UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas de São Carlos

**UM EDITOR GENÉRICO SENSÍVEL À SINTAXE
ARMAZENADA NUMA BASE DE DADOS**

Carlos Roberto Valêncio

Orientador:
Prof. Dr. Caetano Traina Júnior

Dissertação apresentada ao Instituto de Ciências Matemáticas de São Carlos, Universidade de São Paulo, como parte dos requisitos para a obtenção do Título de Mestre - Área: Ciências de Computação e Matemática Computacional.

USP - São Carlos
Estado de São Paulo - Brasil
Fevereiro de 1993

*Dedico este trabalho aos meus pais,
José e Dora, e a minha esposa e
filho, Soraia e Lucas, que sempre
estiveram ao meu lado.*

AGRADECIMENTOS

Ao Prof. Dr. Caetano Traina Jr., orientador e amigo, pela competência, dedicação e compreensão dispensada durante o desenvolvimento deste trabalho. Trabalhar com o Prof. Caetano só fez aumentar minha admiração à sua capacidade profissional e ao seu caráter.

Ao Prof. Dr. Eurípedes (IBILCE-UNESP) e Prof. Dr. Paulo Masiero (ICMSC-USP), que contribuíram para o meu ingresso no programa de mestrado do ICMSC-USP.

À Profa. Dra. Agma Juci M. Traina, pela simpatia que sempre nos tratou e pela ajuda na finalização deste trabalho.

Aos amigos do Grupo de Estudos de Base de Dados - ICMSC-USP: João Ferreira, Luiz Camolesi, Mauro Biajiz, Nivaldi Calônego, Osvaldo Takai, Pedro Pizzigatti e Rildo Ribeiro, pelo companheirismo e troca de conhecimento nas nossas discussões. Em especial ao João Eduardo, pelo total apoio na fase final deste trabalho.

Ao Marcos e Miriam, funcionários do Polo Computacional UNESP-Bauru, pelo esforço e dedicação na elaboração dos desenhos deste trabalho.

Ao Departamento de Computação UNESP-Bauru, que contribuiu para a realização deste trabalho.

Às funcionárias da biblioteca do ICMSC - USP - São Carlos, pela competência e dedicação no cumprimento de suas tarefas, o que contribuiu para o bom desenvolvimento deste trabalho.

Às secretárias, Bete e Laura, da secretaria de pós-graduação do ICMSC-USP, pela competência na execução de suas atividades e simpatia com que sempre nos atendeu.

Aos meus pais pelo constante incentivo e apoio.

À minha esposa Soraia, pelo abnegação e carinho oferecidos quando foi preciso para que este trabalho se concretizasse.

Ao meu filho Lucas, que há de perdoar-me pela ausência.

Ao CNPq, pelo apoio financeiro dado para o desenvolvimento deste trabalho.

ÍNDICE

Lista de Figuras	v
Resumo	vi
Abstract	vii
1. Introdução	1
1.1 - Caracterização do Problema	1
1.2 - Organização da Dissertação	3
2. Revisão Bibliográfica	5
2.1 - Considerações Iniciais	5
2.2 - Conceitos Utilizados pelos Ambientes Orientados para a Estrutura.	6
2.3 - MENTOR	7
2.4 - CORNELL PROGRAM SYNTHESIZER	12
2.5 - Projeto GANDALF	16
2.5.1 - Editores ALOE	17
2.5.2 - Ambientes GANDALF	20
2.5.2.1 - PROTÓTIPO GANDALF	21
2.5.2.2 - Projeto GNOME	25
2.5.2.3 - SMILE	25
2.6 - PECAN	25
2.7 - Considerações Finais	35
3. Protótipo do Editor Genérico Sensível à Sintaxe	37
3.1 - Considerações Iniciais	37
3.2 - Descrição do Protótipo EGSS	37
3.3 - Edição	38

3.4 - Tradução de Programas de uma Linguagem para Outra	43
3.4.1 - Estruturas Sintáticas de Controle de Fluxo	44
3.4.2 - Subrotinas	49
3.4.3 - Definição de Constante, Definição de Tipo, Declaração de Variável e de Parâmetro	52
3.4.4 - Expressões	58
3.5 - Considerações Finais	59
 4. Representação Intermediária dos Programas	60
4.1 - Considerações Iniciais	60
4.2 - Estrutura da Hierarquia	60
4.3 - Apresentação de uma Hierarquia	62
4.4 - Inclusão e Substituição	63
4.5 - Eliminação	66
4.6 - Considerações Finais	66
 5. Estrutura da Base de Dados	68
5.1 - Considerações Iniciais	68
5.2 - Sistema de Gerenciamento de Dados Utilizado	68
5.3 - Estrutura da Base GEO Utilizada pelo PEGSS	70
5.4 - Apresentação da Descrição da Linguagem Abstrata	71
5.4.1 - Estruturas de Controle de Fluxo	72
5.4.2 - Definição de tipos, definição de constantes e declaração de rótulos.	75
5.4.3 - Tipos de dados	76
5.4.4 - Declarações de Parâmetros e Variáveis	78
5.4.5 - Subrotinas	79
5.4.6 - Definições locais e globais de subrotinas	80
5.4.7 - Escopo das variáveis	81
5.4.8 - Blocos	81
5.4.9 - Expressões	82
5.5 - Armazenamento de Programas na Base de Dados	83
5.6 - Esquemas de Recuperação de Programas	85
5.7 - Considerações Finais	97

6. Expressões	98
6.1 - Considerações Iniciais	98
6.2 - Expressões em Geral	98
6.3 - Índices do Tipo de Dado Vetor	106
6.4 - Expressões da Estrutura de Controle "FOR"	112
6.5 - Considerações Finais	126
 7. Aspectos Gerais da Implementação	 128
7.1 - Considerações Iniciais	128
7.2 - Hierarquia	128
7.2.1 - Definição dos elementos de uma hierarquia	129
7.2.2 - Operações básicas em hierarquias	132
7.3 - Criação e Recuperação de Programas	134
7.3.1 - Criação de Programas	134
7.3.2 - Recuperação de Programas	143
7.4 - Expressões	147
7.4.1 - Expressões em geral	148
7.5 - Considerações Finais	149
 8. Conclusões, Decisões de Projeto e Futuras Pesquisas	 150
8.1 - Conclusões	150
8.2 - Decisões de Projeto	151
8.2.1 - Definição da Abrangência do PEGSS	151
8.2.2 - Definição da DLA	152
8.2.3 - Esquemas de Recuperação	153
8.2.4 - Hierarquia	154
8.2.5 - Expressões	154
8.2.6 - Sintaxe Concreta e informações para a edição	155
8.3 - Futuras Pesquisas	155
8.3.1 - Tornar o PEGSS totalmente independente de linguagem	155
8.3.2 - Ampliar a verificação automática de escopo para os identificadores através da DLA	156
8.3.3 - Conclusões sobre a implementação	157
8.3.4 - Criação de novas ferramentas	158

Referências Bibliográficas	160
Apêndice A: Composição Hierárquica dos Gabaritos da Linguagem Pascal	A1
Apêndice B: Definição da Linguagem Abstrata	B1
Apêndice C: Esquema de Recuperação de Programa da linguagem Pascal	C1

LISTA DE FIGURAS

Figura 2.1: Árvore de Sintaxe Abstrata de P.	9
Figura 2.2: Esquema para a construção de editores ALOE.	20
Figura 2.3: Ligação e carregamento no GP.	24
Figura 2.4: Visão do programa "exemplo"	31
 Figura 3.1a: Gabarito com as Estruturas Sintáticas	45
Figura 3.1b: Continuação dos gabarito com as estruturas	45
Figura 3.2: Traduções da estrutura de controle "for".	48
 Figura 5.1: Representação da estrutura de controle "for" na DLA.	74
Figura 5.2: DRI do progrma de nome "par".	85
Figura 5.3: Continuação do DRI do progrma de nome "par".	86
 Figura 6.1: Tabela de equivalência entre operadores	100
Figura 6.2: Continuação da Tabela de equivalência	101
Figura 6.3: Etapas para a transformação de infix para pós-fix	102
Figura 6.4: Prioridades entre os operadores	103
Figura 6.5: Prioridades entre os operadores	104
Figura 6.6: Transformação de genéricas para infix	105
Figura 6.7: Exemplos de expressões que descrevem índices	112
Figura 6.8: DRI de uma estrutura "for".	116
 Figura 7.1: Representação das possíveis inserções na hierarquia.	133
Figura 7.2: Representação das colônias de programas	135
Figura 7.3: DRI sobre as informações iniciais de um arquivo	135
Figura 7.4: Hierarquia inicial do arquivo/programa "teste".	136
Figura 7.5: Colônias contendo o ERPPas e o programa "teste".	144

UM EDITOR GENÉRICO SENSÍVEL À SINTAXE ARMAZENADA NUMA BASE DE DADOS

Autor: Prof. CARLOS ROBERTO VALÊNCIO

Orientador: Prof. Dr. CAETANO TRAINA JÚNIOR

RESUMO

Este trabalho descreve a implementação do Protótipo de um Editor Sensível à Sintaxe (PEGSS). Nesta versão o Editor oferece a edição sensível à sintaxe para um subconjunto das linguagens Pascal, C e FORTRAN. Programas sintaticamente corretos são construídos através da requisição pelo usuário de gabaritos que correspondem aos comandos e pela descrição das expressões.

É proposto um modo único de representação interna dos programas editados sob o contexto desse Editor. Assim, um conjunto de ferramentas pode beneficiar-se dessa forma de representação. Por exemplo, é permitido que programas escritos em uma linguagem sejam obtidos em qualquer das outras linguagens contempladas pelo Editor. Além disso, pode ser efetuada a adição de novas linguagens pertencentes a um mesmo paradigma, pois a construção dessa representação interna prevê esta extensão.

Neste trabalho são apresentadas as características do PEGSS: a estrutura utilizada para a exibição dos seus programas, a representação interna destes programas, a manipulação das expressões e, finalmente, como foram especificados e implementados os módulos principais que compõem este Sistema.

A GENERIC EDITOR SENSIBLE TO STORED SYNTAX IN A DATA BASE

Author: Prof. CARLOS ROBERTO VALÊNCIO

Adviser: Prof. Dr. CAETANO TRAINA JÚNIOR

ABSTRACT

This work describes the construction of the Prototype of an Editor Sensible to Syntax (PEGSS). In this version the Editor offers the edition sensible to syntax to a subgroup of the Pascal, C and FORTRAN languages. Programs syntatically correct are build through the requirement by the templates users that correspond to the command and through the description of the expressions.

It's proposed an only manner of intern representation of programs editored under a context of this Editor. So, a group of tools can be benefited of this representation form. For instance, it is allowed that programs written in a language are obtained in any of the other languages suported by the Editor. Besides, the adition of new languages belonging to a same paradigm can be made, therefore the construction of this intern representation foresees this expansion.

In this work the characteristics of the PEGSS are presented: the structure utilized to an exhibition of its programs, the intern representation of these programs, the manipulation of the expressions and, finally, how were specified and build the main modulus that arrange this system.

Capítulo 1

INTRODUÇÃO

1.1 - Caracterização do Problema.

A falta de critérios para o desenvolvimento de software, a dificuldade em mantê-los e a incapacidade em atender a sua demanda têm contribuído muito para que a sua evolução passe por sérias dificuldades, na tentativa de acompanhar a evolução do "hardware". Estas dificuldades caracterizam a discutida "crise do software".

Qualquer tentativa de solucionar esta crise deve proporcionar assistência prática a quem desenvolve software e melhorar a qualidade do software produto. Um caminho possível consiste em: adequar métodos para cada uma das fases do seu desenvolvimento; tornar disponíveis ferramentas para automatizar estes métodos; utilizar técnicas que assegurem a sua qualidade; e desenvolver mecanismos para coordenação, controle e gerenciamento. Com isso, torna-se possível disciplinar o desenvolvimento de software. O estudo e o desenvolvimento destes métodos, técnicas e ferramentas constituem a disciplina denominada "Engenharia de Software" [Pre_87].

A Engenharia de Software tem como objetivo oferecer o suporte prático a quem desenvolve software e melhorar a qualidade do produto final. O suporte prático é proporcionado pelas ferramentas automatizadas, as quais auxiliam o usuário (analista e ou programador) no desenvolvimento de software, livrando-o de operações repetitivas, onde não existe o fator "criatividade". Promovendo, o máximo possível, a realização de tarefas "braçais" pelo próprio computador, permite-se ao usuário concentrar esforços em outras atividades mais "nobres". Exemplos dessas ferramentas são editores, compiladores e ligadores de módulos, usados desde que surgiram os computadores, e ferramentas que automatizam o emprego de métodos e metodologias. Com o passar do tempo, começou-se a desejar que ferramentas utilizadas isoladamente pudessem constituir-se em uma só ferramenta, ou seja, integradas em um só Ambiente.

Uma ferramenta que integra o editor com todos os recursos necessários à geração de códigos - o compilador, o ligador de módulos, o gerenciador de bibliotecas e o

depurador - recebe o nome de Ambiente de Programação. Esta única ferramenta suporta toda a fase de codificação do ciclo de desenvolvimento de software. Exemplos desses Ambientes são Mentor [Don_75] [Don_84], Cornell Program Synthesizer [Tei_84], Gandalf [Not_85] [Eli_85] [Hab_86], PECAN [Rei_84] [Rei_85], POE [Fis_84] e outros. Esses ambientes possuem ferramentas, tais como Editores sofisticados que oferecem recursos poderosos para o suporte a edição de programas. Alguns desses Editores são sensíveis à sintaxe e possibilitam a edição de programas livres de erros sintáticos e, na maioria deles, de erros semânticos.

Um aspecto vital em qualquer ambiente integrado de apoio ao desenvolvimento de software, é a forma como as informações são representadas e armazenadas internamente. Para isso, emprega-se uma base de dados, adequada para a representação dos dados. A base de dados torna-se assim o elemento central da integração de ferramentas, compartilhando os dados de maneira homogênea para todas elas. Uma base de dados contribui também para a expansibilidade do ambiente, permitindo que sejam agregadas novas ferramentas, que manipulam as mesmas informações, sem afetar as ferramentas já existentes.

Este trabalho descreve o Protótipo de um Editor Genérico Sensível à Sintaxe, onde é possível a edição de programas sintaticamente corretos, escritos em qualquer das linguagens suportadas pelo Editor: Pascal, C e FORTRAN. As informações dos programas são armazenadas de modo único, independentemente de qual das linguagens suportadas pelo Editor tenha sido utilizada para a sua construção. Um conjunto de ferramentas podem beneficiar-se dessa forma de armazenamento, como por exemplo, a possibilidade de se ter um programa representado em qualquer das linguagens suportadas pelo Editor.

A disponibilidade de uma forma única de representação de um programa facilita a construção de analisadores de validade e consistência, otimizadores, geradores de massa de testes, etc. Essa forma de representação é possível - afinal um programa escrito em qualquer dessas linguagens acaba sendo traduzido para a mesma linguagem de máquina - mas, ao contrário de um programa em formato objeto, essa representação é de alto nível, permitindo que toda a semântica do programa, em sua linguagem fonte, seja mantida. Mais do que isso, qualquer informação gerada pelo processo de tradução (equivalente ao processo de compilação) pode ser armazenada, evitando-se ter que reobtê-la em ferramentas usadas posteriormente sobre o programa, que dela necessite.

O PEGSS utiliza-se, para a representação interna de seus programas, do GERenciador de Objetos (GEO) [Tra_91], um sistema de gerenciamento de dados baseado

no Modelo de Representação de Objetos (MRO) [Tra_86] [Tra_88] [Bia_92]. Assim, foi possível alcançar um dos objetivos deste trabalho, qual seja o de validar a base de dados GEO em aplicações de Engenharia de Software. Além desse, foram também objetivos deste trabalho: investigar os requisitos necessários para a implementação de recursos de edição, dirigido pela sintaxe; e verificar a capacidade de suportar, de forma homogênea, múltiplas linguagens de um mesmo paradigma.

A disponibilidade de uma forma única de representação em alto nível de um programa, comum a diversas linguagens, e armazenado em uma base de dados, facilita muito a criação de um ambiente integrado, com um conjunto extenso de ferramentas de análise e manipulação de programas.

Este trabalho desenvolveu esta forma de representação genérica, bem como teve implementado um Editor para a tradução de programas escritos em C, Pascal e FORTRAN, o qual tanto converte programas escritos numa dessas linguagens para a representação genérica, quanto converte da representação genérica para uma dessas linguagens. O Editor foi construído de maneira modular para que possa ser futuramente expandido, para suportar outras linguagens com paradigmas semelhantes aos dessas.

1.2 - Organização da Dissertação.

Este trabalho está organizado da seguinte forma:

- **Capítulo 1:** são apresentadas as idéias que motivaram o desenvolvimento desta dissertação, bem como a sua organização;
- **Capítulo 2:** são apresentados os pontos relevantes do estudo bibliográfico realizado como subsídeos para o desenvolvimento deste trabalho, e mostram-se outras ferramentas sendo desenvolvidas na área;

- **Capítulo 3:** descreve-se como um programa pode ser construído sob o contexto do PEGSS, além de apresentar como se procede a tradução de uma linguagem, para a representação genérica;
- **Capítulo 4:** apresenta-se a representação genérica, utilizada pelo PEGSS para a apresentação e edição dos seus programas;
- **Capítulo 5:** mostra-se a estrutura da base de dados, utilizada pelo PEGSS, para a representação genérica e para o armazenamento das informações por ele manipuladas;
- **Capítulo 6:** faz-se uma apresentação detalhada do tratamento dispensado pelo PEGSS às expressões de um programa. Mostra-se como são introduzidas, armazenadas e recuperadas;
- **Capítulo 7:** são apresentados os principais módulos que compõem o PEGSS. É descrita a especificação e forma de como estes módulos foram implementados.
- **Capítulo 8:** é apresentada uma análise dos resultados obtidos neste trabalho, a descrição das principais decisões adotadas no projeto, e são propostas algumas sugestões para a continuação das pesquisas.

Capítulo 2

REVISÃO BIBLIOGRÁFICA

2.1 - Considerações Iniciais.

