos em detalhes o método utilizado para a análise sintática, o seu processo de funcionamento e a interface do Analisador Sintático com as outras fases.

IV.1. ANALISADOR SINTÁTICO RRP LL(1)

O método escolhido para Análise Sintática foi o método RRPLL(1) que trata, qualquer gramática, com lados direitos regulares. Para melhor entendimento veremos, no ítem abaixo, uma definição deste tipo de gramática.

IV.2. GRAMÁTICAS COM LADOS DIREITOS REGULARES (RRP)

Gramáticas que possuem conjuntos regulares como lados direitos de suas produções, vêm sendo intensamente estudadas nos últimos anos por representarem uma maneira mais natural de se especificar linguagens livres de contexto.

Uma gramática RRP, livre de contexto, é uma 4-tupla $(N, \sum, P, \vdash S \dashv)$, onde:

(a) N e $\sum$ são conjuntos finitos de Não-Terminais e Ter-

minais, respectivamente;

(b) P é um conjunto de pares $(A \rightarrow \pi)$ tal que $A \in N$ e π é um conjunto regular sobre $(N \cup \Sigma)$;

(c) S é um símbolo inicial e $\vdash$ e $\dashv$ são delimitadores.

Uma derivação em G é da forma $u\ A\ v \Rightarrow u\ w\ v$ se $(A \rightarrow w) \in P$ e $w \in \pi$.

No Apêndice II, podemos ver a gramática PASCAL por nós adotada, no formato RRP.

IV.3. RAZÕES PARA ESCOLHA DO MÉTODO

As razões que nos levaram a escolher este método são várias, e as tentaremos explicá-las de uma maneira top-down. Em primeiro lugar, estamos trabalhando com uma linguagem que foi projetada para ter uma gramática do tipo LL(1), razão pela qual era natural a utilização de um analisador LL(1).

Em segundo lugar, paralelo ao fato que o método é excelente do ponto de vista prático, pela sua simplicidade, sua generalidade e facilidades para a recuperação de erros sintáticos, contamos com a existência do Gerador de Analisadores Sintáticos RRPLL(1), do projeto NHÃONHÃO, da COPPE/UFRJ |13|. Com esse gerador, o processo de obtenção da Tabela de controle do Parser torna-se simples, devido ao fato que a gramática de entrada pode conter ε-produções e recursões à esquerda, uma vez que serão eliminadas automaticamente pelo sistema.

A última justificativa para esta escolha, seria de caráter sentimental, pois o Gerador de Analisadores Sintáticos, RRPLL(1), é fruto de um trabalho desenvolvido na COPPE, e consideramos importante que nossos trabalhos sirvam como ferramenta ou como base para projetos futuros.

IV.4. PROCESSO DE GERAÇÃO DO ANALISADOR SINTÁTICO

Vamos discorrer rapidamente sobre o processo de obtenção do A.S., utilizado pelo gerador do NHÃONHÃO, cujos algoritmos podem ser encontrados na referência |13|.

A entrada deste gerador, será a gramática no formato de expressão regular.No Apêndice II, podemos ver a gramática PASCAL em RRP.

O primeiro passo a ser dado pelo gerador, será a codificação das expressões regulares em Árvores Binárias Costuradas. A razão desta codificação é que esta última é a estrutura mais indicada para atuarem algoritmos de transformação, que alteram a gramática, com intuito de eliminar o reconhecimento de sentenças vazias e recursão à esquerda, diretas ou indiretas. Depois disso as ABCs serão codificadas em AFDs por um Construtor de AFDs, que fornecerá, como saída um A.F.D. de múltiplas entradas que representa uma gramática equivalente sem recursão à esquerda e sem ε-produções.

O procedimento seguinte é a construção da Tabela de Controle do Parser, a partir do AFD mínimo e os conjuntos FIRST

e FOLLOW. Maiores informações sobre a utilização do gerador de Analisadores Sintáticos RRPLL(1), podem ser encontradas na referência |13|.

IV.5. TABELA DE CONTROLE DO ANALISADOR SINTÁTICO

A estrutura escolhida para a representação da Tabela de Controle, foi a "de lista". Uma estrutura alternativa seria em forma de matriz, mas esta estava fora de cogitação, uma vez que a matriz que representa a tabela é esparsa, o que gastaria muito espaço de memória.

Neste trabalho, essa lista foi implementada com um vetor de registros, de nome TAB-SINTATICA, e o número de elementos é igual ao número de estados do AFD mínimo. Cada elemento deste vetor representa um nó da lista, e pode ser representado graficamente como na Figura (IV.1).

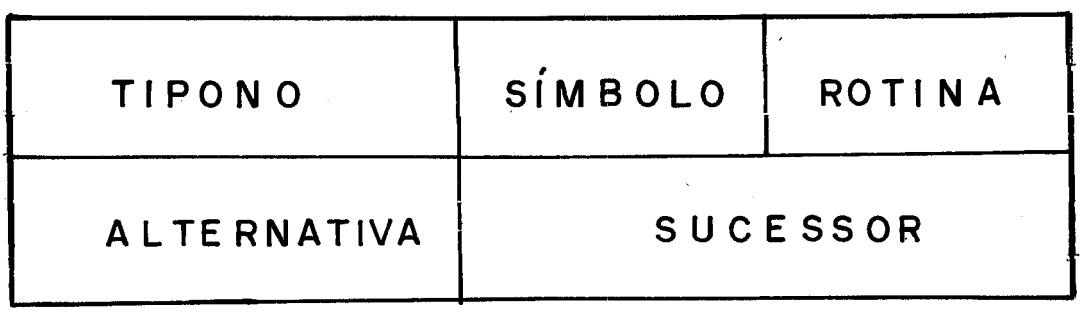


Figura IV.1

O campo TIPO-NO indica a ação associada ao nó, isto é, o procedimento que deve ser tomado pelo AS ao se encontrar neste nó. Esses procedimentos serão vistos no ítem (IV.7).

Os valores que este campo pode assumir são os seguintes:

a) TST --> corresponde a uma transição no Autômato com o reconhecimento do terminal cujo código está representado no campo SÍMBOLO.

b) NT --> corresponde a uma derivação de um Não-Terminal cujo código está representado no campo SÍMBOLO.

c) POP --> indica que o Não-Terminal foi reconhecido.

d) FIM --> indica que a análise está concluída, que corresponde ao reconhecimento do texto.

O campo SÍMBOLO como vimos anteriormente será utilizado para conter o código do Terminal ou Não-Terminal, dependendo do valor do TIPO-NO.

O campo ALTERNATIVA indica se existe uma alternativa, isto é, um outro caminho a ser seguido na lista, quando for o caso em que o símbolo que se encontra representado no TOKEN, não coincidir com o símbolo Terminal caso TIPO-OP seja TST, ou não pertencer ao FIRST do Não-Terminal, caso TIPO-OP seja NT.

Se houver alternativa, o nó alternativo será o nó que se encontra imediatamente abaixo do nó que está sendo analisado.

O campo SUCESSOR aponta para o próximo nó da cadeia de nós, que representam a produção.

Por fim, o campo ROTINA indica se existe um procedimento de geração da FIP e/ou análise semântica,associado a este nó.

IV.6. OUTRAS TABELAS UTILIZADAS PELO ANALISADOR SINTÁTICO

Em adição a TAB-SINTATICA, o analisador sintático trabalha com mais duas tabelas:

a) Tabela ENTRY

Representada por um vetor, onde cada elemento deste vetor corresponde ao número do nó que inicia o Autômato da produção na lista,(conseqüentemente, ao número na Tabela Sintática), correspondente ao Não-Terminal,representado pelo índice do elemento no vetor. Ver Figura (IV.2).

b) Tabela FIRST

Um array de duas dimensões, que indica se o Terminal, representado pelo número da coluna, pertence ao FIRST do Não-Terminal, representado pelo número da linha. Ver Figura (IV.3).

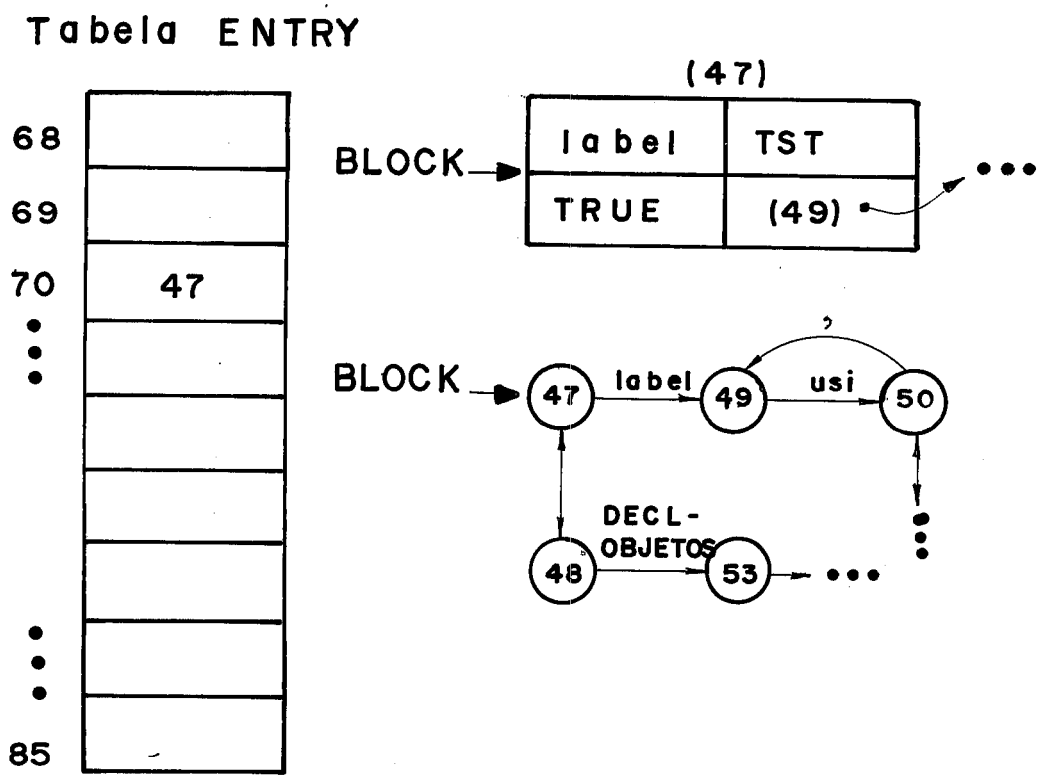


Figura IV.2

Tabela FIRST

	1	•••	13	14	•••	16	•••	20	•••
Terminais / Não-Terminais	program		procedure	function		label		const	
68									
69									
70 BLOCK			X	X		X		X	
•••									
85									

Figura IV.3

IV.7. O PROCEDIMENTO DO ANALISADOR SINTÁTICO

Neste ítem descreveremos o procedimento do Analisador Sintático, de uma maneira quase informal, pois os comentá - rios inseridos entre os comandos da rotina ANALISADOR-SINTÁTICO no programa, devem tornar compreensível o funcionamento deste Analisador.

O procedimento deste analisador se constitui basicamente da estrutura de um laço, para percorrer a TAB-SINTATICA, de acordo com as produções que forem utilizadas no programa fonte. Tendo como ponto de partida o estado 0 (zero), que corresponde ao estado inicial da gramática considerada, o procedimento deste laço será realizar, a cada passada, as implicações de uma das quatro ações sintáticas descritas abaixo, relacionada com o estado em que se estiver, da TAB-SINTÁTICA. Este laço só será interrompido quando houver o reconhecimento do fim do programa fonte, ou seja, encontrando-se a ação FIM, ou o número de erros ocorridos no programa atingir um valor limite, de maneira que não compense a continuação da análise, ou uma terminação anormal ocorra.

Descreveremos a seguir o procedimento a ser realizado, para cada uma das quatro ações sintáticas.

Procedimento para ação TST

Este nó, como vimos no ítem (IV.5) representa uma transição do Autômato Finito. Isto implicará na comparação do

símbolo que representa o nó, com o símbolo que está em TOKEN. Caso coincidam, e não se esteja em estado de erro: uma ação semântica-tradução, se houver para este estado, deverá se realizar, um próximo símbolo deverá ser lido e se caminhará para o sucessor deste estado. Caso não coincidam, é verificado se existe um caminho de alternativa deste estado. Se não existir, não temos qualquer outra ação a realizar senão chamar a rotina TRATA-ERRO-SINTÁTICO.

Procedimento_para_ação_NT

Como vimos no ítem (IV.5), este nó corresponde a uma derivação de um Não-Terminal, cujo código está representado no campo SÍMBOLO.

Antes de tudo, temos que verificar se o símbolo em TOKEN pertence ao FIRST do Não-Terminal. Caso pertença, a derivação deverá tentar ser realizada, percorrendo-se a parte do Autômato-Finito correspondente a este Não-Terminal, retornando no final para o estado que originou esta derivação. Para isto precisaremos de uma estrutura de pilha, de nome PILHA-SINTÁTICA, onde no topo teremos sempre, o nome do estado para o qual teremos que retornar ao ser reconhecida a produção que está sendo analisada. O estado atual então tem que ser empilhado na PILHA-SINTÁTICA, o que será realizado pela rotina EMPILHA-ESTADO.

Se houver ação semântica-tradução para ser realizada, a rotina SEMÂNTICA-TRADUTOR deverá ser chamada neste ponto.

Caso o símbolo em TOKEN não pertença ao FIRST, é verificado se existe um caminho de alternativa para este estado. Se não existiu, a rotina TRATA-ERRO-SINTÁTICO será chamada.

Procedimento para ação POP

Temos neste nó a indicação que foi reconhecido o Não-Terminal, em cuja derivação se estava trabalhando. O procedimento para esta ação é desempilhar o estado que estava no topo da PILHA-SINTÁTICA, chamando-se a rotina DESEMPILHA-ESTADO e caminhar para o sucessor deste estado desempilhado. Caso haja ações semânticas e de tradução para serem realizadas, é chamada a rotina SEMÂNTICA-TRADUTOR.

Procedimento para ação FIM

Apenas um único estado possui esta ação. Ao chegarmos nele, sabemos que a análise foi concluída. Neste ponto então o laço será interrompido.

IV.8. TRATAMENTO DE ERROS SINTÁTICOS

Como vimos no Capítulo I, uma das propostas do nosso projeto, em relação as facilidades oferecidas ao programador, é entre outras, um bom recuperador de erros que forneça mensagens capazes de ajudar o programador a corrigir seu módulo e que sejam detectados no programa-fonte o maior número de erros, para que ele possa posteriormente corrigí-los todos de uma só vez.

É sabido que para estratégia TOP-DOWN não temos a nossa disposição um recuperador de erros eficiente, e o método que mais se adapta a esta estratégia, é a que trabalha basicamente por inserções.

Tratando-se de uma primeira etapa do projeto, optamos por um método simples que opera por inserção, com uma extensão para incluir operações de deleção, e como último recurso utiliza panic-mode.

IV.8.1. DETECÇÃO DE ERROS SINTÁTICOS

A detecção de um erro sintático se dá quando o símbolo lido não coincide com o nó que encabeça a sublista do estado em que o analisador sintático se encontra, como também com suas alternativas, sendo do tipo Terminal ou Não-Terminal, onde serão considerados os FIRST's. Dar-se-á neste momento a chamada da rotina TRATA-ERRO-SINTATICO.

A primeira ação desta rotina é a verificação se esse erro foi produzido pelo programador, ou se é um erro conseqüente de uma correção anterior. O objetivo desta verificação é para evitar que sejam enviadas mensagens de erro que não reflitam a realizada do módulo que está sendo analisado. Para realizar este controle, utilizamos uma booleana chamada ESTADO-DE-ERRO, que será ligada quando for detectado um erro do programador, e permanecerá ligada até que o Analisador Sintático consiga reconhecer o símbolo que se encontra na entrada,ou seja,até que o Analisador SIntático se recupere do erro.Só serão en

viadas mensagens de erro se o Analisador Sintático não estiver em estado de erro.

A mensagem de erro se constitui do caracter '~' seguido de uma frase indicando qual o tipo de erro cometido. Esta mensagem será impressa na linha imediatamente abaixo da que se encontra o erro, e a partir da coluna correspondente ao início do símbolo em questão, o que mostrará ao usuário exatamente em que símbolo da cadeia de entrada deu-se o erro. Isto se faz possível, pois em TOKEN temos a componente COLUNA-OCORRÊNCIA que indica em que coluna se inicia o símbolo recém-lido.

IV.8.2. CORREÇÃO DE ERROS SINTÁTICOS

O tipo de correção e a mensagem a ser enviada, serão obtidos de uma tabela denominada TAB-ERRO. Esta tabela é construída previamente, com base nas transições do Autômato Finito apresentado no Apêndice III.

A tabela TAB-ERRO tem a forma de uma matriz, onde as linhas representam os estados "cabeça" de uma cadeia de alternativas, e as colunas são formadas por **todos** os símbolos existentes na linguagem por nós considerada. Devido a serem considerados, para verificação da ação de correção, apenas os estados "cabeça" de uma cadeia de alternativas, sempre guardamos na variável ESTADO-INICIAL o valor do primeiro estado de uma cadeia que está por ser percorrida.

Os elementos desta tabela são registros onde a com

ponente TIPO-AÇAO representa ação a ser tomada, se for ação de inserção, o símbolo temos em SÍMBOLO, e o diagnóstico do erro em MENSAGEM. A Figura (IV.4) mostra graficamente a **tabela** TAB-ERRO em forma de matriz e também a representação do tipo de seus elementos.

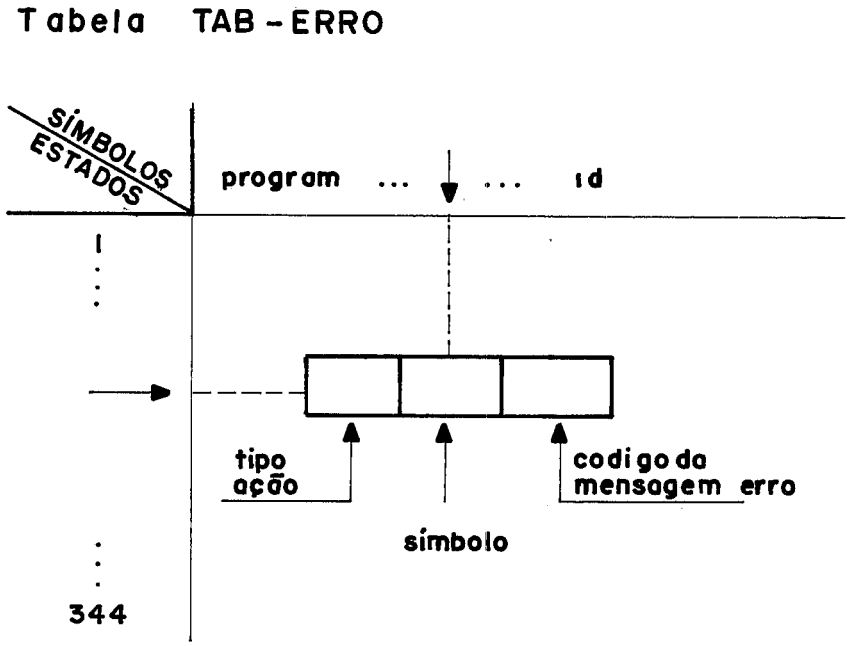


Figura IV.4

A seguir explicaremos em detalhes qual o critério utilizado para a escolha das ações a serem realizadas, e o procedimento seguido, para cada uma delas.

Ação Inserir:

Será realizada quando do estado atual para se chegar em um estado que reconheça o símbolo que se encontra no TOKEN, seria preciso apenas que um determinado símbolo antecedesse, na cadeia de entrada, o símbolo lido. A Figura (IV.5) mostra graficamente um exemplo para este tipo de ação.

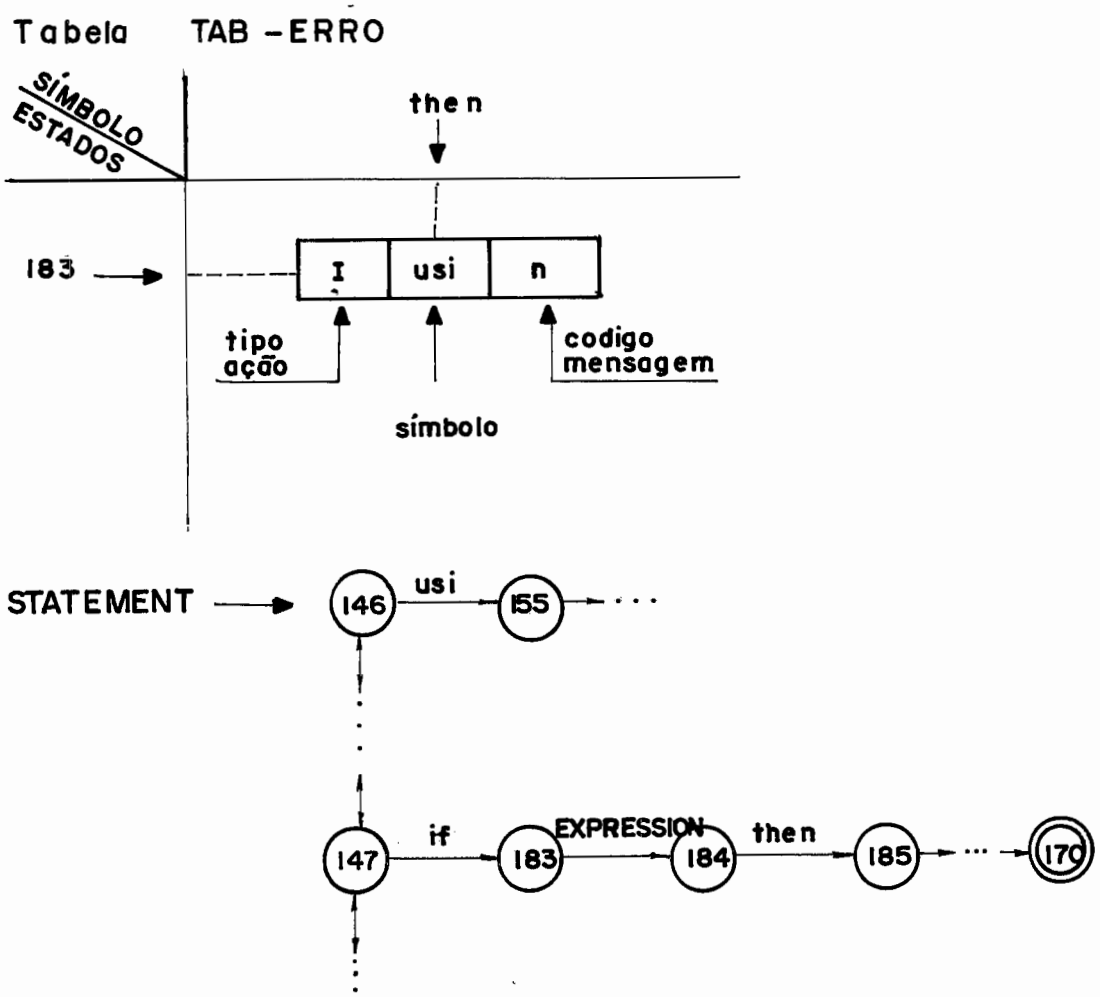


Figura IV.5

O procedimento realizado para ação de inserir, na rotina TRATA-ERRO-SINTATICO é de salvar o último símbolo lido, ou seja, o que está em TOKEN, símbolo em uma variável chamada EXTRA-TOKEN. O símbolo a ser inserido encontra-se como já foi dito anteriormente, na componente SIMBOLO, e será feito o novo Token. Se o símbolo a inserir for um identificador ou um rótulo, o nome será constituído do caracter '$\not\emptyset$'.

Ao se retornar para o Analisador Sintático, a ação a ser realizada será TST ou NT e o símbolo que se encontra em TOKEN, ou seja, o símbolo inserido, será reconhecido. Neste ponto então, como a variável ESTADO-DE-ERRO estará ligada, ao invés de se ler um novo símbolo da cadeia de entrada, o TOKEN voltará a ser o símbolo guardado em EXTRA-TOKEN. Como sabemos que este símbolo vai ser aceito, saímos do estado de erro, isto é, ESTADO-DE-ERRO será desligada.

Ação Deletar:

Deletaremos o símbolo que se encontra na entrada, quando este estiver totalmente fora de contexto, e além do mais, o símbolo não serve para marcar o final da produção em que nos encontramos de maneira que possamos encerrá-la para continuar a análise. Como uma regra geral, podemos dizer que os símbolos ";", "endprogram", "end", nunca serão deletados.

O procedimento realizado no caso desta ação é a leitura de um novo símbolo, pela chamada do ANALISADOR-LÉXICO. Ao retornarmos para o Analisador Sintático este símbolo será examinado e nada podemos afirmar sobre sua aceitação ou não pelo Analisador-Sintático. Note que a chave ESTADO-DE-ERRO estará ligada e só será desligada ao ocorrer o reconhecimento do TOKEN.

Ação Panic-Mode:

Esta ação ocorrerá quando o símbolo lido é do tipo sincronizador para a produção, ou seja, seu aparecimento não deve ser ignorado pois poderemos fechar esta produção, e geralmente, também a produção que a utiliza, e assim sucessivamente até que seja encontrada uma produção que tenha como próximo símbolo, em uma forma sentencial, o símbolo lido.

O procedimento desta ação é bem mais complexo que os das ações anteriores. A PILHA-SINTATICA deverá ser desempilhada até que em TOPO-SINT tenhamos o estado inicial de um Não-Terminal, que possua em seu FOLLOW o símbolo em TOKEN.

Uma situação de erro do tipo catastrófico pode ocorrer neste procedimento, ou seja, a estrutura PILHA-SINTÁTICA pode ficar vazia e ainda sobrarem símbolos da cadeia de entrada que não foram analisados. Neste caso o laço do Analisador Sintático será interrompido e ao se retornar para o corpo do programa uma mensagem de erro será emitida, indicando terminação anormal.

Obviamente teremos que desempilhar também os nós na PILHA-SEMÂNTICA que tenham sido criados durante a Análise Sintática dos estados desempilhados da PILHA-SINTÁTICA. O nó que serviu para ficar no topo da PILHA-SEMÂNTICA nesta situação, terá seu código alterado para SUBARVORE-ERRADA. Quando explicarmos o Analisador-Semântico, no Capítulo VI veremos as conseqüências deste código nesta fase da análise.

Ao retornarmos para o Analisador Sintático, o estado que se encontra no topo da PILHA-SINTATICA tem que ser considerado como reconhecido, para que ele seja desempilhado, de maneira que a análise continue e o símbolo que se encontra na entrada, seja reconhecido. Por isso, a última ação no procedimento para Panic-Mode, é colocar como estado atual, ou seja, atribuir a variável ESTADO, o nome de um estado arbitrário, que tenha como ação o POP, para que o estado que se encontra no topo da PILHA-SINTÁTICA seja desempilhado, e o estado atual passa a ser seu sucessor.

IV.8.3. CONSIDERAÇÕES FINAIS

Existem trabalhos publicados sobre estatísticas de erros mais freqüentes, cometidos por programadores, como o trabalho de Ripley e Druseikis [17], onde os autores chegam à conclusão de que a maior parte das vezes apenas um símbolo está errado, em cada trecho do programa fonte. Então, partindo desta premissa, o método de tratamento de erros sintáticos utilizado, nesta tese, deve conseguir tratar a maior parte desses erros. Através das suas ações de inserção e deleção, acrescidas da ação panic-mode, que garante o fato de não serem eliminados um número exagerado de símbolos de entrada, este método permitirá que o programador conheça a maioria, senão todos, de seus erros, no programa fonte, de uma só vez.

CAPITULO V

TABELA DE SÍMBOLOS

Como vimos no Capítulo II, é na FIP que se encontram os atributos dos identificadores definidos pelo programador, atributos estes, que necessitarão ser consultados neste passo, pelo Analisador Semântico, para verificar a coerência do uso de um nome, com sua declaração/especificação.

Neste capítulo então veremos qual o mecanismo de acesso e a estrutura utilizada, para o nosso esquema de Tabela de Símbolos, a qual o Analisador Semântico construirá para que ele próprio possa consultar informações na FIP, necessárias a realização de sua análise.

V.1. MÉTODO DE ACESSO

A técnica que escolhemos para organizar nossa Tabela de Símbolos, é a de Hash.

O método para se calcular a posição da tabela para um determinado símbolo, é o da Divisão.

Segundo Lum |10|, depois de experimentos com arquivos reais de diversos tamanhos, o método da Divisão foi o que

deu em média os melhores resultados.

Para resolvermos o problema da colisão utilizaremos uma lista seqüencial, que será acoplada à tabela Hash. Teremos um ponteiro AVAIL-POSIÇÃO, que indicará a próxima posição livre da parte seqüencial.

V.2. ESTRUTURA DA TABELA DE SÍMBOLOS

Podemos considerar esta tabela como sendo formada por duas partes, ou seja, uma parte fixa onde estarão as informações permanentes, como por exemplo, nomes do símbolos, e uma parte dinâmica, para informações de escopo do símbolo.

Quando falarmos em símbolo, estamos nos referindo a identificadores não reservados e rótulos.

A parte fixa desta tabela, será formada por 3 vetores. O primeiro vetor, PTALT, será utilizado para apontar a entrada da parte seqüencial, para o caso de colisão. Em PTNOME, o segundo vetor, teremos os ponteiros para o vetor VETNOMES, que é uma estrutura de Packed-Array do tipo Char, onde serão guardados os nomes dos símbolos (Identificadores e rótulos) que se encontram registrados nesta tabela de símbolos.

Finalmente no terceiro vetor, PTNIVEL, será encontrado o ponteiro para o registro da última ocorrência do símbolo, na tabela.

Na parte dinâmica da tabela é onde se encontram os registros das ocorrências dos símbolos, na tabela. É chamada dinâmica pois devido a características de estrutura de blocos da linguagem PASCAL, temos o problema de um mesmo nome poder ser utilizado para representar distintos símbolos, que se encontram em níveis diferentes, e ao se sair de um nível, todos os símbolos que foram declarados neste, devem ser removidos.

Estes registros possuem a seguinte estrutura: o número do nível em que se deu a ocorrência, o ponteiro para o nó da FIP onde se encontra a declaração/especificação do símbolo e ponteiro para o registro da ocorrência anterior.

Para nos informar qual nível que está sendo tratado, uma variável inteira de nome NIVEL será utilizada. Esta variável será incrementada de um, na fase de Análise Semântica, quando for reconhecido o início de um procedimento, e decrementado de um ao reconhecimento do final de um procedimento. O mecanismo de tratamento de escopo será melhor entendido quando estivermos tratando da instalação de um símbolo, no próximo ítem.

Na Figura (V.1) podemos visualizar a estrutura da Tabela de Símbolos com sua parte fixa e a dinâmica, com um símbolo que ocorreu em dois níveis diferentes.

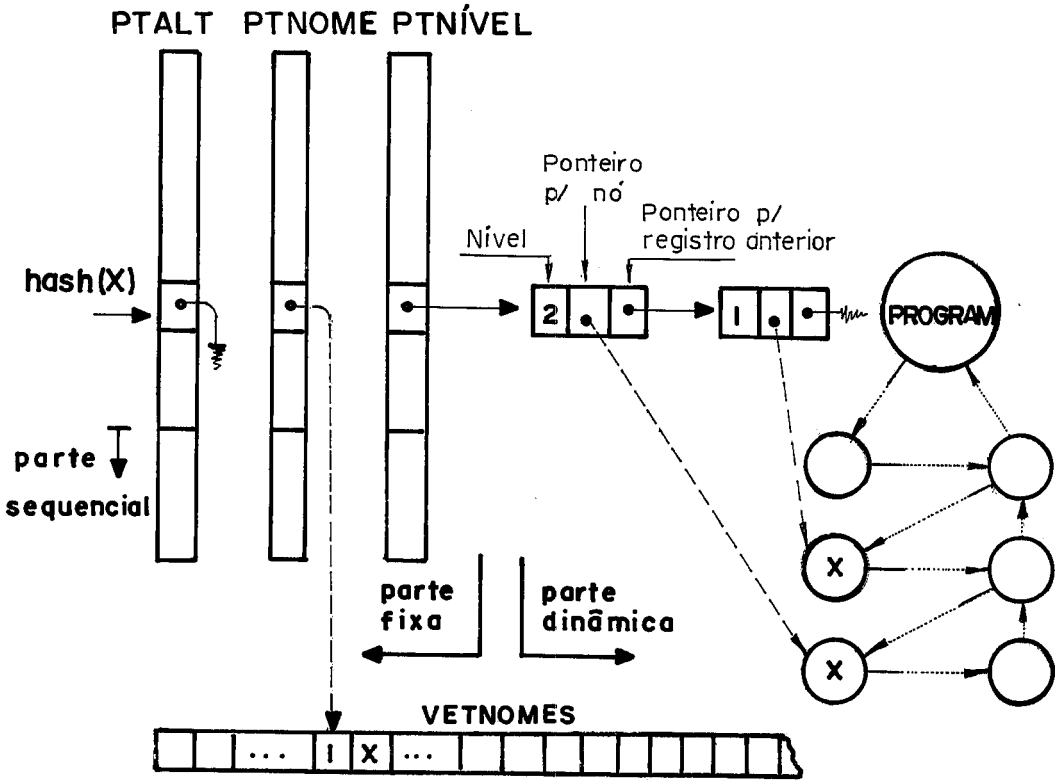


Figura V.1

V.3. INTRODUÇÃO DOS IDENTIFICADORES DE PACKAGES NA TABELA DE SÍMBOLOS

Como foi descrito no Capítulo II, estamos utilizando neste trabalho o conceito de Packages, onde um módulo que esteja sendo compilado pode referenciar nomes que foram definidos em uma package previamente analisada e que se encontra armazenada em memória secundária, sob nossa forma intermediária. Falamos também que os identificadores standards estarão definidos dentro de uma package, a qual denominamos de STANDARD, e cuja presença é obrigatória implicitamente, em qualquer módulo sendo analisado.