O objetivo deste Capítulo é apresentar alguns dos Ambientes Orientados para Estruturas [Dar_87]: MENTOR [Don_75] [Don_84], Cornell Program Synthesizer [Tei_81] [Tei_84] [Rep_84], Projeto Gandalf [Amb_85] [Not_85] [Eli_85] [Hab_86] e PECAN [Rei_84] [Rei_85]. Procura-se ressaltar as principais características e, em alguns casos, o modo pelo qual foram implementados determinados recursos. Esses ambientes foram selecionados por possuírem um conjunto dos principais recursos oferecidos pelos ambientes deste tipo. Além desses, outros Ambientes Orientados para Estruturas podem ser encontrados na literatura, como por exemplo: Emily [Han_84], IPE [Med_81], POE [Fis_84] e GENESE [Ugg_90].

Fez-se um estudo desses Ambientes com o propósito de formar uma base de informações a ser utilizada como subsídios para a construção do Protótipo do Editor Genérico Sensível à Sintaxe. A atenção, ao analisar essas publicações, esteve basicamente voltada às informações referentes aos editores utilizados por esses ambientes, aos recursos oferecidos por estes editores, suas implementações e, principalmente, aos recursos e implementações sensíveis à sintaxe de linguagens de programação.

Antes de se iniciar a apresentação desses ambientes, são fornecidos alguns conceitos importantes utilizados por estes ambientes.

2.2 - Conceitos Utilizados pelos Ambientes Orientados para a Estrutura.

Os conceitos [Pin_91] que são frequentemente citados na sub-seção que segue são:

- Sintaxe concreta: é a representação das estruturas sintáticas de uma linguagem, necessária para a exibição de programas escritos nesta linguagem.
- Árvore de sintaxe abstrata: é a estrutura de dados utilizada como forma intermediária de representação de programas.
- Sintaxe abstrata: descreve a representação interna da árvore de sintaxe abstrata.
- Sintaxe léxica: descreve os itens da linguagem.
- Semântica estática: verifica a consistência da estrutura de dados definidas no programa, isto é, verificação de tipos, regras de escopo, etc.
- Semântica dinâmica: verifica a consistência dos relacionamentos entre as estruturas definidas para o programa.
- "Unparsing": transformação das informações armazenadas na árvore de sintaxe abstrata em um programa legível a ser exibido no vídeo. "Unparser" é o procedimento que efetua esta transformação.
- A gramática de atributos proporciona tanto a incorporação de atributos pelos símbolos não terminais, quanto as regras semânticas proporcionadas por produções em uma gramática livre de contexto. Os tipos de atributos são:
  - atributos herdados: são baseados nos atributos dos ancestrais do nó não terminal;
  - atributos sintetizados: são baseados nos atributos dos nós descendentes do nó não terminal.

2.3 - MENTOR.

O MENTOR [Don_75] [Don_84] é um ambiente de suporte à implementação, documentação, teste, validação e gerenciamento de programas [Pin_91]. O MENTOR e suas ferramentas foram escritos em Pascal e sua implementação iniciou-se em um Honeywell-Bull 68 suportado pelo sistema operacional MULTICS. Também tem sido transportado para o VAX sob o UNIX para Ada e Pico. Encontra-se em versões para cinco linguagens: Pascal, Ada, Metal, Flip e Rapport.

A principal característica do ambiente MENTOR é a utilização de uma estrutura de dados para o armazenamento dos programas editados. Estas estruturas representadas por árvores do tipo "operador-operando" são geralmente denominadas de Árvores de Sintaxe Abstrata (ASA). A maioria dos recursos oferecidos por este ambiente são obtidos por meio de programas que operam sobre a estrutura, como por exemplo, a apresentação de um programa em diferentes níveis de detalhes, produção de comentários estruturados, depuradores, etc.

Para entender melhor como são armazenados os arquivos editados sob o controle do MENTOR é apresentado a seguir a estrutura de dados utilizada para a ASA.

Árvore de Sintaxe Abstrata.

Um programa manipulado pelo MENTOR é armazenado em uma árvore de sintaxe abstrata independentemente de ter sido escrito sob seu controle ou não.

A árvore do tipo "operando-operador" possui três modalidades de nós:

- a. O nó composto por um operador de multiplicidade fixa, isto é, possuidor de um número fixo de descendentes. Exemplos desses operadores e suas classes, que definem os tipos dos seus nós descententes, são:

if: exp X stat X stat -> stat.
ass: varbl X exp -> stat.
call: ident X lexp -> stat.
gtr: exp X exp -> exp.
mult: exp X exp -> exp.
index: ident X lexp -> exp.

- b. O nó composto por um operador de multiplicidade variável, isto é, possuidor de um número variável de descendentes. Exemplos desses operadores utilizados para representar listas e suas classes são:

lstat: stat X stat X ... X stat -> stat.
lexp: exp X exp X ... X exp -> exp.

- c. Nós terminais, denominados de folhas da árvore, que podem ser variáveis ou constantes.

A seguir mostra-se um exemplo de como ficaria a representação abstrata de um trecho P de um programa em Pascal. Considerando o trecho P de um programa:

```
if X>0 then P(X,A[Y,Z]) else  
    begin  
        Y := Y * 2;  
        X := 0  
    end
```

A representação através da árvore de sintaxe abstrata de P seria a seguinte:

O procedimento "Construtor"

Um arquivo de caracteres, construído ou não, sob o controle do editor do MENTOR, transformar-se-á em sua correspondente ASA através de um programa analisador denominado "Construtor".

O "Construtor" monta a Árvore de Sintaxe Abstrata do programa com ausência de verificação de tipo, a qual se faz posteriormente em um processo separado. O "Construtor" não apenas constrói a ASA toda, como também é requisitado sempre que qualquer modificação é feita no texto.

A verificação da correção sintática de um programa editado sob o controle do MENTOR se faz durante a transformação do arquivo para a ASA, pois o "Construtor" realiza tal operação dentro dos parâmetros definidos para a árvore, os quais, por sua vez, foram estabelecidos de acordo com a sintaxe da linguagem corrente. Assim, se a transformação não é possível é acusado o erro sintático.

Figura 2.1: Árvore de Sintaxe Abstrata de P.

Ferramentas do MENTOR

Após a ASA de um arquivo ter sido construída, alguns procedimentos operam sobre ela proporcionando ao usuário alguns recursos importantes na construção do programa:

- O verificador da correção de escopo, que analisa a declaração de identificadores para um escopo.
- O verificador de tipos, que analisa as operações de tipos.
- O verificador de erros de execução, composto por rotinas de análise de fluxo de dados e um interpretador simbólico.
- O interpretador/depurador, que efetua a verificação de programas agindo sobre a árvore de sintaxe e a tabela de símbolos. A verificação é implementada pela introdução de predicados de primeira ordem.
- Mecanismos de otimização de programas. As otimizações locais são feitas por transformações no programa, enquanto otimizações globais são efetuadas ao nível de códigos fonte.
- O procedimento PROFILE, que oferece um panorama da execução do programa, proporcionando recursos de depuração.
- Procedimentos que mostram o texto do programa em diferentes níveis de detalhes. Por exemplo, a representação P acima seria mostrada, de acordo com os níveis de detalhes, do seguinte modo:

```
if # then # else ...
```

```
n = 1
```

<pre>if X>0 then P(X,A[Y,Z]) begin ... end</pre>	n = 2
<pre>if X>0 then P(X,A[Y,Z]) else begin # ; # end</pre>	n = 3
<pre>if X>0 then P(X,A[Y,Z]) else begin Y := Y * 2; X := 0 end</pre>	n = 4

- Procedimentos que permitem mostrar no vídeo o texto de um programa, realçando, através de contraste luminoso, partes interessantes para o usuário, como suas declarações e identificadores.
- Procedimentos que introduzem e atualizam nós de comentários na representação abstrata. Programas avaliadores especiais podem, posteriormente, produzir comentários estruturados para documentação.

A linguagem de manipulação de árvore MENTOL.

O MENTOR possui uma linguagem denominada MENTOL que permite ao usuário comunicar-se com o ambiente. Através de seus comandos é possível mostrar, inserir, eliminar nós de uma árvore, executar e depurar programas. Por exemplo, para mostrar o programa P da Figura 2.1 através do nível 1 de detalhe basta descrever o seguinte comando:

@TOP P1

onde,

@TOP posiciona no início da árvore de um programa e
P1 indica que o nível de detalhe é 1.

Todos os procedimentos descritos na sub-seção anterior estão disponíveis ao usuário através da linguagem MENTOL. Através dessa linguagem é possível também expandir ou substituir este conjunto de ferramentas escrevendo procedimentos externos, aumentando a capacidade do ambiente.

O MENTOR é considerado como um primeiro passo em direção à construção de Ambientes de Programação. Foi pioneiro na utilização de estruturas de dados do tipo árvore, que possibilitando a construção de ferramentas mais poderosas, permitindo que programas concretos possam ser manipulados como objetos abstratos. Além disso, o MENTOR apresenta facilidades que outros ambientes orientados para estruturas mais recentes não possuem, como o suporte concorrente para programas escritos em diferentes linguagens.

2.4 - CORNELL PROGRAM SYNTHESIZER.

O Cornell Program Synthesizer (CPS) [Tei_81] [Tei_84] [Rep_84] é um ambiente de programação interativo com recursos integrados para criar, editar, executar e depurar programas.

O processo de edição e execução é orientado pelas estruturas sintáticas da linguagem de programação suportada pelo ambiente. A gramática da linguagem está embutida em uma coleção de gabaritos pré-definidos que, através da requisição dos mesmos e pela descrição de expressões como textos, permitem que programas sejam construídos sem erros sintáticos. Se o usuário deseja o comando "if-then-else" basta digitar o comando que o requisita, e em seguida, aparecerá no vídeo o seguinte gabarito:

```
if ( condição )  
    then comando  
    else comando
```

O cursor ficará posicionado no início da palavra "condição". Esta posição é denominada lugar de inserção, e o editor ficará esperando que o usuário digite adequadamente uma expressão booleana ("condição"). Após a introdução da expressão o usuário poderá passar para os outros lugares de inserção do tipo "comando", para que se possa continuar a construção do programa. Assim sendo, o usuário fica livre de detalhes sintáticos da linguagem, pois só lhe será permitido digitar uma instrução do tipo "condição" quando o cursor estiver posicionado no primeiro lugar de inserção desta estrutura (qualquer tentativa de digitar algo diferente resultará em erro) e, com os gabaritos, o usuário não comete erros que surgiriam eventualmente ao ter que digitar livremente as estruturas da linguagem.

O editor do CPS é híbrido, ou seja, manipula estruturas de dados do tipo árvore e do tipo texto. Os gabaritos são oferecidos através de comandos, enquanto as expressões e comandos de atribuição devem ser descritas digitando-se caractere por caractere. Os erros digitados pelo usuário são detectados imediatamente, pois um analisador sintático é requisitado pelo editor à cada frase digitada.

Recursos de depuração também são sensíveis à sintaxe. Quando o usuário requisita o "tracing" para a execução, o cursor acompanha no vídeo as respectivas instruções do código fonte. Quando requisita o "single-stepping", o usuário controla a execução de acordo com a estrutura hierárquica dos gabaritos do programa.

Implementação:

A representação interna dos arquivos do Synthesizer é em forma de árvore, onde os nós são gabaritos ou frases. Os nós são de comprimento variável, compostos por três partes:

extensão	código	continuação

- Extensão: identifica o tipo de nó e contém informações para mostrá-lo.
- Código: contém códigos interpretáveis, ou seja, códigos que serão executados por um interpretador.
- Continuação: contém um desvio para o próximo código.

Como exemplo mostra-se a seguir como fica a representação interna do gabarito "if-then-else":

```
if ( condição )  
then comando  
else comando
```

lugares de inserção não preenchidos

No exemplo acima, os lugares de inserção não foram preenchidos, portanto seus respectivos códigos interpretáveis devem indicar o não preenchimento para o interpretador.

Preenchendo os lugares de inserção com a condição $k > 0$ e o segundo comando com `write()`, a estrutura fica:


```

if ( K > 0 )
then comando
else write ( expressão ) ;

```

O interpretador executará as linhas de código e as instruções "cont." (continua), com acessos às variáveis e definições de procedimentos na tabela de símbolos.

Na depuração os recursos de acompanhamento das estruturas executadas no texto do programa e o controle, pelo usuário, da velocidade em que as estruturas dão-se através da ação do interpretador quando passa do "cont." para um novo nó.

Saída da árvore em programa texto:

A impressão do arquivo é gerada a partir da árvore e de uma tabela que contém a representação externa de cada gabarito. Por exemplo, o gabarito "if-then-else" tem a seguinte representação externa:

```
"IF ("  
    descrição da condição  
")\\(\n THEN"  
    descrição do comando 1  
"ELSE"  
    descrição do comando 2  
"\\)\r"
```

Onde os caracteres de controle tem os seguintes significados:

\\	move a margem esquerda para a direita em uma unidade,
\\n	pula uma linha e coloca na atual margem esquerda,
\\)	move a margem esquerda para a esquerda em uma unidade,
\\r	"carriage-return" para a atual margem esquerda.

Um programa é encarregado de percorrer a árvore em pré-ordem e interpretar os caracteres de controle de acordo com a tabela contendo o formato para as representações externas (gabaritos) dos nós encontrados na árvore. Os nós compostos por "expressões" armazenadas na forma pós-fixa são transformados na forma infixa para sua representação externa.

Como a representação externa de um programa é feita percorrendo-se a árvore diretamente, existe uma tabela atualizada que faz o "mapeamento" do endereço do nó para as coordenadas do vídeo. Esta tabela também é utilizada para a movimentação do cursor e, na execução, para os recursos de acompanhamento, no vídeo, dos códigos que estão sendo executados.

Para facilitar o desenvolvimento de recursos sensíveis à sintaxe implementou-se um sistema independente de linguagem para a geração de sistemas como o Cornell Program Synthesizer. O gerador de sistemas denominado de Synthesizer Generator gera os sistemas através de uma especificação gramatical de uma dada linguagem de programação. A gramática de atributos é utilizada para definir a sintaxe, o formato de saída (sintaxe

concreta), a semântica de cada frase e o gabarito da linguagem desejada. Estas informações são especificadas através da SSL ("Synthesizer Specification Language").

2.5 - Projeto GANDALF.

O objetivo do Projeto GANDALF [Amb_85] [Ell_85] [Kai_85] [Not_85] [Hab_86] é a produção, através da geração semi-automática, de ambientes de desenvolvimento de software que integrem as noções de ambiente de programação e de desenvolvimento de sistemas [Hab_86]. Estes ambientes têm características em comum, tais como:

- Integração: uma Base de Dados comum permite o compartilhamento de conhecimento entre as tarefas em um ambiente de desenvolvimento de software Gandalf.
- Interface interativa com o usuário: a comunicação entre os Ambientes Gandalf e seus usuários é feita através de interfaces interativas. Por exemplo, se durante a edição de um programa o usuário tentar utilizar uma variável não declarada, o sistema emitirá uma mensagem, avisando-o do possível erro.
- Uniformidade: a uniformidade e consistência da interface entre todas as ferramentas reduz o freqüente aumento do custo ao serem introduzidas novas tarefas, e o tempo dispensado no treinamento de novos usuários.
- Estado bem definido do sistema: os Ambientes Gandalf incorporam informações que descrevem o estado do sistema de software que está sendo desenvolvido.

Além destas características gerais os Ambientes de Programação e de Desenvolvimento de Software Gandalf possuem características particulares. Os Ambientes de Programação Gandalf são Sistemas de Edição Orientados para Linguagem que suportam depuração ao nível fonte e interpretação ou compilação incremental. Também são

caracterizados tanto pelo suporte ao Controle de Versões de Sistemas como ao Gerenciamento de Projetos.

A necessidade de gerar ambientes de desenvolvimento de software semi-automaticamente se deve ao fato de que cada projeto precisa de diferentes ambientes para o seu desenvolvimento [Hab 86]. O projeto Gandalf resolve este problema através do gerador de editor orientado para estrutura denominado ALOE, é apresentado na sub-seção seguinte.

2.5.1 - Editores ALOE.

É um conjunto de Editores gerados pelo ALOE. Estes editores, sensíveis à estrutura sintática de uma linguagem, são a base dos Ambientes Gandalf.

Todos os editores ALOE possuem dois tipos de comandos, que proporcionam uma uniformidade em suas interfaces:

- Comandos construtivos (dependentes da linguagem) são as estruturas sintáticas da linguagem. Por exemplo, caso o usuário deseje usar o comando "while", basta digitar "wl" que seu respectivo gabarito, representado abaixo, aparecerá no vídeo. O cursor ficará parado no lugar de inserção "%bool-exp", esperando que a expressão do tipo "booleana" seja digitada pelo usuário:

```
while %bool-exp do
    %comando
od
```

- Comandos de edição (independentes da linguagem) são utilizados para manipular a árvore na base de dados, tal como "apagar" uma construção ou "mover" o cursor.

Construção dos Editores:

Para contruir um editor, o implementador deve entrar com a descrição da sintaxe abstrata, da sintaxe concreta e os nomes das rotinas de ação.

- Sintaxe abstrata: descreve os operadores (terminais e não-terminais) e classes. Estas indicam um conjunto de operadores que podem substituir um lugar de inserção em particular. Por exemplo:

```
WHILE = % bool-exp % comando
% bool-exp = ID | TRUE | FALSE | LSS | LEQ | EQL | NEQ
% comando = ASSIGN | IF | FOR | WHILE | RETURN
```

onde o lugar de inserção "% comando" pode ser substituído por um dos operadores entre o ASSIGN, IF, FOR, WHILE ou RETURN.

- Sintaxe concreta: descreve o modo pelo qual cada operador terá sua representação na saída. Por exemplo,

"WHILE = while @1 do @N @2"

onde @1 e @2 indicam os filhos do operador WHILE e @N a inserção de uma nova linha na saída. Esta saída é feita pelo "unparser" que percorre a árvore sintática transformando em representação textual.

- Rotinas de ação: são procedimentos que permitem ao usuário fazer modificações no ambiente corrente, tal como criar, apagar ou modificar um nó da árvore.

A construção de editores pode ser visualizada através da Figura 2.2.

Figura 2.2: Esquema para a construção de editores ALOE.

O ALOEGEN é uma ferramenta que auxilia o implementador na descrição da sintaxe concreta e da sintaxe abstrata do ambiente a ser construído, transformando-a em tabelas na linguagem GC (Gandalf C). Posteriormente essas tabelas são compiladas, transformando-se em código objeto. As rotinas de ação são transformadas em código objeto através do compilador GC que junto com as tabelas da linguagem (na forma de código objeto) e o "kernel" (na forma de código objeto) são ligados para produzir o editor.

Na geração de um ambiente, o implementador constrói uma gramática que representa todo o conjunto de construções (tais como comandos de linguagem de programação, listas de controle de acesso e descritores de módulos) que estão associadas ao processo de desenvolvimento de software. Essa gramática é armazenada em uma base de dados, que permitirá o compartilhamento das informações entre as tarefas que estão interrelacionadas, viabilizando a integração entre os ambientes.

2.5.2 - Ambientes GANDALF.

Entre os ambientes Gandalf implementados descreveremos três: Protótipo Gandalf (GP), Projeto GNOME e SMILE. É importante ressaltar que o GP e o GNOME foram implementados como um editor ALOE.

2.5.2.1 - PROTÓTIPO GANDALF.

Este ambiente suporta a fase de codificação, proporcionando auxílio na construção incremental de programas (compilação incremental e depuração ao nível fonte), controle de versões de sistemas e gerenciamento de projetos.

Primeiramente focalizar-se-á o suporte do controle de versões de sistemas, descrevendo-se a estrutura para a descrição de sistemas usada no GP, a qual será retomada posteriormente para os outros dois tipos de suporte.

Controle de Versões de Sistemas:

O principal recurso utilizado para o controle de versões é a estrutura para descrição dos sistemas, cuja composição é análoga à usada em sistemas de arquivos. Caixas podem conter outras caixas e módulos, do mesmo modo que diretórios podem conter subdiretórios e arquivos. Os módulos contêm: uma interface, composta pelos seus recursos, e um conjunto de versões que a implementam de diferentes maneiras. Cada versão contém um conjunto de revisões que representam a atual implementação de um módulo. Assim, as alterações nos módulos feitas pelo programador e a composição do sistema serão incorporadas por esta estrutura.

Um dos auxílios que o GP oferece para controle de versões de sistemas é estender a interface de um módulo. Por exemplo, supondo-se o módulo M com os recursos de interface (f,g), e se quisesse estendê-los para (f,g,h), sabendo-se que P1 é uma implementação de um programa para h, o novo módulo P com a interface contendo os recursos (f,g,h) definiria-se pela seguinte construção:

```
module P provide f,g,h ;  
    impl P1 with M ;  
end P
```

Outra possibilidade é a composição de módulos, onde módulos existentes podem ser reempacotados. Supondo um módulo M que proporciona o conjunto de recursos (f,g) e um módulo N que proporciona o conjunto de recursos (h,k). A composição:

```
comp C = ( M , N )
```

seria a versão de um módulo C que exporta no mínimo os recursos (f,g,h,k).

O GP oferece também a geração automática de versões executáveis do sistema. Isto é possível através do comando "instantiate", que proporciona ao usuário eleger uma versão do módulo como sua versão padrão. Assim, subsistemas são gerados a partir do agrupamento de versões padrão, sem que haja a necessidade de o usuário ficar lembrando as várias sequências de carregamentos, necessárias para se obter o correto agrupamento de versões dos módulos.

Gerenciamento de Projeto:

Este tipo de auxílio garante a integridade do desenvolvimento do estado da informação e proporciona o acesso a esta informação. O GP oferece este suporte através de:

- **Lista de acesso:** uma para cada caixa, controlando o acesso às informações para cada tipo de usuário.
- **Histórico das revisões:** um para cada caixa, um para cada módulo e um para cada versão do módulo. São mensagens ordenadas pelas datas, descrevendo as revisões efetuadas.

- Controle nas modificações em módulos: através de comandos que evitam conflitos nas atualizações:
  - "reserve": reserva para o usuário o módulo a ser alterado, evitando que outros usuários façam modificações ao mesmo tempo.
  - "release": volta para o estado anterior ao digitar "reserve".
  - "deposit": deposita o módulo modificado na base de dados pública.

Construção incremental de programas:

Consiste em otimizar o ciclo: editar texto, compilar, carregar, executar. No Protótipo Gandalf algumas melhorias foram alcançadas neste ciclo, tais como:

- O uso de editores ALOE permite diminuir o esforço gasto em compilação, porque são construídos programas sintaticamente corretos compilados.
- O carregamento e a ligação são incrementais, assim, depois de qualquer alteração feita dentro de um módulo com o uso de endereçamento invariante para cada procedimento, dispensa-se a religação do sistema todo. Um vetor denominado "vetor de entradas" contém para cada módulo todos os endereços dos códigos objetos de seus procedimentos. Após qualquer modificação no procedimento, ele é recarregado em um endereço novo de código. Evita-se a religação porque se atualiza o endereço da respectiva entrada do procedimento no "vetor de entradas". A Figura 2.3 mostra esta abordagem.

Figura 2.3: Ligação e carregamento no GP.

A coluna da esquerda representa as chamadas de procedimento no código fonte. A coluna da direita representa espaços de armazenamento dos códigos objeto dos respectivos procedimentos. Na coluna do meio está o "vetor de entradas" contendo os endereços de armazenamento dos referidos códigos objeto. O símbolo B* indica que algumas modificações foram feitas no código fonte do procedimento B e seu código objeto foi armazenado em outro espaço de memória. Consequentemente atualiza-se o vetor de entradas, evitando-se que os endereços para as chamadas do procedimento B tenham que ser alterados através de uma nova ligação.

- Depuração ao nível fonte. Através dos comandos "trace" e "pause" o usuário pode interromper a execução de um sistema, retornando o controle ao usuário. Em seguida pode-se recommençar a execução ou continuar a partir do ponto em que foi interrompida. Outro recurso é a possibilidade de alterar, através de uma janela, os pares: valor e nome dos objetos pertencentes às estruturas de dados manipulados pelo sistema durante a execução.

2.5.2.2 - Projeto GNOME.

Consiste em uma família de editores ALOE utilizados como um único ambiente de programação em cursos introdutórios de programação. Programas sintaticamente corretos e semanticamente verificados são desenvolvidos e executados dentro do contexto do editor. Estes editores são:

- GNOME-Family:** usado no treinamento de alunos para a manipulação de árvores;
- GNOME-KAREL:** permite construções simples de programas através de uma linguagem gráfica;
- GNOME-FORTRAN:** editor orientado para estrutura que oferece suporte à edição sob o contexto da linguagem FORTRAN;
- GNOME-Pascal:** editor orientado para estrutura que oferece suporte à edição sob o contexto da linguagem Pascal.

2.5.2.3 - SMILE.

Ambiente que auxilia no gerenciamento de código fonte e de projetos. Atividades principais de suporte:

- Definição e manipulação de interconexão de módulos:
  - comandos que importam e exportam recursos (funções C, tipos de dados, etc) de um módulo.
  - comandos para visualizar a estrutura com os módulos que compõem o sistema corrente.
- Controle de versões públicas e experimentais da base de dados dos projetos.
- Suporte à compilação implícita através do gerenciamento da base de dados que a auxilia.

2.6 - PECAN.

O PECAN [Rei_84] [Rei_85] é um gerador de sistemas de desenvolvimento de programas para linguagens algébricas de programação. Proporciona um ambiente de programação através de especificações simples, em que múltiplas visões concorrentes são permitidas durante o processo de construção de programas. Estas visões constituem-se em representações gráficas e textuais do programa, de sua semântica e de sua execução. Cada visão é definida em um local do monitor de vídeo através de janelas que são ativadas por dispositivos de pontuação.

Este protótipo de ambiente foi construído como parte de uma família de ambientes desenvolvidos na Universidade de Brown, e teve como objetivo principal avaliar a capacidade do poder computacional e gráfico da nova geração de computadores pessoais para o desenvolvimento de programas.

O PECAN é de fácil utilização, oferece um retorno imediato e permite ao usuário visualizar o seu programa sob vários aspectos. A maioria das suas características são compartilhadas por outros ambientes, entre elas, as principais são descritas a seguir:

- Retorno imediato de erros semânticos e sintáticos durante a edição do programa;
- Facilidade para retornar ao início de uma ação que foi considerada errada pelo usuário, desfazendo-a e começando-a novamente;
- Possibilidade de escrever seus programas através de gabaritos das estruturas que estão disponíveis como comandos;
- Flexibilidade na escolha, a qualquer momento, entre escrever textualmente ou através de gabaritos o programa;
- A utilização de menus como alternativa para a maioria dos comandos;
- Utilização da tela através de múltiplas janelas;
- Oferecimento de uma semântica incremental na construção dos programas, possibilitando que estes sejam compilados enquanto são editados;

variável. Isto poderia ser efetuado de dois modos: digitando-se a palavra "var" sobre o lugar de inserção corrente, especificado pela palavra DECLARATION, ou escolhendo-se a opção "var" oferecida pelo menu à direita da janela utilizada pelo Editor. Para o programa cujo nome é "exemplo", mostra-se abaixo como esta inserção é efetuada:

```
PROGRAM exemplo (IDENTIFIER);  
  COMMENT  
  DECLARATION  
  ROUTINE  
  
  BEGIN { Program exemplo }  
    STATEMENT  
  END.
```

Digitando-se a palavra "var" na posição corrente e em seguida especificando a declaração das variáveis x e y, o programa fica representado do seguinte modo:

```
PROGRAM exemplo (IDENTIFIER);  
  COMMENT  
  
  VAR  
    x, y : integer;  
  VARIABLE DEFINITION;  
  
  DECLARATION  
  
  ROUTINE  
  
  BEGIN { Program exemplo }  
    STATEMENT  
  END.
```

Esta sequência de inserções ilustra os três modos oferecidos pelo Editor para se escrever um programa. O usuário pode escolher o gabarito através de um menu de opções, pode inseri-lo por meio de um prefixo digitado sobre a estrutura corrente no programa ou

pode digitar a estrutura toda. Com isto o usuário tem mais liberdade durante a edição, cabendo a ele escolher o modo que melhor lhe convenha.

Alguns comandos simples permitem ao usuário "navegar" pela árvore sintática do programa, possibilitando-se escolher ou colocar um nó nesta árvore. Se o nó corrente for uma variável meta ou um nó folha, o Editor mostra, respectivamente, as possíveis estruturas que podem ser utilizadas para a expansão e uma lista de candidatos a nomes obtidos de uma tabela de símbolos.

A visão Nassi-Schneiderman

Esta visão permite mostrar um programa utilizando o Diagrama de Nassi-Schneiderman. Por meio desta representação o usuário pode visualizar a forma completa do seu programa desenhada como um gráfico de fluxo estruturado.

Como esta visão pertence ao programa que está sendo editado, qualquer alteração que o usuário faça é automaticamente propagada para a representação gráfica.

A visão Declaração

Esta visão é apresentada apenas para as declarações. Ela pode ser acionada selecionando-se a janela utilizada para tal ou escolhendo-se, no texto do programa, o local adequado para descrever uma declaração. Ao introduzir a declaração de um nome é verificado se este já existe neste escopo. Caso isto não ocorra, um novo nome é criado. Para uma declaração selecionada em um programa, o usuário poderá utilizar a visão para adicionar, alterar ou apagar o nome a ela associado e localmente editar campos da declaração, como o tipo de dado da variável. Também podem ser efetuadas alterações no escopo da variável, por exemplo passando-a de local para global.

Os propósitos desta visão são: permitir que o usuário crie interativamente seus programas, efetuando suas declarações a medida que as for utilizando e, por fim, complementar as informações apresentadas em um Diagrama de Nassi-Schneiderman, já que neste tipo de representação não é possível especificar os tipos dos dados utilizados.

Visões semânticas

O PECAN permite também que se apresente informações semânticas sobre um programa que está sendo escrito, pois entende-se que tais informações são bastante úteis se oferecidas durante a sua construção. Estas informações são proporcionadas pelo compilador incremental e estão representadas na forma de: tabela de símbolos, definições de tipos de dados, árvores de expressões e gráficos de controle de fluxo.

A visão Tabela de Símbolos

Uma tabela de símbolos é construída e mantida incrementalmente para os símbolos internos e os definidos pelo usuário. Para cada árvore sintática esta visão mostra os seus símbolos e escopos. Os nomes dos símbolos são descritos nos interiores de caixas, que podem estar aninhadas, delimitando o escopo no programa onde foram definidos. Cada nome é apresentado com sua classe, tal como variável, tipo ou rótulo, e outras informações relevantes, tais como tipo de uma variável. Se a estrutura corrente do programa é uma variável, então esta variável aparecerá realçada na visão dos símbolos. Nesta visão, em apontando-se para um símbolo ou escopo, provoca-se uma definição de nó para aquele escopo ou nome na localização atual. Considerando que o programa "exemplo" tenha sido acrescido de algumas estruturas, a sua representação pode ficar como a descrita na Figura 2.4.

A visão para os símbolos do programa representado na Figura 2.4 é a seguinte:

```

PROGRAM exemplo (IDENTIFIER);
COMMENT
  VAR
    X, Y : integer;
    VARIABLE DEFINITION;
  DECLARATION
FUNCTION gcd (a, b: integer): integer;
COMMENT
  { no declarations }
BEGIN { function gcd }
  IF b = 0 THEN
    gcd := a
  ELSE
    gcd := gcd({ a := } b, { b := } a MOD b);
  STATEMENT
END

ROUTINE

BEGIN { Program exemplo }
  STATEMENT
END.

```

Figura 2.4: Visão do programa "exemplo"

exemplo : [program]

```

SCOPE *INITIAL[1] [block]
  x      : [variable] TYPE(integer)
  y      : [variable] TYPE(integer)
  gcd    : [function] TYPE(PROC)

  SCOPE gcd [subprogram]
    gcd   : [return] TYPE(integer)
    a     : [variable] TYPE(integer)
    b     : [variable] TYPE(integer)

```

Neste exemplo, a variável "b" de tipo inteiro pertencente ao escopo da subrotina "gcd" aparece em destaque pois é a variável corrente na visão que representa o programa na Figura 2.4.

A visão tipo de dados

Os tipos de dados utilizados no programa podem apresentar-se através desta visão. Ao editar um nó da árvore que descreve um tipo, a visão mostra como este é composto. Se o nó corrente que está sendo alterado for uma variável, a visão mostra qual é o seu tipo. Para uma expressão modificada, é apresentado o tipo desta expressão. Estas informações são mostradas em uma primeira linha da visão que descreve a sua classe, por exemplo, RECORD, POINTER ou ARRAY. Nas linhas seguintes são apresentados recursivamente os tipos dos elementos que os compõem, possibilitando ao usuário escolher até em que nível de recursividade a estrutura deva ser exibida.

Se o usuário escolher uma definição de tipo da visão, esta será inserida na posição corrente do programa.

Para a visão do programa "exemplo" da Figura 2.4 representa-se na visão Tipo de Dado a seguinte informação:

```
integer: INTEGER (16)
```

A visão Expressão

Através desta visão é possível representar graficamente a árvore de uma expressão que está sendo editada. Destaca-se o elemento corrente do programa na árvore correspondente, mostrando ao usuário uma perspectiva diferente de toda a expressão e a atual localização deste elemento. A escolha de um nó na árvore torna-o corrente na expressão correspondente do programa.

A visão expressão para o programa "exemplo" representado na Figura 2.4 é a seguinte:

A visão Fluxo

O compilador incremental constrói e mantém uma representação gráfica do programa baseada no Diagrama de Controle de Fluxo. Nestes diagramas representa-se instruções de entrada e saída, estruturas de controle de fluxo e as subrotinas do programa corrente. As atualizações no programa são dinamicamente mantidas na sua representação gráfica de fluxo. Pode-se transformar um nó da árvore que representa o programa em nó corrente, escolhendo a sua respectiva representação no diagrama de fluxo.

Visões da Execução

Durante a execução de um programa, o PECAN oferece um conjunto de visões que permitem seu acompanhamento. Estas visões são proporcionadas pelo interpretador.

Uma destas visões emite mensagens do estado atual da execução, mensagens de erros e possibilita a entrada e saída dos dados para o programa corrente. Um exemplo do conteúdo desta visão para um trecho de programa que lê dois números inteiros, imprime uma mensagem e tem a execução interrompida pelo usuário é:

```
>>> Normal termination
>>> Continue execution ...
>>> Normal termination
>>> Program is ready to run
>>> Begin execution ...
135 63
Result is
>>> User halted execution
```

Alguns recursos de depuração também são oferecidos pelo PECAN. Através de visões é possível monitorar a execução reversa dos programas, passo a passo, mostrando um comando por vez. Nesta visão é permitido também avançar ou retornar à execução, definir pontos de parada e controlar a sua velocidade. As estruturas que são executadas podem ser observadas através das visões do programa ou do diagrama de fluxo, bastando para isto ativar a visão escolhida.

Também poderá ser requisitada pelo usuário uma visão contendo uma pilha dos dados que são utilizados durante a execução. Esta pilha contém as informações separadas pelos blocos do programa que vão sendo executados. A visão contendo a pilha de dados executados é dividida em duas partes: a parte da esquerda apresenta o começo de cada bloco, as variáveis nele contidas e seus respectivos valores; a parte da direita apresenta os valores para as variáveis cuja composição é mais complexa, como por exemplo, variáveis do tipo RECORD, ARRAY e POINTERS. Um exemplo desta visão para o programa da Figura 2.4 é mostrado a seguir:

STACK	DATA
Program	
x	135
y	63
Function teste	
teste	<UNDEFINED>
a	135
b	63
Function teste	
teste	<UNDEFINED>
a	63
b	9
Function teste	
teste	<UNDEFINED>
a	9
b	8

Neste exemplo o símbolo indefinido "teste" aparece rotulado como tal, variáveis inteiras como x, y, a e b aparecem com os valores que assumem durante a execução. Para este exemplo nenhuma variável complexa (RECORD, POINTER ou ARRAY) é executada, portanto o lado direito desta visão não está sendo ocupado. Caso contrário, deste lado apareceriam os valores assumidos pelos elementos que compõem a estrutura.