Neste ítem iremos descrever como os identificadores de uma package, seja a STANDARD ou não, podem ser consultados a-

través da Tabela de Símbolos pela rotina que trata da Análise Semântica.

O primeiro nível que constará obrigatoriamente na estrutura da Tabela de Símbolos, de qualquer módulo sendo compilado, será o nível 0. Neste nível estarão todos os identificadores standard definidos na package STANDARD. Estes identificadores standard serão instalados na Tabela de Símbolos pela rotina **TRAZ-PACKAGE-STANDARD**, que será chamada pela rotina **SEMÂNTICA-TRADUTOR**, quando esta estiver fazendo Análise Semântica do heading do módulo.

Para instalar os identificadores standard na Tabela de Símbolos, a rotina **TRAZ-PACKAGE-STANDARD** deverá percorrer a árvore da Package **STANDARD**, se detendo nos nós que sejam de definição de identificadores. Em cada um desses nós, será aplicada então a função hash, definida no ítem (V.1) na cadeia de caracteres que representa o nome do identificador, e que se encontra registrada em uma estrutura, que fica armazenada junto com a árvore da package **STANDARD**. Então nesta posição da Tabela de Símbolos, encontrada, será instalado o identificador, ou seja, em **PTNOME** será colocado um ponteiro para primeira posição da cadeia, constituída pelo tamanho e o nome do identificador, em **VETNOMES** e em **PTNIVEL** um ponteiro para um registro de ocorrência, que será criado, em que constará: o valor 0 para o nível, o ponteiro para o nó da Package **STANDARD** onde se encontra declarado o identificador e **nil** para referência anterior.

Caso esteja se utilizando package no módulo sendo

compilado, os identificadores de package serão colocados no nível 1. Então para cada package reconhecida, será chamada a rotina TRAZ-PACKAGE-SPEC que entre outras ações, deverá instalar os identificadores que constam da especificação, utilizando para isso o mesmo esquema que foi utilizado em TRAZ-PACKAGE-STANDARD.

Na instalação dos identificadores de uma especificação, pode ocorrer o fato de já existir um registro de ocorrência para o nome, apontado por PTNIVEL. Esta ocorrência pode ter sido na Package STANDARD ou numa Package que consta da lista de USING cujos identificadores então já foram instalados.

A estratégia utilizada para resolver este impasse, é a que também será adotada para resolver o problema do escopo, isto é, valerá a última referência ao símbolo. O programador neste caso, deverá ser notificado da ocorrência deste fato, através de uma mensagem de advertência.

O usuário então terá que ficar atento, na ordem que utilizará as packages em USING, caso estas utilizem nomes iguais. Note com isso, que identificadores standards podem ser redefinidos, aliás, como previsto no PASCAL original.

V.4. OPERAÇÕES REALIZADAS NA TABELA DE SÍMBOLOS

Descreveremos neste ítem quais e como serão realizadas as operações na nossa Tabela de Símbolos. Estas operações, como já dissemos anteriormente, são de total responsabilidade

do Analisador-Semântico.

Neste ponto não podemos deixar de falar sobre uma estrutura que está intimamente ligada com as operações da Tabela de Símbolos. Trata-se da PILHA-SEMÂNTICA. Vamos nos deter um pouco para comentar sobre esta estrutura, apesar de que, ela merecerá outros comentários no Capítulo VI quando descrevermos as ações do Analisador-Semântico, quando falarmos sobre a geração da FIP.

V.4.1. PILHA-SEMÂNTICA

Examinando-se a gramática PASCAL ou o Autômato-Finito no Apêndice III vemos que várias de suas estruturas são definidas recursivamente,como por exemplo TYPE, em que ocorre o próprio Não-Terminal "TYPE" nas transições de array, file e, até indiretamente em FieldList de record. Isso significa que, no caso de declaração de um tipo, é necessário interromper este processo para gerar-se as informações referentes a um outro tipo, interior ao primeiro. Assim como em declarações de tipo, temos essa característica presente em várias produções.

Esses fatos sugerem o uso de uma estrutura de pilha para reter certas informações quando houver uma interrupção na geração das informações de um determinado objeto. Esta estrutura se chama PILHA-SEMÂNTICA e será utilizada pelo Tradutor para geração da FIP, com objetivo de reter os nós de subárvores que estavem sendo gerados quando uma interrupção ocorreu, para ser gerada uma subárvore de nível mais interno.

Neste ponto o que nos interessa saber é como a PILHA-SEMÂNTICA interferirá na Tabela de Símbolos.A resposta é, no que se refere a Tabela de Símbolos a PILHA-SEMÂNTICA influenciará no fato que, o código do nó que se encontra no topo desta pilha, no momento da chamada da rotina TAB-SÍMBOLOS, é quem determinará qual das operações deverá se realizar, a de consulta ou de instalação de um símbolo.

V.4.2. OPERAÇÃO DE INSTALAÇÃO DE UM SÍMBOLO

Em alguns compiladores, é na fase de Análise Léxica que são instalados os símbolos na Tabela de Símbolos. Isto se dá quando este analisador reconhece um identificador que não seja reservado. No nosso trabalho devido ao fato que os atributos estarão na FIP, a função de instalação dos identificadores ficou por conta do Analisador-Semântico.

O processo de instalação de um símbolo, se dará da seguinte maneira: nos estados da TABELA-SINTÁTICA que se referem a derivação da produção de Identificador ou de Rótulo, que esteja sendo declarado, corresponderá a uma ação do Tradutor, de empilhar um nó cujo código será de acordo com as funções sintáticas, do símbolo sendo analisado. Quando se chegar no estado final desta derivação, isto é, após o reconhecimento do símbolo, o Analisador-Semântico chamará a rotina TAB-SIMBOLOS, que reconhecerá o código do nó que se encontra no topo da PILHA-SEMÂNTICA, como pertencente ao conjunto denominado INSTALA-SÍMBOLO.

Neste ponto será verificado se este nome está apare

cendo pela primeira vez, ou seja, se ainda não havia ocorrido uma entrada nesta posição da tabela. Caso isto seja verdade, o tamanho do nome e a cadeia que o forma devem ser instalados em VETNOMES, e em PTNOME será colocado um ponteiro para posição ocupada pelo tamanho do nome, em VETNOMES. Logo a seguir um registro de ocorrência deve ser criado e preenchido da seguinte maneira: a componente NIVEL receberá o valor da variável NIVEL que representa o nível em que ocorreu o símbolo, a componente NO receberá o valor de TOPO, isto é, este registro apontará para o nó da FIP em que está definido o símbolo, e finalmente a componente NIVEL-ANT receberá o valor registrado em PTNIVEL, ou seja, ponteiro para ocorrência anterior, que neste caso é vazia.

O nó que se encontra no topo da PILHA-SEMÂNTICA receberá o ponteiro para a posição em VETNOMES, na qual foi instalado o tamanho e a cadeia que forma seu nome.

Caso já estivesse registrada uma entrada para este símbolo, é necessário se verificar em que nível se deu a última ocorrência. Se o nível da ocorrência for semelhante ao nível atual, estamos na presença de um erro do tipo símbolo declarado duas vezes. Neste caso na componente PTR1 do nó que se encontra no topo da PILHA-SEMÂNTICA, será anotado o endereço do nó da primeira ocorrência no nível, e ao se retornar, este erro será notado pelo Analisador-Semântico, que deverá tratar o erro.

Quando o nível da última ocorrência for inferior que o nível atual, um registro de ocorrência deverá ser criado, e preenchido da mesma maneira a qual citamos no caso em que não tinha sido registrada entrada alguma.

Com este procedimento asseguramos que a ocorrência mais interna de um símbolo, será sempre a primeira a ser encontrada.

O mesmo procedimento que foi feito no caso anterior, quanto a colocação no nó no topo da PILHA-SEMÂNTICA, do valor em PTNOME, se dará também para estes dois últimos casos.

V.4.2. OPERAÇÃO DE CONSULTA À TABELA DE SÍMBOLOS

Assim como temos os códigos que indicam a instalação de um símbolo na Tabela de Símbolos, possuímos também certos códigos que apenas indicam uma consulta.

Durante a fase de uso do símbolo, o Analisador Semântico precisará verificar como foi definido o símbolo para analisar a validade de seu uso. Como é na FIP que se encontra os atributos de um símbolo, esta consulta na verdade nada mais é do que o preenchimento no nó que se encontra no topo da PILHA-SEMÂNTICA, do endereço na FIP, onde foi definido o símbolo no nível mais interno, e também onde começa a cadeia que forma o nome, no vetor VETNOMES.

Estas informações se fazem necessárias também para os passos seguintes, ou seja, para o Interpretador e o Gerador de Códigos, pois estes precisarão consultar informações sobre a definição dos símbolos para realização de suas tarefas.

Ao se consultar o endereço da definição de um símbolo, poderá ser verificado que o símbolo não tinha sido declarado, ou seja, não existe qualquer entrada na Tabela de Símbolos para este nome. A ocorrência deste fato significa que o programador está utilizando um símbolo que esqueceu de declarar; sendo necessário uma mensagem de erro para notificá-lo do fato. Provavelmente este nome será referenciado outras vezes, no mesmo nível, e seria extremamente desagradável a repetição da mensagem indicando novamente que o símbolo não foi declarado.

O procedimento adotado por nós, para evitar esta duplicidade de mensagem, foi a criação de um registro de ocorrência que apontará para um nó especial denominado NOAUXILIAR de código SUBARVORE-ERRADA, e que apenas será utilizado para este tipo de ocorrência. Uma mensagem de erro deverá ser emitida notificando que o identificador não foi declarado.

Garantimos assim que qualquer outra referência, neste nível, ao identificador, será tratada como no caso anterior, ou seja, será encontrado um registro de ocorrência e anotado o endereço do nó, suposto como de definição do identificador. No próximo Capítulo descreveremos qual a conseqüência de um identificador voltar da consulta à Tabela de Símbolos, apontando para um nó de código SUBARVORE-ERRADA.

V.4.3. OPERAÇÃO DE DESTRUIÇÃO DOS REGISTROS DE OCORRÊNCIA, NO FINAL DE UM BLOCO

O Analisador Semântico ao final de um procedimento,

deverá retirar todos os registros de ocorrência criados no nível. Isto será feito através da chamada da rotina FIM-NÍVEL, que percorrerá o vetor PTNIVEL, desde a primeira posição até a última, verificando se existe uma ocorrência no nível em questão. Cada vez que for encontrado um registro de ocorrência do determinado nível, esta posição em PTNIVEL receberá o ponteiro que estava na componente NIVEL do registro a ser destruído, cujo valor pode ser nil ou endereço do registro de ocorrência de um nível mais externo.

O registro removido será devolvido ao Heap.

V.5. JUSTIFICATIVA PARA NÃO EXISTÊNCIA DE UMA TABELA DE CONSTANTES

Geralmente os compiladores possuem uma tabela própria para armazenar os valores das constantes, numéricas ou não, encontradas no programa. Neste trabalho optamos pela não existência desta tabela e temos várias razões para isso.

Como vimos no Capítulo III nesse passo trabalharemos a nível simbólico, isto é, as constantes, não serão convertidas, operação que ficará a cargo do passo seguinte (Interpretação/Geração de Código).

Outro fato levado em consideração é que as constantes das Packages compiladas deverão ser armazenadas juntamente com a Forma Intermediária, e os nomes de seus símbolos, que estarão em VETNOMES. Resolvemos então ter um local comum para ar-

mazenar as cadeias de caracteres, seja do nome de um símbolo ou uma constante.

Ao ser reconhecida uma constante o Analisador_Semântico chamará a rotina ANOTA-NOME-CTE que será responsável pelo armazenamento da cadeia que forma a constante, no vetor VETNOMES, e fazer a componente PTNOME, do nó que representa a constante, apontar para a posição, em VETNOMES onde inicia a cadeia.

CAPÍTULO VI

ANALISADOR-SEMÂNTICO/TRADUTOR

Neste capítulo iremos descrever as ações relacionadas com a Análise Semântica e com a geração da FIP. A razão pela qual estamos trabalhando a Análise Semântica e a Tradução para Forma Intermediária juntas, se dá pelo fato que as ações Semânticas serão feitas percorrendo-se a subárvore construída para cada produção, no procedimento de tradução.

Como foi dito no Capítulo II, a Forma Intermediária PASCAL (FIP) proposta por este projeto, deve retratar ao máximo as características do programa fonte, tendo o aspecto de uma árvore sintática. Outra característica é que os atributos dos símbolos encontra-se na FIP. Então a forma que nos pareceu mais natural para geração da FIP foi de associar a construção da árvore ao processo de Análise Sintática.

A responsável pela realização dessas ações, é a rotina SEMÂNTICA-TRADUTOR que será chamada pelo Analisador Sintático quando a componente ROTINA do estado onde se encontra o Analisador Sintático, tiver o valor TRUE.

Antes de começarmos a descrever o processo de Tradução e Análise Semântica, vamos dar uma noção de como a Rotina SEMÂNTICA-TRADUTOR foi estruturada.

VI.1. ROTINA SEMÂNTICA-TRADUTOR

A rotina SEMÂNTICA-TRADUTOR se constitui de um comando CASE, onde o seletor é a variável ESTADO, que representa o estado atual da TAB-SINTATICA ou seja, onde estava se efetuando a Análise Sintática e havia uma ação semântica/tradução a ser realizada.

As constantes que rotulam os comandos representam os estados para os quais se faz necessário uma análise de contexto e/ou uma ação para geração da FIP.

Na Figura VI.1 podemos ver o esquema desta rotina.

```
PROCEDURE     SEMÂNTICA-TRADUTOR ;
  :
 BEGIN
   CASE ESTADO OF
   1: (* ESTADO 1 *)
      :
   2: (* ESTADO 2 *)
      :
   END (* CASE *)
 END;     (* PROCEDURE SEMANTICA-TRADUTOR *)
```

Figura VI.1

VI.2. DESCRIÇÃO DA GERAÇÃO DA FIP E ANÁLISE SEMÂNTICA REALIZADA

Neste ítem veremos a descrição geral do processo de geração da FIP bem como a representação de cada subárvore que a forma. Veremos também as ações semânticas realizadas por este módulo.

VI.2.1. DESCRIÇÃO GERAL DO PROCESSO

Como foi dito anteriormente a construção da FIP está associada ao processo de Análise Semântica, através da existência de uma componente de nome ROTINA encontrada em cada estado da TAB-SINTATICA, que determina a chamada ou não da Rotina SEMÂNTICA-TRADUTOR, cuja estrutura foi vista no ítem anterior. Vimos também que a estratégia utilizada para a Análise Sintática é a de TOP-DOWN.

Por estes fatos podemos considerar como semelhantes,o processo de construção da FIP com a geração da árvore de pare no que se refere a ordem de construção das subárvores, associados as produções, utilizadas para análise do texto submetido.

Os primeiros nós a serem gerados correspondem a produção associada ao símbolo inicial da gramática. Ao ser encontrado um Não-Terminal na forma sentencial desta primeira produção, esta geração será interrompida para que seja gerada a subárvore correspondente a produção associada a este Não-Terminal. Este processo se repetirá para cada Não-Terminal encontrado, no sentido da esquerda para direita.

As subárvores que forem sendo geradas devem ser "pen duradas" nas subárvores correspondentes a produção que as tenha gerado. Para isto necessitaremos de uma estrutura de pilha, que manterá os nós das subárvores, cuja geração foi interrompida, pe lo aparecimento de um Não-Terminal na forma sentencial associada assim como os nós da subárvore sendo gerada no momento.

Esta estrutura chama-se PILHA-SEMÂNTICA e já foi a-presentada no Capítulo V. No decorrer do próximo ítem, na exposição do processo de construção de cada subárvore que forma a FIP, a necessidade da existência e atuação desta pilha, será entendida por completo.

A Análise Semântica será realizada percorrendo-se a subárvore construída para a produção a ser analisada. No estado que corresponde ao estado final da produção, no topo da PILHA-SEMÂNTICA, teremos a raiz desta subárvore. A Análise Semântica será feita nesta ocasião, percorrendo a subárvore, filha do nó raiz, que se encontra no topo da pilha. Os nós que corres ponderem a construções erradas semanticamente, serão marcados com o código de SUBARVORE-ERRADA.

Para que fique mais claro o entendimento da construção das subárvores que serão descritas recomenda-se acompanhar a explicação examinando os Autômatos Finitos, de cada produção,que podem ser encontrados no Apêndice III.

Como em todos os Capítulos anteriores, não descreveremos a nível de detalhes, a implementação do procedimento desta fase. Isto se justifica pelo fato que no Apêndice IV encontra-

se na sua íntegra, a Rotina SEMÂNTICA-TRADUTOR, onde praticamente para cada linha de comando existe um comentário associado, tornando-a auto-documentada.

VI.2.2. ESTRUTURA DE UM NÓ

Vejamos agora a estrutura considerada para os nós da FIP.

Embora a quantidade de informações varie com o tipo de nó não havendo portanto uniformidade, não trabalharemos com nós de tamanho variável. Os nós serão padronizados em função do maior nó possível. A estrutura adotada para o nó é a seguinte:

```
NO = RECORD
       CODIGO       : Código do nó.
       BAIXO        : Aponta para o nó filho.
       LADO         : Aponta para o nó irmão.
       CIMA         : Aponta para o nó pai.
       PTNOME       : Aponta para o nome no vetor VETNOMES. (Caso
                      o nó corresponda a um identificador ou uma
                      constante).
       PTR1         : Aponta para o nó raiz da subárvore que cor-
                      responde a declaração/especificação deste nó.
                      (Caso o nó corresponda a um identificador).
       PTR2         : Contém o código do tipo com que o identifica
                      dor foi declarado.
       NÍVEL        : Nível estático do Bloco.
       DESLOCAMENTO: Off-set do identificador, no Bloco em que
                      foi declarado.
    END;
```

VI.2.3. DESCRIÇÃO DAS ROTINAS DE CONSTRUÇÃO DA FIP

Os nós da FIP serão criados e ligados entre si, para construção das subárvores utilizando-se basicamente 4 rotinas: EMPILHA-NO, FAZ-FILHO, FAZ-PAI-FILHO e FAZ-PAI-MEIO. A seguir veremos a descrição de cada uma dessas ações:

a) EMPILHA-NO - Cria um nó através do procedimento NEW, atribui o código que consta como parâmetro desta rotina e empilha o nó criado na PILHA-SEMÂNTICA.

b) FAZ-FILHO - Faz o nó que se encontra no topo da PILHA-SEMÂNTICA ser filho do nó que se encontra na posição TOPO-1. Desempilha-se o nó filho e o nó que ficará no topo será o pai.

c) FAZ-PAI-FILHO - Corresponde a aplicação de FAZ-FILHO duas vezes. O nó que ficará no topo será o que estava na posição TOPO-2.

d) FAZ-PAI-MEIO - O nó que se encontra na posição TOPO-1 será feito o pai dos nós nas posições TOPO-2 e TOPO. Os filhos serão desempilhados e o pai ficará no topo da pilha.

VI.2.4. DESCRIÇÃO DAS SUBÁRVORES E AÇÕES SEMÂNTICAS RELACIONADAS

Neste ítem serão descritas todas as subárvores relacionadas com as produções existentes na gramática RRP por nós considerada. Faremos esta exposição de uma maneira top-down tal como foram geradas as subárvores, ou seja, mostraremos a geração da FIP como um todo, que corresponde a produção associada ao símbolo inicial desta gramática, depois veremos a geração das subárvores correspondentes aos Não-Terminais que se encontram na produção do símbolo inicial, da esquerda para direita. Este processo se repetirá para cada Não-Terminal encontrado.

PRODUÇÃO COMPILATION

Esta é a produção inicial da gramática por nós adotada (Formato RRP).

A geração da Forma Intermediária para esta produção, corresponde à construção da árvore para o módulo sendo analisado, ou seja, a árvore resultante da análise/tradução desta produção, retrata a Forma Intermediária final do módulo submetido.

Primeiramente iremos apresentar o formato da árvore gerada para um módulo Program, a seguir, o da árvore correspondente a um módulo Package.

Módulo PROGRAM

Na Figura (VI.2) podemos visualizar a árvore gerada para um programa que utilize Packages, através da cláusula USING.

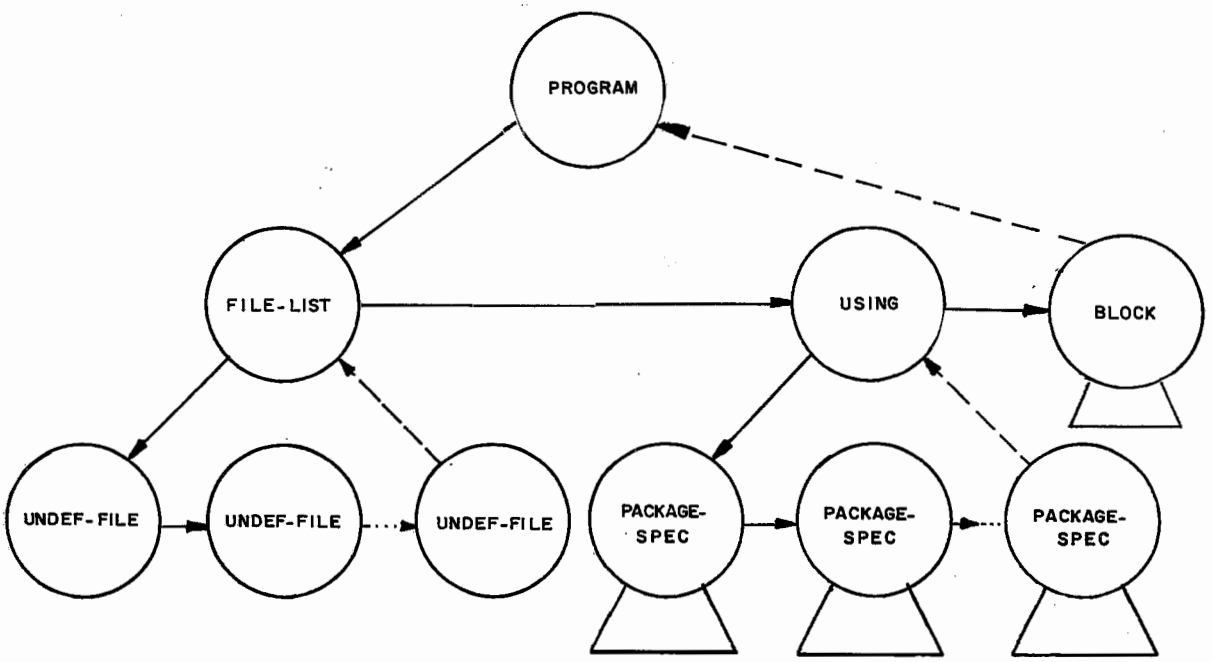


Figura VI.2

Na Figura (VI.3) temos representada a árvore de um programa que não faz uso de packages.

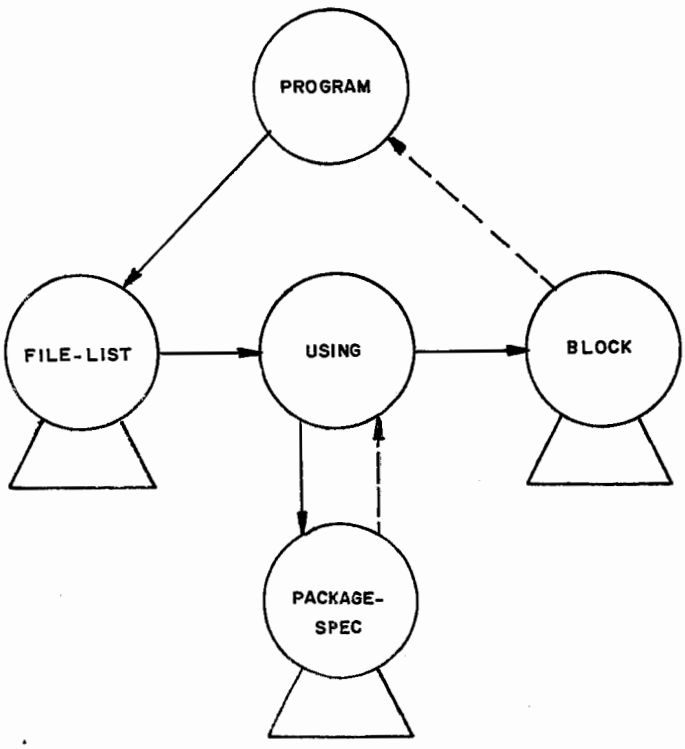


Figura VI.3

Note que independente de haver packages ou não, o nó USING sempre estará presente na FIP. Isto se dá pelo fato que a subárvore de especificação da Package STANDARD sempre estará presente na FIP de um módulo.

O nó raiz desta árvore cujo código é PROGRAM possui na componente PTNOME um ponteiro para a cadeia que forma seu nome, no vetor VETNOMES. O nó FILE-LIST é a raiz de uma subárvore, constituída de uma lista de nomes de arquivos, definidos no heading deste programa.Estes nós de arquivos terão o código UNDEF-FILE, que será modificado para DEF-FILE quando for encontrada a declaração do arquivo através de uma declaração FILE. Por default teremos os arquivos INPUT e OUTPUT no nó FILE-LIST.

O nó USING tem como filhas as subárvores referentes a especificação das packages referenciadas no programa. Essas subárvores serão instaladas pela Rotina TRAZ-PACKAGE-SPEC. No caso da Package STANDARD a responsável pela sua instalação será a Rotina TRAZ-PACKAGE-STANDARD. Maiores detalhes sobre packages poderão ser encontrados no Capítulo VII.

Por fim o nó BLOCK é a raiz da subárvore que representa o corpo do programa.

Ações Semânticas

A única ação semântica realizada na árvore de módulo program é a verificação se todos os nós, filhos do nó FILE-LIST, estão com o código DEF-FILE. Caso isto não ocorra, ou seja, exista algum nó que permaneça com o código UNDEF-FILE, será emiti

da uma mensagem de Advertência, apenas para alertar deste fato, ao programador.

Módulo PACKAGE

Neste ítem iremos apresentar as árvores geradas na análise de uma Package. Não entraremos em detalhes sobre a utilização desse módulo pois para isto foi criado um capítulo a parte, neste trabalho. O que podemos adiantar neste momento é que uma Package possui duas representações de árvores que coexistem na memória auxiliar: a árvore cuja raiz tem código PACKAGE e a subárvore de raiz PACKAGE-SPEC, que é a especificação da package.

A primeira representa a Package propriamente dita, com todas as suas declarações e procedimentos. A especificação representa apenas as declarações de objetos e especificações das rotinas, enfim a parte que será visível ao usuário da package, então a subárvore que será feita filha do nó USING de um módulo que utilize a Package.

Árvore PACKAGE

Na Figura(VI.4) podemos observar a árvore gerada para o módulo de uma package.

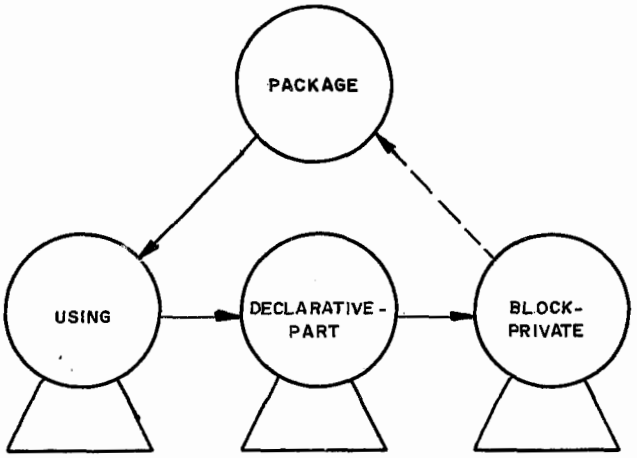


Figura VI.4

O nó raiz da árvore, conterá a informação do nome da Package, em PTNOME, tal qual o nó PROGRAM.

O mesmo comentário já feito para a subárvore USING no ítem anterior, cabe para esta árvore de Package.

A subárvore cuja raiz é o nó DECLARATIVE-PART é semelhante a subárvore BLOCK, com exceção da subárvore LABEL-PART que não é definida em uma DECLARATIVE-PART. Esta subárvore corresponde a declaração da parte que ficará visível na Package.

A subárvore cuja raiz é o nó BLOCK-PRIVATE corresponde a declarações e procedimentos internos da Package, que não podem ser acessados pelo usuário da Package. Sua estrutura é semelhante a subárvore BLOCK.

Ações Semânticas

Caso alguma Package referenciada no nó USING não tenha sido armazenada ainda, uma mensagem de erro será emitida notificando o ocorrido.

Árvore PACKAGE-SPEC

Na Figura(VI.5) podemos observar a árvore gerada para a especificação de uma Package. Esta árvore é a que ficará sob o nó USING de um módulo que a referencie.

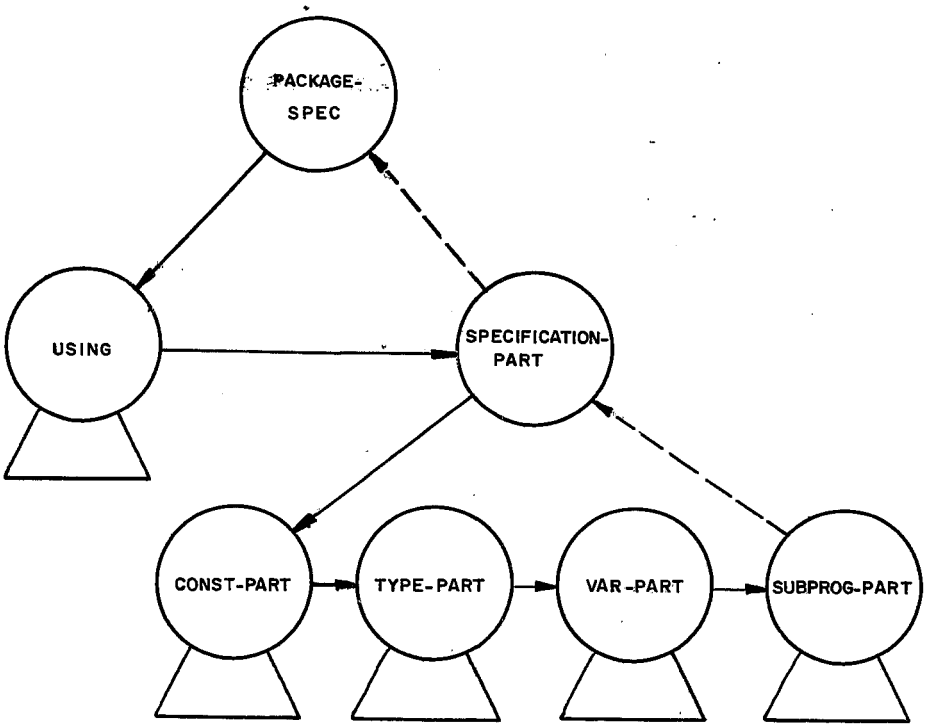


Figura VI.5

O nó PACKAGE-SPEC, raiz desta árvore, possui na compo

nente PTNOME a posição no vetor VETNOMES onde está armazenada a cadeia de caracteres que forma seu nome. O nó USING, assim como no módulo PROGRAM, sempre estará presente independente da Package utilizar ou não outras Packages, pois a PACKAGE STANDARD também estará presente para análise deste módulo.

O nó SPECIFICATION-PART é a raiz da subárvore onde se encontra os objetos da parte visível da Package.

As subárvores cujas raízes são os nós: CONST-PART, TYPE-PART, VAR-PART serão detalhadamente explicada nos próximos itens. A subárvore de raiz SUBPROG-PART, para o módulo de especificação de uma Package, possui apenas subárvores das especificações das rotinas definida na Package. Na Figura VI.6 podemos observar a especificação de uma procedure e de uma função, ambas definidas com parâmetros.

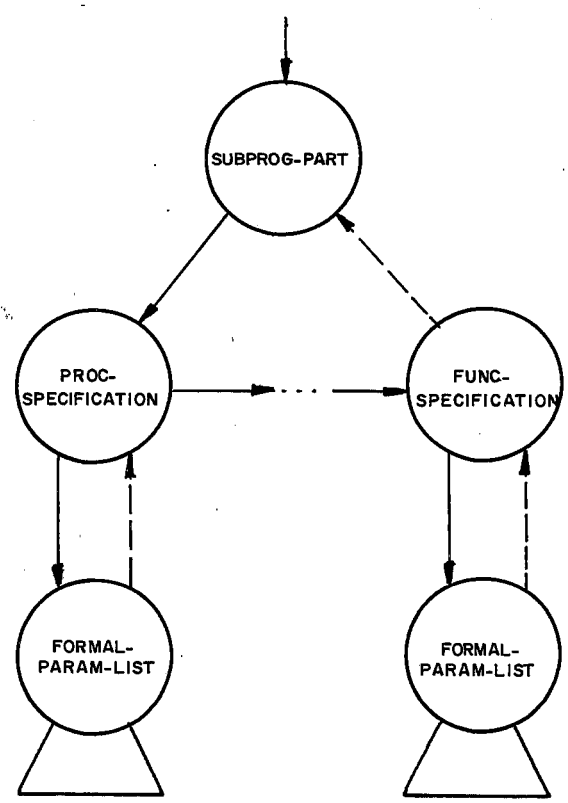


Figura VI.6

Na Figura (VI.7) apresentamos a especificação de rotinas definidas sem parâmetros.