2.7 - Considerações Finais.

Uma característica importante desses Ambientes encontra-se na estrutura de dados utilizada para o armazenamento de seus arquivos. Através dessa estrutura, uma variedade de recursos são oferecidos, permitindo uma maior praticidade aos usuários. Em alguns casos, programas sintaticamente corretos e semanticamente verificados são possíveis ao final da edição, graças à interação entre o ambiente e o usuário, proporcionando maior rapidez e produzindo programas diretamente compilados. Oferecem-se sofisticados recursos de edição, execução e depuração devido ao armazenamento de informações sintáticas e semânticas da linguagem de programação utilizada pelo Ambiente. Finalmente, uma contribuição importante é a geração de instâncias de ambientes orientados para estrutura,

a partir da descrição de propriedades semânticas e sintáticas de uma linguagem através de gramáticas, introduzindo a noção de geradores de ambientes.

Capítulo 3

PROTÓTIPO DO EDITOR GENÉRICO SENSÍVEL A SINTAXE

3.1 - Considerações Iniciais.

O Protótipo do Editor Genérico Sensível à Sintaxe (PEGSS) tem como características oferecer suporte à edição sensível à sintaxe das linguagens por ele suportadas e permitir que programas escritos em uma linguagem possam ser representados através das suas outras linguagens. Este Capítulo apresenta estas características, através de uma visão do usuário, onde se aproveita para mostrar as vantagens e limitações desta versão do PEGSS.

3.2 - Descrição do Protótipo EGSS.

Um objetivo deste trabalho é construir um Protótipo de um Editor Genérico Sensível à Sintaxe armazenada em uma base de dados. É denominado genérico porque permite a edição de programas em qualquer linguagem entre as várias suportadas pelo Editor. Além disso, uma vez editado o programa na linguagem escolhida, é possível obter-se o mesmo programa representado em qualquer uma das outras linguagens, dentro do que seja possível para a tradução de um programa escrito em uma linguagem, para a outra. Diz-se sensível à sintaxe por ter o conhecimento da construção sintática da linguagem utilizada na edição do programa, possibilitando, ao final, um programa livre de erros léxicos e sintáticos. Além das características descritas acima, o Editor é dirigido por estruturas, pois a edição é feita, basicamente, através da requisição de gabaritos que são alocados em uma composição hierárquica de objetos sintáticos, ao invés de uma sequência de caracteres. A construção dessa estrutura, durante o processo de edição, permite apresentar o programa

de modo estruturado, onde as estruturas sintáticas do mesmo nível estão uma abaixo da outra e as dos níveis mais inferiores aparecem obedecendo a uma endentação.

Os programas editados são armazenados em uma base de dados, segundo a concepção lógica de uma construção sintática genérica para as linguagens suportadas pelo PEGSS, denominada de Descrição da Linguagem Abstrata (DLA). Através da DLA é possível obter-se a representação de um programa em qualquer uma das linguagens, pois um programa é armazenado de modo único para todas elas.

Este trabalho capacitou o PEGSS a suportar a edição sensível à sintaxe de um subconjunto considerável das linguagens Pascal, C e FORTRAN. A escolha dos subconjuntos das construções sintáticas das linguagens foi efetuada de modo que fossem representadas as suas principais características e que a adição das demais estruturas seja feita sem alterar o que está implementado.

A opção por essas linguagens deve-se basicamente a: as construções sintáticas dessas linguagens são em muitos casos parecidas - isso possibilita uma representação genérica das suas estruturas sintáticas; e devido à grande utilização dessas linguagens na implementação de projetos de software.

Estas características visam fazer do PEGSS uma ferramenta de suporte à fase de codificação do processo de desenvolvimento de software, possibilitando que programas sintaticamente corretos sejam escritos sob o seu controle e que módulos escritos em uma linguagem possam ser utilizados para a composição de outros, em outra linguagem.

3.3 - Edição.

O Protótipo do EGSS é uma combinação de editor orientado para estrutura e editor de texto. As expressões e comandos de atribuição são introduzidos digitando-se caractere por caractere, enquanto as demais estruturas da linguagem são oferecidas como gabaritos, através dos menus de opções.

Um gabarito representa uma estrutura sintática da linguagem composto por palavras-chave e lugares para as possíveis introduções de outros gabaritos ou expressões. Os lugares para as novas introduções definem a classe sintática das possíveis inserções, evitando erros sintáticos na introdução de novas estruturas. Erros léxicos também são evitados, pois não é permitido ao programador alterar as palavras-chave que compõem o gabarito, além da verificação léxica das expressões inseridas. Por exemplo, um gabarito de uma estrutura

desvio condicional utilizada pelo PEGSS é:

```
if ( <condição> )  
  then  
    <bloco>  
  else  
    <bloco>
```

onde as palavras <condição> e <bloco> formam os lugares para as possíveis inserções e o "if", "then" e "else" são palavras-chave.

O PEGSS permite criar novos arquivos ou carregar arquivos já existentes, mesmo que tenham sido editados sob o contexto de outra linguagem.

Após a escolha da linguagem e introdução do nome do arquivo, o processo de criação de um programa é iniciado a partir de um primeiro gabarito, oferecido pelo Editor. Considerando que a linguagem corrente é o Pascal, o gabarito inicial será o seguinte:

```
program <nome>;  
  <declaração>  
  <decl.subrot.>  
  <bloco>
```

Este primeiro gabarito aparece com a palavra-chave e os lugares de inserção em destaque ("highlight") indicando que esta é a estrutura corrente no momento. A tecla "return" é utilizada para as inserções de novos gabaritos ou expressões e, se pressionada nesta posição, não causará efeito algum, pois tais inserções devem ser efetuadas quando o elemento corrente for um lugar de inserção. Consequentemente, as palavras-chave não são acessíveis ao usuário impossibilitando-o de quaisquer tentativas de alterá-las.

A movimentação ("navegação") sobre o gabarito é proporcionada através da utilização das teclas "seta para cima", "para baixo", "para a esquerda", "para a direita", "home" e "end". Se a tecla "seta para a direita" for pressionada, a palavra que ficará em destaque será <nome> e, pressionando a tecla "seta para baixo", a palavra em destaque

será <declaração>. Repetindo o último passo duas vezes o lugar de inserção posicionado será o <bloco>. Esta sequência de movimentação sobre o gabarito é mostrado abaixo, através de (1), (2), (3) e (4):

<pre> program <nome>; <declaração> <decl.subrot.> <bloco> </pre> (1)	<pre> program <nome>; <declaração> <decl.subrot.> <bloco> </pre> (2)
<pre> program <nome>; <declaração> <decl.subrot.> <bloco> </pre> (3)	<pre> program <nome>; <declaração> <decl.subrot.> <bloco> </pre> (4)

Ao pressionar a tecla "return", um novo gabarito será apresentado em seu lugar:

```

program <nome>;
  <declaração>
  <decl.subrot.>
  begin
    <comando>
  end.

```

Após posicionar no lugar de inserção <comando> e pressionar a tecla "return", é apresentado um menu com as opções de escolha entre comandos com ou sem rótulos. A escolha de comandos sem rótulos proporciona a apresentação de um outro menu de opções contendo as possíveis estruturas sintáticas a serem introduzidos neste ponto, como é apresentado a seguir:

```

program <nome>;
  <declaração>
  <decl.subrot.>
  begin
    <comando>
  end.

```

atribuição
chamada subr.
for/do
goto
if/then
if/then/else
retorno de func.
repeat
while/do
nenhuma....

Escolhida a estrutura, um gabarito, representando-a, será colocado no lugar de <comando>. Se a estrutura escolhida for "while/do", o programa ficará do seguinte modo:

```

program <nome>;
  <declaração>
  <decl.subrot.>
  begin
    while( <condição> )do
      begin
        <comando>
      end;
    <comando>
  end.

```

Sempre que um lugar de inserção é substituído por um gabarito, um outro lugar será oferecido pelo Editor, para que a construção do programa possa prosseguir através de inserções de mais estruturas sintáticas, pertencentes à mesma classe sintática do lugar substituído. No exemplo acima, o lugar de inserção <comando> foi gerado pelo Editor, após a inserção da estrutura selecionada.

Os lugares de inserção como <nome> e <condição> permitem que identificadores e expressões sejam, respectivamente, introduzidos. Este processo é efetuado através de uma janela na parte inferior do vídeo que permite a introdução dos caracteres que os compõem.

Após sua descrição, estes serão substituídos nos seus respectivos lugares de inserção. Na introdução de um identificador já utilizado no programa é emitida uma mensagem avisando o programador de tal fato. Um exemplo de inserções de um nome e expressão é apresentado no trecho de programa que segue:

```
program <nome>;  
  <declaração>  
  <decl.subrot.>  
  begin  
    while( <condição> )do  
      begin  
        <comando>  
      end;  
    <comando>  
  end.
```

nome: teste

posicionando o cursor até a palavra de inserção <condição>:

```
program <nome>;  
  <declaração>  
  <decl.subrot.>  
  begin  
    while( <condição> )do  
      begin  
        <comando>  
      end;  
    <comando>  
  end.
```

expressão: a>b+c

após pressionar a tecla "return":

```
program teste;  
  <declaração>  
  <decl.subrot.>  
  begin  
    while( [a>b+c] ) do  
      begin  
        <comando>  
      end;  
    <comando>  
  end.
```

Deste modo o programa é escrito através do processo "top-down", onde as introduções são dirigidas pelos lugares de inserção definindo a correção sintática do mesmo. Somente a análise sintática das expressões ainda não é efetuada nesta versão do PEGSS, ocorrendo apenas a sua verificação léxica.

A eliminação de estruturas em um programa só é possível, na atual versão do PEGSS, para as expressões. Isto é feito do mesmo modo para a introdução de uma nova expressão, onde a estrutura corrente deve ser a expressão que será eliminada dando início à sua substituição por outra.

3.4 - Tradução de Programas de uma Linguagem para Outra.

No PEGSS um programa pode ser editado sob o domínio de qualquer uma das linguagens por ele suportadas. Enquanto o programa é construído, informações sobre as estruturas que os compõem são armazenadas na base de dados, segundo a construção sintática genérica, definida pela Descrição da Linguagem Abstrata (DLA). Portanto, todo programa, independentemente da linguagem em que foi escrito, estará armazenado na base de dados de forma única. Esta forma de armazenamento permite que um programa editado sob o contexto de uma linguagem possa ser traduzido para qualquer outra linguagem disponível no Editor.

Algumas estruturas sintáticas existentes na linguagem fonte (linguagem utilizada durante o processo de criação do programa) não possuem representação na linguagem destino da tradução (linguagem em que se deseja ter o programa representado). Este problema é resolvido pelo PEGSS de três maneiras:

- . conversão em estruturas sintáticas equivalentes na linguagem destino ou
- . mensagem indicando que a estrutura sintática não possui representação na linguagem destino ou
- . a estrutura sintática é desprezada na linguagem destino.

Exemplos de estruturas com as características descritas acima, serão apresentadas nas sub-seções que seguem.

3.4.1 - Estruturas Sintáticas de Controle de Fluxo.

Entre as estruturas sintáticas de controle de fluxo, existem as que possuem conversão imediata de uma linguagem para outra, quando, para as linguagens, estas estruturas são sintaticamente semelhantes. Um exemplo, entre outros implementados pelo PEGSS, é a estrutura de desvio condicional, denominada, pelo Editor, de "ifte", representada na Figura 3.1a e 3.1b. Nesta Figura são mostradas outras estruturas, através dos seus respectivos gabaritos que as descrevem em cada uma das linguagens suportadas pelo Editor. Para a maioria delas, a diferença é de apenas algumas palavras chave que constituem seus gabaritos.

Existem estruturas cuja representação nas outras linguagens não é imediata, pois nelas não há estruturas sintaticamente parecidas, mas permitem que uma combinação de algumas de suas estruturas possam representá-las. As estruturas "repeat", "do", "while" e "ifa", que aparecem na Figura 3.1a e 3.1b, possuem esta característica. Por exemplo, a estrutura "repeat" é obtida na linguagem FORTRAN, através de uma composição de algumas de suas estruturas sintáticas.

Os gabaritos que representam estas combinações sintáticas são utilizados pelo Editor, durante a tradução do programa de uma linguagem para outra e não estão disponíveis no menu de opções de estruturas de controle da linguagem oferecidas pelo Editor, durante a edição.

A estrutura "repeat", na linguagem C, possui, no gabarito que a representa, o operador "!" antes do lugar de inserção <condição>. Como este operador faz parte do gabarito, não será permitido ao programador, alterá-lo. Caso contrário, por exemplo, a eliminação deste operador, provocaria uma inconsistência na informação, que ficaria armazenada na base de dados a respeito desta estrutura. Outros gabaritos possuem esta característica, como por exemplo, o "repeat" na linguagem FORTRAN, o "do" e o "ifa" na

nome da estrutura	Pascal	C	FORTRAN

ifte	if(<condição>)
	then
	<bloco>
	else
	<bloco>
		if(<condição>)
		<bloco>
		else
		<bloco>
			if(<condição>) \$then
			<bloco>
			else
			<bloco>
			endif

ifa	if(<expr.> = 0)
	then
	goto <rotulo>
	else
	if(<expr.> > 0)
	then
	goto <rotulo>
	else
	goto <rotulo>;
		if(<expr.> == 0)
		goto <rotulo>
		else
		if(<expr.> > 0)
		goto <rotulo>;
		else
		goto <rotulo>;
			if(<expr.>) <rotulo>,
			<rotulo>, <rotulo>

ifl	if(<condição>) then <comando>	if(<condição>) <comando>	if(<condição>) <comando>
case	case <expr.> of <const:comando> end;	switch(<expr.>){ <const:comando> <const:comando> }	goto(<rótulos>), <expr.> goto <rotulo> <rotulo> continue

Figura 3.1a: Gabarito com as estruturas sintáticas das linguagens Pascal, C e FORTRAN.

linguagem Pascal, o "ifa" na linguagem C, e outros. Estes gabaritos estão representados na Figura 3.1a e 3.1b.

Algumas estruturas, como por exemplo a "gotoc" na linguagem FORTRAN, não possuem estruturas equivalentes, nem permitem que combinações de estruturas sintáticas existentes nas outras linguagens possam representá-las. Neste caso, quando programas editados sob o contexto da linguagem FORTRAN, contendo estruturas como esta, são traduzidos para as outras linguagens, no lugar destas estruturas, aparecerá uma mensagem indicando sua inexistência na linguagem destino.

A estrutura "ifl" apresenta uma característica diferente das demais. Na linguagem FORTRAN e no seu respectivo gabarito, o lugar de inserção representado pela palavra <comando> permite que apenas uma entre as estruturas "goto", "gotoc", "atrib" ou "cham_sub" possa ser escolhido em um menu de opções, durante o processo de edição. Nas

nome da estrutura	Pascal	C	FORTTRAN

repeat	repeat
	<bloco>
	until(<condição>);
		do
		<bloco>
		while(!<condição>);
			<rótulo>continue
			<bloco>
			if(.NOT.<condição>)
			goto<rótulo>
			endif

do	repeat
	<bloco>
	until(not<condição>);
		do
		<bloco>
		while(<condição>);
			<rótulo>continue
			<bloco>
			if(<condição>)goto<rótulo>
			endif

while	while(<condição>)do <bloco> <bloco>	while(<condição>) <bloco>	<rot>if(.NOT.<condição>) goto<rótulo> <bloco> goto<rótulo> <rótulo>continue endif
gotoc	_____	_____	goto(<rótulos>)
goto	goto<rótulo>;	goto<rótulo>;	goto<rótulo>
atrib	<atribuição>;	<atribuição>;	<atribuição>
cham_sub	<nome>(<lista expr.>);	<nome>(<lista expr.>);	call<nome> (<lista expr.>)

Figura 3.1b: Continuação dos gabaritos com as estruturas sintáticas das linguagens Pascal, C e FORTRAN.

construções sintáticas das linguagens Pascal e C este tipo de restrição não é possível, pois este define uma classe sintática composta por todas as estruturas de controle. No entanto, na tradução desta estrutura em programas editados sob o contexto da linguagem FORTRAN para as linguagens Pascal ou C são utilizados os correspondentes gabaritos, como é mostrado na Figura 3.1a e 3.1b. Durante a edição, após esta tradução, não é permitida a introdução, no lugar de injeção <comando>, destes gabaritos, sob o contexto da linguagem Pascal ou C. Com isto, consegue-se a tradução desta estrutura da linguagem FORTRAN para as outras linguagens, sem que suas construções sintáticas possam ser afetadas durante a seção de edição.

A tradução da estrutura sintática "for" é feita após uma análise de suas expressões.

Dependendo de quais tenham sido as expressões introduzidas nesta estrutura sob o domínio de uma linguagem, poderá ou não ocorrer a tradução, em uma estrutura equivalente na linguagem destino.

Na Figura 3.2 são mostrados alguns exemplos das possíveis traduções suportadas pelo PEGSS para a estrutura "for". Na última coluna está indicada a linguagem sob a qual foi introduzida a estrutura de controle, através das letras q (qualquer linguagem), C (linguagem C) e F (linguagem FORTRAN).

É possível efetuar algumas traduções da estrutura "for", quando editadas sob o domínio de uma linguagem, mas pode ser impossível determiná-las quando editadas sob o contexto de outra linguagem. Por exemplo, a tradução 6 da Figura 3.2, onde a estrutura "for" é editada sob o domínio da linguagem C e a tradução 7, onde a estrutura é editada sob o contexto da linguagem FORTRAN. No caso da tradução 7, a impossibilidade de tradução para a respectiva estrutura na linguagem C, consiste em determinar sua expressão de parada, pois a análise feita pelo Editor, durante a edição desta estrutura, não permite decidir se a expressão que define o passo de sua variável de controle é de incremento ou decremento.

A estrutura "for", editada sob o contexto da linguagem C, como por exemplo as representadas nas traduções 13 e 14 da Figura 3.2, possui em suas expressões, uma sintaxe não suportada pelas outras linguagens. Portanto, sua tradução é impossibilitada pelo Editor.

Nos casos em que não é possível traduzir as expressões que compõem uma estrutura "for", será apresentado apenas o seu gabarito, com os respectivos lugares de inserção <expr>. Deste modo, fica representado, no programa traduzido que, naquele ponto, foi impossibilitada a representação de uma estrutura "for", ficando a critério do usuário redefini-lá ou não. Portanto, para a tradução 14 da Figura 3.2, as respectivas representações, para o exemplo das estrutura "for" escrito sob o controle da linguagem C, seriam as seguintes:

Pascal	C
for<expr>to<expr>do <bloco>	for(;i<a+b;i++) <bloco>
 FORTRAN 	
do<rótulo>,<expr>,<expr>,<expr> <rótulo>continue	

Pascal	C	FORTRAN	Edição
1. for i=1 to 10 do <bloco>	for (i=1;i<=10;i=i+1) <bloco>	do x,i=1,10,1 <bloco> x continue	q
2. for i=1 to 10 do <bloco>	for (i=1;i<=10;i++) <bloco>	do x,i=1,10,1 <bloco> x continue	C
3. for i=c to a+b do <bloco>	for (i=c;i<=a+b;i=i+1) <bloco>	do x,i=c,a+b,1 <bloco> x continue	q
4. for i=c to a+b do <bloco>	for (i=c;i<=a+b;i=i+1) <bloco>	do x,i=c,a+b <bloco> x continue	F
5. for i=c to a+b do <bloco>	for (i=c;i<=a+b;i+=1) <bloco>	do x,i=c,a+b,1 <bloco> x continue	C
6. _____	for(i=c;i<=a+b;i=i+(e+f)) <bloco>	do x,i=c,a+b,e+f <bloco> x continue	C
7. _____	_____	do x,i=c,a+b,e+f <bloco> x continue	F
8. for i=c to b-1 do <bloco>	for (i=c;i<b;i++) <bloco>	do x,i=c,b-1,1 <bloco> x continue	C
9. _____	for(i=c;i>=a+b;i=i-(e+f)) <bloco>	do x,i=c,a+b,-(e+f) <bloco> x continue	q
10. _____	for (i=c;i>=a+b;i=i-1) <bloco>	do x,i=c,a+b,-1 <bloco> x continue	q
11. _____	for (i=c;i>=a+b;i--) <bloco>	do x,i=c,a+b,-1 <bloco> x continue	C
12. _____	for (i=c;i>=a+b;i-=e) <bloco>	do x,i=c,a+b,-e <bloco> x continue	C
13. _____	for(i=c;j=d;i<5;j<7;i++j++) <bloco>	_____	C
14. _____	for(;i<a+b;i++) <bloco>	_____	C

Figura 3.2: Traduções da estrutura de controle "for".

onde apenas é apresentado, em Pascal e em FORTRAN, o gabarito da estrutura "for" definida em C.

3.4.2 - Subrotinas.

São consideradas subrotinas pelo PEGSS as funções definidas pela linguagem C o programa principal, os procedimentos e as funções definidos pela linguagem Pascal e FORTRAN.

A sintaxe da linguagem Pascal permite às subrotinas definirem novas subrotinas sob seu escopo. Tal característica não é compartilhada pelas sintaxes das linguagens C e FORTRAN, onde todas subrotinas são definidas no escopo global de um arquivo (programa).

Na tradução de programas editados sob o contexto da linguagem Pascal para as outras linguagens a característica descrita acima é levada em consideração. Uma subrotina que foi definida dentro do escopo de outra subrotina será omitida na tradução para as outras linguagens.

Um exemplo de como ficaria está tradução para o programa "teste", editado sob o contexto da linguagem Pascal, é mostrado a seguir. O programa "teste" é um programa em que a maioria de seus lugares de inserção não foram preenchidos, não estando, portanto, concluída sua edição, o que não impossibilita a sua tradução para as outras linguagens.

```
program teste;  
  <declaração>  
  procedure procl( <decl.param.> )  
    <declaração>  
    function func1( <decl.param.> ): real;  
      <declaração>  
      <decl.subrot.>  
      <bloco>  
    <decl.subrot.>  
    <bloco>  
  <decl.subrot.>  
begin  
  a := 0;  
  procl( <lista expr.> );  
  <comando>  
end.
```

Programa "teste" obtido sob o contexto da linguagem C:

```
<define const.>
<define tipo>
<declara var.>
void procl( <lista param.> )
    <decl.param>
    {
        <define const.>
        <define tipo>
        <declara var.>

        <comando>
    }
<decl.função>
main()
{
    <define const.>
    <define tipo>
    <declara var.>

    a = 0;
    procl( <lista expr.> );
    <comando>
}
```

Representação do programa "teste" sob o contexto da linguagem FORTRAN:

```
program teste
  <declaração>

  a = 0
  proc1( <lista expr.> )
  <comando>
stop
end
subroutine proc1( <lista param.> )
  <declaração>
  <bloco>
<decl.subrot.>
```

Nota-se que a subrotina "func1" definida no escopo da subrotina "proc1" é totalmente omitida, nesta tradução. O mesmo processo ocorrerá, se este programa for requisitado, sob o contexto da linguagem FORTRAN. Deve-se observar também um outro detalhe considerado pelo Editor nas traduções deste exemplo, o tipo do retorno da subrotina "proc1", quando apresentado sob o contexto da linguagem C é fixado como "void", pois esta subrotina foi definida para não retornar valor algum.

Um outro problema tratado pelo PEGSS na tradução de subrotinas é a expressão de retorno de uma função. No Pascal e FORTRAN, a expressão retornada pela função é especificada através de uma estrutura de atribuição, onde a variável à esquerda da atribuição é o identificador da função. Na linguagem C a expressão de retorno é descrita através da estrutura sintática "return", sem que seja especificado o identificador da função a que pertence. Portanto, na tradução de uma estrutura "return", escrita na linguagem C, para as respectivas representações em Pascal e FORTRAN, o Editor deve contornar tal situação, para que estas possam ser efetivadas. A solução encontrada para este caso e implementada nesta versão do PEGSS é a de apresentar uma expressão de atribuição, onde o identificador da função é apenas indicado, deixando para o usuário defini-lo corretamente, como é apresentado no exemplo de tradução a seguir:

Pascal	C	FORTRAN
<code><ident.> := n*(n-1);</code>	<code>return(n*(n-1));</code>	<code><ident.> = n*(n-1)</code>

onde, a palavra `<ident.>` é colocada na atribuição pelo Editor.

Na solução do contrário, ou seja, quando a estrutura de atribuição que descreve o retorno de uma função deve ser introduzida pelo programador, sob o domínio da linguagem Pascal ou FORTRAN, isto é efetuado através de um gabarito oferecido pelo menu de opções que permite ao Editor identificar que a expressão introduzida é o retorno de uma função, facilitando a tradução. O exemplo abaixo mostra como ficaria o retorno de uma função para as linguagens suportadas pelo Editor:

Pascal	C	FORTRAN
<code>fatorial := n*(n-1);</code>	<code>return(n*(n-1));</code>	<code>fatorial = n*(n-1)</code>

onde, o identificador "fatorial" da função é omitido na representação desta estrutura, sob o contexto da linguagem C.

3.4.3 - Definição de Constante, Definição de Tipo, Declaração de Variável e de Parâmetro.

As linguagens suportadas pelo PEGSS possuem estruturas sintáticas muito semelhantes para a definição de constante, o que facilita a conversão destas estruturas de uma linguagem para outra. A seguir, são apresentados alguns exemplos de traduções de definições de constantes, nas linguagens suportadas pelo Editor:

Pascal	C	FORTRAN
<code>TAM_MAX = 10;</code>	<code>#define TAM_MAX 10</code>	<code>parameter(TAM_MAX = 10)</code>
<code>LIN = A+B;</code>	<code>#define LIN A+B</code>	<code>parameter(LIN = A+B)</code>

Na linguagem C, a estrutura utilizada para a definição de constante permite definir, muito mais do que apenas associar, um identificador a uma expressão numérica ou literal, mas para limitar a complexidade, nesta atual versão do PEGSS e possibilitar a tradução entre as suas linguagens, esta estrutura está restrita à definição de constantes, quando utilizada sob o contexto desta linguagem.

A tradução de declaração de variável é imediata para as variáveis de tipos básicos (inteiro, real, caracter, etc) e, mais elaborada, para variáveis de tipos estruturados (vetor, registro, etc). Alguns exemplos de traduções de declarações de variáveis suportadas pelo PEGSS são mostradas a seguir:

Pascal	C	FORTRAN
a:integer;	int a;	integer a
b:char;	char b;	character*1 b
c1:array[0..9]of real;	float c1[10];	real c1(0:9)
c2:array[1..9]of real;	float c2[<indices>];	real c2(1:9)
c3:array[0..k]of real;	float c3[k-1];	real c3(<indices>)
c4:array[a..k]of real;	float c4[<indices>];	real c4(<indices>)
c5:array[0..3,0..3]of integer;	int c5[4,4];	int c5(0:3,0:3)
d:record e : integer; <decl.campos> end;	struct(int e; <decl.campos>) d;	_____
f:boolean;	int f;	logical f
_____	_____	character*5 g
h:vetor;	vetor h;	<tipo> h

Os identificadores de tipos são traduzidos para as representações equivalentes nas linguagens em declarações envolvendo tipos básicos, mantendo-se o nome das variáveis. Para as declarações que utilizam o tipo vetor, a tradução é efetuada na expressão que define o seu índice, no seu tipo, e na própria descrição do tipo vetor. Nas declarações cujos tipos das variáveis são registros, a tradução ocorre na descrição do tipo registro e nas declarações de seus campos, sendo que uma declaração desta será omitida, quando a

tradução é feita sob o contexto da linguagem FORTRAN, pois este tipo não é definido para esta linguagem. Para as traduções envolvendo identificadores de tipos definidos pelo usuário entre as linguagens Pascal e C, o identificador é conservado junto com o nome da variável. Já para as traduções deste tipo para a linguagem FORTRAN, o identificador de tipo é omitido, mostrando apenas o nome da variável.

As linguagens Pascal e C permitem a definição de tipos novos, o que não acontece com a linguagem FORTRAN. Logo, as definições de tipos, em programas escritos em Pascal ou C, serão omitidas quando traduzidas para FORTRAN. Seguem alguns exemplos de traduções possíveis no PEGSS:

Pascal	C	FORTRAN
inteiro=integer;	typedef int inteiro;	_____
vetor=array[0..5]of integer;	typedef int vetor[6];	_____
a=array[1..5]of integer;	typedef int vetor[<índice>];	_____
pessoa=record	typedef pessoa struct (	
idade:integer;	int idade;	_____
sexo:char;	sexo char;	
end;	);	

A tradução das definições de tipos novos é feita de acordo com a sintaxe utilizada para descrevê-las, em cada linguagem, mantendo-se o identificador de tipo novo e traduzindo-se, como nas declarações, a representação dos tipos definidos pela linguagem.

A definição da lista formal dos parâmetros de uma subrotina, escrita em Pascal, é efetuada, durante a definição desta subrotina, onde são utilizados apenas os identificadores de tipos, para definir o tipo de cada um destes parâmetros. Todavia, esta restrição não se emprega para as declarações de parâmetros nas subrotinas escritas, ou traduzidas em C e FORTRAN, devendo, portanto, ser consideradas nas traduções, ou edição de programas que contêm definições de subrotinas. No exemplo apresentado a seguir, mostra-se como ficam as declarações dos parâmetros da subrotina de nome "func1" escrita na linguagem C, traduzida para a sua respectiva representação, nas linguagens Pascal e FORTRAN:

C	FORTRAN

<pre>int func1(a, s)
  int a;
  float s[10];
  {
    .
    .
    .
  }
}</pre>
	<pre>integer func1(a, s)
	  integer a
	  real s(0:9)
	  .
	  .
	  .
	  return
	end</pre>

Pascal

```
func1( a:integer; <decl.param.> ): integer;  
begin  
  .  
  .  
  .  
end;
```

Neste exemplo, a declaração do parâmetro de nome "s" é omitida, quando a subrotina é mostrada sob o contexto da linguagem Pascal, o que não ocorre, quando a linguagem corrente é o FORTRAN ou C. As declarações que não envolvem tipos estruturados são traduzidas para a respectiva representação do identificador de tipo, na linguagem corrente, como é o caso do parâmetro de nome "a". Deve-se notar, também, que nas linguagens C e FORTRAN um parâmetro é especificado na lista formal de parâmetros e na descrição da sua declaração; porém, na linguagem Pascal, este parâmetro é descrito apenas na lista formal de parâmetros. Logo, para cada parâmetro declarado em um programa escrito em Pascal, o Editor deve apresentá-lo, nestes dois modos, em uma subrotina traduzida para a sua respectiva representação em C e FORTRAN.

As definições de constantes, tipos e declarações de variáveis definidas no escopo da função "main", em um programa escrito na linguagem C, não estarão representadas por estruturas equivalentes, nas traduções para a linguagem Pascal ou FORTRAN, pois, para estas linguagens, as definições e declarações no escopo de suas subrotinas principais são globais ao programa e não apenas locais à subrotina principal, como ocorre em C. As

definições e declarações globais, em programas escritos em Pascal ou FORTRAN, serão traduzidos em suas equivalentes definições e declarações na linguagem C, porém globais ao programa. Um exemplo, mostrando esta situação, é apresentado na tradução abaixo:

Considere-se o trecho de programa P escrito na linguagem C:

```
<define const.>
<define tipo>
int a;
float b;
<declara var.>
<decl.função>
main()
{
    <define const.>
    <define tipo>
    int e;
    float i;
    <declara var.>

    i = 0;
    i = i + e * 5 - a / b;
    <comando>
}
```

Tradução do programa P para a linguagem Pascal:

```
program <nome>;
label <decl.rotulo>;
const
    <def.const.>
type
    <def.tipo>
```

```
var
  a : integer;
  b : real;
  <declara var.>
<decl.subrot.>
begin
  i := 0;
  i := i + (e * 5) - (a / b);
  <comando>
end.
```

Tradução do programa P para a linguagem FORTRAN:

```
program <nome>
  integer a
  real b
  <declara var.>
  <def.const>

  i = 0
  i = i + (e * 5) - (a / b);
  <comando>
  stop
end
<decl.subrot.>
```

Neste exemplo, as variáveis declaradas globalmente, de nomes "a" e "b" no programa escrito em C, são traduzidas para o escopo da subrotina principal dos programas escritos em Pascal e FORTRAN, mantendo-se como variáveis globais, enquanto as variáveis de nomes "e" e "i" declaradas como locais à função "main()" do programa escrito em C, são omitidas, nas respectivas traduções em Pascal e FORTRAN, pois tais declarações não são possíveis para estas linguagens.

3.4.4 - Expressões.

Como já mencionado anteriormente, a introdução de expressões é feita, digitando-se caractere por caractere. Após a sua inserção, uma análise léxica é feita, evitando que expressões lexicamente incorretas sejam introduzidas.

Uma vez aceita a expressão pelo Editor, seus operadores são convertidos em operadores genéricos, isto é, operadores únicos para as linguagens suportadas pelo PEGSS, obtendo-se uma expressão genérica que será armazenada na base de dados.

A tradução de uma expressão escrita em uma linguagem para outra é efetuada através da conversão dos operadores genéricos nos respectivos operadores da linguagem destino da tradução, mantendo-se a prioridade entre os operadores, definida durante a edição da expressão.

Considere-se a expressão abaixo escrita na linguagem C:

$$a + = b + c \% d - 10 \quad (1)$$

A expressão (1) convertida para a linguagem Pascal:

$$a \#27\# ((b + (c \text{ mod } d)) - 10)$$

A expressão (1) convertida para a linguagem FORTRAN:

$$a \#27\# ((b + (c \#6\# d)) - 10)$$

Um operador genérico é representado através do símbolo "#num#", onde num é um número inteiro e único para cada operador.

Os operadores que não possuem representação na linguagem destino da tradução são mantidos na forma genérica.

Nota-se que a prioridade entre os operadores na expressão é conservada, mesmo com os operadores genéricos resultantes da sua tradução, com a inserção dos parênteses (ver Capítulo 6).

Uma expressão obtida após uma tradução poderá ser editada normalmente, com prejuízo restrito à sua análise sintática, pois, se nesta expressão houver operadores

genéricos, estes não serão entendíveis léxica e sintaticamente pela linguagem corrente. Uma solução é efetuar a análise sintática apenas nas expressões que não possuam operadores genéricos. Na atual implementação do PEGSS, a análise sintática das expressões, ainda não foi implementada.

3.5 - Considerações Finais.

Neste Capítulo foram apresentadas as principais estruturas sintáticas manipuladas pelo PEGSS, descrevendo-se a maneira como a representação interna unificada pode ser utilizada para representar programas escritos em diferentes linguagens, e como essa representação interna pode ser novamente traduzida para uma das linguagens Pascal, C ou FORTRAN.

Embora apenas três linguagens tenham sido contempladas, o Editor é extensível, e outras linguagens que sigam paradigmas semelhantes podem ser suportadas.

Deve-se notar que a forma de armazenamento interno é genérica a ponto de suportar a representação de qualquer estrutura sintática de qualquer das três linguagens contempladas. Note-se que existem determinadas formas de representação interna não suportadas por alguma das três linguagens, sendo assim é impossível a sua exibição nesta linguagem. Assim, determinadas construções são gerais, e atendem a qualquer linguagem, enquanto outras são específicas a um menor número de linguagens.

Algumas das construções particulares de uma linguagem podem ser representadas de uma maneira "composta" em outras linguagens, permitindo uma maior quantidade de construções sintáticas intercambiáveis. Porém isto é feito de modo distinto do que seria a representação nativa desta linguagem, caso a mesma fosse introduzida a partir dela.

A grande vantagem da existência de uma representação interna genérica é na construção de analisadores, tais como Analisadores de Controle de Fluxo de Dados, geração automática de códigos, a partir de Geradores de Aplicação, e outros.

Capítulo 4

REPRESENTAÇÃO INTERMEDIÁRIA DOS PROGRAMAS

4.1 - Considerações Iniciais.

A maioria dos ambientes orientados para estruturas representam seus programas em árvores de sintaxe, de onde são obtidas informações não apenas para a exibição do programa, mas também para a verificação semântica, outros recursos de suporte à compilação incremental e auxílio à depuração.