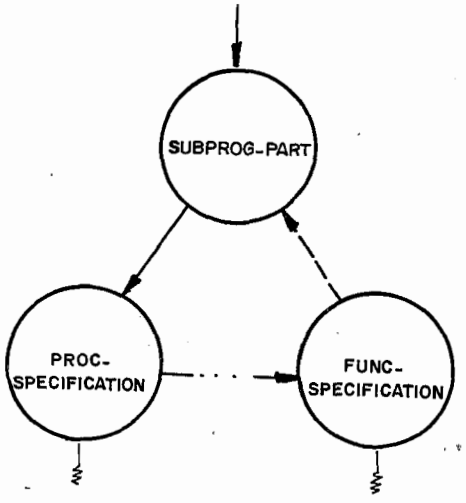


Figura VI.7

Ações Semânticas

Caso alguma package referenciada no nó USING não tenha sido armazenada ainda, uma mensagem de erro será emitida notificando o ocorrido.

PRODUÇÃO BLOCK

A Figura (VI.8) apresenta a subárvore referente a esta produção.

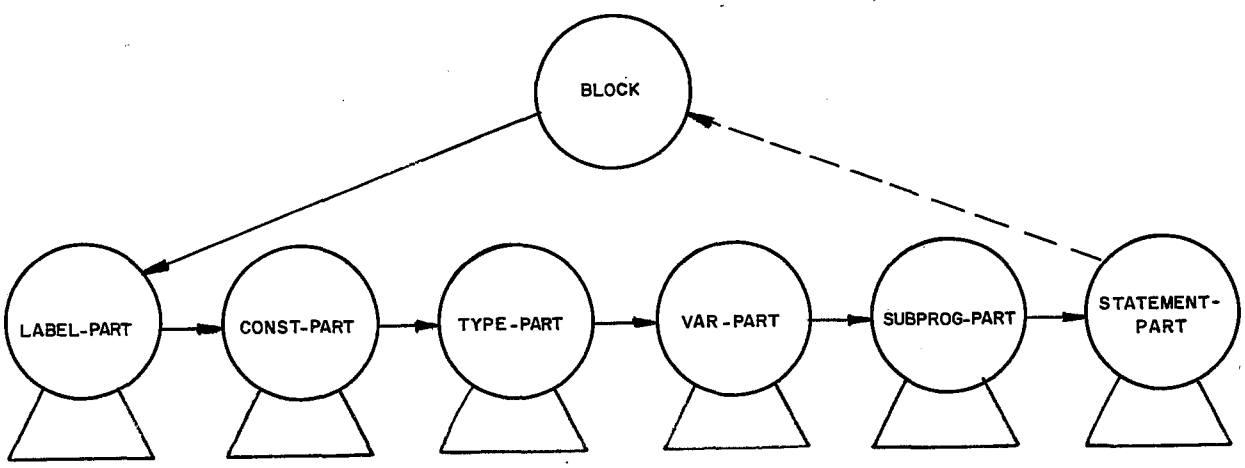


Figura VI.8

As 5 primeiras subárvores, representam a parte de declaração dos símbolos onde todos os objetos locais ao bloco são definidos. A primeira subárvore, indica os rótulos definidos neste bloco. A segunda subárvore define os sinônimos para as constantes, isto é, introduz identificadores que serão usados no lugar de constantes. A terceira contêm as definições dos tipos utilizados e a quarta as variáveis consideradas. Por fim, a última subárvore deste grupo define as procedures e funções válidas para o bloco.

Sabemos que na linguagem PASCAL qualquer parte referente a declarações, pode ser omitida. Neste caso não será gerada subárvore para a parte omitida.

A subárvore cuja raiz é o nó STATEMENT-PART contém as subárvores referentes aos comandos utilizados no Bloco.

Ações Semânticas

Ao ser construída esta subárvore, deverá ser verificado se todos os rótulos declarados neste bloco foram definidos. Isto poderá ser realizado percorrendo-se a subárvore LABEL-PART, e verificando se na lista formada por seus filhos, existem nós cujo código ainda é UNDEF-LABEL. Ao descrevermos a subárvore LABEL-PART este procedimento ficará mais claro.

Subárvore LABEL-PART

Na Figura(VI.9) podemos ver a subárvore referente a declaração de rótulos, antes que tenham sido definidos.

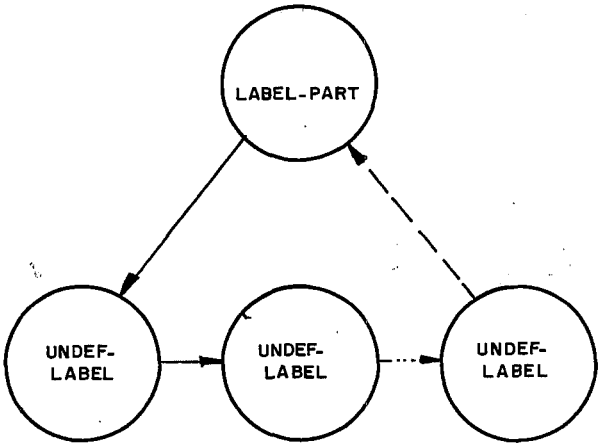


Figura VI.9

Como vimos no Capítulo V a ocorrência da declaração de um rótulo gerará a ação de instalação deste na Tabela de Símbolos, onde será criado um registro de ocorrência, que apontará para o nó UNDEF-FILE correspondente a este rótulo. Quando este rótulo for definido, na componente PTR1 do nó UNDEF-LABEL correspondente, será colocado o endereço do nó de definição do label. Então o código do nó de declaração, passará a ser DEFINIDO, e não mais UNDEF-LABEL. Por este motivo todos os nós de uso do rótulo

apontarão para o nó de declaração.

Na Figura (VI.10) podemos ver a subárvore LABEL-PART quando todos os labels declarados, foram definidos.

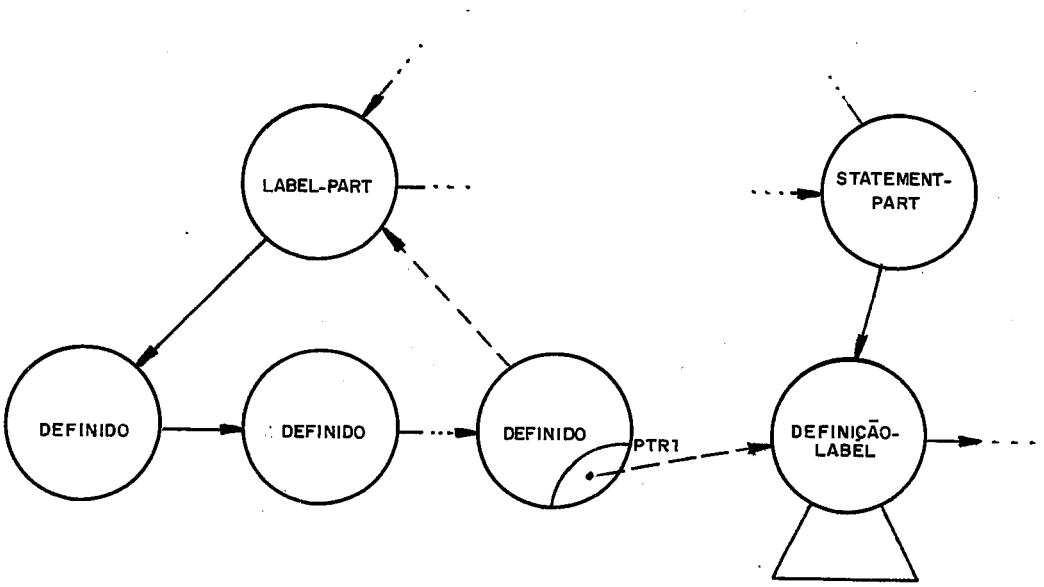


Figura VI.10

Ações Semânticas

A lista de nós de declaração dos rótulos deve ser percorrida para verificar se o tamanho dos rótulos excede de 4 caracteres.

Caso isto ocorra será emitida mensagem de erro notificando o programador, deste fato.

PRODUÇÃO DECL-OBJETOS

Esta produção é responsável pela declaração das constantes, tipos e variáveis do Bloco. Durante sua análise, serão geradas as 3 subárvores: CONST-PART, TYPE-PART e VAR-PART.

Subárvore CONST-PART

Na Figura (VI.11) podemos ver a subárvore de declaração de constantes.

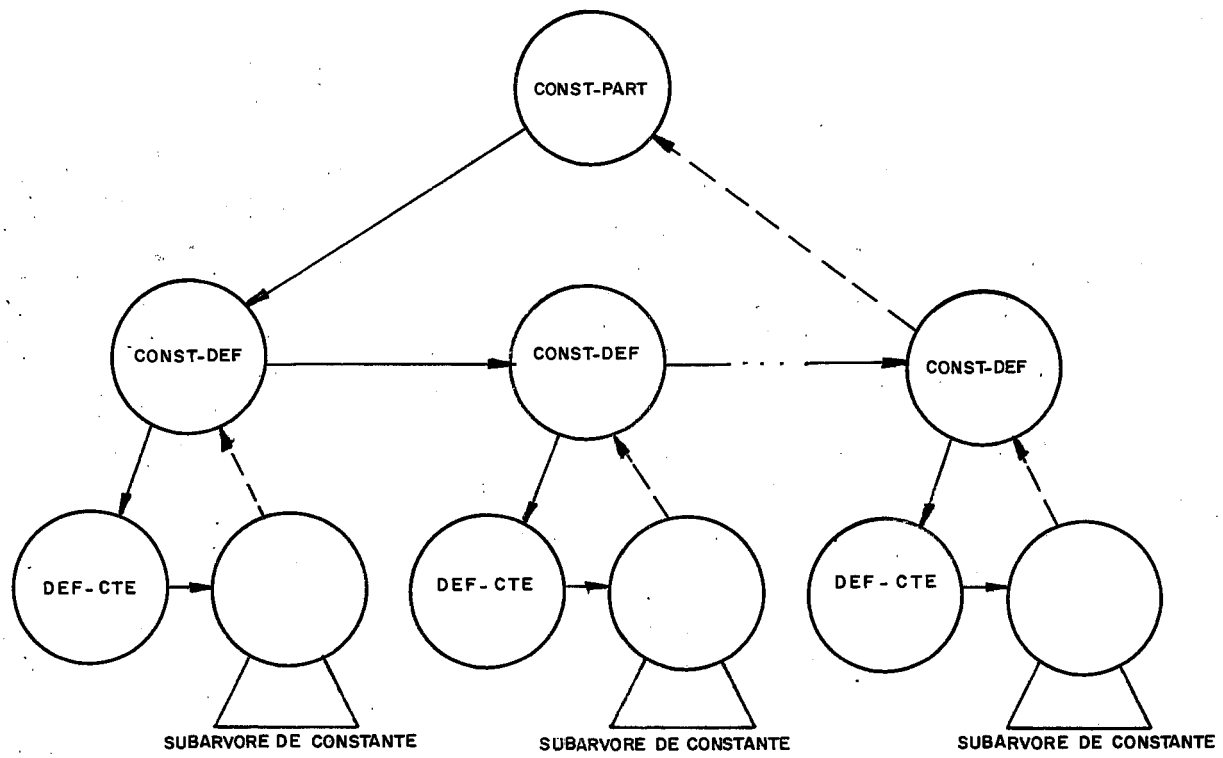


Figura VI.11

Os códigos de nós de constantes são:

- USI
- USR
- MUNARIOI
- MUNARIOR
- STRING
- USED-CTE

Todos os nós representam constantes com exceção dos nós USED-CTE, MUNARIOI e MUNARIOR, têm na componente PTNOME um ponteiro para a posição de VETNOMES onde foi instalado a cadeia de caracteres que forma a constante. O nó USED-CTE corresponde a utilização de um identificador, como valor da constante.

Ações Semânticas

Para cada subárvore CONST-DEF será verificado se o identificador que representa o nome da constante, já tinha sido declarado previamente, neste Bloco. Caso isto ocorra será enviada uma mensagem de erro, mas não será necessário mudar o código para SUBARVORE-ERRADA pois este nó não será nunca alcançado, visto que já foi declarado em outra ocasião.

Os identificadores correspondentes aos nós USED-CTE deverão ser pesquisados para verificar se foram definidos realmente como constantes. Caso esta condição seja satisfeita, na componente PTR1 deste nó teremos o endereço do nó de definição desta constante, e em PTR2 o código do tipo (INTEIRO, REAL ou STRING).

No caso do identificador não ter sido declarado como uma constante, será emitida uma mensagem de erro notificando este fato. O nó com código USED-CTE passará a ter o código SUBARVORE-ERRADA.

Quando a constante vier acompanhada de um sinal unário, durante a construção da subárvore, este terá o código MUNARIO para sinal menos e MAISU para sinal mais. A ação semântica a ser efetuada nesta subárvore é primeiramente verificar se a constante pode ser sinalizada, ou seja, se esta não é uma String. Caso o seja, uma mensagem de erro será emitida e o código será modificado para SUBARVORE-ERRADA.

Depois desta verificação, caso o nó seja MAISU (mais unário), este será descartado pois não produz efeito algum.

No caso de nó MUNARIO, o tipo da constante deverá ser verificado para que seja colocado explicitamente no nó raiz desta subárvore, pois isto facilitará a análise semântica de operações que necessitem conhecer o tipo de seus operandos. Feito isto o código do nó será modificado para MUNARIOI se o tipo inteiro e MUNARIOR se o tipo for real.

Subárvore_TYPE-PART

Em PASCAL podemos definir um identificador que representará um tipo ou definir o tipo diretamente na declaração de uma variável.

Na Figura(VI.12)veremos a subárvore TYPE-PART, que declara identificadores de tipos. A seguir veremos todas as subárvores que formam os tipos propriamente ditos permitidos em PASCAL e que correspondem às subárvores geradas na produção TYPE.

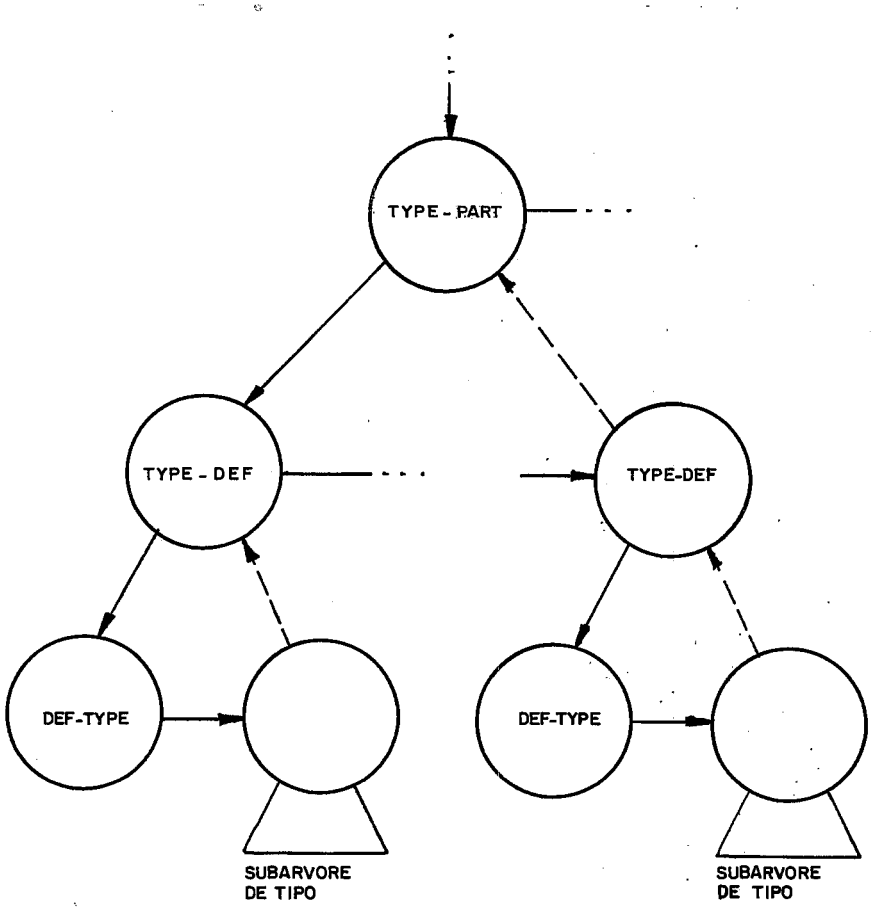


Figura VI.12

Ações Semânticas

Para cada subárvore TYPE-DEF será verificado se o identificador que representa o nome do tipo, já tinha sido declarado anteriormente no Bloco. Caso isto ocorra será enviada uma mensagem de erro referente a este acontecimento.

PRODUÇÃO TYPE

Aqui serão mostradas todas as subárvores geradas na análise desta produção. Primeiramente veremos os tipos não estruturados, que são obtidos na produção SIMPLE-TYPE mas que englobaremos a esta produção. A seguir veremos os tipos estruturados.

Tipos Não Estruturados

Os tipos considerados não estruturados são: os escalares standard, escalares definidos pelo programador e os subranges.

Tipo Escalar Standard

São quatro os tipos escalares standard: integer, real, Boolean e char.

Como foi visto no Capítulo V estes tipos constarão da Package STANDARD e serão instalados no nível 0 da Tabela de Símbolos.

Ao ser encontrado um desses tipos o código gerado será o USED-TYPE pois seus nomes serão considerados simples identificadores.

Dentro da Rotina TAB-SIMBOLOS, ao ser pesquisado o endereço do nó onde foi definido este tipo, será reconhecido, a-

través do valor da função hash, que se trata de um dos quatro tipos standard. Então na componente PTR2 do nó USED-TYPE, será colocado o código do tipo correspondente, que são: INTEIRO, REAL, BOOLEANO e CHAR.

Tipo_Escalar_Definido_pelo_Programador

Na Figura(VI.13)podemos ver a subárvore gerada para este tipo

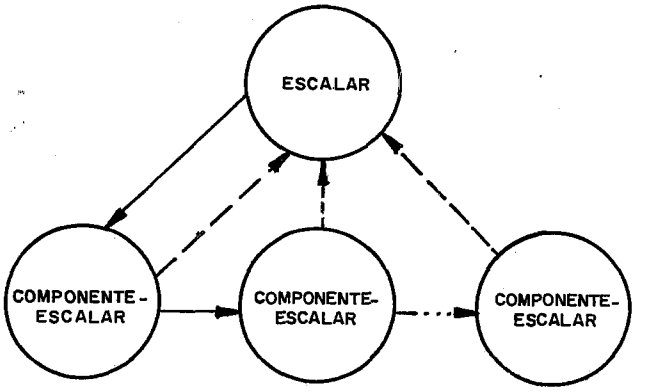


Figura VI.13

Ações Semânticas

A lista de componentes do escalar deverá ser percorrida para verificar se o identificador que representa a componente já tinha sido declarado neste Bloco. Caso isto ocorra, o nó desta componente terá seu código modificado para SUBARVORE-ERRADA. Uma mensagem de erro será enviada notificando o fato ocorrido.

Todos os nós COMPONENT-ESCALAR apontam para o nó ESCALAR, através da componente PTR1. Isto se faz necessário para análise semântica de operações com componentes de escalar, que necessitam pertencer a mesma subárvore ESCALAR.

O nó ESCALAR possuirá na componente PTR1 um ponteiro para o nó DEF-TYPE correspondente, caso o escalar seja definido em uma declaração do tipo. No caso de ser um tipo definido na declaração de uma variável, a componente PTR1 terá o valor NIL. Esta informação nos será útil na análise semântica de operações com Set.

Subrange

Como é sabido um subrange é um subconjunto finito, de um tipo escalar previamente declarado. Então poderemos ter um subrange de um escalar definido pelo programador como também de um escalar standard (exceto real). Na Figura (VI.14) podemos ver a subárvore gerada para um subrange de um escalar definido e na Figura (VI.15) os subranges dos tipos standard.

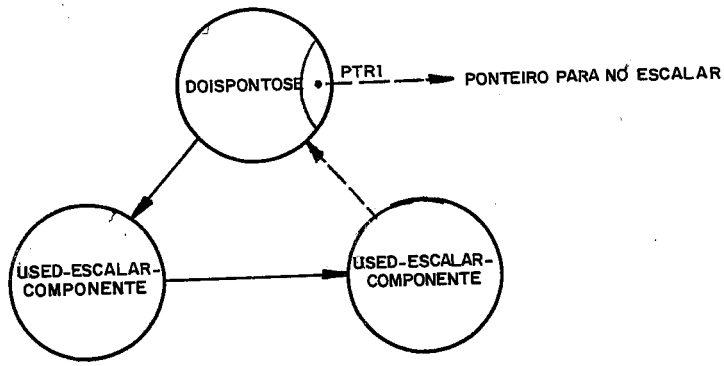


Figura VI.14

Figura VI.15

Ações Semânticas

Note que as subárvores apresentadas nas Figuras (VI.14) e (VI.15) já estão em suas formas definitivas, ou seja, já passaram pelo processo de Análise Semântica.

Se observarmos o Autômato da produção SIMPLE-TYPE veremos que um subrange é formado pela produção CONSTANT seguida do símbolo .. que por sua vez é seguido pela produção CONSTANT.

No estado final da produção SIMPLE-TYPE no caso de subrange, teremos no topo da PILHA-SEMÂNTICA um nó de código DOIS PONTOS, pai de dois nós que podem ser dos seguintes códigos: USED-CTE, USED-ESCALAR-COMPONENT, e todos os códigos referentes a valores de constantes inclusive os códigos MUNARIOI e MUNARIOR.

Se o código do primeiro filho for USED-ESCALAR-COMPONENT o código do segundo também terá que ser este. Além disso devem ter sido ambos declarados no mesmo tipo escalar (e este deve ser um tipo declarado). Isto poderá ser verificado utilizando-se as informações contidas nas componentes PTR1, dos dois nós.

Caso seja encontrado algum erro, o código no nó pai mudará para SUBARVORE-ERRADA, caso contrário mudará para DOISPONTOS, e em PTR1 será colocado o endereço do nó ESCALAR que corresponde ao tipo base do subrange.

Quando o código do primeiro filho não for de componente de Escalar, também não poderá ser uma constante do tipo real, nem string de tamanho maior que 1. Isto vale também para o segundo filho. Os tipos dos dois nós filhos devem ser iguais, isto é: inteiro, booleano ou char. Depois desta verificação o código será modificado para DOISPONTOSI, se inteiro, DOISPONTOSB se booleano, DOISPONTOSC se char ou SUBARVORE-ERRADA, caso os tipos não coincidam ou, um ou os dois filhos já tiverem o código SUBARVORE-ERRADA.

Tipo já Declarado

Ainda na produção SIMPLE-TYPE podemos ter a geração de uma subárvore que se constitui apenas de um nó. Trata-se da referência a um tipo já declarado. O código para este tipo é o USED-TYPE. Este nó, caso seja a referência de um tipo estruturado terá na componente PTR1 o endereço do nó correspondente ao tipo definido na sua declaração, em TYPE-PART. Caso o tipo seja um escalar ou subrange, em PTR1 teremos o endereço do nó onde realmente foi declarado o tipo, isto é, a origem do tipo. Caso seja um tipo standard, já tecemos comentários sobre este no início deste ítem.

O motivo desta distinção no conteúdo da componente PTR1 de um USED-TYPE definido como estruturado e um USED-TYPE escalar, se faz necessário pois a estratégia escolhida para a verificação de igualdades de tipos para as operações do PASCAL foi a seguinte: dois objetos são do mesmo tipo, quando estruturados, se constam da mesma declaração ou se foram declarados com um mesmo identificador de tipo.

Quando os objetos são de tipo escalar definido pelo programador valerá este mesmo princípio; se foram dos seguintes tipos: subrange de escalar definido pelo programador, escalar standard ou subrange de tipo standard, bastará que o tipo base seja o mesmo.

Ações Semânticas

A componente PTR1 do nó USED-TYPE deve vir da consulta à Tabela de Símbolos, apontando para um nó DEF-TYPE. Caso isto não ocorra o código será feito SUBARVORE-ERRADA, caso contrário o nó USED-TYPE será devidamente preenchido de acordo com o critério exposto anteriormente.

Tipos Estruturados

Tipo ARRAY

Na Figura(VI.16)podemos ver a subárvore para o tipo Array.

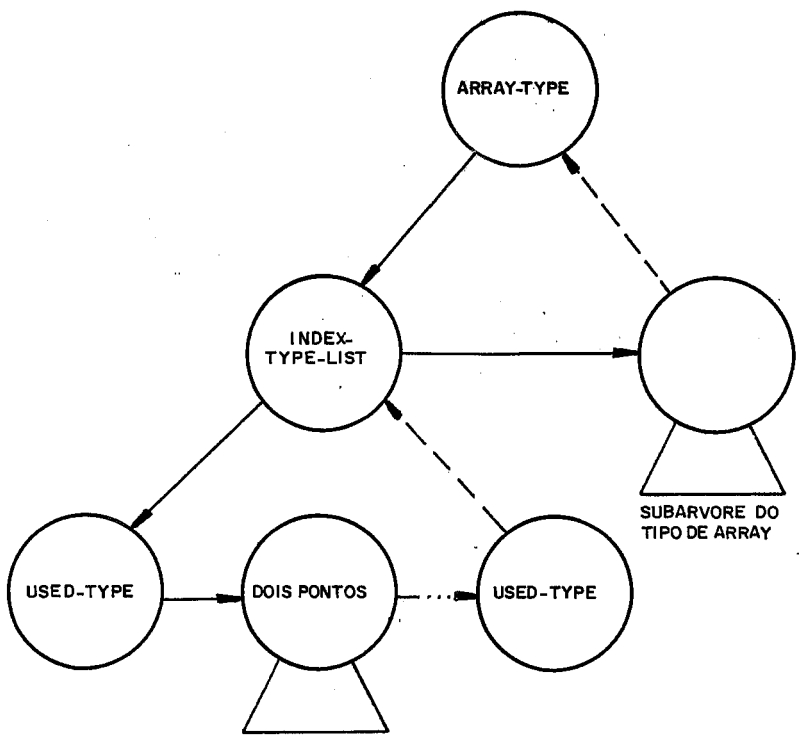


Figura VI.16

Ações Semânticas

A lista de nós que estão sob o nó INDEX-TYPE-LIST deve ser percorrida para que seja verificado, caso o nó tenha o código USED-TYPE, se o tipo deste é: inteiro, real ou um tipo estruturado. Caso isto ocorra uma mensagem de erro será emitida e o código do nó deste index será feito SUBARVORE-ERRADA.

Tipo_RECORD

Na Figura (VI.17) podemos ver a subárvore gerada para uma estrutura de Record, definida com as duas partes: a FIXED-PART e a VARIANT-PART.

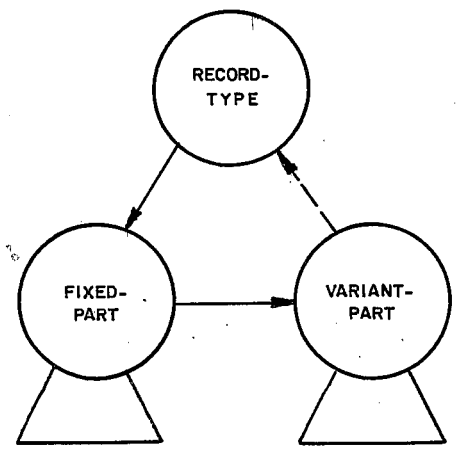


Figura VI.17

Na Figura (VI.18) mostramos como seriam as subárvores de um tipo Record no caso em que estrutura não contém (a) componente, (b) a Variant-Part e (c) a Fixed-Part.

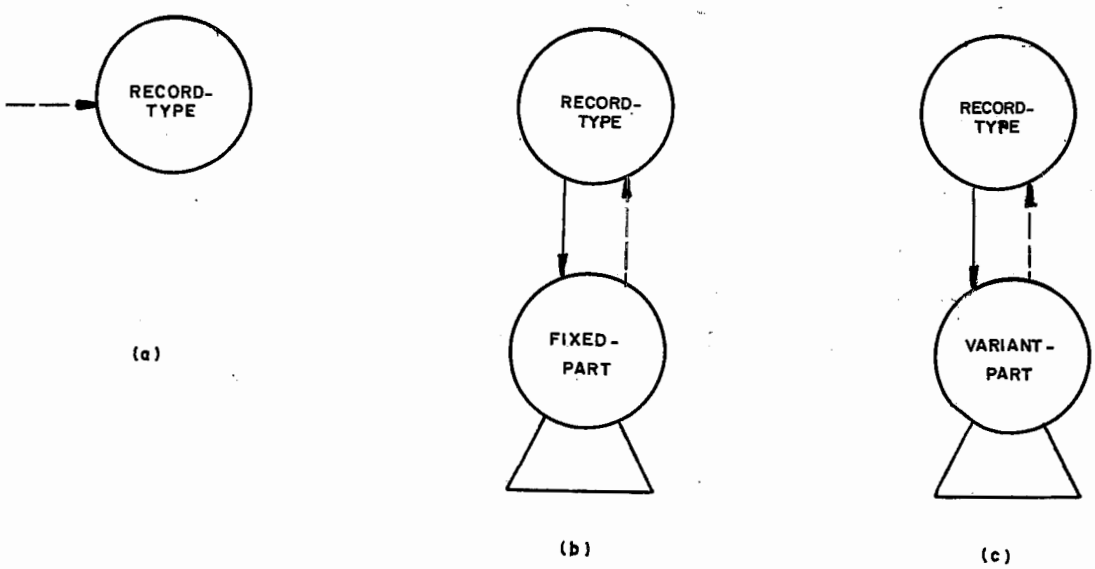

RECORD-
TYPE
(a)
RECORD-
TYPE
FIXED-
PART
(b)
RECORD-
TYPE
VARIANT-
PART
(c)

Figura VI.18

Subárvore FIXED-PART

A Figura (VI.19) apresenta a subárvore cuja raiz é o nó FIXED-PART.

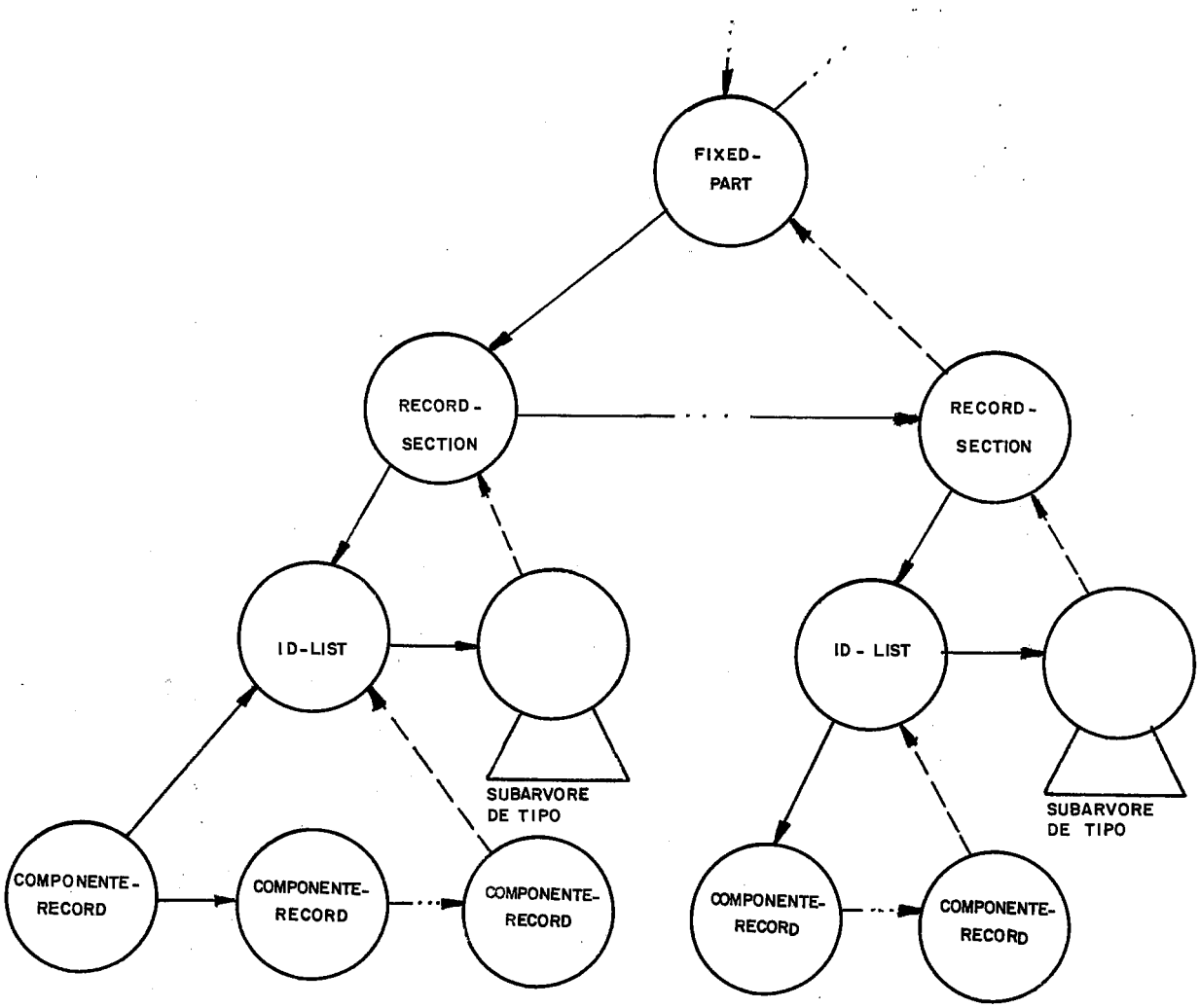


Figura VI.19

Os nós COMPONENT-RECORD têm na componente PTNOME a posição em VETNOMES, onde foi registrada a cadeia de caracteres que forma seu nome, apesar de não serem inseridos na Tabela de Símbolos.