O PEGSS utiliza-se de uma estrutura hierárquica para a representação intermediária de seus programas, através da qual são obtidas as informações para a exibição das estruturas sintáticas nela representadas e as correspondentes informações na base de dados do programa corrente.

O objetivo deste Capítulo é a descrição desta estrutura hierárquica e a disposição das informações manipuladas pelo PEGSS, em seus nós. Complementando esta descrição, as figuras que compõem o Apêndice A especificam todas as subhierarquias que representam as estruturas sintáticas, definidas para a linguagem Pascal. A descrição completa das subhierarquias das estruturas sintáticas das linguagens Pascal, C e FORTRAN pode ser encontradas em [Val_92].

4.2 - Estrutura da Hierarquia.

A representação de um programa escrito através do PEGSS é feita em uma estrutura hierárquica, ao invés de cadeias de caracteres. Este tipo de estrutura permite que o programa seja construído e apresentado de forma estruturada, ou seja, as estruturas sintáticas que o compõem obedecem a uma endentação pré-estabelecida, facilitando sua legibilidade. Além disso, ferramentas de apoio à edição, compilação incremental e

depuração podem ser construídas, aproveitando-se esta estrutura.

A hierarquia de um programa é composta de dois tipos de elementos: uma estrutura de controle e seus nós. A estrutura de controle contém informações gerais de uma hierarquia, como por exemplo, seus nós primeiro e último, o nó corrente e o seu nível, durante o processamento da hierarquia, etc. Um nó contém informações como o item sintático nele armazenado, código do objeto da base de dados que representa a estrutura sintática especificada neste nó, o nível em que está localizado na hierarquia, etc. Como um nó não está isolado em uma hierarquia, também estão armazenadas nele as informações do seu nó pai, seu nó irmão de cima (prévio), seu nó irmão de baixo (próximo) e seu primeiro nó filho. No apêndice A são omitidas algumas destas ligações, apenas para tornar mais legíveis as representações das subhierarquias ali descritas.

A ilustração abaixo mostra os dois tipos de elementos de uma hierarquia.

Representação de uma estrutura de controle de hierarquia:

primeiro nó	último nó	nó corrente	nível corrente	...	outras informações

Representação de um nó da hierarquia:

nó pai	nó prévio	nó próximo	item sintático	...

...	código do objeto	nível	...	nó filho

A endentação apresentada pelas estruturas sintáticas, em um programa, é proporcionada pela disposição das palavras-chave e dos lugares de inserção nos nós da hierarquia, pois esta estrutura de dados permite que um gabarito tenha os seus itens hierarquicamente dependente de um outro, no nível imediatamente acima. Por exemplo, a estrutura "ifte" tem as palavras-chave "if(@)", "then", "else", além dos lugares de inserção <condição> e <bloco>, dispostos de modo que o item "if(@)" seja apresentado algumas colunas à esquerda do "then" e "else", indicando que esta estrutura tem início no item

"if(@)" e que compreende os demais, inclusive seus respectivos lugares de inserção. Isto ocorre porque o item "if(@)" é alocado em um nó de nível superior aos outros itens "then" e "else", estando estes dois últimos, um nível acima dos itens <condição> e <bloco>. A composição hierárquica desta estrutura é mostrada a seguir:

Na representação da estrutura "ifte", o item "if(@)" possui três itens subordinados e os dois lugares de inserção subordinados a dois destes itens. Mas em geral, um item pode possuir um número qualquer de subordinados, e cada subordinado também, em um número qualquer de níveis.

4.3 - Apresentação de uma Hierarquia .

Para que as estruturas de um programa representadas em uma hierarquia sejam mostradas ao usuário, esta terá de ser preparada anteriormente e, depois, processada. Preparar uma hierarquia significa determinar, após uma atualização qualquer, as linhas e colunas iniciais nas quais deve ser exibido cada um dos itens, nos respectivos nós, na janela de apresentação dos programas do PEGSS. Depois disso, a hierarquia poderá ser processada, isto é, o programa, representado em uma hierarquia preparada, é mostrado ao usuário.

Durante o processamento ("unparsing") de uma hierarquia, o usuário pode escolher um, entre os lugares de inserção, para que seja introduzida uma nova estrutura no programa, ou seja, um novo nó na hierarquia. Como já descrito no Capítulo 3, a "navegação" sobre as estruturas, ou hierarquia, é feita através das teclas de seta, onde a estrutura posicionada permanecerá em destaque. "Seta para cima" ou "seta para baixo" permitem subir e descer no mesmo nível da hierarquia. "Seta para a direita" e "seta para a

esquerda" permitem ir para os níveis posterior e anterior ao corrente, tornando menos ou mais abrangentes os itens em destaque. As teclas "home" e "end" posicionam-se, respectivamente, nos primeiros e últimos nós do primeiro nível da hierarquia, ou seja, primeira e última estrutura do programa.

Durante a preparação e processamento de uma hierarquia, o tratamento dado aos itens que contêm o caractere "@" é diferente daquele dado aos outros. Este caractere indica que deverá ser substituído durante o seu processamento pelo item do nó filho ao nó que o contém. Portanto, diferentemente dos demais, os itens com este caractere determinam a sobreposição de item na apresentação.

Exemplificando, a estrutura "ifte" representada na hierarquia acima terá, através do seu processamento, a seguinte apresentação:

if(<condição>)	if(<condição>)	if(<condição>)
then	then	then
   <bloco>	 <bloco>	 <bloco>
else	else	else
   <bloco>	 <bloco>	 <bloco>
(1)	(2)	(3)

Na representação (1) da estrutura "ifte" aparecem em destaque todos os itens e lugares de inserção que compõem o seu gabarito, pois o nó corrente durante o processamento é o que contém o item "if(@)". Pressionando-se a "seta para a direita", o nó corrente torna-se o que contém o lugar de inserção <condição>, deixando apenas este em destaque na sua representação, como mostrado em (2), pois este nó não possui qualquer nó filho. Através da "seta para baixo", a "navegação" na hierarquia segue, posicionando-se no nó abaixo do corrente do mesmo nível, deixando em destaque o item do nó corrente e seus filhos, como se observa em (3).

4.4 - Inclusão e Substituição.

A inserção de nós na hierarquia é feita, pressionando-se a tecla "return" com o cursor posicionado em um lugar de inserção. Seguindo o modo "top-down" de construção dos programas, os nós da hierarquia são introduzidos nos níveis de valor mais baixo para

os mais altos, geralmente como nós terminais da hierarquia. Em algumas situações, os nós são introduzidos entre dois nós existentes em níveis diferentes, ou no mesmo nível.

A substituição de um nó da hierarquia ocorre através da substituição de algumas informações do nó corrente por outras, mantendo as suas ligações com os outros nós. Entre as informações substituídas em um nó destacam-se o item e o código do objeto que o representa na base de dados.

(1)

As inserções e substituições dos nós de uma hierarquia são todas controladas pelo Editor, que determina os novos itens, lugares de inserção ou expressões a serem introduzidos. Estes deverão estar representados na hierarquia a partir de um lugar de inserção, escolhido pelo usuário. No exemplo mostrado a seguir, o lugar de inserção escolhido é o <bloco>

pertencente ao item "then" da estrutura "ifte", mostrada na ilustração (1).

(2)

Na ilustração (2) é apresentada a hierarquia da estrutura "ifte" após a introdução dos nós que representam a estrutura "bloco". Pode-se notar que o nó contendo <bloco> teve este lugar de inserção substituído pelo item "begin" e os outros nós adicionados à hierarquia. Na ilustração (1) está representado o processamento desta hierarquia.

Outro exemplo é apresentado nas ilustrações (3) e (4), onde mostra-se situações diferentes das já descritas. Considere-se um trecho de programa escrito em Pascal contendo o nó com o lugar de inserção <declara var.> e a sua respectiva hierarquia:

Se o lugar de inserção escolhido pelo usuário for o <declara var.>, a substituição

```

program <nome>;
.
.
var
    <declara var.>
.
.

```

(3)

e inserções na hierarquia seriam feitas pelo Editor, utilizando-se dos nós representados na hierarquia (6).

(4)

```

program <nome>;
.
.
.
var
    <nome> = <tipo>;
    <declara var.>
.
.
.

```

(5)

Na ilustração (5) é apresentado o processamento da hierarquia representada em (6). Nota-se que a sequência de caracteres "@ : @;" substituiu o lugar de inserção <declara var.> em um nó e outro nó foi introduzido com este mesmo lugar de inserção, além dos nós contendo os lugares de inserção <nome> e <tipo>. Como já citado anteriormente, cada caractere "@" indica que durante o processamento da hierarquia este deverá

ser substituído pelo item no primeiro nó filho do nó que o contém. No caso de ocorrer mais do que um caractere "@", como para a sequência de caracteres "@ : @;", estes deverão ser substituídos, respectivamente, pelos primeiros nós dos níveis seguintes, começando pelo primeiro nó filho do nó que contém a sequência de caracteres, o que contém o lugar de inserção <nome>, e o primeiro filho do próximo nível, o nó que contém o lugar de

(6)

inserção <tipo>.

No Apêndice A estão representados, para cada lugar de inserção em um nó da hierarquia, os respectivos nós, o que o substitui e os que são introduzidos, caso este seja o lugar de inserção escolhido pelo usuário em um programa.

4.5 - Eliminação .

A eliminação de uma estrutura em um programa, correspondente a um nó ou mais na hierarquia, não está implementada nesta versão do Editor. Apenas é permitido que uma hierarquia inteira seja eliminada. Assim como na edição, a eliminação exige que cada estrutura sintática seja tratada individualmente, tanto em relação aos nós que devem ser eliminados da hierarquia, quanto aos objetos que devem ser eliminados da base de dados.

4.6 - Considerações Finais.

Nota-se que a composição hierárquica utilizada pelo PEGSS armazena apenas informações sobre a exibição dos itens que compõem as estruturas sintáticas do programa (escrito em qualquer das três linguagens), os códigos dos objetos que representam estas estruturas na base de dados e informações sobre as ligações de um nó na hierarquia. As informações, de semântica estática, por exemplo, são obtidas pelo PEGSS através da base de dados, diferentemente dos ambientes orientados para estrutura tradicionais que as armazenam nas suas árvores de sintaxe.

Esta estrutura, ao contrário das árvores de sintaxe utilizadas pelo MENTOR, CPS, PG e outros, não define nós com números fixos ou variáveis de filhos. Todos os seus nós possuem um número variável de filhos.

Capítulo 5

ESTRUTURA DA BASE DE DADOS

5.1 - Considerações Iniciais.

O PEGSS utiliza uma base de dados orientada a objetos, tanto para armazenar e obter informações dos programas editados sob o seu controle, quanto para armazenar e obter informações sobre as construções sintáticas das linguagens por ele suportadas. Este Capítulo mostra como estas informações estão dispostas nesta base de dados e como são utilizadas pelo Editor.

5.2 - Sistema de Gerenciamento de Dados Utilizado

O PEGSS utiliza uma base de dados orientada a objetos denominada de GEO (GEreenciador de Objetos) [Tra_91] para o armazenamento da maioria das informações manipuladas. O GEO é um Sistema de Gerenciamento de Dados para Aplicações de Projeto, baseado no Modelo de Representação de Objetos (MRO) [Tra_86] [Tra_88] [Bia_92]. Este sistema tem como característica atender um amplo espectro de aplicações, incluindo aplicações integradas de projetos e produção de sistemas em geral e que, além disso, pode tirar proveito de arquiteturas de processamento paralelo, em especial, que pode ser executado com eficiência em arquiteturas desenvolvidas especialmente para a sua execução [Tra_86]. Logo, a construção de máquinas de base de dados podem proporcionar elevado desempenho com o MRO, apesar da grande demanda de recursos computacionais, criada por modelos de alto conteúdo semântico.

Como indicado no nome, o Modelo de Representação de Objetos segue o enfoque de orientação a objetos. Através do MRO, foi possível representar todas as informações utilizadas pelo Editor na construção, recuperação e tradução dos seus programas,

aproveitando-se de características que permitem especificar as informações durante a sua manipulação.

Alguns conceitos do MRO são mencionados rapidamente a seguir, por serem bastante utilizados nas sub-seções seguintes.

Para o MRO o mundo real constitui-se por um conjunto de objetos. Cada objeto possui um conjunto de identificadores externos e um identificador interno, atribuído pelo sistema, o qual identifica-o unicamente em toda a Base de Dados. Objetos semelhantes são caracterizados através de **Tipos de Objetos**.

As interações entre objetos do mundo real podem ser modeladas no MRO através de vários conceitos, entre eles os relacionamentos. Os relacionamentos semelhantes são caracterizados através da **modalidade de relacionamentos**. Os relacionamentos podem ser:

.binários ou apenas relacionamentos - associam dois objetos, denominados objeto origem e objeto destino do relacionamento. Cada relacionamento binário possui um relacionamento oposto, para representar sua semântica oposta. Todo relacionamento segue um padrão definido no esquema da aplicação denominado de **Tipo de relacionamento binário**, onde define-se o tipo do objeto origem, tipo do objeto destino e a modalidade permitida para este relacionamento;

. triplo - envolve a associação entre três ou mais objetos. Todo relacionamento triplo também segue um padrão definido no esquema da aplicação denominado de **Tipo de relacionamento triplo**.

Tanto objetos como relacionamentos são caracterizados por **atributos**. Cada atributo possui um tipo e um conjunto não vazio de valores.

Os objetos da Base de Dados são divididos em partições denominadas de **Colônias de objetos**, ou simplesmente de colônias. As colônias são classificadas em **Tipos de colônias**, e em cada uma podem estar presente um ou mais tipos diferentes de objetos. No entanto, objetos de um determinado tipo podem habitar colônias de um único tipo. Os objetos que pertencem a uma colônia, são chamados de habitantes dessa colônia.

Os **Tipos de colônias** estão organizados em uma hierarquia, onde cada colônia de um determinado nível é constrita por um objeto habitante de uma colônia localizada em um nível hierárquico imediatamente acima. O tipo da hierarquia é definido pela colônia global, que apresenta todos os seus objetos constantemente disponíveis, e corresponde ao nível menos detalhado de acesso ao Sistema.

Os tipos de objetos no esquema podem ser especializados em **Subtipos de objetos**. Subtipos podem ser outra vez especializados em novos subtipos. Um subtipo também é um tipo de objeto.

Os elementos do MRO são representados graficamente através do Diagrama de Representação de Objetos (DRO). As instâncias dos Tipos indicados no DRO são especificados pelo Diagrama de Representação de Instâncias (DRI). Na Figura 5.1 é mostra-se um exemplo de DRO e nas Figuras 5.2 e 5.3, exemplos de DRI. Um tipo de objeto é representado no DRO na metade superior de um círculo (por exemplo, na Figura 5.1, "comando") . A metade inferior especifica a colônia em que habita este tipo de objeto (por exemplo, na Figura 5.1, "Arquivo"). Uma modalidade de relacionamento é descrita através de arcos de circunferências (por exemplo, na Figura 5.1, "for-desvia-para"). Um objeto é representado, no DRI, por um círculo (por exemplo, na Figura 5.2, objeto de nome "par") e, um relacionamento, por um arco dirigido.

5.3 - Estrutura da Base GEO Utilizada pelo PEGSS.

Um programa escrito, utilizando o PEGSS, poderá ter a sua representação em qualquer das linguagens por ele suportadas, dentro dos limites da construção sintática da linguagem em que se deseja tê-lo representado. Por exemplo, a definição de novos tipos de dados em um programa editado sob o contexto da linguagem Pascal ou C não poderá ter uma representação equivalente em uma tradução deste programa para a linguagem FORTRAN, pois tal especificação sintática não é possível nesta última linguagem. Deste modo, outras estruturas sintáticas definidas para uma linguagem podem não ter uma representação equivalente em outra linguagem. Como as linguagens suportadas pelo Editor pertencem a um mesmo paradigma, a maioria de suas estruturas sintáticas possui uma representação equivalente entre elas.

A representação de um programa, nas várias linguagens, é possível para o PEGSS graças ao modo único segundo o qual são armazenadas as informações sobre as estruturas sintáticas que o descrevem. Este modo único, ou genérico, de armazenamento de um programa está definido através da Descrição da Linguagem Abstrata (DLA).

Uma vez que um programa está armazenado na base de dados, a sua recuperação é feita pelo Editor, sob o contexto da linguagem escolhida. Isto é feito, obtendo-se as informações de sua recuperação, também armazenadas na base de dados, além das próprias informações do programa. As informações para a recuperação de um programa estão armazenadas através de colônias de objetos, permitindo a cada uma destas colônias obtenção do programa em um das linguagens suportadas pelo Editor. As informações contidas nestas colônias são denominadas de Esquemas de Recuperação de Programas.

Portanto, como a atual implementação do Editor suporta as linguagens Pascal, C e FORTRAN existe um Esquema de Recuperação de Programas para cada uma destas linguagens.

As informações manipuladas pelo Editor através da edição, armazenamento e recuperação de programas podem ser divididas em:

- **Descrição da Linguagem Abstrata (DLA)** - esquema lógico da base de dados GEO utilizada pelo PEGSS, onde está especificada uma Construção Sintática Genérica para as linguagens por ele suportadas;
- **Esquemas de Recuperação de Programas** - informações que permitem ao Editor obter a representação de um programa armazenado na base de dados de acordo com a linguagem corrente. Os esquemas para cada linguagem são denominados: Esquema de Recuperação de Programas na linguagem Pascal (ERPPas), Esquema de Recuperação de Programas na linguagem C (ERPC) e Esquema de Recuperação de Programas na linguagem FORTRAN.
- **Programas** - informações de um programa escrito sob o controle do Editor, segundo a especificação da DLA.

Cada uma destas informações é descrita mais detalhadamente nas sub-seções seguintes.

5.4 - Apresentação da Descrição da Linguagem Abstrata

A DLA foi concebida de modo que apenas as informações comuns às linguagens de uma estrutura sintática fossem armazenadas na base de dados e as informações particulares para cada linguagem da estrutura fossem obtidas, quando sob o seu contexto. Por outro lado, algumas informações representadas na DLA só têm sentido para algumas linguagens e não para as outras, cabendo ao Editor selecionar as informações da DLA referentes a uma linguagem, quando sob o seu contexto.

Como já mencionado na seção 5.2, para a representação da DLA, foi utilizado o MRO, de acordo com o qual cada estrutura sintática é definida através de um tipo de objeto, e as estruturas, que podem estar a ela associadas são definidas por meio das suas

modalidades. Em um tipo de objeto pode estar representada toda uma classe sintática, sendo que as estruturas possíveis de serem descritas neste tipo de objeto definem esta classe.

Para a representação da DLA foi utilizado o Diagrama de Representação de Objetos (DRO) e as figuras que a descrevem são encontradas no Apêndice B. As figuras estão dispostas em conjuntos que representam as estruturas sintáticas da DLA, relevantes para o Editor, quando sob o domínio de uma dada linguagem. Deste modo, nas Figuras Pascal-1 até a Pascal-13 estão representadas as estruturas da DLA manipuladas pelo Editor, quando sob o contexto da linguagem Pascal; as Figuras C-1 até a C-13 contém as estruturas da DLA consideradas pelo Editor, quando sob o contexto da linguagem C; e as Figuras FORTRAN-1 até a FORTRAN-11 possuem representadas as estruturas da DLA tratadas pelo Editor, quando sob o contexto da linguagem FORTRAN.

Nas sub-seções que seguem são apresentadas as estruturas das linguagens descritas na DLA e sua representação.

5.4.1 - Estruturas de Controle de Fluxo.

Cada estrutura de controle definida na DLA é representada pelo tipo de objeto "comando", sendo cada um dos subtipos "atrib", "case", "cham_sub", "do", "for", "goto", "gotoc", "ifa", "ifl", "ift", "ifte", "repeat" e "while" definido por diferentes atributos para o seu tipo de atributo "tipo", correspondendo a cada uma das estruturas. Logo, o tipo de objeto "comando" representa a classe sintática composta por todas as estrutura de controle de uma linguagem.

A terminologia utilizada neste texto para referenciar uma estrutura sintática de controle associa a cada estrutura o subtipo do tipo de objeto "comando" utilizado na DLA para representá-la. Por exemplo, o nome da estrutura "if/then/else" é "ifte", como já foi utilizado na Figura 3.1a, 3.1b e no Capítulo anterior.

A estrutura de controle denominada pelo Editor de "ifte" está representada na DLA, através do tipo de objeto "ifte". Este tipo de objeto está relacionado ao tipo de objeto "bloco" através das modalidades "executa_caso_v" e "executa_caso_f" e ao tipo de objeto "condição" através da modalidade "verifica_cond", como pode ser visto nas Figuras Pascal-6, C-6 e FORTRAN-5. Cada uma destas modalidades define um conjunto de relacionamentos possíveis de serem efetuados entre a estrutura "ifte" e outras estruturas. Por exemplo, a estrutura "bloco" define a classe sintática composta por uma sequência de estruturas de controle que podem estar associadas ao "ifte", caso a condição seja verdadeira ou falsa.

Portanto, para uma estrutura "ifte", em um programa escrito sob o contexto de uma linguagem, as informações que deverão estar armazenadas na base de dados são a expressão que descreve a sua condição, o bloco a ser executado caso a condição seja verdadeira e o bloco a ser executado caso a condição seja falsa. O modo como estas informações devem ser apresentadas, isto é, os gabaritos utilizados para representá-las é definido, quando sob o domínio da linguagem escolhida. Logo, apenas o que é genérico sobre as estruturas será armazenado na base de dados, deixando-se as particularidades da representação das estruturas, para contexto de cada linguagem.

Algumas estruturas de controle possuem na sua representação, segundo a DLA, informações utilizadas apenas quando são representadas em determinadas linguagens. Por exemplo, a estrutura denominada "for", representada pelo tipo de objeto "for", tem associada a ela as modalidades "for_desvia_para" e "rotulo_do_for" que representam os rótulos em uma dessas estruturas, quando apresentada sob o contexto da linguagem FORTRAN, como se mostra na Figura 5.1. Estas informações não são consideradas, quando esta estrutura é apresentada sob o contexto das linguagens Pascal ou C; portanto estas informações, apesar de estarem representadas na DLA, são descritas apenas na Figura FORTRAN-7 e não aparecem nas Figuras Pascal-8 e C-8.

Uma estrutura "for" pode ter, por exemplo, sua representação possível apenas na linguagem Pascal, ou apenas na linguagem C, ou nas linguagens FORTRAN e C, ou em qualquer das linguagens suportadas pelo Editor. Estas possibilidades estão especificadas na DLA através dos subtipos:

."forg": especifica que a estrutura "for" é representável em qualquer das linguagens suportadas pelo Editor;

."forfc": especifica que a estrutura "for" é representável nas linguagens FORTRAN e C;

."forc": especifica que a estrutura "for" é representável apenas na linguagem C;

."forf": especifica que a estrutura "for" é representável apenas na linguagem FORTRAN;

."forp": especifica que a estrutura "for" é representável apenas na linguagem Pascal.

Para uma estrutura desta armazenada na base de dados, o Editor determina sob quais linguagens ela poderá ser apresentada através do atributo que determina o seu subtipo.

Figura 5.1: Representação da estrutura de controle "for" na DLA.

A Figura 5.1 mostra como as informações sobre a estrutura "for" são representadas segundo a DLA. Nas Figuras Pascal-8, C-8 e FORTRAN-7 estão representadas informações desta estrutura relevantes ao Editor, quando sob o contexto das respectivas linguagens Pascal, C e FORTRAN.

A estrutura de controle "ifl" é representada, segundo a DLA, através do tipo de objeto "ifl". Esta estrutura tem a particularidade de efetuar o desvio condicional para apenas algumas estruturas de controle que são: o "gotoc", o "cham_sub", o "atrib" e o "goto". Esta restrição às possíveis estruturas que podem estar relacionadas à estrutura "ifl" é representada na DLA pelas modalidades "possui_cmd" ligando o tipo de objeto "ifl" com os tipos de objetos "gotoc", "cham_sub", "atrib" e "goto". As Figuras Pascal-7, C-7 e FORTRAN-8 mostram as informações na DLA da estrutura "ifl" de acordo com as estruturas equivalentes nas construções sintáticas das respectivas linguagens Pascal, C e

FORTTRAN.

As estruturas de controle podem possuir rótulos que indicam um local para onde o controle do fluxo do programa deve ser transferido, durante a execução. Estas informações estão representadas na DLA através da modalidade "possui_rótulo", relacionando o tipo de objeto que representa a estrutura de controle ao tipo de objeto "rótulo". Nas Figuras Pascal-10, C-10 e FORTRAN-9 está representada a estrutura de controle com rótulo "ifte_r". Para cada estrutura de controle sem rótulo, também existe uma representação com rótulo desta estrutura na DLA, sendo que, nas Figuras do Apêndice B, a única representada é a "ifte_r". A representação das outras estruturas é efetuada através dos tipos de objetos "atrib_r", "case_r", "cham_sub_r", "do_r", "for_r", "goto_r", "gotoc_r", "ifa_r", "ifl_r", "ift_r", "repeat_r" e "while_r", onde cada um corresponde a uma estrutura com rótulo. Estas estruturas estão associadas aos mesmos tipos de objetos pelas mesmas modalidades como as suas respectivas representações sem rótulos, exceto no que se refere à adição da modalidade "possui_rótulo" relacionando-as ao objeto "rótulo". Por exemplo, a estrutura "ifte_r" está associada através das mesmas modalidades e com os mesmos tipos de objetos que a representação da estrutura "ifte", a menos da modalidade "possui_rótulo" que relaciona-a ao tipo de objeto "rótulo".

5.4.2 - Definição de tipos, definição de constantes e declaração de rótulos..

A linguagem Pascal permite que qualquer subrotina descreva, em seu escopo, declarações de rótulos, definições de constantes e definições de tipos. A linguagem C permite que as subrotinas descrevam em seu escopo definições de constantes e de tipos. E a linguagem FORTRAN, entre as declarações e definições citadas para as subrotinas escritas na linguagem Pascal, permite que suas subrotinas descrevam apenas definições de constantes. Estas informações estão representadas, segundo a DLA, nas Figuras Pascal-3, C-3 e FORTRAN-3.

As definições de constantes em uma subrotina são representadas através do tipo de objeto "def_const", onde cada definição de constante é especificada por um identificador e uma expressão, representados, respectivamente, através dos tipos de objetos "nome_const" e o tipo de objeto "constante". A representação de que uma subrotina pode ter várias destas definições é efetuada através das modalidades "prox_def_const".

Como foi citado no Capítulo 3, a estrutura que permite definir constantes na linguagem C também possibilita outros tipos de definições, tal como a definição de macros

com parâmetros, tornando-a mais abrangente do que as estruturas que definem constantes nas linguagens Pascal e FORTRAN. Assim como foram modelados os subtipos para a estrutura "for", poderiam ser definidos também subtipos para o tipo de objeto "def_const", onde um subtipo representaria as definições de constantes comuns às todas as linguagens e um outro subtipo representando apenas as definições possíveis para a linguagem C. Deste modo estariam representadas, genericamente, todas as possibilidades destas estruturas, nas respectivas linguagens, o que no momento está restrito apenas ao que é comum entre elas.

As definições de tipos estão representadas na DLA, através da associação do tipo de objeto "def_tipo", que representa as definições de tipos de dados novos de uma subrotina, com o tipo de objeto "ident_tipo_novo", que representa o identificador de tipo novo, em uma definição desta, e com o tipo de objeto "dado_var" que corresponde ao seu tipo de dado. As várias definições de tipo de uma subrotina estão relacionadas através da modalidade "prox_def_tipo".

A declaração de rótulos de uma subrotina, definida sob o contexto da linguagem Pascal, está representada pelo tipo de objeto "decl_rótulo" relacionada ao tipo de objeto "subrotina". Esta declaração especifica apenas os identificadores de rótulos representados pelo tipo de objeto "rótulo".

5.4.3 - Tipos de dados.

Os tipos de dados existentes para as linguagens Pascal, C e FORTRAN podem ser divididos em tipos de dados básicos e estruturados. Os tipos básicos são os tipos primitivos definidos pelas linguagens e os tipos estruturados são tipos mais elaborados e definidos pelo programador. Para a DLA, foi especificado apenas um subconjunto dos tipos de dados, definidos nas construções sintáticas destas linguagens.

Os tipos de objetos e modalidades, que representam os tipos de dados especificados na DLA e descritos nesta seção, estão nas Figuras Pascal-11, C-11 e FORTRAN-10.

O tipo de objeto "tipo_básico" representa os tipos básicos de dados definidos pelas linguagens, como por exemplo: "integer", "real" e "char" para a Linguagem Pascal; os tipos "int", "float", "void" e "char" para linguagem C; e "integer", "real" e "character*1" para a linguagem FORTRAN.

Os tipos de dados estruturados estão representados como: os vetores multidimensionais representados através do tipo de objeto "array" para as linguagens Pascal, C e FORTRAN; os registros representados através do tipo de objeto "registro", para as

linguagens Pascal e C; e os tipos "character", representado pelo tipo de objeto "charac", para a linguagem FORTRAN.

Um vetor pode possuir um ou mais índices, onde cada uma das definições de seus índices constitui-se em uma expressão. A representação na DLA de cada definição de índice de um vetor é efetuada através do par de tipos de objetos "índice" e "expr_ind" (um subtipo de "expressão").

O tipo estruturado registro é definido somente para as linguagens Pascal e C e é representado através do tipo de objeto "registro". As definições de seus campos e respectivos tipos são descritas na DLA através do relacionamento com objetos de tipo "variável" (subtipo de "declaração").

O tipo de dado "character" é definido apenas pela linguagem FORTRAN e é representado pelo tipo de objeto "charac". A especificação do tamanho de um tipo "character" é representada através do relacionamento do tipo de objeto "charac", com o tipo de objeto "tam".

Nas linguagens Pascal e C é possível definir novos tipos de dados, a partir de tipos básicos e estruturados definidos para a linguagem. Como já foi descrito na sub-seção anterior, esta possibilidade está representada através do tipo de objeto "def_tipo" que permite associar um identificador de tipo definido pelo programador a um tipo de dado. Este tipo de definição é mostrado nas Figuras Pascal-4 e C-4. Após a definição de um tipo novo, este poderá ser utilizado em definições de novos tipos e declarações. Isto é descrito na DLA por meio do relacionamento dos tipos de objetos "tipo_de_dado" e "ident_tipo_novo".

O tipo de objeto "tipo_de_dado" possui os seguintes subtipos: "dado_parâm" e "dado_var". O subtipo "dado_parâm" está relacionado apenas aos tipos de objetos "tipo_básico" e "ident_tipo_novo". Isto significa que um tipo de dado representado por um objeto de tipo "dado_parâm" só poderá ser um tipo básico ou um identificador de tipo definido pelo usuário. Já, o subtipo "dado_var" representa todos os tipos de dados definidos para a linguagem, pois objetos deste subtipo podem estar relacionados com objetos cujos tipos podem ser: "tipo_básico", "array", "registro", etc. A separação dos possíveis tipos de dados de uma linguagem em identificadores de tipos e todos os tipos de dados se faz necessária, pois, de acordo com a construção sintática de algumas delas, só é possível que determinadas estruturas estejam associadas a identificadores de tipos de dados. Por exemplo, os tipos de dados possíveis de serem especificados nas declarações de parâmetros de uma subrotina, na linguagem Pascal, são apenas os definidos por identificadores de tipos. Estes subtipos também devem ser considerados pelo Editor, quando a linguagem corrente

for C ou FORTRAN, pois, apesar de não fazer sentido esta especialização para estas linguagens, os subtipos são utilizados, para permitir a tradução destas estruturas, para as respectivas estruturas na linguagem Pascal.

5.4.4 - Declarações de Parâmetros e Variáveis.

As declarações na DLA são representadas pelo tipo de objeto "parâmetro" (subtipo de "declaração"), representando as declarações de parâmetros, e "variável" (subtipo de "declaração"), representando as declarações de variáveis. Estes subtipos são assim definidos, pois para o Editor, durante o processo de edição e recuperação de programas, é necessário diferenciar uma declaração de parâmetros de uma declaração de variáveis. Estas declarações estão representadas nas Figuras Pascal-2, Pascal-5, Pascal-11, C-1, C-3, C-5, C-11, FORTRAN-2, FORTRAN-4 e FORTRAN-10.

Nada impede que estas declarações sejam representadas através de tipos de objetos diferentes. A escolha pela atual representação deve-se ao fato de estas informações ficarem agrupadas como declarações, e também como declarações de parâmetros e de variáveis, possibilitando que ferramentas de suporte a documentação ou depuração possam usufruir dos três tipos de informações, ao invés de duas.

As declarações de parâmetros de uma subrotina são definidas por um nome de parâmetro e o seu tipo. Isto é representado na DLA pelos tipos de objetos "nome_parâmetro" e "tipo_de_dado" associados ao tipo "parâmetro". Para as linguagens C e FORTRAN, o tipo "parâmetro" está associado diretamente com o tipo de objeto "tipo_de_dado", enquanto em Pascal a associação é efetuada com o tipo "dado_parâm". Esta diferenciação ocorre, porque, apenas na linguagem Pascal, o tipo de um parâmetro na declaração deve ser um identificador de tipo.

As declarações de variáveis, assim como as declarações de parâmetros, são definidas pelo nome de variável e pelo seu tipo. Estas informações estão representadas através do tipo "variável", associado aos tipos de objetos "nome_var" e "dado_var", onde o último tipo de objeto representa os tipos de dados possíveis de serem definidos para uma variável em uma declaração.

5.4.5 - Subrotinas.

Uma subrotina está representada na DLA através do tipo de objeto "subrotina", sendo que os tipos de subrotinas possíveis nas linguagens suportadas pelo Editor podem ser principal, função e procedimento. Essas informações são descritas por meio dos subtipos "principal", "função" e "procedimento" do tipo "subrotina".

Na DLA, uma "subrotina" de subtipo "principal" é uma subrotina em Pascal e FORTRAN que descreve as declarações, definições e corpo do programa principal. Em C, esta subrotina é descrita através da função "main()". Uma "subrotina" de subtipo "função" é uma subrotina em Pascal e FORTRAN que especifica o seu retorno através de uma estrutura de atribuição. A variável à esquerda desta atribuição deve ser o identificador da função. Em C, o seu retorno é especificado através de uma expressão, em uma estrutura "return", onde não é especificado o identificador da função. Uma "subrotina" de subtipo "procedimento" é uma subrotina em Pascal e FORTRAN que não possui uma expressão a ser retornada e, em C, é uma função cujo valor de retorno é o "void".

Qualquer subtipo de "subrotina" em Pascal e FORTRAN pode ter um nome, como mostrado nas Figuras Pascal-2 e FORTRAN-2, descrito através do tipo de objeto "nome_subrot". Isto não ocorre do mesmo modo em C, onde apenas a "subrotina" de subtipo "principal" não possui nome, como mostra a Figura C-1.

O tipo de objeto "parâm_da_lista" e "nome_parâmetro" mostrados nas Figuras Pascal-5, FORTRAN-4 e C-5 permitem representar os vários nomes de uma lista de parâmetros pertencente ao subtipo "função" ou "procedimento". Mesmo não sendo possível a uma subrotina na linguagem Pascal possuir uma lista de nome de parâmetros na sua definição, estes tipos de objetos aparecem na Figura Pascal-5. Apenas o Editor manipula estes tipos e cria automaticamente objetos destes tipos para cada declaração de parâmetro. Isto é necessário para facilitar a tradução de subrotinas destes subtipos para as outras linguagens, que, no caso, necessitam que objetos destes tipos estejam definidos.

A "subrotina" de subtipo "procedimento" está associada ao tipo de objeto "tipo_básico" através da modalidade "retorna_nulo", como é apresentado nas Figuras Pascal-1, FORTRAN- 1 e C-1 pois, mesmo para as linguagens Pascal e FORTRAN, estas peças devem ser manipuladas pelo Editor, quando são criados estes subtipos de subrotinas, em programas escritos sob o domínio de tais linguagens. Com isto, facilita-se a tradução de uma linguagem para outra de subrotinas representadas neste subtipo.