Subárvore VARIANT-PART

Na Figura (VI.20) apresentamos a subárvore cuja raiz é o nó VARIANT-PART.

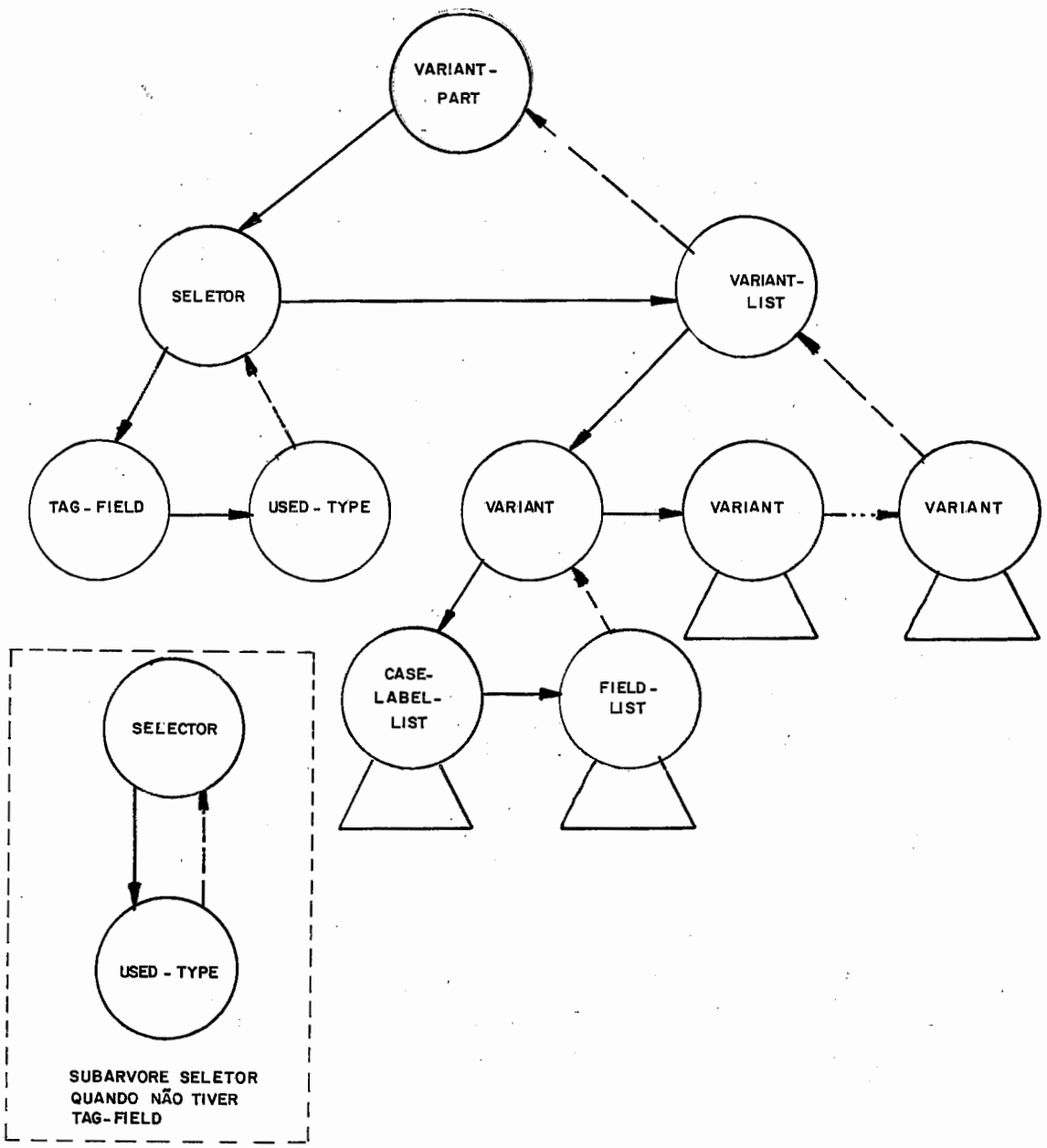


Figura VI.20

Na Figura(VI.21) podemos ver a subárvore de VARIANT-PART quando VARIANT-LIST não tiver componentes.

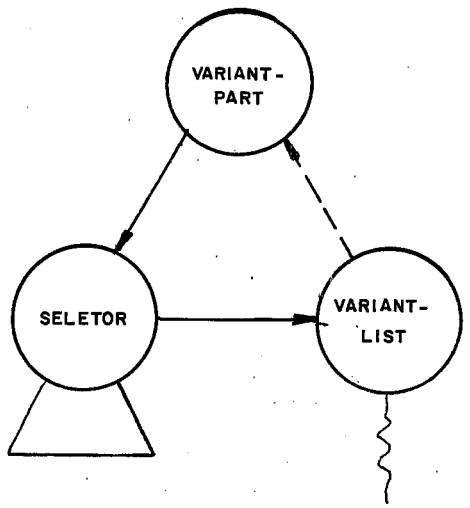


Figura VI.21

O nó FIELD-LIST que aparece na subárvore VARIANT-PART é raiz de uma subárvore semelhante a subárvore RECORD-TYPE. Vejamos na Figura(VI.22) a subárvore FIELD-LIST, um nível mais abaixo.

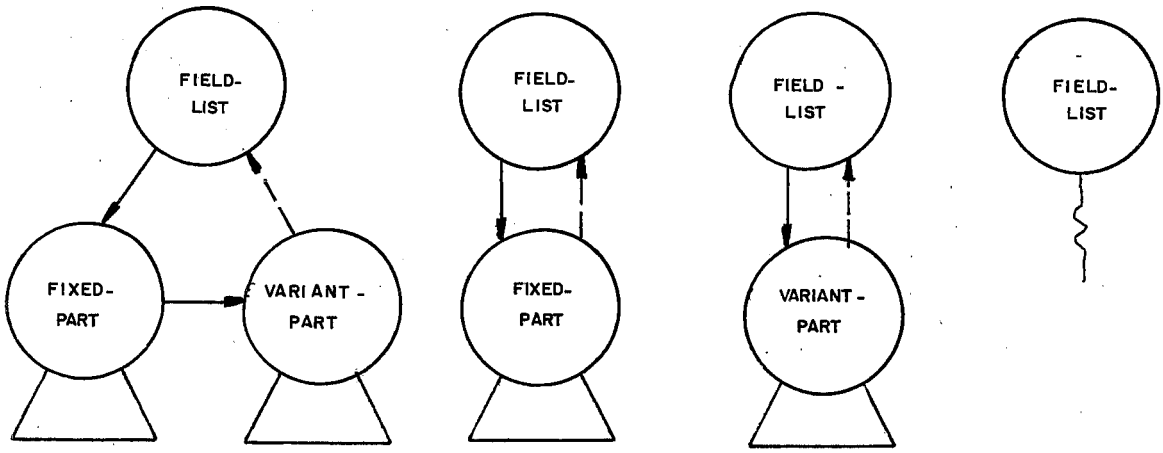


Figura VI.22

Ações Semânticas

Em uma estrutura de Record todas as componentes devem ter nomes distintos, até mesmo as que ocorrerem em diferentes Variants.Para realizar esta verificação um ponteiro será utilizado para fixar cada componente da lista, começando da primeira, enquanto que outro ponteiro percorrerá a lista de componentes seguintes, que se encontrem no mesmo nível.

A subárvore VARIANT-PART além desta verificação, deverá ter checado se o tipo do SELETOR, representado pelo nó USED-TYPE, é escalar. Isto será verificado através das componentes PTR1 e PTR2 do nó USED-TYPE.

As constantes de cada subárvore CASE-LABEL-LIST devem ser percorridas e checadas uma a uma para confirmar se o tipo é o mesmo do que consta em USED-TYPE, na subárvore SELETOR.

Como dissemos anteriormente o nó FIELD-LIST, na subárvore VARIANT, é raiz de uma árvore semelhante a do RECORD-TYPE. Por isso todas as verificações ditas realizadas para a subárvore RECORD-TYPE serão também feitas para cada subárvore FIELD-LIST.

Tipo SET

Na Figura(VI.23)podemos ver a subárvore gerada para o tipo Set.

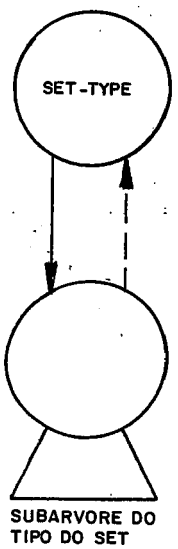


Figura VI.23

A Figura (VI.24) apresenta os tipos possíveis para a subárvore do Tipo Set.

Figura VI.24

Ações Semânticas

O tipo do set será gerado na produção SIMPLE-TYPE. Isto significa que a subárvore do tipo já está checada inclusive podendo ter o código SUBARVORE-ERRADA.

Se o tipo for USED-TYPE, precisaremos checar se é de tipo estruturado. Este fato pode ser verificado através da componente PTR2, ou seja, se PTR2 tiver o valor ESCALAR, DOISPONTOS, ou os tipos standard INTEIRO, BOOLEANO e CHAR está correto. Caso contrário houve um erro, o que causará uma men-

sagem de erro, e a troca do código USED-TYPE para SUBARVORE-ERRADA.

Tipo_FILE

Na Figura (VI.25) apresentamos a subárvore para o tipo FILE.

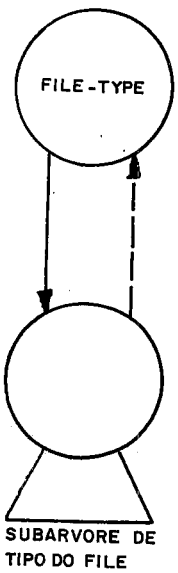


Figura VI.25

Ações_Semânticas

Não será permitido um file do tipo file, assim como, de um tipo estruturado que de alguma maneira tenha um tipo file na definição de seu tipo.

Tipo_POINTER

Em algumas implementações de compiladores PASCAL, o tipo de um pointer pode ser declarado com um tipo, cuja declaração encontra-se mais adiante, dentro do mesmo bloco. Consideramos que este fato dá uma certa comodidade ao programador, além de que, em algumas ocasiões se faz necessário, por isso, foi permitido no nosso trabalho.

A princípio o código do nó gerado para o tipo do pointer será UNDEF-TYPE. Este nó ao entrar na produção Ident será examinado pela rotina TABELA-SIMBOLOS, que deverá verificar se o identificador correspondente já tinha sido declarado anteriormente. Caso este fato não tenha ocorrido ainda, este identificador será instalado na Tabela de Símbolos, sendo criado um registro de ocorrência para o nó UNDEF-TYPE.

Quando este tipo for declarado, será encontrado um registro de ocorrência apontando para um nó de código UNDEF-TYPE. Então o código UNDEF-TYPE será modificado para USED-TYPE, e na sua componente PTR1 será colocado o ponteiro para o nó de declaração do tipo (DEF-TYPE). O registro de ocorrência passará a apontar para o nó de declaração do tipo, ou seja, o nó DEF-TYPE.

Caso o identificador já tivesse sido registrado na Tabela de Símbolos, a componente PTR1 receberá o endereço do nó onde se encontra declarado o tipo.

Na Figura (VI.26) podemos observar a subárvore gerada

para o tipo Pointer, de um tipo ainda não declarado.

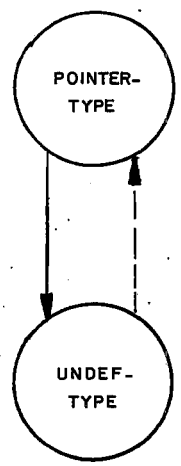


Figura VI.26

Na Figura (VI.27) temos a subárvore resultante para um tipo Pointer, cujo tipo já tinha sido previamente declarado.

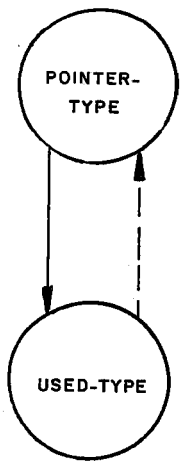


Figura VI.27

Ações Semânticas

No caso em que o tipo do pointer já tiver sido declarado, isto é, o nó UNDEF-TYPE vier da produção IDENT com o valor da componente PTR1 diferente de nil, deverá ser checado se o código do nó, apontado por PTR1 é DEF-TYPE. Caso isto ocorra o código UNDEF-TYPE será modificado para USED-TYPE, e este nó será completado no que se refere a informação de tipo (vide ítem Tipo já declarado).

Quando este identificador de tipo não tiver sido de-

clarado como um tipo, o código do nó UNDEF-TYPE será modificado para SUBARVORE-ERRADA e uma mensagem de erro será emitida, notificando este fato.

Tipo_PACKED

Como podemos verificar no autômato da produção Type, existem 4 tipos de packeds: Packed-Array, Packed-File, Packed_Set e Packed-Record. Na Figura(VI.28)estão representadas as 4 subárvores geradas para estes tipos relacionados acima.

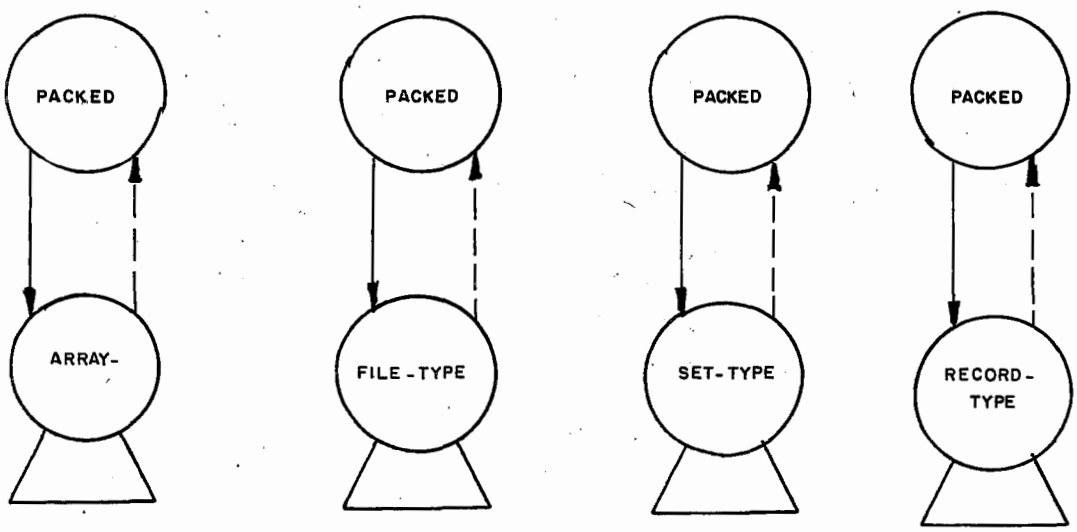


Figura VI.28

Como podemos observar na Figura acima, as subárvores que formam os tipos do Packed são as mesmas que são formadas para o tipo propriamente dito.

Ações_Semânticas

O tipo dos elementos de uma estrutura Packed, não pode ser estruturado. O tipo Packed-Set não precisa ser veri-

ficado quanto a isto, e nos tipos Packed-Array, Packed-File esta verificação não traz qualquer dificuldade. A verificação se torna complexa quando o Packed é de um tipo Record. Esta análise, será feita através da rotina VERIFICA-PACKED-RECORD, que verificará se alguma componente foi declarada com um tipo estruturado.

Observe aqui o quanto ajuda o nó USED-TYPE possuir as informações de seu tipo nas componentes PTR1 e PTR2, de seu nó.

Subárvore VAR-PART

A Figura (VI.29) apresenta a subárvore gerada na parte de declarações das variáveis do Bloco.

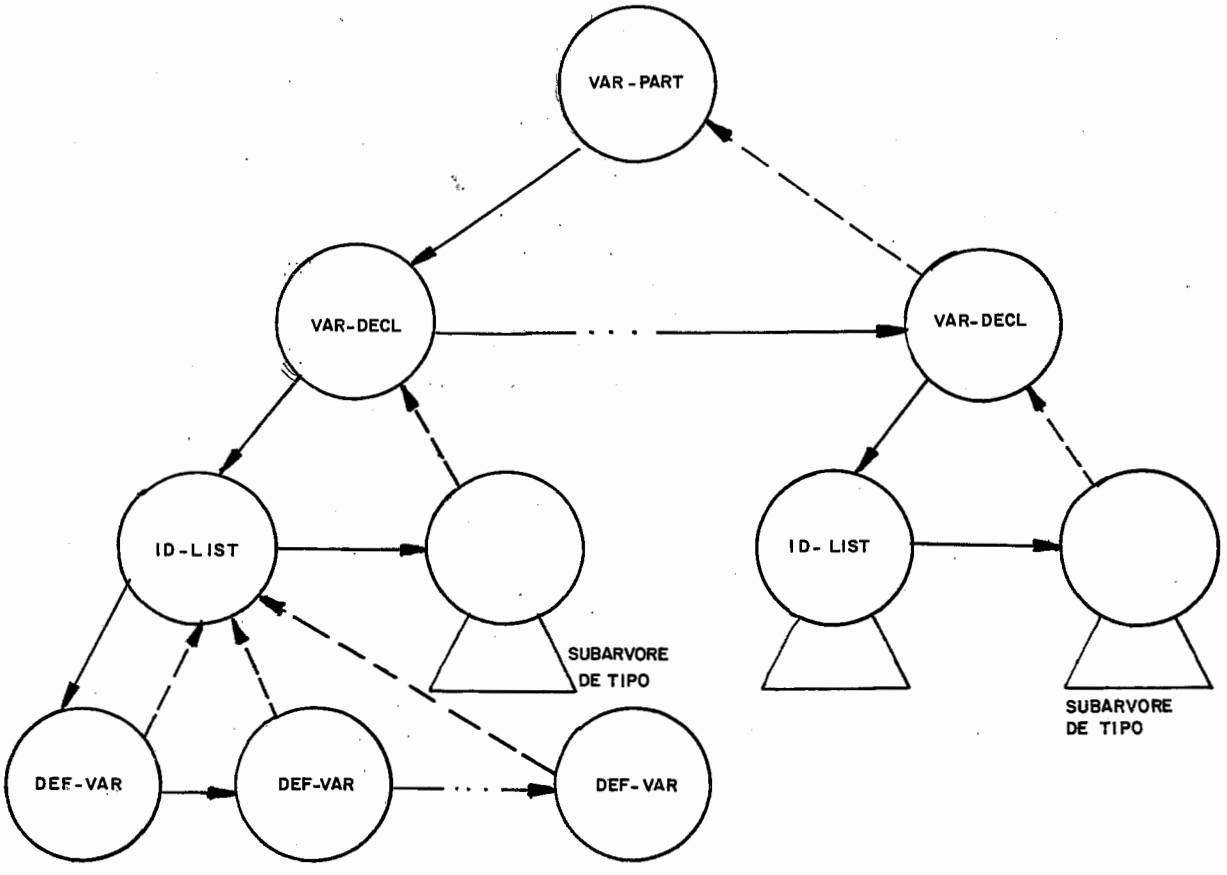


Figura VI.29

Note que mesmo que a lista de variáveis contenha apenas um identificador, o nó ID-LIST existirá, como pai desta subárvore unitária, formada apenas por um nó DEF-VAR.

Os nós de código DEF-VAR apontam para o pai ID-LIST, para facilitar a verificação do tipo a que pertencem.

Ações Semânticas

Para cada subárvore VAR-DECL, a lista formada pelos nós DEF-VAR deve ser percorrida para verificar-se se algum dos identificadores utilizados, já tinha sido declarado anteriormente neste Bloco.

Subárvore SUBPROGRAM-PART

Na Figura (VI.30) apresentamos a subárvore SUBPROG-PART sem especificar seus filhos, o que faremos logo a seguir.

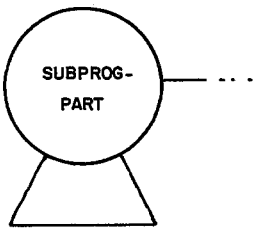


Figura VI.30

Na Figura (VI.31) podemos ver a subárvore gerada para declaração de uma Procedure, com parâmetros e na Figura (VI. 32) a subárvore gerada para declaração de uma Function, também definida com parâmetros

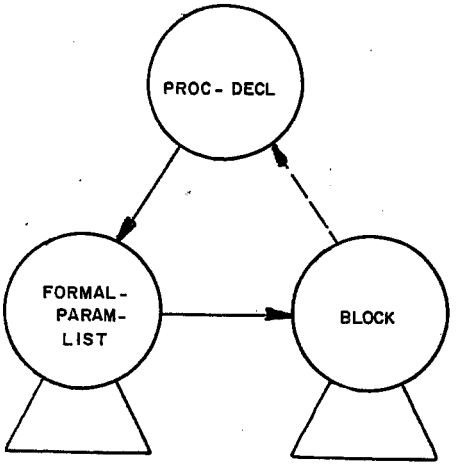


Figura VI.31

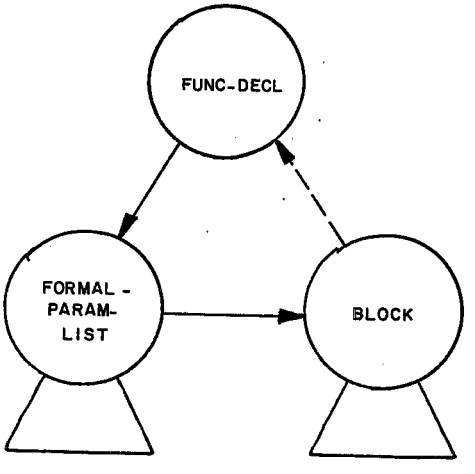


Figura VI.32

Os nós PROC-DECL e FUNC-DECL possuem em PTNOME a posição em VETNOMES, onde está armazenado a cadeia de caracteres que forma seus nomes

O tipo da função será armazenado nas componentes PTR1 e PTR2 do nó FUNC-DECL, tal como se fosse um nó USED-TYPE.

Vejamos a seguir como seriam as subárvores da declaração de uma Procedure, e de uma Função, definidas sem parâmetros. Nas Figuras (VI.33) e (VI.34) podemos observar essas subárvores.

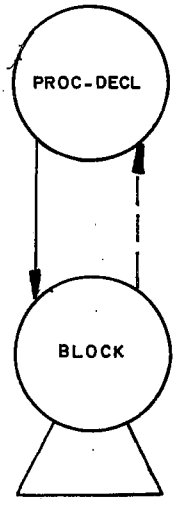


Figura VI.33

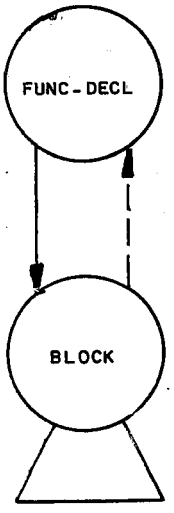


Figura VI.34

Na linguagem PASCAL a chamada de uma rotina pode ocorrer antes de sua declaração, caso a rotina seja referenciada como Forward. Na Figura (VI.35) podemos ver a subárvore da declaração Forward para uma Procedure com parâmetros e na Figura (VI.36) para a Procedure sem parâmetros. Na Figura (VI.37) temos a subárvore da declaração Forward para uma Função com parâmetros e na Figura (VI.38) para a Função declarada sem parâmetros.

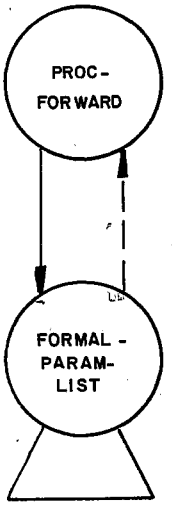


Figura VI.35

Figura VI.36

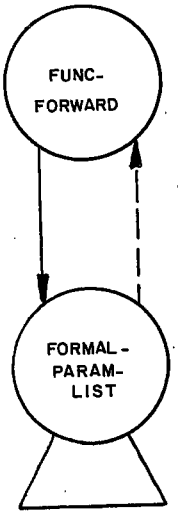


Figura VI.37

Figura VI.38

Os nós PROC-FOWARD e FUNC-FORWARD diferem dos nós PROC-DECL e FUNC-DECL, apenas no código, no que se refere às informações, são semelhantes.

Na referência Forward de uma rotina, será criado um registro de ocorrência na Tabela de Símbolos, que apontará para o nó de declaração Forward.

Quando ocorrer a declaração de uma rotina já declarada como FORWARD, a subárvore BLOCK que corresponde ao corpo da rotina, será pendurada no nó FORWARD; que já possuirá a lista de parâmetros, se a rotina tiver parâmetros, e no caso de função, o nó FUNC-FORWARD já terá o tipo da função registrado. Então o código deverá ser mudado para FUNC-DECL caso seja uma função ou PROC-DECL, caso seja uma procedure.

Ações Semânticas

Quando uma rotina for previamente declarada como Forward, seus parâmetros e tipo, no caso de Função devem aparecer apenas na declaração Forward.

Após a formação subárvore que representa o heading de uma rotina, teremos que verificar se esta já foi anteriormente declarada como Forward.

Quando isto acontecer se a rotina apresentar a ocorrência de parâmetros, e/ou tipo, no caso de uma Função, será emitida uma mensagem de erro notificando este fato. Independen-

te de ter havido erro, o nó que se encontra no topo da PILHA-SEMÂNTICA será descartado e o topo da pilha receberá o endereço do nó da declaração Forward. No caso de terem havido erros, conforme dito acima, valerá a referência feita na declaração Forward.

PRODUÇÃO FORMAL-PARAM-LIST

A Figura(VI.39) apresenta a subárvore para esta produção, apresentando uma subárvore para cada tipo de parâmetro.

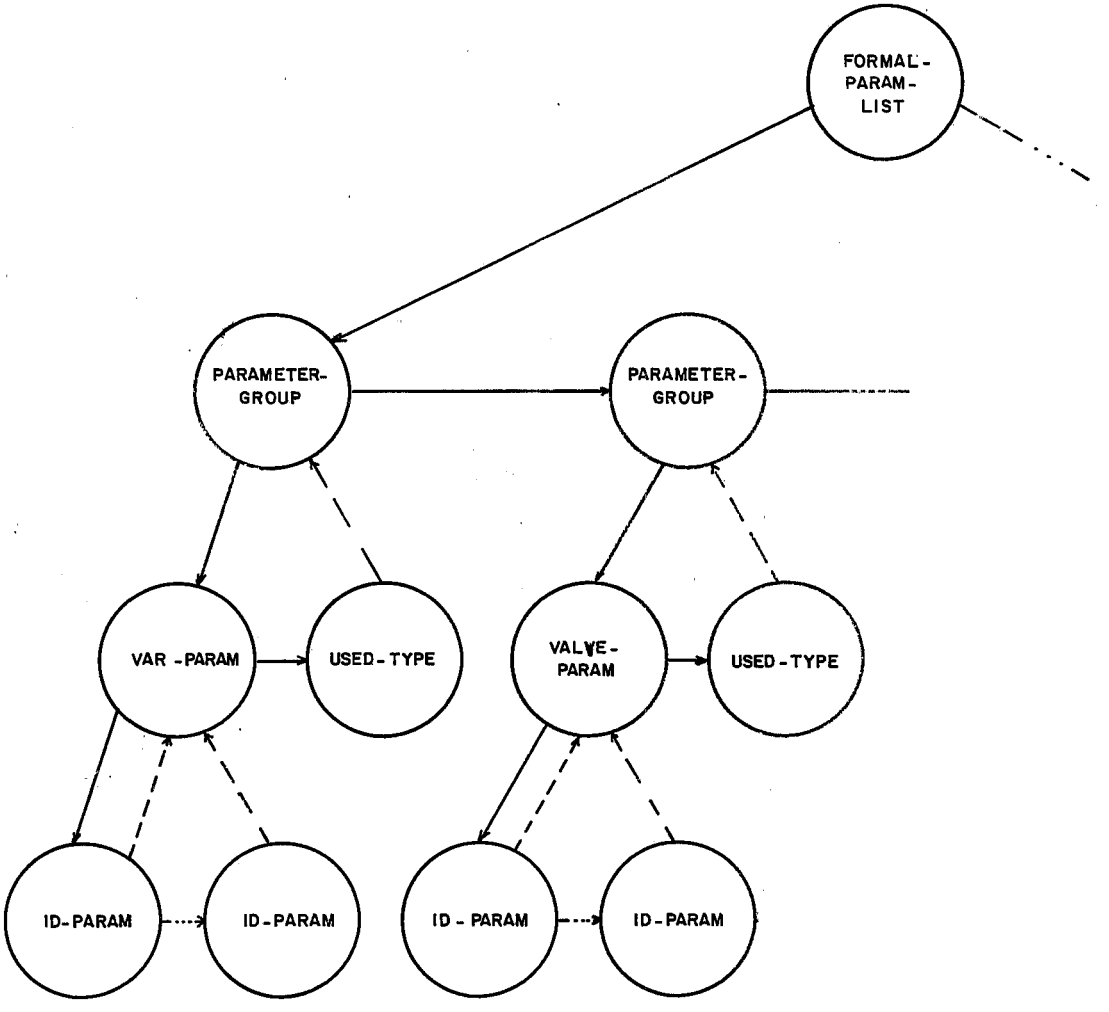


Figura VI.39

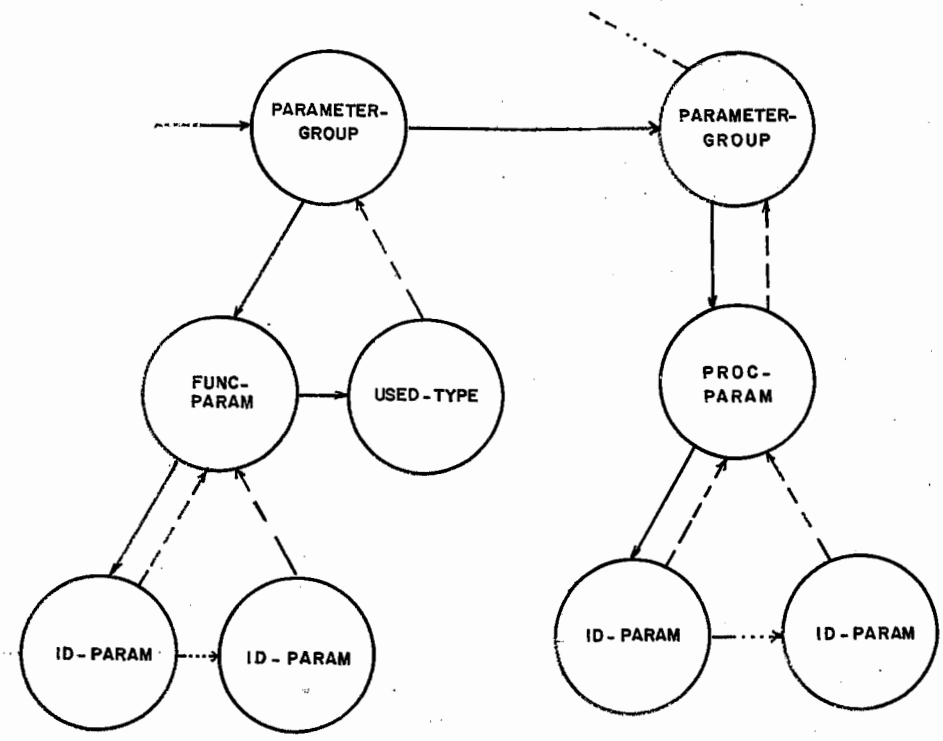


Figura VI.39 (continuação)

Como podemos observar, existe um código para cada tipo de parâmetros, isto é, para passagem por valor temos VALUE-PARAM, para passagem por referência temos VAR-PARAM, para parâmetros funções, FUNC-PARAM e no caso de parâmetros do tipo procedure, PROC-PARAM.

Os nós ID-PARAM apontam para o nó pai, para tornar mais fácil a verificação do tipo.

Ações Semânticas

Os nós USED-TYPE devem ser checados para verificar se o identificador utilizado realmente foi declarado como um tipo.

Nas subárvores PARAMETER-GROUP cujo tipo de parâmetro é um FUNC-PARAM, o nó USED-TYPE não pode representar um tipo estruturado com exceção do tipo pointer. Caso isto não ocorra, será emitida uma mensagem de erro, e o código do nó PARAMETER-GROUP correspondente será modificado para SUBARVORE-ERRADA.

PRODUÇÃO VARIABLE

Para esta produção apresentaremos as subárvores, através de exemplos, pois não existe uma estrutura de árvores fixa para a representação de uma variável.

Vejamos então na Figura (VI.40) a subárvore gerada para a seguinte referência:

X [A, Y + Z, 5] . W

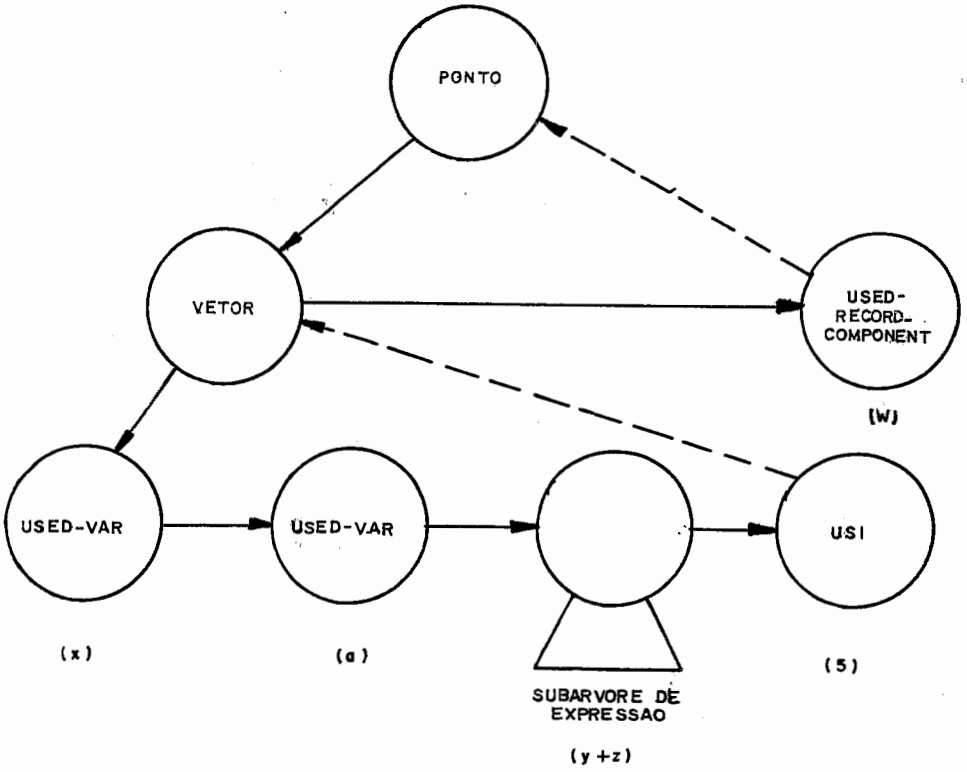


Figura VI.40

No nó de código PONTO, teremos nas componentes PTR1 e PTR2 as informações do tipo da componente W, que é a variável representada pela estrutura.

A Figura (VI.41) ilustra a subárvore da referência: X.Y.Z.

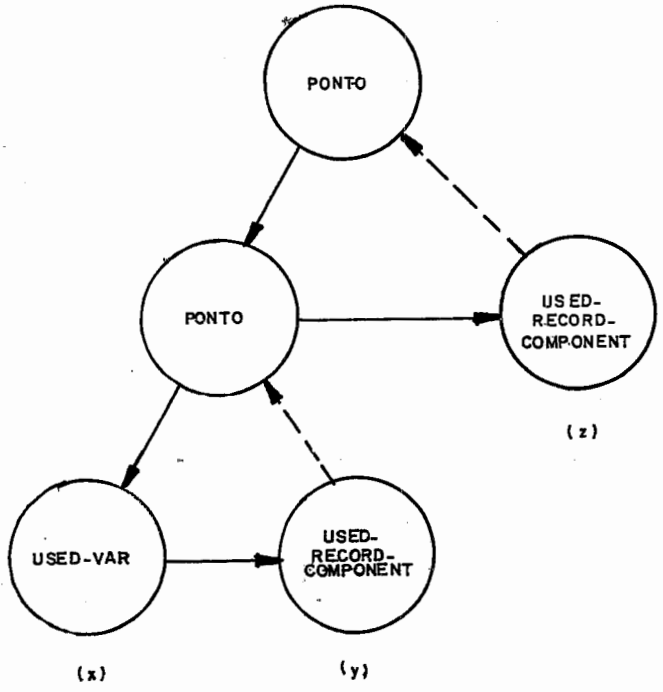


Figura VI.41

Note que X deve ser uma variável do tipo Record, por isso tem seu código igual a USED-VAR, e Y e Z devem ser componentes desta estrutura de Record pois aparecem a direita de um ponto, e então têm seus códigos igual a USED-RECORD-COMPONENT.

Para uma variável do tipo pointer de um tipo escalar qualquer, teríamos a subárvore apresentada na Figura (VI.42).

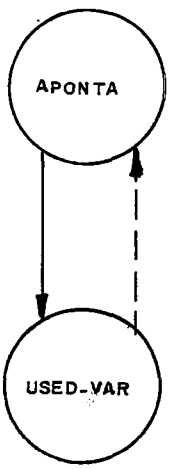


Figura VI.42

Observe que quando a variável é do tipo estruturado, e não estiver sendo referenciado como parâmetro, a raiz da subárvore de referência a esta variável possui um dos três códigos, de acordo com seu tipo, e são eles: PONTO, APONTA ou VETOR. Neste nó raiz, nas componentes PTR1 e PTR2, teremos as informações do tipo do objeto que está sendo referenciado.

Caso só apareça um identificador na produção variable, será gerado o nó apresentado na Figura (VI.43).

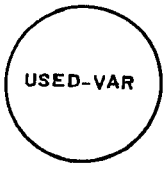


Figura VI.43

Neste nó na componente PTR1 teremos o endereço do nó onde o identificador foi declarado.

Ações Semânticas

A subárvore gerada deverá ser percorrida, de baixo para cima, para verificar se a referência está correta. Caso esteja errada, o código raiz será feito SUBARVORE-ERRADA.

Se a subárvore gerada foi a de código USED-VAR, será verificado se a componente PTR1 aponta para um nó DEF-VAR. Caso isto ocorra é verificado o tipo com que foi declarada a variável. Se este tipo for estruturado, o código do nó USED-VAR será modificado para PARAM-VAR. Quando a componente PTR1 não apontar para um nó DEF-VAR, o código será mudado para SUBARVORE-ERRADA.

SUBÁRVORES DE EXPRESSÃO

Expressões são construções denotando determinadas regras de computação para obtenção de valores, através da aplicação de operadores em operandos.

As regras de composição especificam a precedência dos operadores de acordo com quatro classes existentes. Na linguagem PASCAL o operador NOT possui a maior precedência, seguido pelos operadores multiplicativos (produção TERM) depois os operadores aditivos (produção SIMPLE-EXPRESSION), e finalmente os de menor prioridade, que são os operadores relacionais (produção EXPRESSION). Uma seqüência de operadores da mesma precedência serão executados da esquerda para direita.

As regras de precedência são refletidas pela sintaxe, e podem ser vistas nos Apêndices I e II.

Para melhor entendimento da geração das subárvores de Expressões iremos apresentar primeiro as subárvores geradas na produção FACTOR, que representa a formação dos operandos de uma expressão. A seguir veremos as produções TERM, SIMPLE-EXPRESSION e por último a produção EXPRESSION, que é a responsável pela geração de uma expressão como um todo.

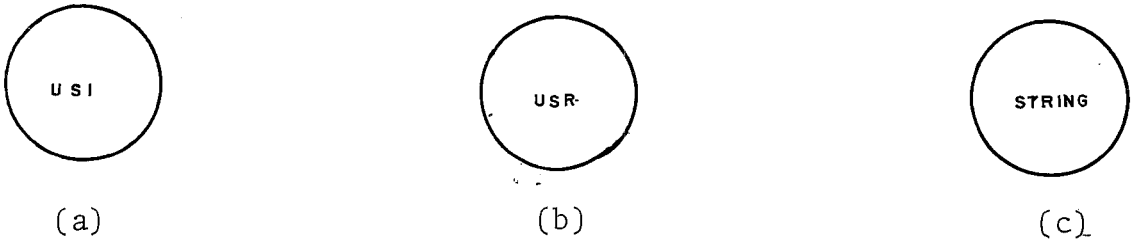
PRODUÇÃO FACTOR

As subárvores geradas nesta produção, as quais iremos apresentar neste ítem, representam as unidades mínimas de uma expressão, isto é, consistem nos operandos de uma expressão

PASCAL.

Constantes

Primeiramente veremos as subárvores geradas para as constantes que aparecem em uma expressão. Na Figura (VI.44) podemos observar em (a) uma constante inteira, em (b) uma constante real e em (c) a representação de uma string.



(a) (b) (c)

Figura VI.44

Valor_NIL

Para o valor NIL o nó gerado poderá ser visto na Figura(VI.45).

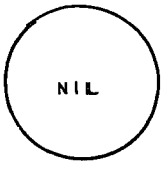


Figura VI.45

Ações Semânticas

Nenhuma ação semântica será necessária para esta subárvore.

Operador_NOT

Na Figura(VI.46) podemos visualizar a subárvore gerada para o operador unário NOT

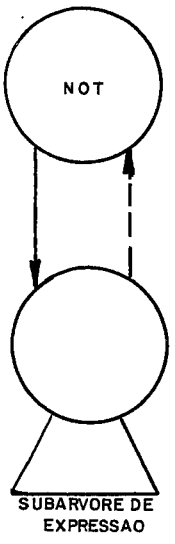


Figura VI.46

Ações_Semânticas

O tipo do fator representado pela subárvore filha do nó NOT tem que resultar um valor booleano.

Uma VARIÁVEL

Como vimos na produção VARIABLE, se o identificador fosse reconhecido como uma variável estruturada, seria gerada uma subárvore constituída apenas de um nó, com o código PARAM-VAR. Caso este identificador não tivesse sido declarado como variável, ou seja, na componente PTR1 não tivesse o endereço de um nó DEF-VAR, seria gerada uma subárvore também de apenas um nó, cujo código seria NAO-USED-VAR.

Na Figura (VI.47) podemos observar todas as subárvores possíveis que podem ser recebidas pelo reconhecimento da produção VARIABLE, dentro de FACTOR.

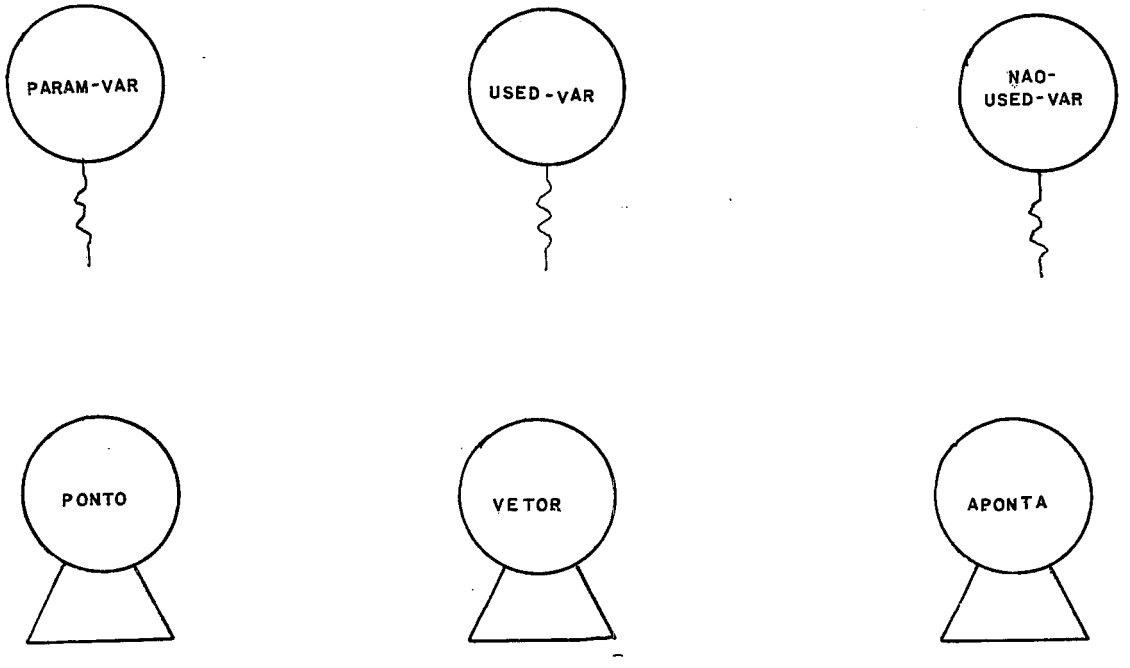


Figura VI.47

Ações Semânticas

Se o nó retornado tiver o código PARAM-VAR, seu tipo deve ser verificado, pois se for um set, seu código deverá mudar para SET-VAR. Isto se faz necessário pois identificadores de set podem estar representados como parâmetros mas também como operandos.

Retornando o nó NAO-USED-VAR, teremos 3 possibilidades: é uma componente de um Escalar, um identificador de constante ou um identificador de uma rotina, declarada sem parâmetros, standard ou não, que esteja como argumento de uma chamada de rotina ou simplesmente como operando de uma expressão, no caso de ser uma função.

No caso de componente de um Escalar, o código do nó será feito USED-SCALAR-COMPONENT. Neste nó a componente PTR1 aponta para o nó ESCALAR em que foi declarada.

Se verificarmos que o identificador representa uma constante, o nó mudará seu código para USED-CTE. Neste nó a componente PTR1 aponta para o nó DEF-CTE.

Ao esgotarmos as duas possibilidades descritas acima, teremos que verificar a última delas ou seja uma rotina. A primeira providência será verificar se é uma rotina standard. Isto pode ser feito verificando se a componente PTR1 aponta para um nó cujo código é FUNC-STAND ou PROC-STAND. Caso o seja a rotina TRATA-SUBPROG-STANDARD será chamada para realizar determinadas a

ções que serão explicadas no próximo capítulo quando, tratarmos de Packages.

Caso não seja subprograma standard, teremos que verificar se este identificador representa um subprograma, e que tenha sido definido sem parâmetros. A componente PTR1 deste nó deverá estar apontando para um nó com um dos três seguintes códigos: PROC-DECL, FUNC-DECL, PROC-SPECIFICATION ou FUNC-SPECIFICATION. Se esta condição for satisfeita, o nó apontado não poderá ter como primeiro filho a subárvore FORMAL-PARAM-LIST.

Caso haja uma ocorrência de erro, será emitida uma mensagem e o nó NAO-USED-VAR será feito SUBARVORE-ERRADA, senão o código será modificado para FUNC-CALL ou PROC-CALL de acordo com o tipo de subprograma.

Chamada de Rotina com Parâmetros

Se observarmos o autômato da produção FACTOR veremos que a entrada do estado 326 corresponde a uma chamada de subprograma que deve ter sido declarada com parâmetros.

A rotina SEMANTICA-TRADUTOR neste estado empilhará um nó cujo código é SUBPROG-CALL. A seguir serão geradas as subárvores de expressão, que correspondem aos argumentos deste subprograma, e que devem ser feitas filhas deste nó SUBPROG-CALL.

Caberá à parte semântica da rotina SEMANTICA-TRADUTOR, no estado final desta produção, verificar se o identificador que nomeia o suposto subprograma corresponde a declaração de uma Procedure ou uma Function, standard ou não.

Na Figura (VI.48) podemos observar a subárvore gerada na produção FACTOR para a chamada de uma Procedure, standard ou não.

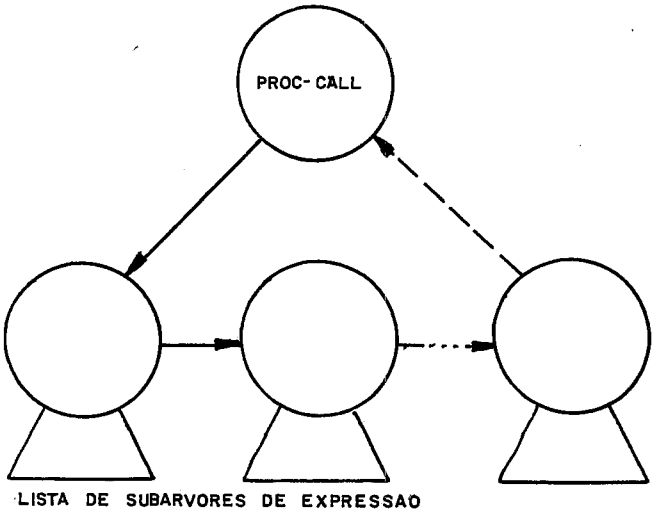


Figura VI.48

Na Figura (VI.49) podemos observar a subárvore gerada para a chamada de uma Function, standard ou não.

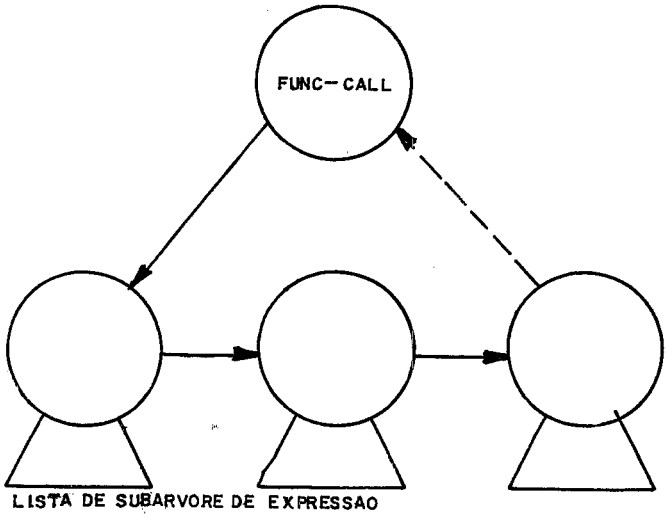


Figura VI.49

Ações Semânticas

A primeira ação semântica a ser realizada será de identificação do tipo de subprograma, ou seja, Procedure ou Function, ou até mesmo nenhum dos dois, caso em que será enviada mensagem do erro e trocado o código do nó para SUBARVORE-ERRADA.

Caso o subprograma seja standard, será chamada a rotina TRATA-SUBPROG-STANDARD.

No caso do subprograma não ser standard, uma extensa e complexa ação se dará: a de verificação de compatibilidade de tipo entre os parâmetros reais e os parâmetros formais.

A rotina responsável pela verificação dos parâmetros chama-se VERIFICA-PARÂMETROS, que trabalhará basicamente com dois ponteiros. O primeiro deles percorrerá a lista de expressões que formam os parâmetros reais, e inicialmente estará apontando para a primeira subárvore da lista. O segundo ponteiro marcará os parâmetros formais, na subárvore de definição e será inicializado com o endereço da primeira subárvore, que está sob o nó FORMAL-PARAM-LIST.

Os ponteiros serão avançados a medida que os parâmetros sejam comparados entre si. Caso algum ponteiro apontar "terra" antes do outro, uma mensagem de erro deve ser enviada, pois os parâmetros não conferem em número.

Obviamente no caso de VAR-PARAM, VALUE-PARAM e FUNC-PARAM, os tipos dos parâmetros devem coincidir entre si, além disso, se o parâmetro formal estiver sob o nó VAR-PARAM, a expressão que representa o parâmetro real deve constituir-se de uma variável.

No caso de passagem de subprogramas como parâmetro, as ações tornam-se ainda mais complexas. Cabe neste momento mostrarmos como seria a utilização de chamadas de subprogramas dentro do corpo de um subprograma, sendo essas rotinas chamadas parâmetros formais.

Na Figura (VI.50) podemos observar a cadeia que será formada pelos nós de chamada de subprogramas declarados como parâmetros, começando pelo nó ID-PARAM correspondente, que será construída na análise do corpo do subprograma que os utiliza.

Então ao depararmos com um parâmetro real do tipo Procedure ou Function, teremos que percorrer o encadeamento formado, comparando os parâmetros formais que constam na subárvore da definição da rotina, que está sendo enviada como parâmetro, com os parâmetros reais das subárvores que fazem parte da cadeia, utilizando para esta verificação a mesma rotina VERIFICA-PARÂMETROS.

Caso seja encontrado um erro, a mensagem associada ao fato ocorrido será emitida, mas nenhum código será mudado.

Sets

Neste ítem iremos tratar das subárvores geradas para set.

Para um set vazio a subárvore a ser gerada pode ser vista na Figura (VI.51).

Figura VI.51

Quando o set possui elementos, a subárvore a ser gerada pode ser vista na Figura (VI.52).

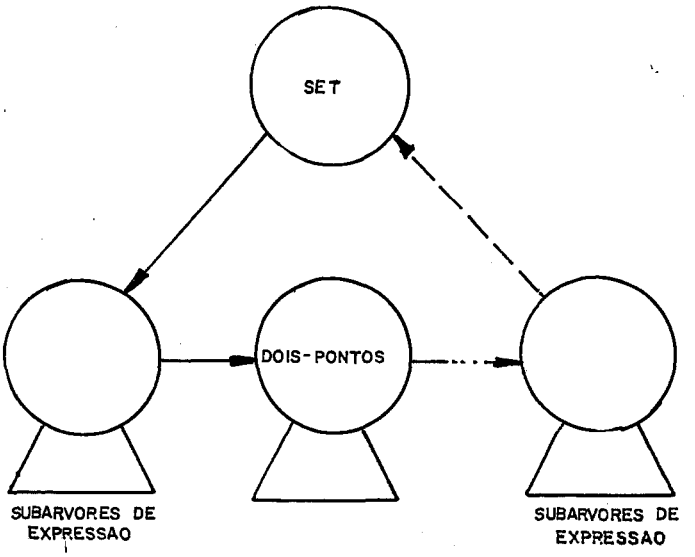


Figura VI.52

No nó SET teremos na componente PTR2 o código do tipo do set, os quais só poderão ser: INTEIRO, CHAR, BOOLEANO ou ESCALAR. Em PTR1 teremos ponteiro para o nó ESCALAR a que pertencem os elementos do set.

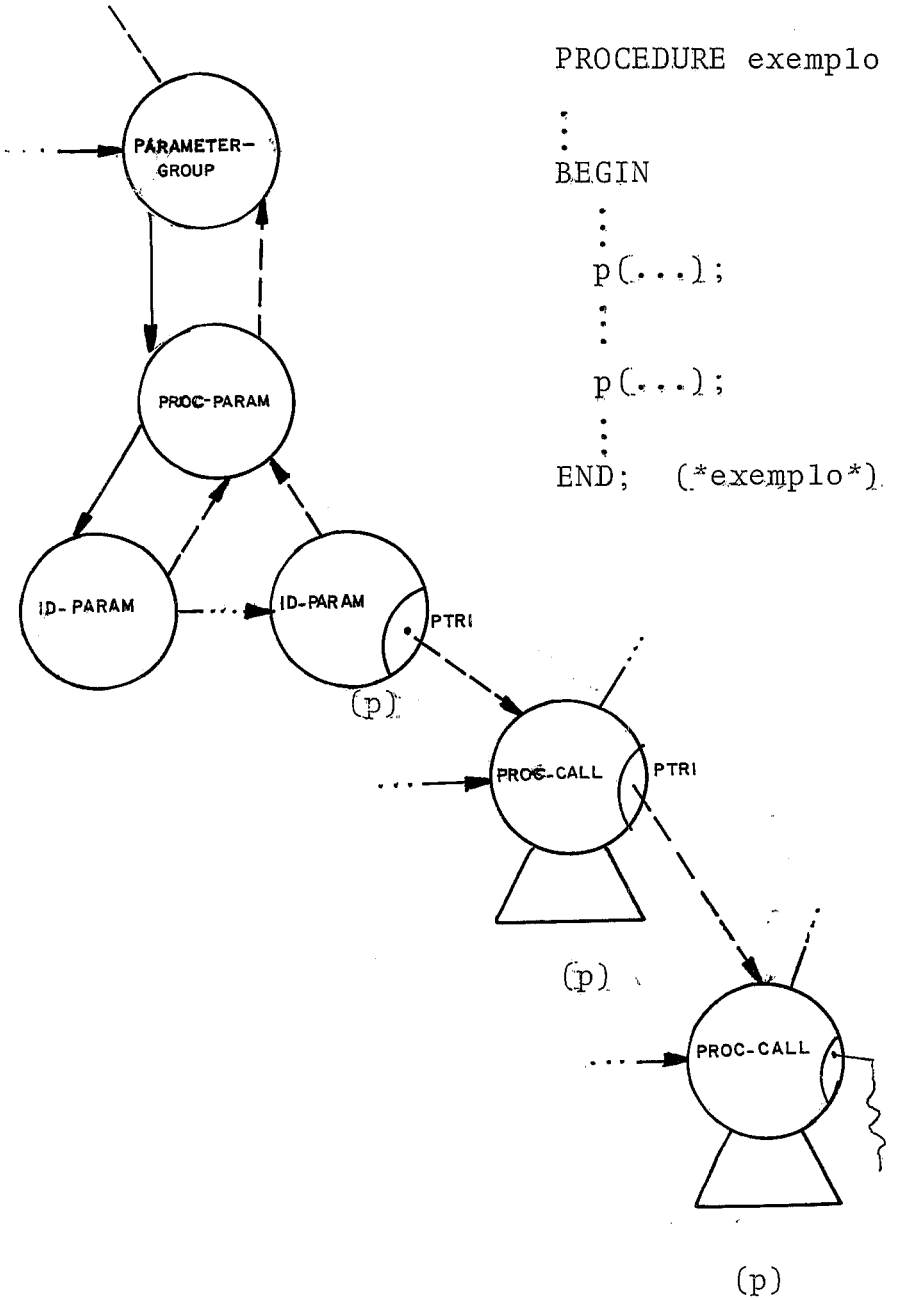
Ações Semânticas

Para a subárvore SET-VAZIO não serão efetuadas ações semânticas.

Quando o SET não for vazio, seus elementos deverão ser checados primeiro isoladamente pois devem ser de um tipo

escalar, sendo que no caso de subrange de expressão, as duas expressões devem resultar tipos escalares iguais. Feito isto deverão ser checados entre si pois todos os elementos de um set devem ser do mesmo tipo escalar.

Esta verificação será feita através de um ponteiro que percorrerá a lista, verificando a validade do tipo da expressão ou do subrange de expressão, e gravando na componente PTR2 do nó que SET, que se encontra no topo da pilha, o tipo do elemento. Caso o tipo não seja válido ou esteja incoerente com um tipo já anotado em PTR2, serão enviadas mensagens de erro referentes a falha ocorrida.




```
PROCEDURE exemplo (...;PRODECURE...,p);
  :
BEGIN
    :
   p(...);
    :
   p(...);
    :
END;   (*exemplo*)
```

Figura VI.50

Operador_AND

Na Figura (VI.53) podemos observar a subárvore gerada para o operador lógico AND.

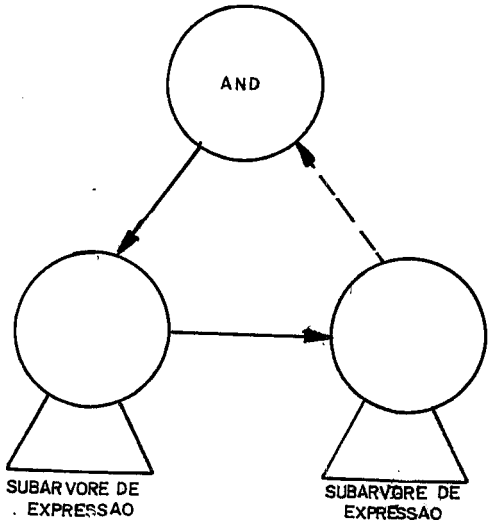


Figura VI.53

Ações_Semânticas

Os dois operandos devem ser do tipo booleano, ou seja, as expressões que representam os operandos podem ser subárvores de operadores que dêem como resultado o tipo booleano tais como: AND, OR, NOT, IN e os operadores relacionais, ou variáveis declaradas com tipo booleano, ou também uma Function do tipo booleano.

Caso esta condição não seja satisfeita será enviada mensagem para o erro ocorrido, mas o código do nó operador AND não será modificado.

Operadores DIV e MOD

Agrupamos esses dois operadores pois ambos só podem ser aplicados em operandos inteiros, resultando também um valor do tipo inteiro.

Na Figura (VI.54) está representada a subárvore para o operador DIV e na Figura (VI.55) a subárvores do operador MOD.

Ações Semânticas

As mesmas ações semânticas, serão realizadas para esses dois operadores. Essas seriam verificar se as duas expressões que representam os operadores desses operadores resultam um valor do tipo inteiro, podendo suas raízes serem nós dos seguintes códigos: DIV, MOD, MAISI, MENOSI, VEZESI, MUNARIOI, USED-VAR do tipo inteiro, uma constante do tipo inteiro, ou uma Function também do tipo inteiro.

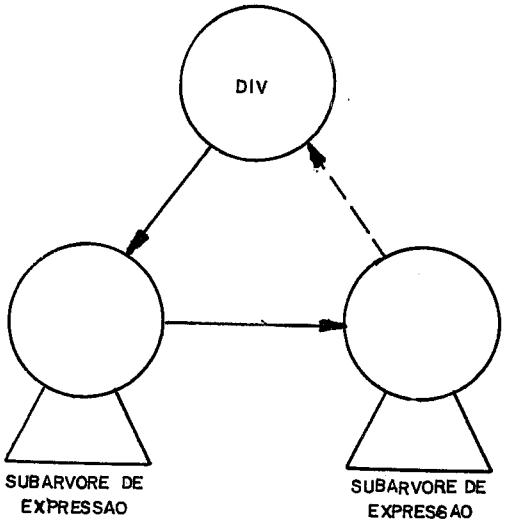


Figura VI.54

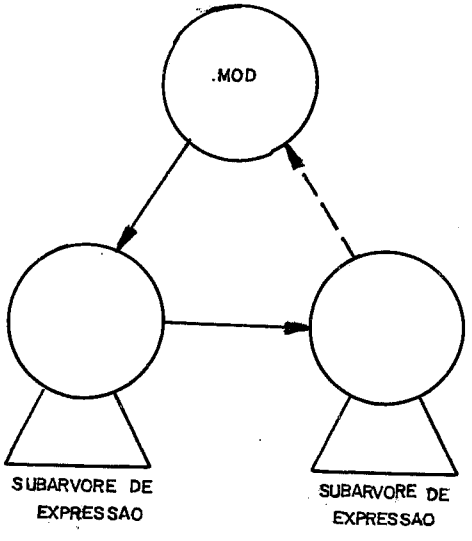


Figura VI.55

Operador /

Na Figura (VI.56) podemos observar a subárvore gerada para este operador.

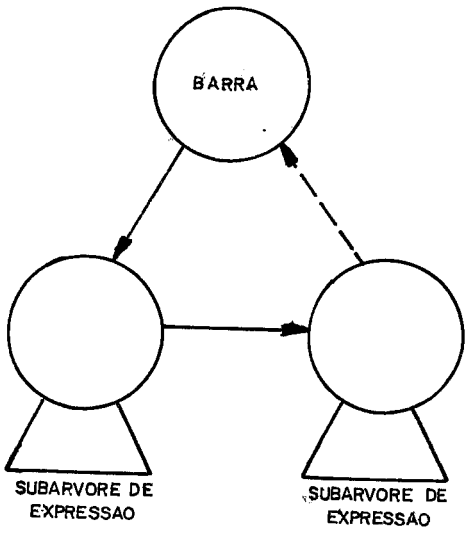


Figura VI.56

Ações Semânticas

O tipo dos operandos deste operador só podem ser real ou inteiro. Esta condição deverá ser checada, podendo as expressões resultarem tipos diferentes.

Caso algum operando não seja numérico, ou dos dois tipos permitidos, será enviada mensagem de erro correspondente a este fato ocorrido. O código da raiz da subárvore não será modificado.

Operador *

Poderão ser geradas três subárvores diferentes para este operador. Na Figura (VI.57) está representada a subárvore gerada quando ambos os operandos resultam valores reais, ou um real e outro inteiro. Na Figura (VI.58) podemos observar a subárvore gerada no caso em que ambos os operandos resultam valores do tipo inteiro.

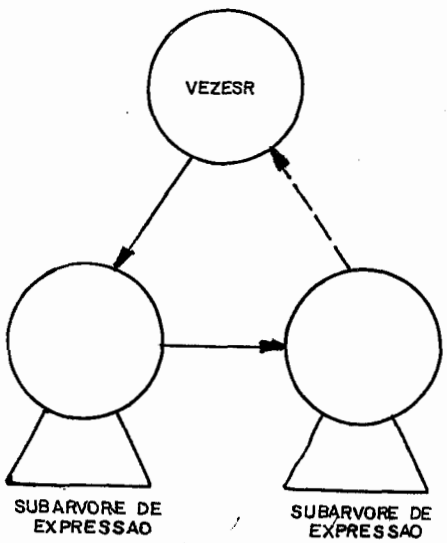


Figura VI.57

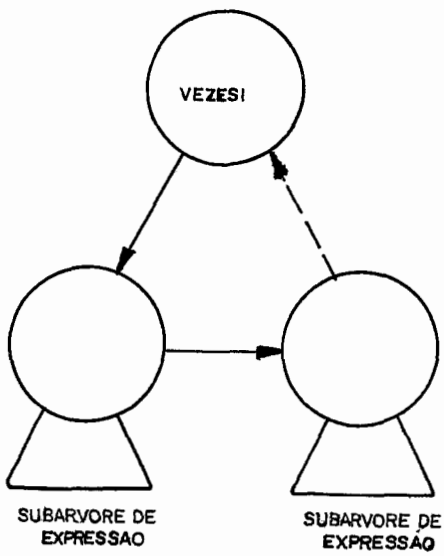


Figura VI.58

Na linguagem PASCAL este operador também pode ser utilizado para operar dois sets, equivalendo à operação de interseção entre dois conjuntos.

Na Figura (VI.59) temos a subárvore gerada para esta operação em que as raízes das subárvores que formam seus operandos podem ter os seguintes códigos: VEZESSET, MENOSSET, MAISSET, SET, SET-VAR ou SET-VAZIO.

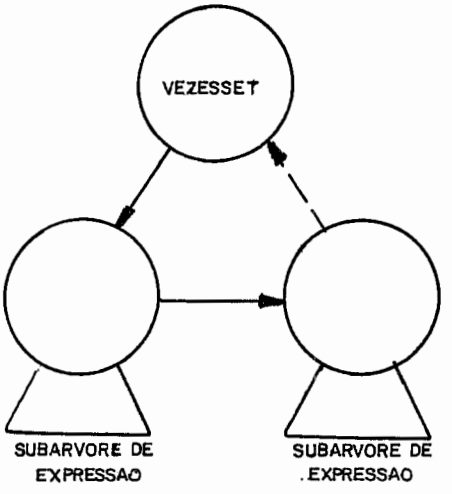


Figura VI.59

Note que no nó VEZESSET, na componente PTR2 teremos o código do tipo dos sets que estão sendo operados. No caso de ESCALAR ou DOISPONTOS teremos na componente PTR1 o ponteiro para o nó de declaração do tipo-base dos sets.

Ações Semânticas

A subárvore depois de montada terá examinada seus operandos para determinar qual o tipo de operação está sendo realizada. No caso do primeiro operando ser numérico, seu tipo deverá ser registrado em PTR2 do nó operador e o segundo operando deverá ser examinado. Caso este nó seja numérico, será enviada mensagem de erro notificando o ocorrido, e o código da raiz terá o código SUBARVORE-ERRADA, procedimento que também será feito quando se concluir que o primeiro operando não é numérico nem uma expressão set. Caso os operandos sejam numéricos os tipos resultantes determinarão a mudança no código da raiz da subárvore para VEZESI ou VEZESR.

Quando o primeiro operando tratar-se de uma expressão set, o tipo que se encontra na componente PTR2 do primeiro operando será anotado na componente PTR2 do nó operador em questão. Se este tipo for ESCALAR ou DOISPONTOS na componente PTR1 do nó operador será anotado o valor da componente PTR1 do nó operando. Ao examinar-se o nó do segundo operando, deverá ser verificado se este também é uma expressão set. Caso o seja, deverá haver uma comparação entre as componentes PTR2 e PTR1 dos nós operando e operador, e se houver diferença, será enviada uma mensagem do erro ocorrido e o código do nó operador será colocado SUBARVORE-ERRADA. Caso contrário o código do nó será modificado para VEZESSET.

Quando a segunda expressão não for de um tipo set uma mensagem deverá ser enviada, e o código do nó operador será colocado SUBARVORE-ERRADA.

PRODUÇÃO SIMPLE-EXPRESSION

Operadores Unários

No Autômato Finito que representa esta produção podemos observar que ao entrarmos pelo estado 305, estaremos reconhecendo o operador - (menos) unário, e pelo estado 304 teremos um + (mais) unário. Apenas o primeiro operador merecerá um código pois o sinal + precedendo um valor numérico, aritmeticamente não surte efeito algum.

Na Figura (VI.60) temos a subárvore gerada para o operador menos unário tendo como operando uma inteira, e na Figura (VI.61) temos subárvore gerada, no caso da expressão resultar um tipo real.

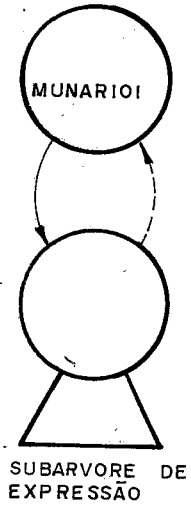


Figura VI.60

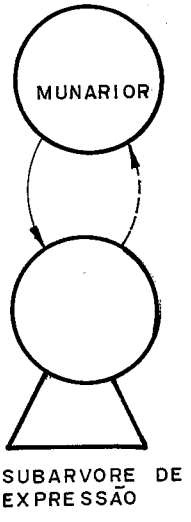


Figura VI.61

Ações Semânticas

A expressão que será operada pelo operador menos unário deverá ser verificada se corresponde a uma expressão numérica. Caso não o seja, o código do operador será feito SUBARVORE-ERRADA, e uma mensagem de erro deverá ser enviada. Se esta condição for satisfeita o tipo resultante da expressão deverá ser verificado para que se faça a distinção no código do nó do operador, em relação o tipo, isto é, código MUNARIOI ou MUNARIOR.

Apesar do operador + unário ser ignorado no que se refere à formação de uma subárvore, uma ação semântica deverá se realizar na verificação do operando deste operador. A expressão que segue este operador deverá ser do tipo numérica e caso isto não se realize, será criado o nó de código SUBARVORE-ERRADA, como pai desta expressão.

Operador_OR

Na Figura(VI.62)podemos observar a subárvore gerada para o operador lógico OR.

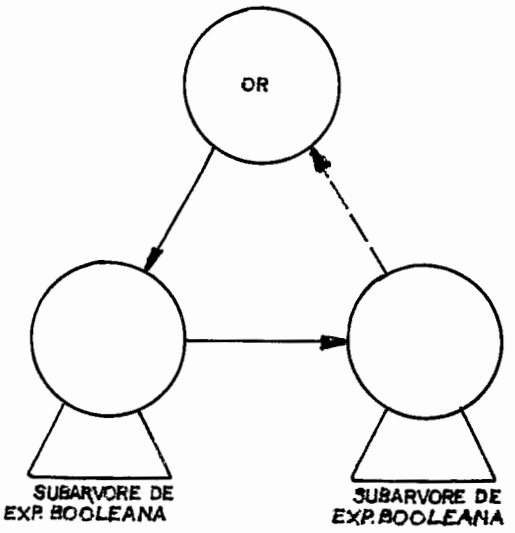


Figura VI.62

Os operandos deste operador devem ser expressões que resultem um tipo Booleano, tal qual os outros operadores lógicos vistos anteriormente.

Ações_Semânticas

Como trata-se de um operador lógico, seus operandos devem ser do tipo Booleano. Os tipos resultantes das expressões que estão como operandos, devem ser examinados para verificar a

validade desta operação.

Caso encontremos um dos tipos resultantes, ou ambos, não sendo booleanos, uma mensagem de erro deve ser emitida, mas o código do nó operador não necessitará ser modificado.

Operadores + e -

Iremos expor estes dois operadores juntos, pois suas subárvores têm formatos iguais e as ações semânticas a serem realizadas para estas, são semelhantes.

Poderemos ter três tipos de operações com esses operadores. Estes tipos são: Inteiro, Real e Set.

Os primeiros dois tipos, decorrem de operações entre expressões numéricas. O operador + quando aplicado a operandos do tipo set. equivale a operação União de conjuntos, e o operador - equivale a operação Complemento de conjuntos.

Na Figura (VI.63) podemos ver a subárvore gerada para uma subtração entre valores inteiros, e na Figura (VI.64) a subárvore gerada para uma subtração entre valores reais, ou quando pelo menos um deles é do tipo real.

Na Figura (VI.65) temos a subárvore para a operação soma entre valores inteiros, e na Figura (VI.66) a soma entre valores reais ou quando pelo menos um deles é do tipo real.

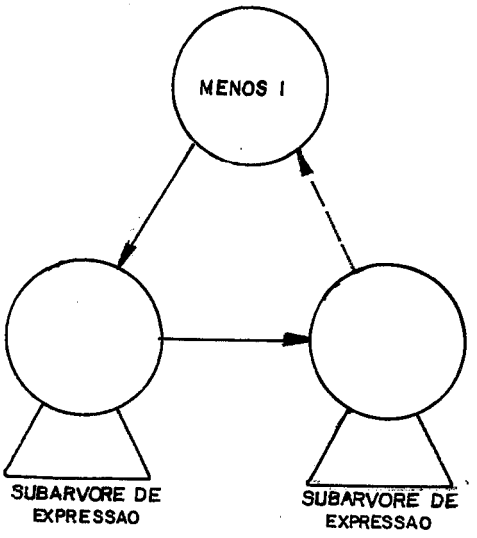


Figura VI.63

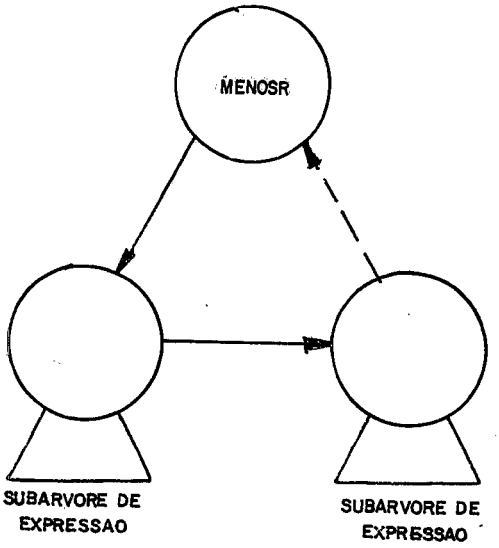


Figura VI.64

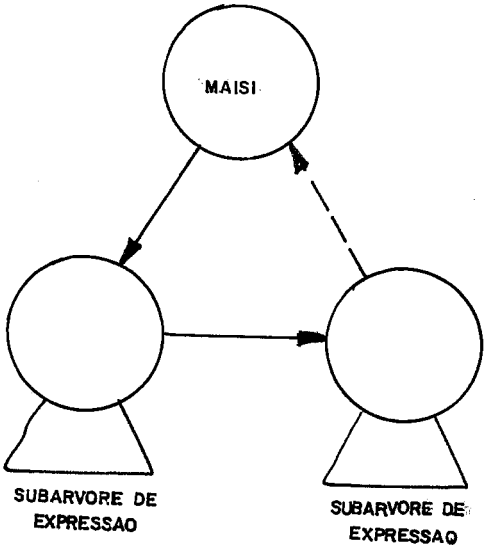


Figura VI.65

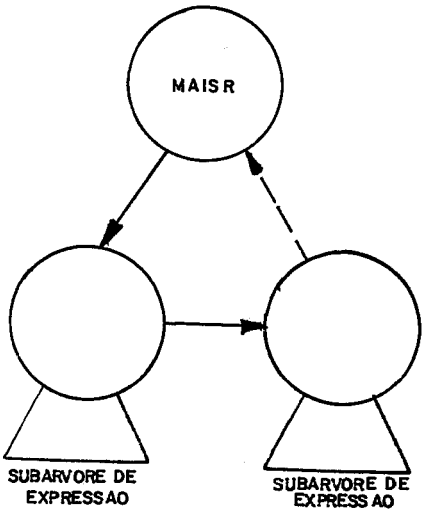


Figura VI.66

Para a operação de União de sets, temos a subárvore que está representada na Figura (VI.67).

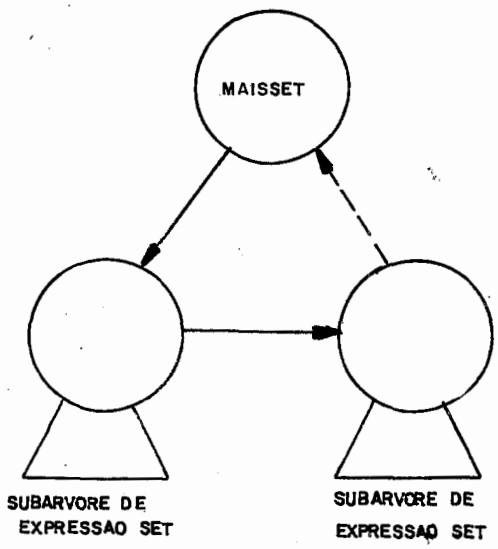


Figura VI.67

A subárvore que representa a operação de complemento entre sets pode ser vista na Figura (VI.68).

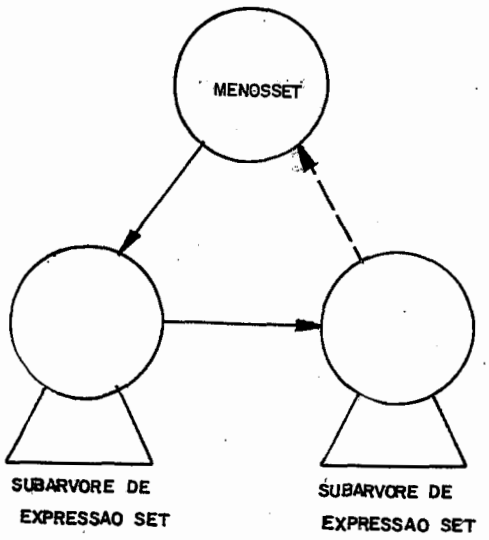


Figura VI.68

Note que por tratar-se de operações entre sets, os nós MENOSSET e MAISSET devem ter na componente PTR2 o tipo do set resultante, e em PTR1 o ponteiro para o nó ESCALAR ou DOISPONTOS que é o tipo-base do set resultante, caso em PTR2 tenhamos o código ESCALAR ou DOISPONTOS.

Ações Semânticas

Os operandos dessas subárvores devem ser examinados para verificar que tipo de operação está representada.

Quando o primeiro operando, da esquerda para direita, for numérico, o código do nó operador deverá ser modificado para MENOSR ou MENOSI, no caso da subtração e MAISR ou MAISI para a soma, conforme o tipo deste operando. O segundo operando deverá ser examinado, para que se verifique a validade da operação, ou seja, deve ser uma expressão numérica, e também para que se determine o tipo final. Se este segundo operando não corresponder a uma expressão numérica, uma mensagem de erro deve ser emitida, e o código do operador passará a ser SUBARVORE-ERRADA.

Caso o segundo operando seja numérico, e do tipo real, o código da operação deverá ser modificado, quando estiver registrado como inteiro.

Tratando-se o primeiro operando de uma expressão set, o mesmo procedimento realizado para o operador VEZESSET deverá ocorrer, resultando então, como d código do nó operador: MAISSET, MENOSSET ou SUBARVORE-ERRADA.

Caso o primeiro operando não satisfaça os dois primeiros tipos de operandos, vistos acima, uma mensagem de erro deverá ser enviada, e o código do nó operador deverá ser modificado para SUBARVORE-ERRADA.

PRODUÇÃO EXPRESSION

Operadores Relacionais

Os códigos que representam estes operadores são: IGUAL, DIFERENTE, MAIOR, MENOR, MAIORIGUAL, MENORIGUAL e IN.

Na Figura (VI.69) apresentamos as subárvores geradas para as operações com operadores relacionais. Note que todas as subárvores têm o mesmo formato, só diferindo no código da raiz, e em algumas delas nos tipos dos operandos dos quais se aplicam. Essas diferenças ficarão mais claras ao comentarmos as ações semânticas referentes a esta produção.

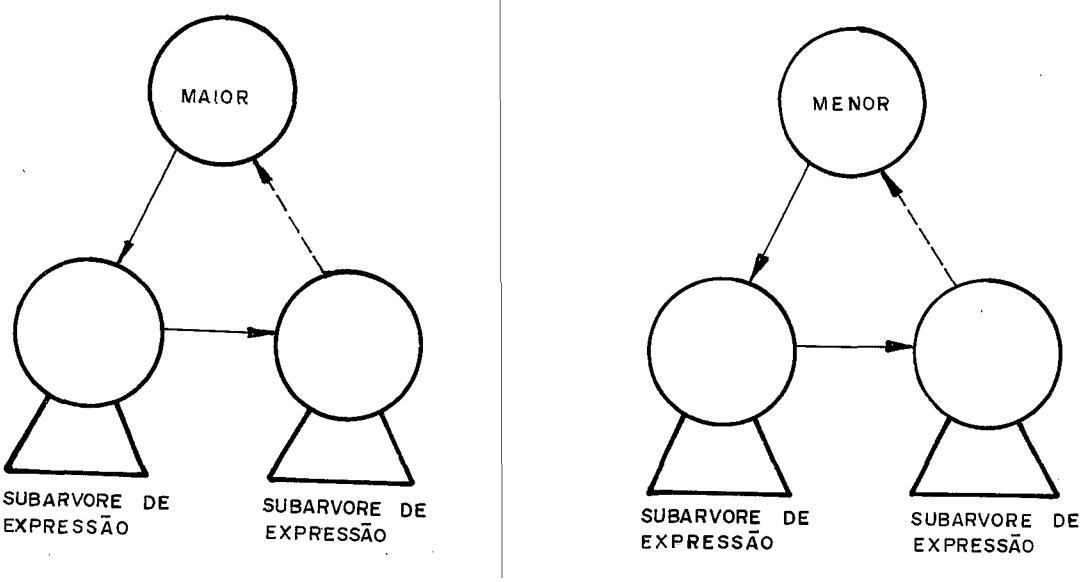
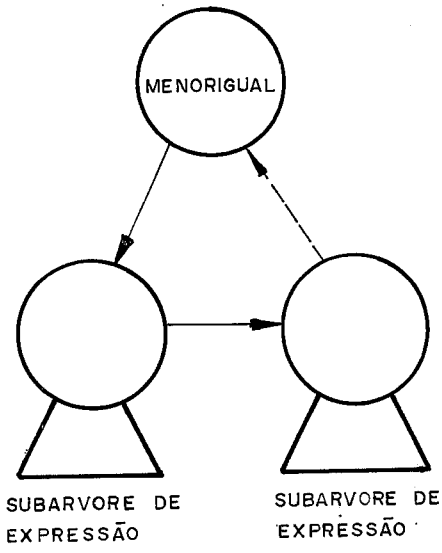
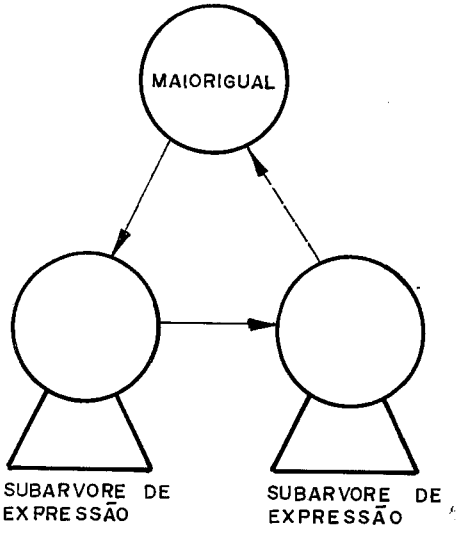
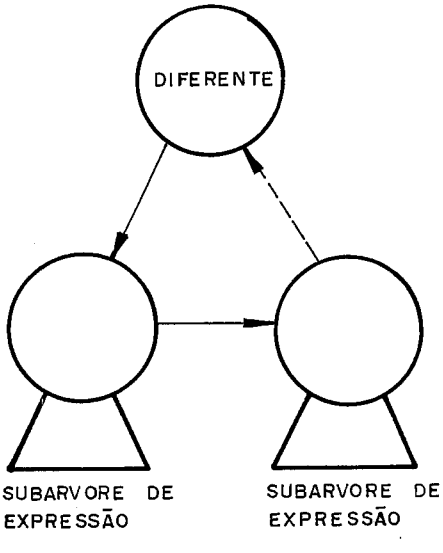


Figura VI.69







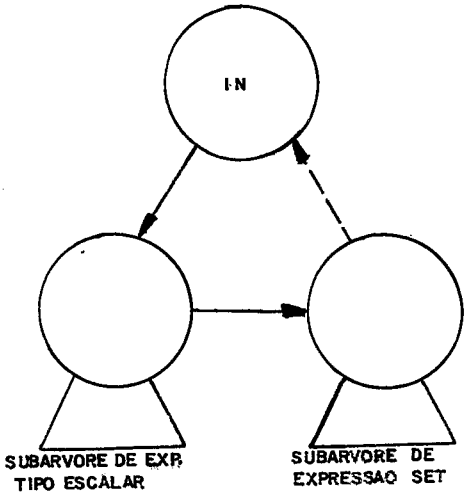


Figura VI.69 (continuação)

Ações Semânticas

Todos os operadores relacionais com exceção de IN devem operar com operandos do mesmo tipo escalar, sendo que os operadores: IGUAL, DIFERENTE, MENORIGUAL e MAIORIGUAL, podem também operar com operandos set, que sejam do mesmo tipo-base.

As ações para verificação de tipos iguais, já foram demonstradas por diversas vezes nos itens anteriores, achamos então desnecessário repetirmos estes procedimentos.

Quando o operador for o IN, o primeiro operando deverá ser examinado para verificar se este é de tipo escalar. Se esta condição não for satisfeita uma mensagem de erro será enviada. O segundo operando será examinado para verificar se é uma expressão do tipo set. Uma mensagem de erro será emitida caso isto não ocorra.

Ao encontrarmos os dois operandos válidos para o tipo do operador, deverá ser verificada a validade da operação. Isto se dará simplesmente verificando-se a igualdade entre as componentes PTR2 e PTR1, dos dois nós, no caso em que o primeiro operando é um nó cujas informações de tipo encontram-se nestas componentes. Quando o tipo do primeiro operando estiver representado no código de sua raiz (tipo standard), através de uma estrutura de um case, facilmente poderemos comparar a igualdade entre seu tipo e a do tipo do set.

No caso de operações entre sets, com os operadores:

IGUAL, DIFERENTE, MENORIGUAL e MAIORIGUAL, a compatibilidade dos tipos dos operandos set pode ser verificada utilizando-se das componentes PTR1 e PTR2.

PRODUÇÃO STATEMENT

Subárvore STATEMENT-PART

Na Figura (VI.70) podemos observar a subárvore gerada para o que corresponde ao conjunto de ações a serem realizadas no Bloco.

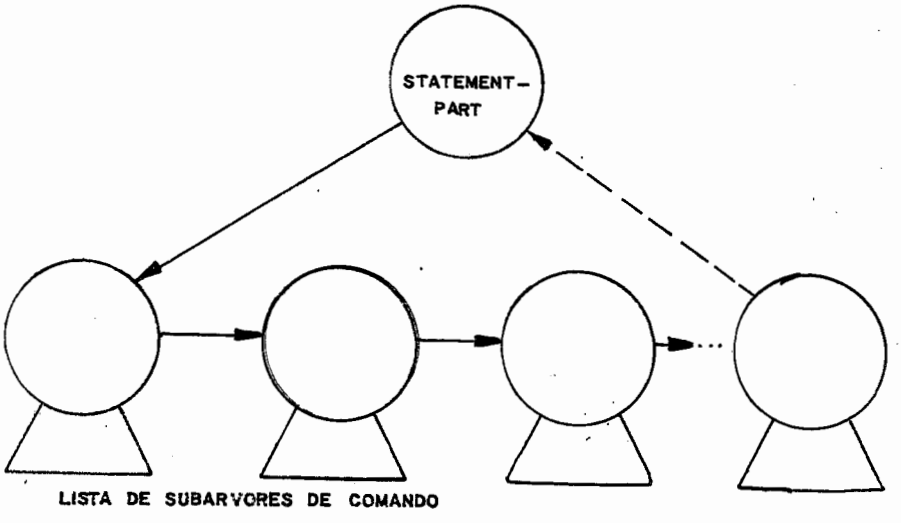


Figura VI.70

Note que mesmo se não existirem ações a serem realizadas, o nó STATEMENT-PART não deixará de figurar na subárvore de BLOCK. Na Figura(VI.71) podemos visualizar como seria esta subárvore, que representa um comando vazio.

Figura VI.71

Ações Semânticas

Nenhuma ação semântica será necessária para esta subárvore.

Descreveremos neste ítem as subárvores geradas para cada comando da linguagem PASCAL. Estas subárvores estarão na FIP como filhas do nó STATEMENT-PART, ou dos nós do tipo cabeça de comando, os quais veremos a seguir.

Comando CASE

A Figura (VI.72) representa a subárvore gerada para um comando Case.

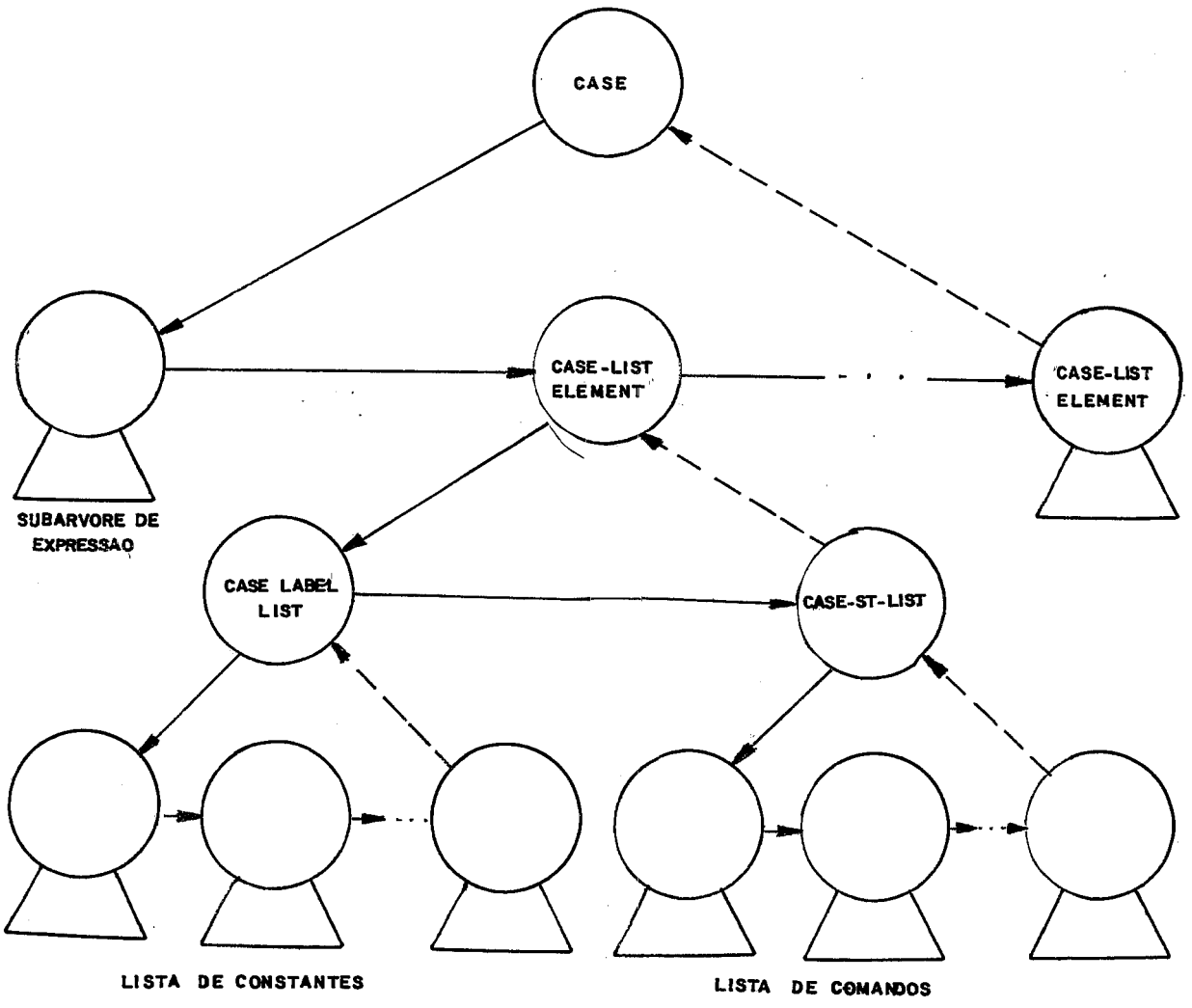


Figura VI.72

Ações Semânticas

A primeira ação a ser realizada é verificar se a expressão resulta um tipo escalar, com exceção do tipo real. Feito isto, deverão ser percorridas as litas de constantes de cada nó CASE-LABEL-LIST para que sejam efetuadas as seguintes verificações: a constante não pode ser de tipo real, a constante deve ser do mesmo tipo da expressão e não pode ser utilizada novamente como rótulo neste comando Case.

A verificação da reutilização de constantes dentro do comando Case, é feita seguindo-se o mesmo procedimento utilizado na verificação das componentes em uma estrutura de Record. Serão utilizadas dois pontos um que fixará a constante a ser analisada e um segundo ponteiro que visitará as constantes seguintes. O teste de unicidade será feito comparando-se as componentes PTNOME das duas constantes sendo analisadas.

Caso alguma das verificações não obtiver êxito, uma mensagem de erro será emitida notificando o fato ao programador.

Comando IF

Na Figura (VI.73) podemos ver a subárvore gerada para um comando IF, sem o símbolo ELSE, e na Figura (VI.74) a subárvore para comando IF definido com o ELSE.

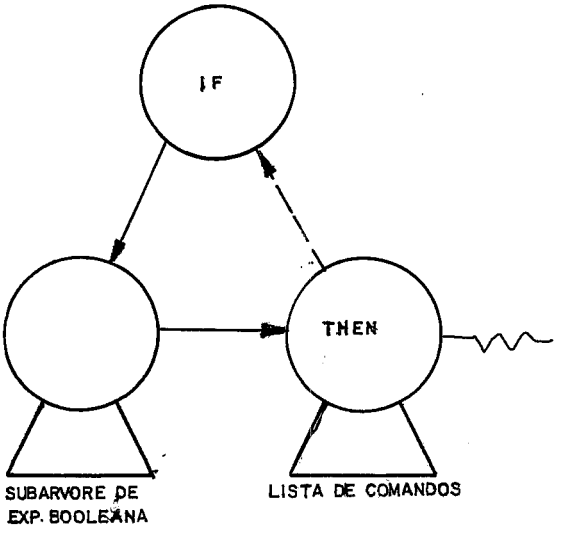


Figura VI.73

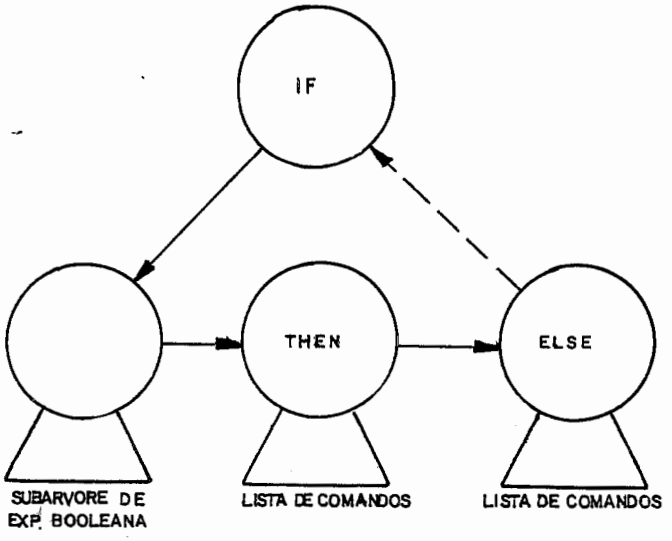


Figura VI.74

Note que a lista de comandos que está abaixo tanto do nó THEN quanto no nó ELSE, podem constituir-se apenas de um comando, quando este não for um comando composto.

Ações Semânticas

A expressão representada nas subárvores acima, deve resultar um tipo Booleano. Isto deverá ser verificado e emitida uma mensagem de erro caso esta condição não ocorra.

Comando_WHILE

Vejamos na Figura (VI.75) como se constitui a subárvore de um comando WHILE.

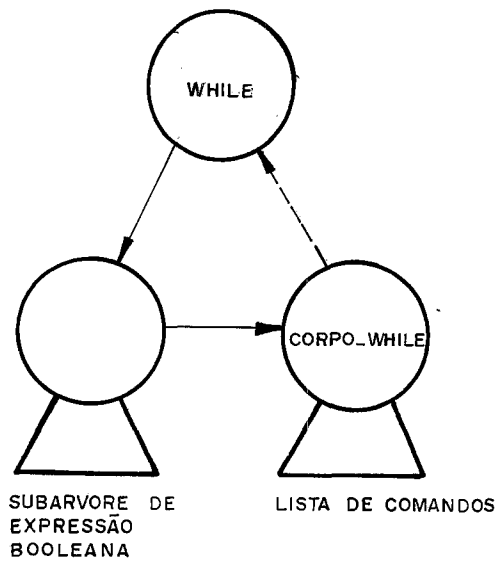


Figura VI.75

Note que a lista de comandos abaixo do nó CORPO-WHILE pode constituir-se de apenas um comando.

Ações_Semânticas

A expressão que controla o laço de um comando WHILE deve resultar um valor do tipo Booleano. Uma mensagem de erro deve ser emitida caso isto não se verifique.

Comando REPEAT

Na Figura (VI.76) podemos observar a subárvore gerada para o comando REPEAT.

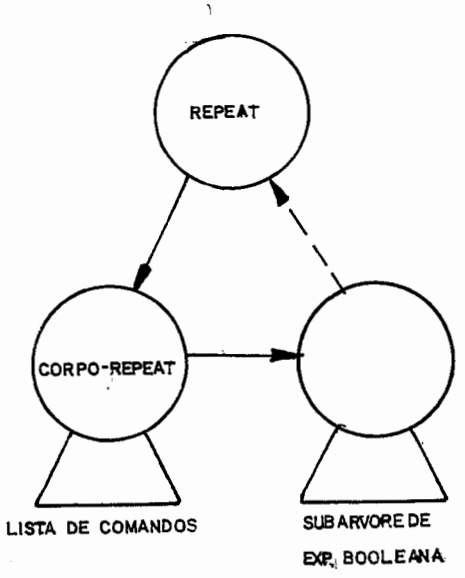


Figura VI.76

O mesmo comentário sobre a lista de comandos feito no WHILE, se faz necessário para o comando REPEAT.

Ações Semânticas

A expressão que controla a seqüência de repetições do comando REPEAT, deve resultar um valor do tipo Booleano. Uma mensagem de erro deve ser emitida caso isto não ocorra.

Comando FOR

Vejamos na Figura (VI.77) a subárvore gerada para o comando FOR, utilizando ação de incremento.

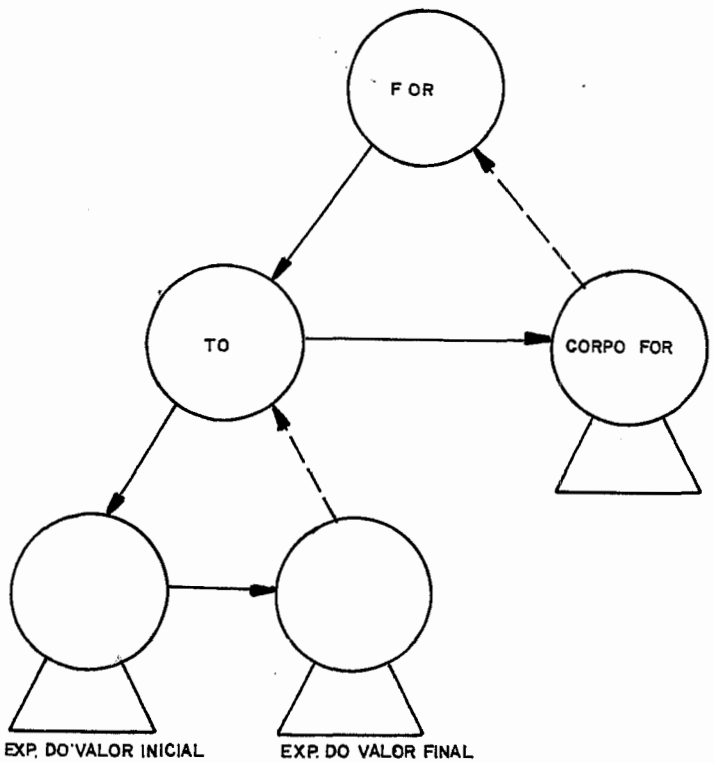


Figura VI.77

Caso o comando FOR utilizasse decremento, o código DOWNTO figuraria no nó cujo código é TO.

Note que as informações sobre a variável de controle do FOR estarão no nó TO (DOWNTO), ou seja, na componente PTR1 teremos ponteiro para o nó DEF-VAR que a define, em PTR2

o código de seu tipo e em PTNOME o ponteiro para o vetor VETNOMES, aonde encontra-se registrada a cadeia de caracteres que formam seu nome.

Ações Semânticas

As expressões que representam: o valor inicial, o valor final e a variável de controle, devem ser do mesmo tipo escalar, com exceção do tipo real. Se isto não se verificar, uma mensagem de erro deve ser emitida notificando o ocorrido.

Não foi previsto o teste para verificar se a variável de controle está sendo alterada dentro do FOR.

Comando Atribuição

Na Figura (VI.78) podemos observar a subárvore gerada para o comando de atribuição.

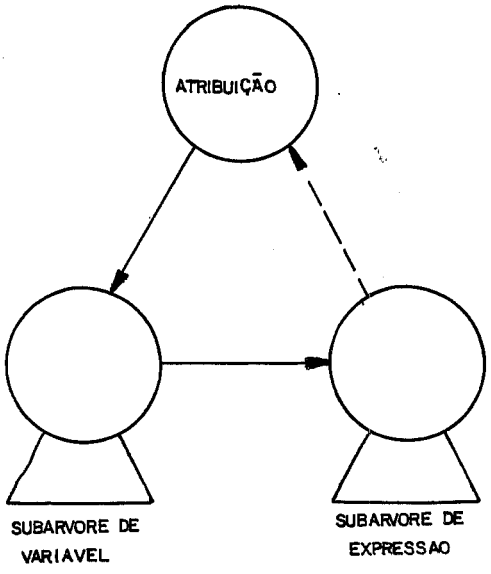


Figura VI.78

O tipo da variável deve ser comparado com o tipo resultante da subárvore de expressão; visa-se assim verificar se são semelhantes, de acordo com o critério mencionado na produção TYPE. Esta comparação não se aplica quando o tipo da variável for real e o da expressão for inteiro, pois este último será convertido para real.

Caso a semelhança não ocorra, será emitida uma mensagem de erro notificando o fato ocorrido.

Comando Rotulado

Na Figura (VI.79) podemos ver a subárvore gerada para um comando simples rotulado, e na Figura (VI.80) temos a subárvore gerada para um comando composto rotulado.

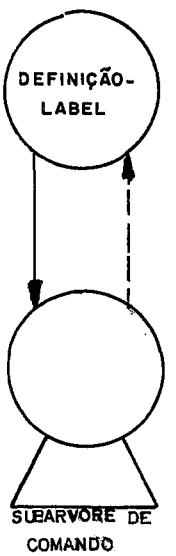


Figura VI.79

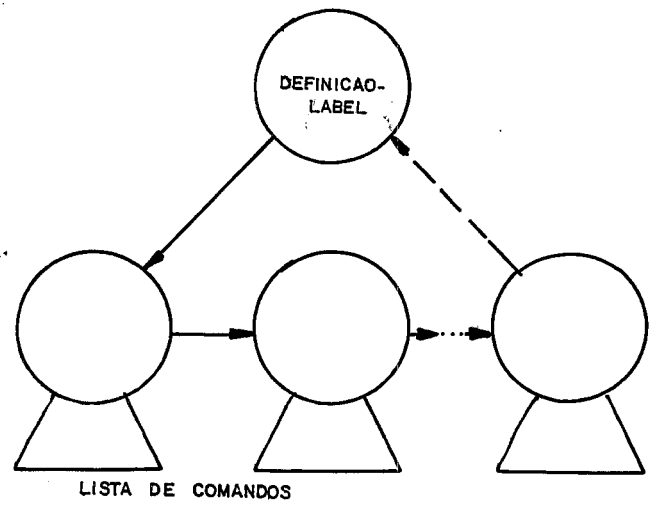


Figura VI.80

Ações Semânticas

As ações a serem realizadas são as seguintes: verifica-se se o rótulo já tinha anteriormente sido definido, isto é, se o nó de declaração do rótulo encontra-se com o código DEFINIDO, pois caso isso não ocorra, o código de UNDEF-LABEL será mudado para DEFINIDO, e na componente PTR1 deste nó será colocado um ponteiro para o nó DEFINIÇAO-LABEL. Caso o rótulo já tivesse sido definido, uma mensagem de erro deve ser emitida notificando o fato ocorrido.

Uma situação de erro que pode acontecer é de não ha

ver registro de ocorrência na Tabela de Símbolos, para o rótulo procurado, ou seja, o rótulo não ter sido declarado. Para que não haja duplicidade de mensagens de erro, o mesmo procedimento realizado para os identificadores, será aplicado neste caso. Será criado então um registro de ocorrência fictício para este rótulo, e que apontará para a variável NO-ERRO, que possui a estrutura de um nó, com o código SUBARVORE-ERRADA.

Comando GOTO

Na Figura (VI.81) podemos observar a subárvore gerada para o comando GOTO.

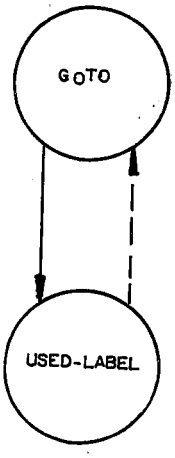
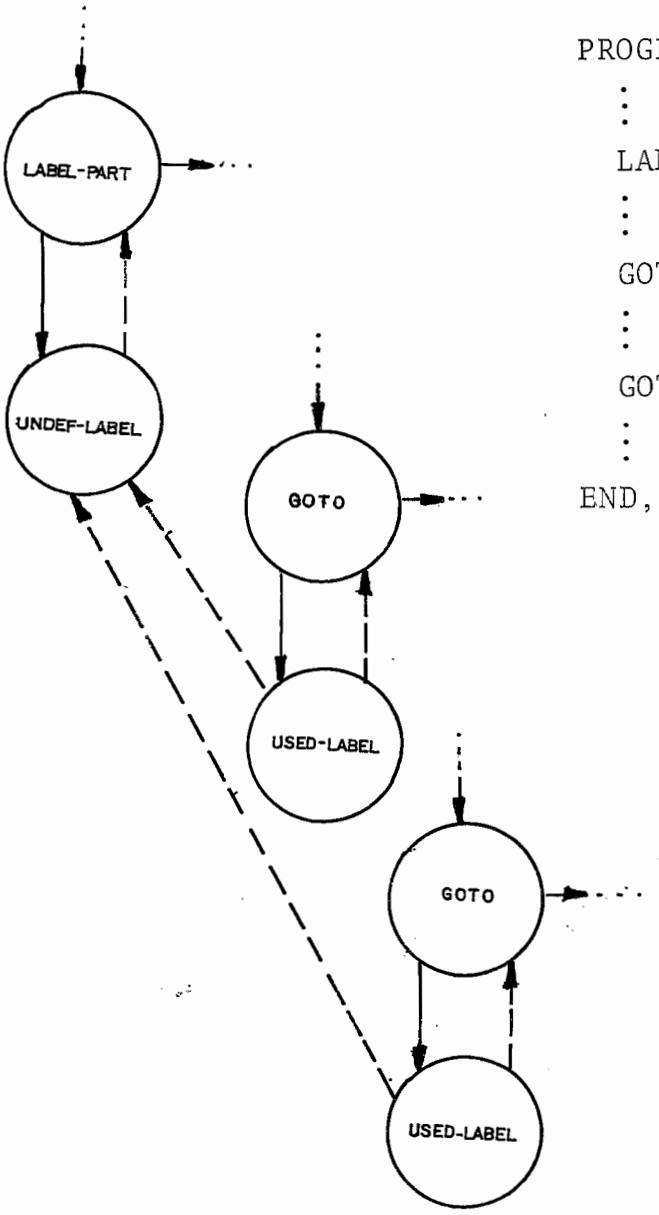


Figura VI.81

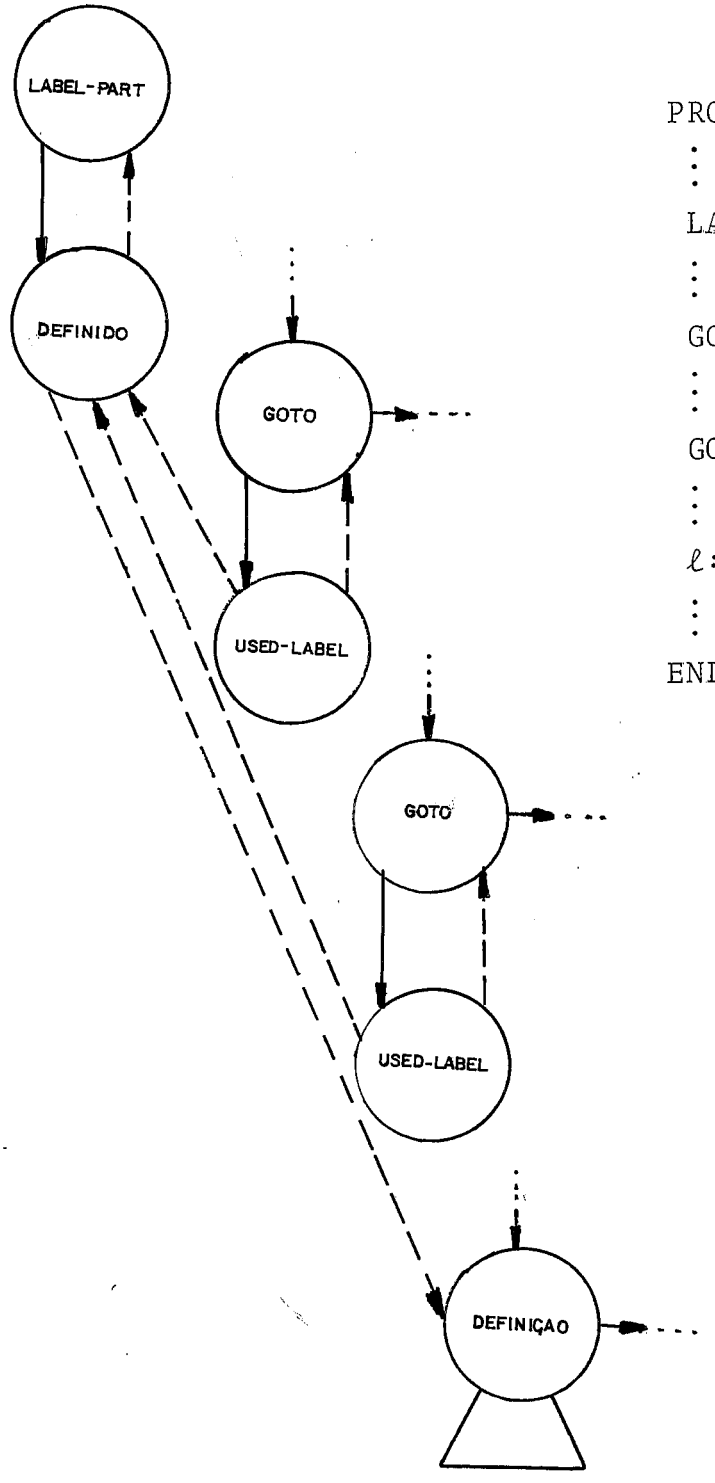
Vejamos na Figura (VI.82) como seria a ligação entre os nós USED-LABEL com o nó em que o label foi declarado, antes da ocorrência de sua definição.




```
PROGRAM exemplo;
  :
  LABEL ℓ;
  :
  GOTO ℓ;
  :
  GOTO ℓ;
  :
END,
```

Figura VI.82

Na Figura (VI.83) podemos observar a configuração das subárvores de declaração e utilização após o label ser definido.




```
PROGRAM exemplo;
 :
 LABEL ℓ;
 :
 GOTO  ℓ;
 :
 GOTO  ℓ;
 :
 ℓ:...
 :
END;
```

Figura VI.83

Ações Semânticas

Caso o label não tenha sido declarado, uma mensagem de erro deve ser emitida.

Não será indicado como erro um goto para fora ou para dentro de qualquer estrutura de laço, desde que a regra de escopo do label esteja sendo obedecida.

Chamada de Procedure

Vejamos na Figura (VI.84) o formato da subárvore para a chamada de uma Procedure, com parâmetros.

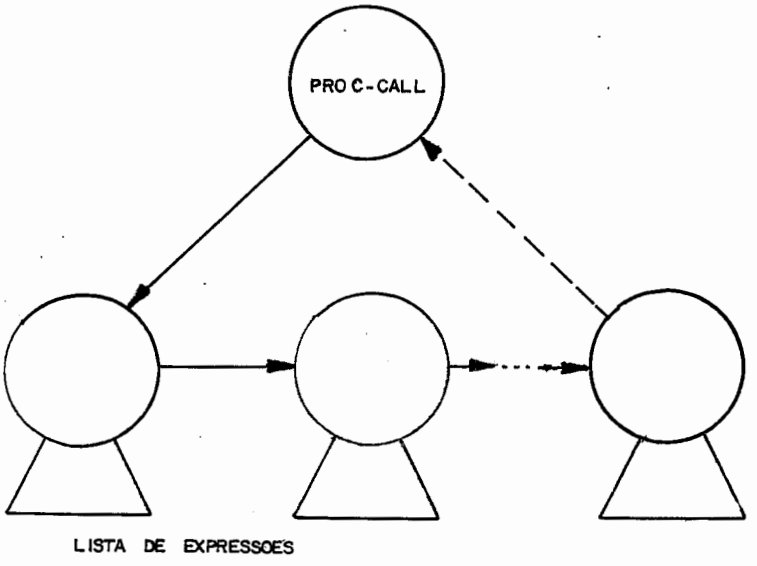


Figura VI.84

Na Figura (VI.85) está representada a subárvore constituída de apenas um nó, para chamada de uma procedure sem parâmetros.

Figura VI.85

Note que o nó PROC-CALL possui na componente PTR1 o endereço da subárvore de especificação ou declaração do subprograma, dependendo em que módulo foi definido.

Ações Semânticas

No caso da Procedure ser chamada com parâmetros, o procedimento a ser realizado, para verificação da sua subárvore é semelhante ao efetuado na produção FACTOR para chamada de rotina com parâmetros.

Quando a chamada for feita sem parâmetros, a primeira verificação a ser feita também será em relação a seu tipo, ou seja, se é standard ou não. Caso seja standard, a rotina TRATA-SUBPROG-STANDARD será chamada, caso contrário, a subárvore de especificação/declaração da Procedure deverá ser examinada para se verificar se a Procedure foi definida com parâmetros. Se for o caso uma mensagem de erros será emitida notificando o ocorrido.

A componente PTR1 do nó PROC-CALL apontará para o nó onde foi definida a Procedure.

Comando WITH

A análise semântica e geração da forma intermediária deste comando deverá ser feita em duas etapas. A primeira etapa trata da criação de todas as subárvores possíveis que podem ser construídas com a lista de estruturas que segue a palavra reservada WITH. A construção dessas subárvores é realizada de acordo com o relacionamento apresentado na Figura(VI.86). Ao detalharmos as ações semânticas realizadas nesta fase, o processo de construção será melhor esclarecido.

Na Figura(VI.87)podemos visualizar a subárvore gerada ao final desta primeira etapa.

A forma:

WITH r_1, r_2,...,r_n DO S;

é equivalente à:

```
WITH r1 DO
   WITH r2 DO
      ....
      WITH rn DO  S;
```

Figura VI.86

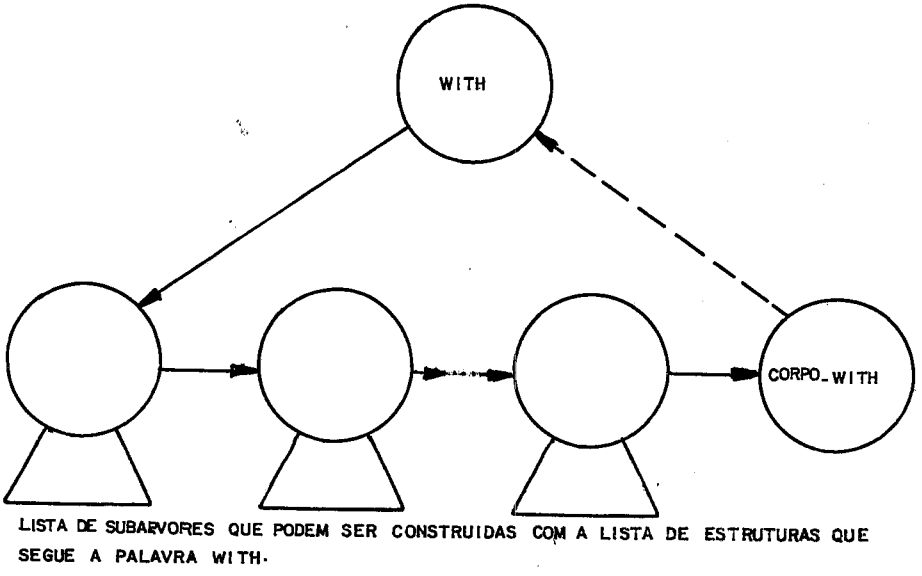


Figura VI.87

A ação principal realizada na segunda etapa constitui-se da construção da subárvore final para cada estrutura de variável, encontrada nos comandos do corpo do WITH, que sejam componentes de alguma das subárvores construídas na primeira etapa.

A subárvore resultante dessa segunda etapa está apresentada na Figura (VI.88):

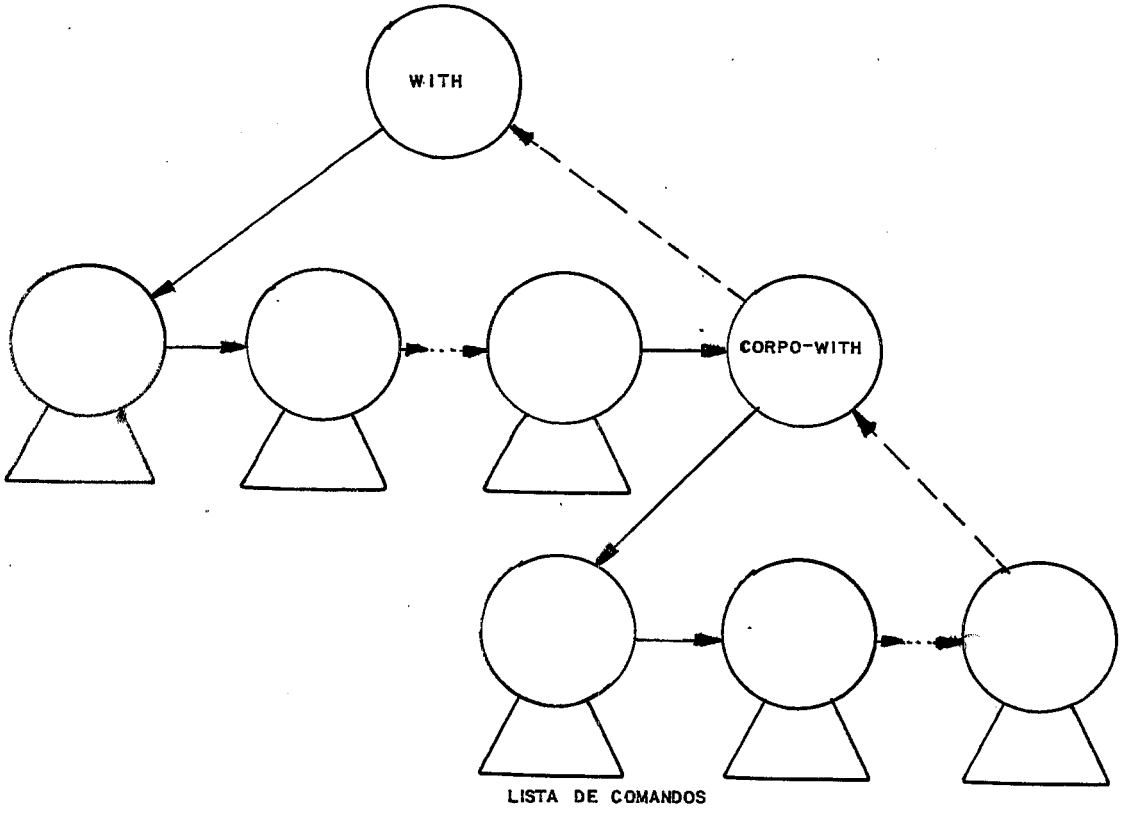


Figura VI.88

Ao final desta segunda etapa todas as componentes de Record referenciadas no corpo do WITH, deverão estar na sua forma real ou seja a subárvore que corresponde a utilização da componente já estará montada. Sendo assim, as subárvores construídas na primeira etapa, não serão mais necessárias, assim como os nós WITH e CORPO-WITH. Todos esses nós serão eliminados e as subárvores que correspondem aos comandos do WITH substituirão o nó WITH.

Ações Semânticas

Ao entrarmos no estado 151 que corresponde ao reconhecimento da palavra reservada WITH, uma variável booleana, ESTOU-NUM-WITH será feita TRUE. Esta variável será consultada cada vez que for realizada a ação semântica em algum comando, pois, caso se esteja num WITH o tratamento a uma estrutura de variável deverá seguir o procedimento exposto anteriormente.

Toda análise semântica realizada neste trabalho é feita no estado final da produção sendo analisada ou seja, "em cima" da árvore construída para esta. O comando WITH será uma exceção.

Vimos anteriormente que o processo de construção da árvore do WITH, divide-se em duas fases. Na primeira fase para cada estrutura da lista, será verificado se esta é componente de alguma subárvore já construída que esteja sob o nó WITH. Caso não o seja, esta estrutura tem que ser do tipo Record, pois se encontra na lista de um WITH. Uma mensagem de erro será emitida se a condição não for satisfeita, e a subárvore correspondente a esta estrutura será destruída.

Na segunda fase, a cada reconhecimento de um comando dentro de um WITH, serão montadas as subárvores de cada componente referenciada, tornando o comando usual.

Vale aqui mencionarmos que caso haja em um comando dentro do WITH, uma variável com o mesmo nome de uma componente, pertencente a uma estrutura da lista, o identificador será considerado como componente do Record.

Comando Composto

O comando composto será considerado por nós como uma seqüência de comandos, sem merecer uma raiz própria.

Quando o comando composto aparecer no corpo de um Bloco, seus comandos ficarão sob o nó STATEMENT-PART. Se o comando composto estiver dentro de um comando repetitivo, seus comandos aparecerão como filhos do nó tipo corpo, tais como: CORPO-WHILE, CORPO-REPEAT etc... .

No caso do comando CASE, os comandos que formam o conjunto de ações deste, ficarão sob o nó CASE-ST-LIST. Para o comando condicional IF, temos os nós THEN e ELSE.

Ações Semânticas

Todas as ações semânticas referentes a este comando, já foram descritas nesta seção.

CAPÍTULO VII

PACKAGES

A utilização desse conceito no nosso trabalho, como já foi dito no Capítulo I, seria a principal facilidade proposta por nós, no sentido de prover ferramentas que simplifiquem o processo de produção de software, através da compilação em separado, ao mesmo tempo que induz o programador a uma disciplina modular de projeto.

Para projetos grandes envolvendo vários programadores, a utilização de Packages permite que uma interface comum possa ser especificada, evitando assim várias interpretações de um mesmo objeto, durante a confecção dos programas do sistema.

Outro aspecto importante da utilização de Packages é a delineação da quantidade de informações que ficarão acessíveis aos usuários de uma Package. Este conceito é uma ferramenta importante para desenvolvimento de sistemas que requerem uma certa privacidade de determinadas informações.

Tais conceitos podem ser encontrados na especificação da linguagem ADA.

Nada de tão radical se propõe para este projeto, naturalmente, uma vez que se trata de uma primeira etapa.

VII.1. ESTRUTURA DE UMA PACKAGE

Como foi visto no capítulo anterior na exposição das árvores de um módulo de Package, temos duas representações para uma Package: a especificação da Package e a Package propriamente dita.

A primeira, a de especificação, corresponde a parte visível da Package, isto é, apresenta as declarações que se tornarão diretamente acessíveis ao módulo em que a Package for especificada na cláusula USING.

Na representação de uma Package, figuram as declarações visíveis e as informações que devem ficar escondidas.

Na Figura (VII.1) ilustramos a forma geral de uma Package, com declaração de um Bloco privado, e a especificação correspondente.

Package

-- Seqüência de declarações de constantes, tipos e variáveis que ficarão visíveis aos usuários da Package.

-- Seqüência de declarações de subprogramas declarados com a cláusula PRIVATE.

-- Seqüência de declarações de subprogramas cujos heading ficarão expostos na especificação da Package.

-- Bloco Privado

  -- Seqüência de declarações de labels, constantes, tipos e variáveis que ficarão invisíveis aos usuários da Package.

  -- Seqüência de declarações de subprogramas que ficarão invisíveis aos usuários da Package.

  -- Corpos de subprogramas cujo heading ficará exposto, mas se utilizam de objetos escondidos.

  -- Procedimentos para inicialização dos objetos da Package.

Package Specification

-- Seqüência de declarações de constantes, tipos e variáveis que formam a parte visível da Package.

-- Seqüência de heading dos subprogramas que serão utilizados pelos usuários da Package.

Figura VII.1

VII.2. MÓDULO PACKAGE

Na Figura (VII.1) podemos observar que a estrutura de uma Package é composta por duas partes distintas: a parte declarativa e a parte privada.

Veremos a seguir as características de cada uma.

Parte Declarativa

Como vimos no ítem anterior, esta parte determina as informações que ficarão visíveis aos usuários da Package. É constituída por declarações de: constantes, tipos, variáveis e subprogramas.

Os subprogramas que são declarados com a cláusula PRIVATE não ficarão expostos. Esses são procedimentos utilizados pelos subprogramas cujos "heading" ficarão visíveis.

Parte Privada

A parte privada de uma Package corresponde a estrutura de um Bloco precedida da palavra reservada PRIVATE, contendo declarações de objetos e procedimentos que não ficarão visíveis aos usuários da Package.

Este Bloco trata da inicialização dos objetos, tais como variáveis e arquivos, declarados tanto na parte declarativa quanto os desta parte. Note que qualquer estrutura

de variável declarada na package, existirá fisicamente, enquanto a representação da Package existir.

No caso de subprogramas que sejam visíveis, e portanto devendo constar da parte declarativa, mas que necessitem de alguma informação desta parte privada, deverão ser declarados como FORWARD na parte declarativa e terem o corpo representado na parte privada.

VII.3. PACKAGE SPECIFICATION

A especificação de uma package é um módulo independente, e por isso, pode ser compilada separadamente da Package que a origina; essa especificação também pode ser gerada a partir de sua Package de origem.

Todas as declarações de objetos, tais como: constantes, tipos e variáveis, que se encontram na parte declarativa da package, devem constar na especificação dessa. No que se refere a subprogramas, com exceção dos subprogramas declarados, com a cláusula PRIVATE, todos que figuram na parte declarativa, terão seus "heading" representados na especificação.

Independente do processo que fornece a especificação da Package, será gerada a árvore da Forma Intermediária. Essa árvore é quem funciona como interface entre o usuário e a package desejada, e portanto, é quem ficará sob o nó USING do módulo que a solicita.

VI.4. GERAÇÃO DA ESPECIFICAÇÃO DE UMA PACKAGE

A geração de uma especificação será feita percorrendo-se os nós da parte declarativa na árvore da Package.

Como vimos no Capítulo VI a árvore do módulo de especificação é semelhante a árvore do módulo Package, no que se refere as declarações de objetos tais como constantes, tipos e variáveis. Então para essas declarações, o processo de geração constitui-se de criar nós semelhantes, armazenando os caracteres que formam o nome dos identificadores, num vetor semelhante ao VETNOMES, criado para esta geração.

No caso dos subprogramas, na especificação constarão apenas os "heading" destes, excetuando-se o caso quando o subprograma for do tipo PRIVATE. A geração será feita de maneira semelhante a descrita anteriormente, apenas chamamos atenção para a estrutura da subárvore SUBPROG-PART que no módulo de especificação de uma package difere em alguns códigos de nós, da subárvore que consta no módulo de Package.

VII.5. ARMAZENAMENTO DE UMA PACKAGE

Na Figura (VII.2) podemos observar o fluxo lógico para gravação da Forma Intermediária do módulo de uma Package, na memória auxiliar.

Note-se que, caso já exista uma Package gravada com o mesmo nome, deverá ser emitida uma advertência indicando

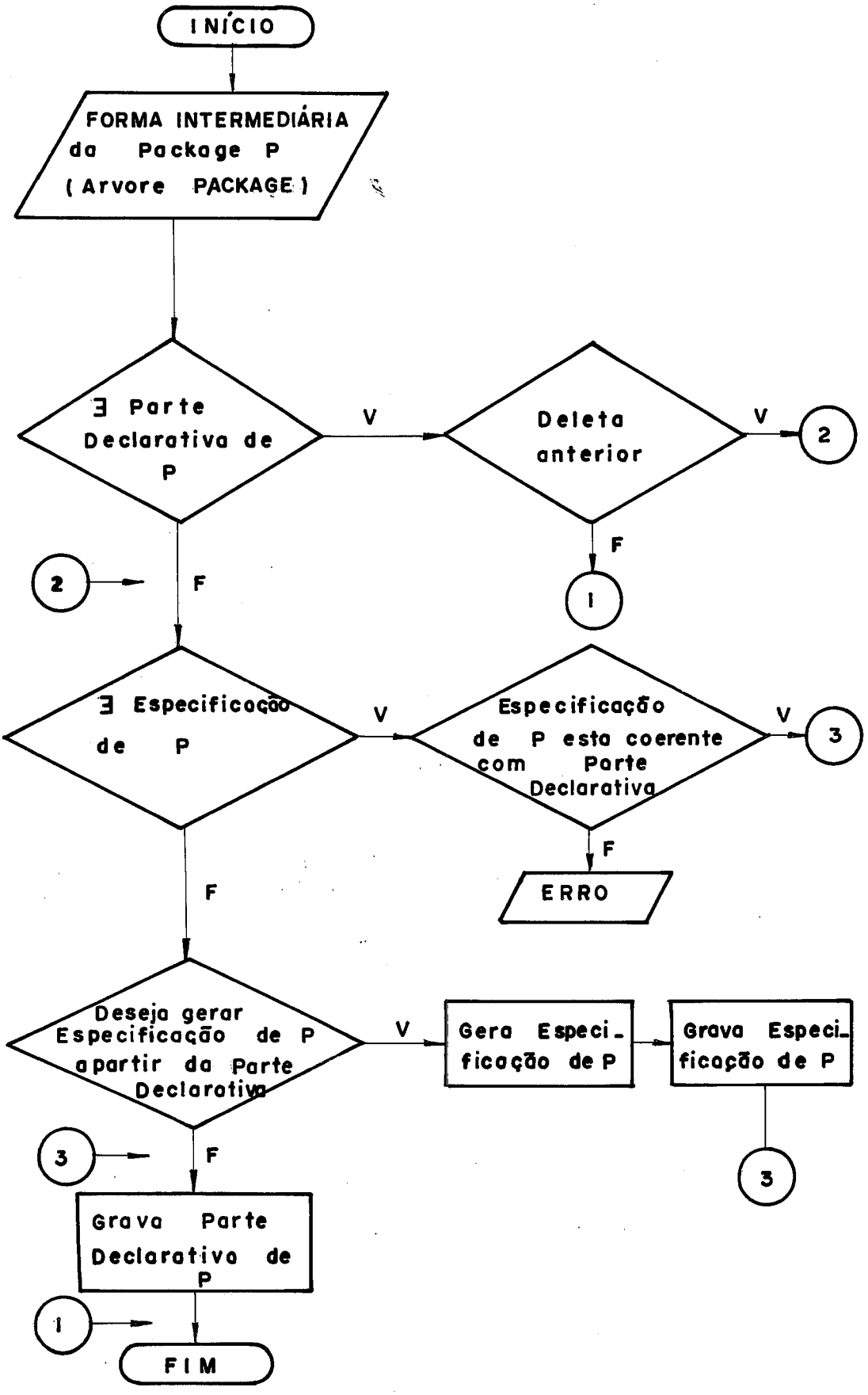

INÍCIO
FORMA INTERMEDIÁRIA da Package P (Arvore PACKAGE)
∃ Parte Declarativa de P
V
Deleta anterior
V
2
F
1
2
F
∃ Especificação de P
V
Especificação de P esta coerente com Parte Declarativa
V
3
F
ERRO
F
Deseja gerar Especificação de P apartir da Parte Declarativa
V
Gera Especificação de P
Grava Especificação de P
3
3
F
Grava Parte Declarativo de P
1
FIM

Figura VII.2

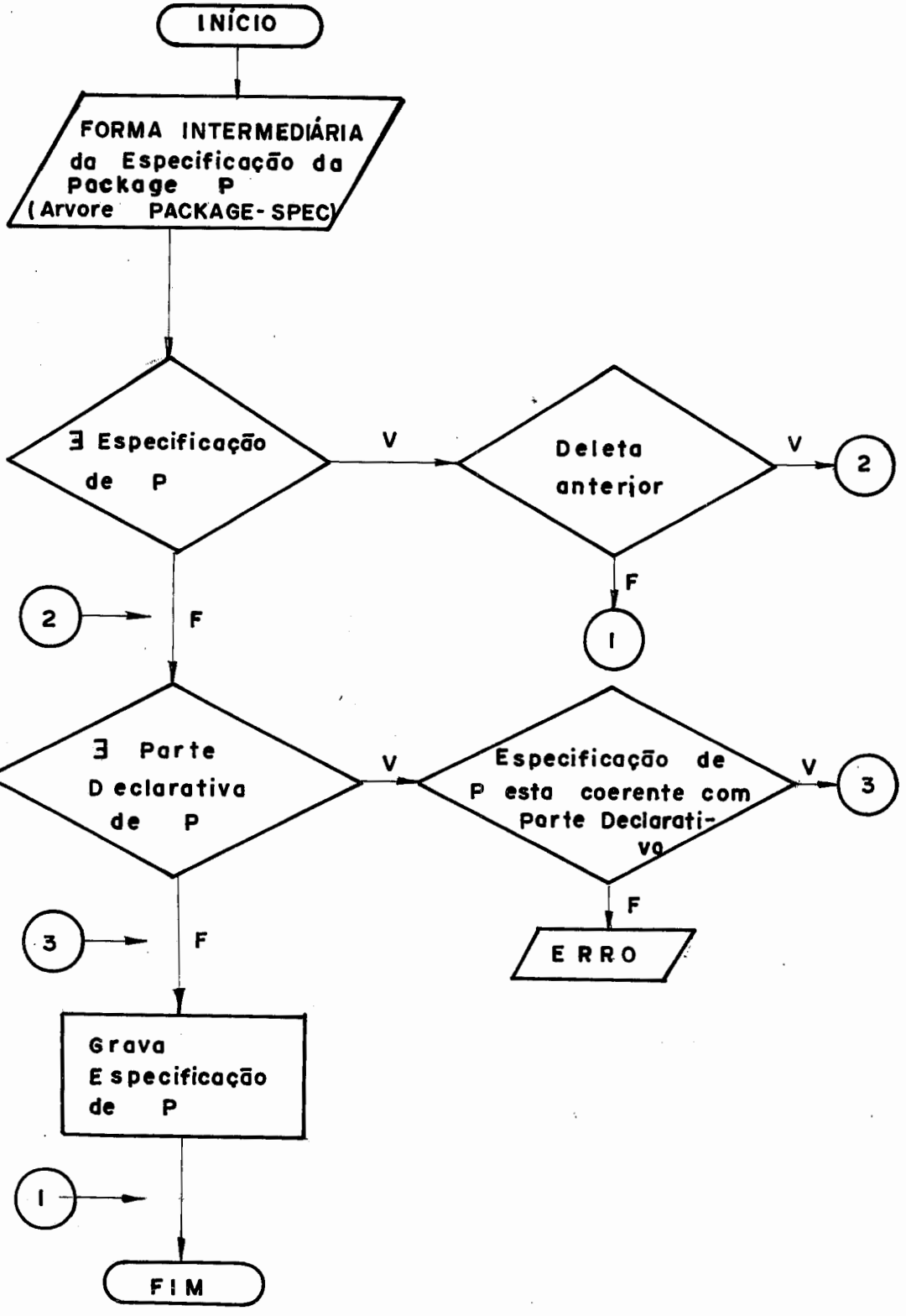

INÍCIO
FORMA INTERMEDIÁRIA da Especificação da Package P (Arvore PACKAGE-SPEC)
∃ Especificação de P
V
Deleta anterior
V
2
F
1
2
F
∃ Parte Declarativa de P
V
Especificação de P esta coerente com Parte Declarativa
V
3
F
ERRO
3
F
Grava Especificação de P
1
FIM

Figura VII.3

o fato ocorrido e uma opção para que a Package já existente seja deletada ou não.

Outra situação de erro passível de acontecer é quando já existir a especificação da package e, através de procedimentos de teste, chegar-se à conclusão que esta especificação não representa a verdadeira.

O fluxo para gravação da Forma Intermediária da especificação de uma Package, pode ser visto na Figura(VII.3). As situações de erro que ocorrem nesse procedimento têm o mesmo sentido lógico.

Podemos observar, no Autômato Finito correspondente à produção que define uma Package, que é permitida a utilização desta, dentro de um módulo Package.

A forma para usar uma Package dentro de outra, é semelhante a da utilização, dentro de um Program, ou seja, através da cláusula USING.

Todos os conceitos ditos para o módulo Program em relação a utilização de uma Package, vale para o módulo Package. Deve ficar bem claro que qualquer módulo que utilize uma Package, que por sua vez utilize outra Package, não poderá ter acesso a parte visível desta última. Isto só ocorrerá se o módulo tiver o nome dessa Package na sua cláusula USING.

VII.6. PACKAGE STANDARD

Como já foi mencionado, em diversos capítulos ao longo deste trabalho, os indicadores standard serão definidos dentro de uma Package de nome STANDARD.

Dois aspectos importantes fazem esta Package diferir das demais. O primeiro deles, é o que se refere ao nível em que se encontram seus identificadores, no nível 0, ou seja no nível mais baixo existente. Isso permitirá que os identificadores Standard sejam redeclarados, tanto dentro de uma Package, quanto num Program, como é previsto na especificação da linguagem.

O outro aspecto, é o fato de que as rotinas standard devem ter um tratamento diferente das definidas pelo programador, visto que as rotinas standard aceitam um número variável de parâmetros, e também não implicam num tipo fixo definido. Essas rotinas quando encontradas serão tratadas pela rotina TRATA-SUBPROG-STANDARD.

Na Figura(VII.4) podemos observar um trecho do que seria a especificação da Package Standard.

```
PACKAGE SPECIFICATION STANDARD;
  :
CONST
MAXINT = ...   ;                    (* Dependente da implementação *)
MAXREAL = ... ;                     (* Dependente da implementação *)
TYPE
BOOLEAN = (FALSE, TRUE);
INTEGER = - MAXINT .. MAXINT;
REAL =
CHAR=(A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,X,Z,W,Y,0,1,
     2,3,4,5,6,7,8,9);
TEXT = FILE OF CHAR;
VAR
INPUT,
OUTPUT : TEXT ;
EOF : BOOLEAN ;
  :
(* SUBPROGRAMAS STANDARD * )
  :
ENDPACKAGE  STANDARD ;
```

Figura VII.4

VII.7. CONSIDERAÇÕES FINAIS SOBRE PACKAGES

A possibilidade de se declarar entidades dentro de uma Package, tais como: subprogramas, tipos, constantes e variáveis, provê uma poderosa ferramenta para se realizar complexos sistemas, que necessitem de diversos programadores. Além disso, com a delineação das informações acessíveis aos usuários da Package, cria-se uma proteção contra o uso de certas informações, proteção essa, tão necessária em alguns sistemas.

Outra facilidade encontrada, é a de se substituir a parte privada de uma Package por outra, durante a confecção do sistema, desde que seja mantida sua parte declarativa. Isso permite que um sistema, depois de sua especificação concluída, possa ser construído por fases,até a sua conclusão.

Obviamente todos os aspectos descritos para a utilização de uma Package, não estão levando em consideração um ambiente de implementação para o projeto. Nosso intuito nesse capítulo, foi apenas de formalizar o procedimento para utilização de packages.

CAPÍTULO VIII

CONSIDERAÇÕES FINAIS

Apresentamos neste trabalho a especificação de um Analisador/Tradutor para uma Forma Intermediária PASCAL, que é parte de um projeto em andamento na COPPE/UFRJ. Tal projeto,um Ambiente de Programação PASCAL, inclui também um Interpretador/ Depurador, já concluído, e um Gerador/ Otimizador de Código, ainda em andamento.

A escolha da Linguagem PASCAL teve dois objetivos: permitir a abordagem de uma gama muito típica e variada de construções em linguagens e, ao mesmo tempo, tratar de uma linguagem aceita internacionalmente como a primeira linguagem moderna destinada ao ensino.

Abraçamos aqui uma tarefa um pouco árdua: cobrir praticamente todas as características dessa linguagem. Com isso, esta tese ficou bastante trabalhosa, devido à Análise Semântica de algumas construções.

O método de Análise Sintática escolhido,o RRP LL (1), é um método prático no sentido de nos haver poupado parte do esforço de preparação da gramática utilizada, uma vez que pode conter ε-produções e recursões à esquerda. Além disso contamos com o Gerador de Analisadores Sintáticos RRP LL (1) do projeto NHÃONHÃO |13|.

A Forma Intermediária PASCAL proposta, em formato de árvore e com atributos coletados, evitou a manipulação e o armazenamento de tabela de símbolos, facilitando assim a modularização do projeto.

Procuramos assim, apesar da diversidade de propostas com que deparávamos, optar pelo aspecto da clareza, em detrimento algumas vezes, da eficiência, uma vez que não haveria implementação. Preocupamo-nos principalmente em documentar para a comunidade interessada, uma moderna concepção de Forma Intermediária e o processo de geração desta, acrescida das facilidades de compilação em separado a qual, sem sombra de dúvida, é uma poderosa ferramenta para o desenvolvimento de um Ambiente de Programação.

Apesar da falta de incentivo que está sujeita à Universidade Brasileira, e principalmente os seus centros de pesquisa, poderemos um dia alcançar a concepção do verdadeiro Ambiente de Desenvolvimento, pois ainda contamos com a criatividade e perseverança de nossos pesquisadores.

O caminho é longo e árduo, mas esperamos ter contribuído com este trabalho, para tal objetivo.

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

|4| BAUER, F.L. e outros - Compiler Construction, an Advanced Course, Springer-Verlag, 1976.

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

|6| GRIES, D. - Compiler Construction for Digital Computers, Wiley International Edition, 1971.

|7| GROGONO, P. - Programming in PASCAL, Addison-Wesley,1978.

|8| JENSEN, K. e WIRTH, N. - PASCAL, User Manual and Report, Springer-Verlag, 1978.

|9| LEDGARD, H. - "ADA: An Introduction", Springer-Verlag, 1981.

|10| LUM, V.Y., LING, H. - Proc. ACM. Nat. Conf., 26, 1971.

|11| OLIVEIRA, A.C. - "Ambiente de Programação PASCAL: Forma Intermediária e Interpretador/Depurador Simbólico", Tese M.Sc., COPPE/UFRJ, 1983.

|12| SETZER, V.W., MELO, I.S.H. - A Construção de um Compilador, , Segunda Escola de Computação, Campinas, 1981.

|13| TELES, A.A.S., SIMONE, E. - Gerador de Analisadores Sintáticos RRPLL(1), , Anais do 8º SEMISH, 1981.

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

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

|16| WIRTH, N. - The Design of a Pascal Compiler, Software Practice and Experience, Volume 1, 309-333, 1971.

|17| RIPLEY, G.D. e DRUSEIKIS, F.C. - Towards a Compiler Error Recovery Effectiveness Rating, Technical Report, Computer Science Department, The University of Arizona, Tucson, Arizona, april, 1976.

APÊNDICE I

GRAMÁTICA PASCAL

```
COMPILATION =
   COMPILATION_LIST ;
COMPILATION_LIST =
   COMPILATION_UNIT
   | COMPILATION_LIST COMPILATION_UNIT ;
COMPILATION_UNIT =
   PROGRAM
   | PACKAGE
   | PACKAGE_SPECIFICATION ;
PROGRAM =
   PROGRAM_HEADING BLOCK IDENTIFIER_OPTION '.' ;
PACKAGE =
   PACKAGE_HEADING DECLARATIVE_PART PRIVATE_PART 'ENDPACKAGE'
      IDENTIFIER '.' ;
PACKAGE-SPECIFICATION =
   PACKAGE_SPECIFICATION_HEADING SPECIFICATION_PART
      'ENDPACKAGE' IDENTIFIER '.' ;
PROGRAM_HEADING =
   'PROGRAM' IDENTIFIER '(' IDENTIFIER_LIST ')' ','
      CONTEXT_SPECIFICATION ;
PACKAGE_HEADING =
   'PACKAGE' 'IDENTIFIER ';'
       CONTEXT_SPECIFICATION ;
PACKAGE_SPECIFICATION_HEADING =
   'PACKAGE' 'SPECIFICATION' IDENTIFIER ';'
       CONTEXT_SPECIFICATION ;
CONTEXT_SPECIFICATION =
   'USING' IDENTIFIER_LIST ';'
   | ;
IDENTIFIER_LIST =
   IDENTIFIER
   | IDENTIFIER_LIST ',' IDENTIFIER ;
IDENTIFIER_OPTION =
   ';' 'ENDPROGRAM' IDENTIFIER
   | ;
IDENTIFIER =
   'ID' ;
BLOCK =
   LABEL_DECLARATION_PART CONSTANT_DEFINITION_PART
   TYPE_DEFINITION_PART VARIABLE_DECLARATION_PART
      SUBPROGRAM_DECLARATION_PART STATEMENT_PART ;
DECLARATIVE_PART =
   CONSTANT_DEFINITION_PART TYPE_DEFINITION_PART
      VARIABLE_DECLARATION_PART SUBPROGRAM_DECLARATION_PART ;
PRIVATE_PART =
   'PRIVATE' BLOCK ';'
   | ;
SPECIFICATION_PART =
   CONSTANT_DEFINITION_PART TYPE_DEFINITION_PART
      VARIABLE_DECLARATION_PART SUBPROGRAM_SPECIFICATION_PART ;
LABEL_DECLARATION_PART =
   'LABEL' LABEL_LIST ;
LABEL_LIST =
   LABEL
```

```
    | LABEL_LIST ',' LABEL ;
LABEL =
    'USI' ;
CONSTANT_DEFINITION_PART =
    'CONST' CONSTANT_DEFINITION_LIST ';'
    | ;
CONSTANT_DEFINITION_LIST =
    CONSTANT_DEFINITION
    | CONSTANT_DEFINITION_LIST ';' CONSTANT_DEFINITION ;
CONSTANT_DEFINITION =
    IDENTIFIER '=' CONSTANT ;
CONSTANT =
    'USI'
    | 'USR'
    | IDENTIFIER
    | SIGN 'USI'
    | SIGN 'USR'
    | SIGN IDENTIFIER
    | 'STRING' ;
SIGN =
    '+'
    | '-' ;
TYPE_DEFINITION_PART =
    'TYPE' TYPE_DEFINITION_LIST ';'
    | ;
TYPE_DEFINITION_LIST =
    TYPE_DEFINITION
    | TYPE_DEFINITION_LIST ';' TYPE_DEFINITION ;
TYPE_DEFINITION =
    IDENTIFIER '=' TYPE ;
TYPE =
    SIMPLE_TYPE
    | STRUCTURED_TYPE
    | POINTER_TYPE ;
SIMPLE_TYPE =
    SCALAR_TYPE
    | SUBRANGE_TYPE
    | IDENTIFIER ;
SCALAR_TYPE =
    '(' IDENTIFIER_LIST ')' ;
SUBRANGE_TYPE =
    CONSTANT '..' CONSTANT ;
STRUCTURED_TYPE =
    UNPACKED_STRUCTURED_TYPE
    | 'PACKED' UNPACKED_STRUCTURED_TYPE ;
UNPACKED_STRUCTURED_TYPE =
    ARRAY_TYPE
    | RECORD_TYPE
    | SET_TYPE
    | FILE_TYPE ;
ARRAY_TYPE =
    'ARRAY' '[' SIMPLE_TYPE_LIST ']' 'OF' TYPE ;
SIMPLE_TYPE_LIST =
    SIMPLE_TYPE
```

```
    | SIMPLE_TYPE_LIST ',' SIMPLE_TYPE ;
RECORD_TYPE =
    'RECORD' FIELD_LIST 'END' ;
FIELD_LIST =
    FIXED_PART
    | FIXED_PART ';' VARIANT_PART
    | VARIANT_PART ;
FIXED_PART =
    RECORD_SECTION_LIST ;
RECORD_SECTION_LIST =
    RECORD_SECTION
    | RECORD_SECTION_LIST ';' RECORD_SECTION ;
RECORD_SECTION =
    IDENTIFIER_LIST '!' TYPE
    | ;
VARIANT_PART =
    'CASE' TAGFIELD_IDENTIFIER 'OF' VARIANT_LIST ;
VARIANT_LIST =
    VARIANT
    | VARIANT_LIST ';' VARIANT ;
TAGFIELD_IDENTIFIER =
    IDENTIFIER '!' IDENTIFIER
    | IDENTIFIER ;
VARIANT =
    CASE_LABEL_LIST '!' '(' FIELD_LIST ')'
    | ;
CASE_LABEL_LIST =
    CASE_LABEL
    | CASE_LABEL_LIST ',' CASE_LABEL ;
CASE_LABEL =
    CONSTANT ;
SET_TYPE =
    'SET' 'OF' SIMPLE_TYPE ;
FILE_TYPE =
    'FILE' 'OF' TYPE ;
POINTER_TYPE =
    '↑' IDENTIFIER ;
VARIABLE_DECLARATION_PART =
    'VAR' VARIABLE_DECLARATION_LIST ';'
    | ;
VARIABLE_DECLARATION_LIST =
    VARIABLE_DECLARATION
    | VARIABLE_DECLARATION_LIST ';' VARIABLE_DECLARATION ;
VARIABLE_DECLARATION =
    IDENTIFIER_LIST '!' TYPE ;
SUBPROGRAM_DECLARATION_PART =
    SUBPROGRAM_DECLARATION_PART PROCEDURE_DECLARATION ';'
    | SUBPROGRAM_DECLARATION_PART FUNCTION_DECLARATION
    | ;
SUBPROGRAM_SPECIFICATION_PART =
    SUBPROGRAM_SPECIFICATION_PART PROCEDURE_HEADING
    | SUBPROGRAM_SPECIFICATION_PART FUNCTION_HEADING
    | ;
PROCEDURE_DECLARATION =
```

```
   PROCEDURE_HEADING PRIVATE_CLAUSE BLOCK_OR_FORWARD ;
PROCEDURE_HEADING =
   'PROCEDURE' IDENTIFIER ';'
   | 'PROCEDURE' IDENTIFIER '(' FORMAL_PARAMETER_SECTION_LIST
     ')' ';' ;
FORMAL_PARAMETER_SECTION_LIST =
   FORMAL_PARAMETER_SECTION
   | FORMAL_PARAMETER_SECTION_LIST ';'
     FORMAL_PARAMETER_SECTION ;
FORMAL_PARAMETER_SECTION =
   PARAMETER_GROUP
   | 'VAR' PARAMETER_GROUP
   | 'FUNCTION' PARAMETER_GROUP
   | 'PROCEDURE' IDENTIFIER_LIST ;
PARAMETER_GROUP =
   IDENTIFIER_LIST ':' IDENTIFIER ;
FUNCTION_DECLARATION =
   FUNCTION_HEADING PRIVATE_CLAUSE BLOCK_OR_FORWARD ;
FUNCTION_HEADING =
   'FUNCTION' IDENTIFIER ':' IDENTIFIER ';'
   | 'FUNCTION' IDENTIFIER '(' FORMAL_PARAMETER_SECTION_LIST ')'
     ':' IDENTIFIER ';' ;
STATEMENT_PART =
   COMPOUND=STATEMENT ;
STATEMENT =
   UNLABELLED_STATEMENT
   | LABEL ':' UNLABELLED_STATEMENT ;
UNLABELLED_STATEMENT =
   SIMPLE_STATEMENT
   | STRUCTURED_STATEMET ;
RESTRICTED_STATEMENT =
   SIMPLE_STATEMENT
   | RESTRICTED_STRUCTURED_STATEMENT ;
SIMPLE_STATEMENT =
   ASSIGNMENT_STATEMENT
   | PROCEDURE_STATEMENT
   | GOTO_STATEMENT
   | EMPTY_STATEMET ;
ASSIGNMENT_STATEMENT =
   VARIABLE ':=' EXPRESSION ;
VARIABLE =
   ENTIRE_VARIABLE
   | COMPONENT_OR_REFERENCED_VARIABLE ;
ENTIRE_VARIABLE =
   IDENTIFIER ;
COMPONENT_OR_REFERENCED_VARIABLE =
   INDEXED_VARIABLE
   | FIELD_DESIGNATOR
   | FILE_BUFFER_OR_REFERENCED_VARIABLE ;
INDEXED_VARIABLE =
   VARIABLE '[' EXPRESSION_LIST ']' ;
EXPRESSION_LIST =
   EXPRESSION
   | EXPRESSION_LIST ',' EXPRESSION ;
```

```
FIELD_DESIGNATOR =
   VARIABLE '.' IDENTIFIER ;
FILE_BUFFER_OR_REFERENCED_VARIABLE =
   VARIABLE '^' ;
EXPRESSION =
   SIMPLE_EXPRESSION
   | SIMPLE_EXPRESSION RELATIONAL_OPERATOR SIMPLE_EXPRESSION ;
RELATIONAL_OPERATOR =
   '='
   | '<>'
   | '<'
   | '<='
   | '>='
   | '>'
   | 'IN' ;
SIMPLE_EXPRESSION =
   TERM
   | SIGN TERM
   | SIMPLE_EXPRESSION ADDING_OPERATOR TERM ;
ADDING_OPERATOR =
   '+'
   | '-'
   | 'OR' ;
TERM =
   FACTOR
   | TERM MULTIPLYING_OPERATOR FACTOR ;
MULTIPLYING_OPERATOR =
   '*'
   | '/'
   | 'DIV'
   | 'MOD'
   | 'AND' ;
FACTOR =
   VARIABLE
   | UNSIGNED_CONSTANT
   | '(' EXPRESSION ')'
   | FUNCTION_DESIGNATOR
   | SET
   | 'NOT' FACTOR ;
UNSIGNED_CONSTANT =
   UNSIGNED_NUMBER
   | 'STRING'
   | 'NIL' ;
UNSIGNED_NUMBER =
   'USI'
   | 'USR' ;
FUNCTION_DESIGNATOR =
   IDENTIFIER '(' EXPRESSION_LIST ')' ;
SET =
   '[' ELEMENT_LIST ']'
   | '[' ']' ;
ELEMENT_LIST =
   ELEMENT
   | ELEMENT_LIST ',' ELEMENT ;
```

```
ELEMENT =
    EXPRESSION .. EXPRESSION
    | EXPRESSION ;
PROCEDURE_STATEMENT =
    IDENTIFIER
    | IDENTIFIER '(' WRITE_EXPRESSION_LIST ')' ;
GOTO_STATEMENT =
    'GOTO' LABEL ;
EMPTY_STATEMENT =
      ;
STRUCTURED_STATEMENT =
    COMPOUND_STATEMENT
    | CONDITIONAL_STATEMENT
    | REPETITIVE_STATEMENT
    | WITH_STATEMENT ;
RESTRICTED_STRUCTURED_STATEMENT =
    COMPOUND_STATEMENT
    | RESTRICTED_CONDITIONAL_STATEMENT
    | RESTRICTED_REPETITIVE_STATEMENT
    | RESTRICTED_WITH_STATEMENT ;
COMPOUND_STATEMENT =
    'BEGIN' STATEMENT_LIST 'END' ;
STATEMENT_LIST =
    STATEMENT
    | STATEMENT_LIST ';' STATEMENT ;
CONDITIONAL_STATEMENT =
    IF_STATEMENT
    | CASE_STATEMENT ;
RESTRICTED_CONDITIONAL_STATEMENT =
    RESTRICTED_IF_STATEMENT
    | CASE_STATEMENT ;
IF_STATEMENT =
    'IF' EXPRESSION 'THEN' STATEMENT
    | 'IF' EXPRESSION 'THEN' RESTRICTED_STATEMENT 'ELSE'
      STATEMENT ;
RESTRICTED_IF_STATEMENT =
    'IF' EXPRESSION THEN RESTRICTED_STATEMENT 'ELSE'
      RESTRICTED_STATEMENT ;
CASE_STATEMENT =
    'CASE'EXPRESSION 'OF' CASE_LIST_ELEMENT_LIST 'END' ;
CASE_LIST_ELEMENT_LIST =
    CASE_LIST_ELEMENT
    | CASE_LIST_ELEMENT_LIST ';' CASE_LIST_ELEMENT ;
CASE_LIST_ELEMENT =
    CASE_LABEL_LIST '!' STATEMENT
    | ;
REPETITIVE_STATEMENT =
    WHILE_STATEMENT
    | REPEAT_STATEMENT
    | FOR_STATEMENT ;
RESTRICTED_REPETITIVE_STATEMENT =
    RESTRICTED_WHILE_STATEMENT
    | REPEAT_STATEMENT
    | RESTRICTED_FOR_STATEMENT ;
```

```
WHILE_STATEMENT =
   'WHILE' EXPRESSION 'DO' STATEMENT ;
RESTRICTED_WHILE_STATEMENT =
   'WHILE' EXPRESSION 'DO' RESTRICTED_STATEMENT ;
REPEAT_STATEMENT =
   'REPEAT' STATEMENT_LIST 'UNTIL' EXPRESSION ;
FOR_STATEMENT =
   'FOR' IDENTIFIER '|=' FOR_LIST 'DO' STATEMENT ;
RESTRICTED_FOR_STATEMENT =
   'FOR' IDENTIFIER '|=' FOR_LIST 'DO' RESTRICTED_STATEMENT ;
FOR_LIST =
   EXPRESSION 'TO' EXPRESSION
   | EXPRESSION 'DOWNTO' EXPRESSION ;
WHITH_STATEMENT =
   'WHITH'VARIABLE_LIST 'DO' STATEMENT ;
RESTRICTED_WHITH_STATEMENT =
   'WHITH' VARIABLE_LIST 'DO' RESTRICTED_STATEMENT ;
VARIABLE_LIST =
   VARIABLE
   | VARIABLE_LIST ',' VARIABLE ;
PRIVATE_CLAUSE =
   'PRIVATE'
   | ;
BLOCK_OR_FORWARD =
   BLOCK
   | 'FORWARD' ;
WRITE_EXPRESSION_LIST =
   WRITE_EXPRESSION
   | WRITE_EXPRESSION_LIST ';' WRITE_EXPRESSION ;
WRITE_EXPRESSION =
   EXPRESSION
   | EXPRESSION '|' EXPRESSION
   | EXPRESSION '|' EXPRESSION '|' EXPRESSION ;
```

APÊNDICE II

GRAMÁTICA PASCAL RRP

```
COMPILATION =
   ( ( ('PROGRAM' IDENT '(' (IDENT & ',') ')' ';'
       ('USING'(IDENT & ',' )';' )? BLOCK ( ';' ' ENDPROGRAM'
         IDENT)? ) /
       ( ('PACKAGE' (( IDENT ';' ('USING' (IDENT & ',') ';')?
           DECLARATIVE_PART ('PRIVATE' BLOCK ';' )? ) /
         ('SPECIFICATION' IDENT ';'
         ('USING' (IDENT & ',') ';' )? SPECIFICATION_PART) ) )
         ('ENDPACKAGE' IDENT) ) '.' )
;
DECLARATIVE_PART =
   ( DECL_OBJETOS DECL_PROC_FUNC )                                    ;
SPECIFICATION_PART =
   DECL_OBJETOS
     ( ( ('PROCEDURE' IDENT ('('FORMAL_PARAM & ';' ')' )? )
         ('FUNCTION' IDENT ('('FORMAL_PARAM & ';' ')' )?
          ':' IDENT ) ) ';' )*                                        ;
BLOCK =
   ('LABEL' 'USI' & ',' ';' )?
   DECL_OBJETOS
   DECL_PROC-FUNC
   ('BEGIN' STATEMENT & ';' 'END')                                    ;
DECL_OBJETOS =
   ('CONST' (IDENT '=' CONSTANT ';')+ )?
   ('TYPE' (IDENT '=' TYPE ';')+ )?
   ('VAR' ( (IDENT & ',' ) ':' TYPE ';')+ )?                          ;
DECL-PROC-FUNC =
   ( ( ('PROCEDURE' IDENT ( '('FORMAL_PARAM & ';' ')' )? )           /
       ('FUNCTION' IDENT ( '('FORMAL_PARAM & ';' ')' )?
        ':' IDENT ) )
   ( ( (';' 'PRIVATE')? ';' BLOCK ';') / (';' 'FORWARD' ',' ) )
     )*                                                               ;
CONSTANT =
   ( ('+'/'-')? (IDENT/'USI'/'USR')) / 'STRING'                       ;
TYPE =
   SIMPLETYPE / ('@' IDENT ) /
   ('PACKED')? ( ('ARRAY' '[' SIMPLETYPE & ',' ']' 'OF' TYPE)        /
                 ('FILE' 'OF' TYPE )                                  /
                 ('SET' 'OF' SIMPLETYPE )                             /
                 ('RECORD' FIEL_LIST 'END' ) )                        ;
SIMPLETYPE =
   IDENT / ('(' IDENT & ',' ')' )                                     /
             ( CONSTANT '..' CONSTANT )                               ;
FIELDLIST =
   ( ( ( (IDENT & ',' ) ':' TYPE)? ) & ';' )                          /
   ( ( (IDENT & ',' ) ';' TYPE)?  ';' )*
       ('CASE' (IDENT ':')? IDENT 'OF'
   ( ( ( (CONSTANT & ',' ) ':' '('FIELDLIST')' )? )& ':') )           ;
FORMAL_PARAM =
   ( ('VAR' / 'FUNCTION')? (IDENT & ',') ':' IDENT )                  /
   ( ('PROCEDURE') ( IDENT & ',' ) )                                  ;
STATEMENT =
   ('USI' ':')?
   ( ( VARIABLE ':=' EXPRESSION )                                     /
```

```
   ( IDENT ( '(' (EXPRESSION (':' EXPRESSION (':'
                    EXPRESSION )? )? )& ',' ')' )? )                    /
   ( 'GOTO' 'USI' )                                                     /
   ( 'BEGIN' STATEMENT & ';' 'END' )                                    /
   ( 'IF' EXPRESSION 'THEN' STATEMENT ('ELSE' STATEMENT)? )             /
   ( 'CASE' EXPRESSION 'OF' ( (CONSTANT & ',' ':'
      STATEMENT )? ) & ';' 'END' )                                      /
   ( 'WHILE' EXPRESSION 'DO' STATEMENT )                                /
   ( 'REPEAT' STATEMENT & ';' 'UNTIL' EXPRESSION )                      /
   ( 'FOR' IDENT ':=' EXPRESSION ('DOWNTO' / 'TO' )
                    EXPRESSION) ('DO' STATEMENT) )                      /
   ( 'WITH' VARIABLE & ',' 'DO' STATEMENT) )?                           ;
VARIABLE =
   IDENT ( '[' EXPRESSION & ',' ']' / '.' IDENT / '@')*                 ;
EXPRESSION =
   ( SIMPLE_EXPRESSION ( ( '=' / '<' / '>' / '<>' / '<=' /              /
                           '>=' / 'IN' )
     SIMPLE_EXPRESSION )? )                                             ;
SIMPLE_EXPRESSION =
   ('+' / '-' )? (TERM & ('+' / '-' / 'OR') )                           ;
TERM       =
   ( FACTOR & ('*' / '/' / 'DIV' / 'MOD' / 'AND' ) )                    ;
FACTOR     =
   VARIABLE / 'USI' / 'USR' / 'STRING' / 'NIL'                          /
      ( IDENT ( '(' EXPRESSION & ',' ')' ) )                            /
      ( '(' EXPRESSION ')' )                                            /
      ( 'NOT' FACTOR )                                                  /
      ( '[' ( ( EXPRESSION ( ',,' EXPRESSION )? ) &
                              ',' )? ']'                                ;
IDENT =
   'ID'                                                                 ;
```

APÊNDICE III

AUTÔMATOS FINITOS