5.4.6 - Definições locais e globais de subrotinas .

Na linguagem Pascal, uma subrotina do tipo principal, um procedimento ou uma função, podem definir novas subrotinas, dentro do seu escopo. Já para as linguagens C e FORTRAN, este tipo de definição é possível apenas no escopo global de um arquivo ou programa.

Estas possibilidades de definições de subrotinas são representadas na DLA através dos tipos de objetos "decl_subrot_glob" e "decl_subrot_loc", onde para as linguagens C e FORTRAN, o último tipo de objeto não é definido. Estes tipos de objetos apresentam os subtipos "func_glob", "proc_glob", "func_loc" e "proc_loc" como pode ser verificado nas Figuras Pascal-1, Pascal-2, C-1 e FORTRAN-2 que representam definições globais ou locais de subrotinas, sejam elas procedimentos ou funções.

Nas três linguagens, uma subrotina principal pode definir novas subrotinas do tipo procedimento ou função como é representado através do tipo de objeto "decl_subrot_glob" associado ao tipo "principal". Os tipos "proc_glob" e "func_glob" estão associados, respectivamente, aos tipos "procedimento" e "função", representando os tipos de subrotinas que podem ser definidos globalmente em um arquivo.

Como já mencionado acima, apenas para a linguagem Pascal é possível definir novas subrotinas, no escopo de outra subrotina do tipo procedimento e função. Esta possibilidade de definição de subrotinas locais a outra subrotina é representada pelo relacionamento entre os tipos "procedimento" e "função" e o tipo de objeto "decl_subrot_loc". As subrotinas definidas no escopo da subrotina principal, como foi dito no parágrafo anterior, são consideradas definições globais de subrotinas.

As definições locais de subrotinas permitem que se descrevam, para cada uma delas, apenas subrotinas do tipo função ou procedimento. A representação desta possibilidade na DLA é feita através da associação entre os tipos "proc_loc" e "func_loc" "dec_subrot_loc", indicando se a definição é de uma subrotina do tipo função ou procedimento, com os respectivos tipos "procedimento" e "função".

O modo como estão representadas na DLA as definições locais e globais de subrotinas permite ao Editor identificar, durante a tradução de um programa escrito em Pascal para a sua representação em C ou FORTRAN, apenas as definições globais de subrotinas, omitindo as definições locais, caso existam.

5.4.7 - Escopo das variáveis.

As declarações das variáveis de uma subrotina estão representadas na DLA através da associação dos tipos de objetos "subrotina" e "declaração", por intermédio de seus devidos subtipos, como pode ser observado nas Figuras Pascal-2, C-1 e FORTRAN-2.

Na linguagem Pascal e FORTRAN as variáveis globais são definidas no escopo do programa principal e as variáveis locais, no escopo das subrotinas do tipo procedimento ou função. A representação das informações que descrevem as declarações das variáveis globais de um programa é efetuada através do relacionamento entre o tipo de objeto "principal" e o tipo de objeto "variável" pela modalidade "decl_var_glob". As informações que descrevem as declarações das variáveis locais de uma subrotina são representadas pela modalidade "dec_var_loc", relacionando os tipos de objetos "procedimento" e "função", ao tipo de objeto "variável".

Para a linguagem C, as representações destas declarações na DLA diferem das citadas acima, pois à subrotina principal, ou seja à função "main()", estão relacionadas as declarações das variáveis globais ao arquivo e as declarações das variáveis descritas no seu escopo. Às outras subrotinas, ou funções que não sejam a "main()", estão associadas apenas as declarações locais das variáveis. As informações sobre as declarações locais das variáveis de uma subrotina estão representadas através da modalidade "decl_var_loc", relacionando o tipo de objeto "subrotina" ao tipo de objeto "variável". A representação das declarações das variáveis globais é efetuada pelo relacionamento entre o tipo de objeto "principal" e o tipo de objeto "variável" pela modalidade "decl_var_glob". É importante ressaltar que as subrotinas principais, descritas sob o contexto das linguagens Pascal e FORTRAN, não possuem declarações locais de variáveis, portanto, nas suas representações na DLA, o tipo de objeto "principal" não possui a modalidade "decl_var_loc" ligando-o ao tipo de objeto que representa as declarações de variáveis.

5.4.8 - Blocos.

A estrutura sintática que permite o agrupamento de uma sequência de estruturas de controle no corpo de uma subrotina ou de uma estrutura de controle é representada na DLA pelo tipo de objeto "bloco". Este tipo de objeto possui os subtipos definidos pelos atributos "início", "subrot" e "composto" que representam, respectivamente, o bloco de uma subrotina principal, o bloco de uma subrotina do tipo procedimento ou função e o bloco

de uma estrutura de controle composta. Esta estrutura está representada nas Figuras Pascal-12, C-12 e FORTRAN-11.

Estes subtipos são definidos assim porque, durante a tradução de uma estrutura de bloco para outra linguagem, a informação de como esta estrutura sintática deverá ser exibida é obtida a partir de um dos seus subtipos que a representa na base de dados.

O tipo de objeto "composto" também possui subtipos definidos pelos atributos "1", "2" e "3" para o seu tipo de atributo "categoria". Isto porque um bloco em um programa cuja representação na base de dados é efetuada pelo subtipo definido pelo atributo "composto" possui três modos distintos de serem exibidos, portanto cada um dos seus subtipos corresponde a um destes modos. O tipo "composto", para a linguagem C, deveria estar associado aos tipos de objetos "def_tipo", "def_const" e "declaração" de subtipo definido pelo atributo "var", porque todo bloco, em uma estrutura de controle, pode definir novos tipos, constantes e declarar variáveis. Nesta versão do Editor, optou-se por excluir essa possibilidade apenas para proporcionar uma maior praticidade durante o processo de criação de programas sobre o contexto desta linguagem, além de torná-los mais claro ao programador. Se estas definições e declarações forem efetuadas, tal procedimento ocorrerá do mesmo modo como já descrito nas sub-seções 5.4.2 e 5.4.4.

5.4.9 - Expressões.

As expressões nas linguagens Pascal, C e FORTRAN são descritas como uma sequência de caracteres que compõem operadores, constantes e nomes de variáveis ou de funções. Uma expressão pode descrever uma estrutura de atribuição, ou uma condição em uma estrutura de controle, ou ser uma das expressões que compõem uma estrutura de controle ou descrever o índice de uma definição do tipo vetor.

Como é mostrado nas Figuras Pascal-13, C-13 e FORTRAN-12, as expressões são representadas através do tipo de objeto "expressão", e o que ela descreve é especificado pelo subtipo definido, pelo atributo do tipo de atributo "modo".

É descrito abaixo o que representam os respectivos subtipos do tipo de objeto "expressão":

• "atribuição"	- uma estrutura de atribuição.
• "condição"	- uma condição de uma estrutura de controle.
• "constante"	- uma definição de constante.
• "lista_expr"	- expressões de uma estrutura de chamada de subrotina.
• "início"	- expressão inicial de uma estrutura "for".
• "parada"	- expressão de parada de uma estrutura "for".
• "incremento"	- expressão de incremento de uma estrutura "for".
• "retorno"	- expressão de retorno de uma função, nas linguagens Pascal e FORTRAN é uma estrutura de atribuição.
• "expr_aritm"	- expressão da estrutura "ifa".
• "ativa"	- expressão de uma estrutura "case".
• "testa_case"	- expressão que ativa um bloco de uma estrutura "case".
• "expr_gotoc"	- expressão utilizada pela estrutura "gotoc".
• "expr_ind"	- expressão que define uma índice de vetor.

Exceção feita ao subtipo "expr_ind" do tipo de objeto "expressão", todos os seus demais subtipos possuem o tipo de atributo "descrição", onde, através dos atributos de "descrição", são armazenadas as expressões na forma pós-fixa que o subtipo representa.

O subtipo capaz de permitir que as expressões definidoras de um índice de vetor sejam armazenadas na base de dados também possui subtipos: "índice_g", "índice_pc", "índice_pf" e "índice_p". Estes subtipos do subtipo permitem ao Editor determinar se um índice é representável sob o contexto da linguagem corrente e como deve ser apresentado.

Para a representação interna utilizada pelo Editor, uma definição de índice é composta das subexpressões determinantes do seu limite inferior e superior. Estas subexpressões são descritas como atributos dos tipos de atributos "inf" e "sup" do subtipo "expr_ind". Esta representação foi assim determinada, para facilitar a apresentação de um índice, nas várias linguagens suportadas pelo Editor.

5.5 - Armazenamento de Programas na Base de Dados.

Qualquer programa editado sob o controle do PEGSS será armazenado na base de dados em uma colônia de objetos, onde as informações que o descrevem estão na forma de objetos, atributos e relacionamentos, segundo o Esquema lógico definido para a DLA.

Por exemplo, considere-se o seguinte trecho de programa editado sob o contexto da linguagem Pascal e armazenado em um arquivo de nome "par":

```
program soma_par;
  label
    <decl.rótulo>;
  const
    <def.const.>
  type
    <def.tipo>
  var
    soma : integer;
    par : integer;
    <declara var.>
  <decl.subrot.>
  begin
    soma := 0;
    par := 100;
    repeat
      soma := soma + par;
      par := par + 2;
      until( par > 200 );
    <comando>
  end.
```

A representação das informações deste programa armazenadas na base de dados é feita através do Diagrama de Representação de Instâncias (DRI), como se mostra nas Figuras 5.2 e 5.3.

Nas Figuras 5.2 e 5.3 os objetos cujos identificadores são "s1", "b1", "d1", "c1", etc, estão representando identificadores definidos pelo GEO.

A colônia do tipo "Arquivo" que contém estas informações é constrita pelo objeto de tipo "arquivo" cujo identificador definido pelo usuário é "par". Este identificador é utilizado pelo Editor para localizar o arquivo entre os demais armazenados na base de dados. Uma vez que esta colônia é aberta, estarão disponíveis ao Editor apenas os objetos, relacionamentos e atributos que a habitam, permitindo deste modo que somente tais peças sejam manipuladas.

Figura 5.2: DRI do programa de nome "par".

5.6 - Esquemas de Recuperação de Programas.

Como já foi dito na seção 5.3, para cada linguagem existe um Esquema de Recuperação de Programas (ERP). Como a versão atual do PEGSS suporta subconjuntos das linguagens Pascal, C e FORTRAN, então os Esquemas definidos são: o Esquema de Recuperação de Programas para a linguagem Pascal (ERPPas), o Esquema de Recuperação de Programas para a linguagem C (ERPC) e o Esquema de Recuperação de Programas para a linguagem FORTRAN (ERPFOR). Estes Esquemas compreendem as informações para a leitura de um programa armazenado na base de dados e de sua apresentação na linguagem corrente.

As informações de cada um dos ERP estão armazenadas na base de dados, através de colônias de objetos. Cada objeto possui informações sobre a recuperação e apresentação de uma estrutura armazenada na base de dados, através de um objeto de mesmo tipo. Por exemplo, na colônia que contém o ERPPas existe um objeto do tipo "def_const" dotado das informações para a recuperação e apresentação de uma definição de constante qualquer, em um programa para a linguagem Pascal. O modo de obter das informações de um objeto

Figura 5.3: Continuação do DRI do programa de nome "par".

do tipo "def_const", e de sua representação, segundo a linguagem Pascal, se obtém através do objeto de tipo "def_const" em ERPPas. Do mesmo modo existem nas colônias que contém o ERPC e o ERPFOR um objeto do tipo "def_const" que permite a recuperação e apresentação de uma estrutura de definição de constante armazenada na base de dados para as respectivas linguagens C e FORTRAN.

Todos os objetos das três colônias que compreendem os ERP possuem o tipo de atributo "lebase", pois a maioria das informações de obtenção e apresentação de um programa na base de dados estão na forma de atributos para este tipo de atributo. Estas informações também se encontram nos objetos cujos tipos possuem subtipos, como atributos de outros tipos diferente do "lebase".

A título de ilustração mostram-se os atributos do tipo "lebase" e "ifte" do objeto de tipo "comando" pertencente a colônia que contém o ERPPas:

tipo de objeto	tipo de atributo	atributos
comando	lebase	TIPO FIM
ifte	ifte	SUBS if(%) 0 INSE <condição> 1 RELA "verifica_cond" INSE then 0 INSE <bloco> 1 RELA "executa_caso_v" INSE else -1 INSE <bloco> 0 RELA "executa_caso_f" INSE <comando> -2 RELA "segue" FIM

O tipo de atributo "ifte" contém as informações de recuperação e apresentação de objetos de subtipo "ifte" da colônia que contém o programa que está sendo lido. Como o tipo de objeto "comando", na DLA, possui outros subtipos ("while", "atrib", "ift" e etc), existem no objeto de tipo "comando", no ERPPas, os tipos de atributos "while", "atrib", "ift" e etc. com as informações de leitura e apresentação correspondente a estes subtipos.

As informações de leitura do programa na base de dados se constituem em instruções para a rotina de leitura, determinantes da ação a ser tomada em relação ao objeto corrente na colônia que contém o programa. Algumas destas instruções completam-se com outras informações que compõem o atributo. Estas instruções têm a seguinte forma:

Função	Modalidade ou Tipo de Atributo

ou

Função

Uma Função pode ser:

TIPO: obter o atributo do tipo de atributo "tipo" do objeto corrente na colônia que contém o programa que está sendo lido.

RELA: a partir do objeto corrente, na colônia que contém o programa, posicionar no objeto destino através da modalidade especificada neste atributo corrente. Se existir o relacionamento com origm nesse objeto, o objeto destino torna-se o objeto corrente, e lêem-se os atributos de seu tipo de atributo "lebase".

ATRI: ler o atributo do objeto corrente na colônia que contém o programa cujo tipo é especificado neste atributo corrente.

FIM: determina o final de leitura dos atributos do tipo de atributo "lebase" do objeto corrente.

Uma função pode vir ou não seguida de uma modalidade ou um tipo de atributo, determinando a peça sobre a qual a função deverá ser aplicada. O elemento que acompanha a função especifica a peça¹ do objeto corrente na colônia que contém o programa que deverá ser aplicada a função.

Podem ser exemplos de atributos que compõem as instruções descritas acima:

ATRI "descrição"
RELA "segue"
FIM

onde, ATRI, RELA e FIM são funções já descritas acima, "descricao" é um tipo de atributo e "segue" é uma modalidade.

As informações segundo as quais a estrutura representada no objeto corrente deve

---

¹peça: pode ser tipo de objeto, tipo de atributo, modalidade, etc

ser exibida sob forma de gabaritos também se constituem em instruções nos atributos do tipo "lebase". Estas informações são utilizadas pelas rotinas que tratam da construção da hierarquia de um programa. As instruções que descrevem estas informações possuem o seguinte formato:

Função	Texto	Incremento de nível

ou

Função

Uma função pode ser:

INSE: inserir um novo nó na hierarquia com o texto e incremento de nível especificado no restante desta instrução.

SUBS: substituir o texto do nó corrente da hierarquia pelo texto especificado nesta instrução.

AVAN: posicionar no nó filho do nó corrente da hierarquia.

ELIM: extrair o nó corrente da hierarquia.

CONC: concatenar ao texto do nó corrente da hierarquia o texto especificado nesta instrução.

As funções descritas acima podem ser seguidas de um texto e um incremento de nível. Um texto é uma cadeia de caracteres que descreve um item sintático de um gabarito ou uma expressão. O incremento de nível é o incremento positivo ou negativo de nível que deverá sofrer o nó corrente da hierarquia.

São exemplos de atributos que compõem algumas das instruções descritas acima:

SUBS	if(@)	0
INSE	<condição>	1
ELIM		

onde,

SUBS, INSE e ELIM são funções já descritas acima, if(@) é um texto que descreve um item sintático, <condição> é um texto que descreve um lugar de inserção de um gabarito, 0 e 1 são incrementos de nível.

Através do modo segundo a qual as informações de recuperação e apresentação foram definidas, é possível obter as várias representações de uma estrutura genérica. Para isto, basta estar no contexto da linguagem alvo. Por exemplo, a leitura e apresentação, sob o contexto da linguagem Pascal, de uma subrotina principal seria efetuada pelo Editor, sob o controle das seguintes informações nos respectivos Esquemas:

.ERPPas:

tipo de objeto	tipo de atributo	atributos
principal	principal	INSE program @; 0 INSE <nome> 1 RELA "nome_da_subrot" INSE label @; 0 INSE <decl.rótulo> 1 RELA "decl_rótulos" INSE const -1 INSE <def.const.> 1 RELA "define_const" INSE type -1 INSE <def.tipo> 1 RELA "define_tipo" INSE var -1 INSE <declara var.> 1 RELA "decl_var_glob" INSE <decl.subrot.> -1 RELA "decl_global" INSE <bloco> 0 RELA "contem_blc_inic" FIM

.ERPC:

tipo de objeto	tipo de atributo	atributos
principal	principal	INSE <def.const.> 0 RELA "define_const" INSE <def.tipo> 0 RELA "define_tipo" INSE <declara var.> 0 RELA "decl_var_glob" INSE <decl.função> 0 RELA "decl_global" INSE main() 0 INSE { 1 INSE <def.const.> 1 RELA "define_const_loc" INSE <def.tipo> 0 RELA "define_tipo_loc" INSE <declara var.> 0 RELA "decl_var_loc" INSE <comando> 0 RELA "contem_blc_inic" INSE } -2 FIM

.ERPFOR:

tipo de objeto	tipo de atributo	atributos
principal	principal	INSE program @; 0 INSE <nome> 1 RELA "nome_da_subrot" INSE <declara var.> 0 RELA "decl_var_glob" INSE <def.const.> 0 RELA "define_const" INSE <bloco> 1 RELA "contem_blc_inic" INSE <decl.subrot.> -1 RELA "decl_global" FIM

Considerando apenas as informações descritas nos atributos acima, o objeto na base de dados que representa uma subrotina principal seria apresentado através das seguintes representações sob o contexto das linguagens Pascal, C e FORTRAN:

PASCAL

```

program <nome>;
  label <decl.rótulo>;
  const
    <def.const.>
  type
    <def.tipo>
  var
    <declara var.>
    <decl.subrot.>
    <bloco>

```

C

```

<def.const.>
<def.tipo>
<declara var.>
<decl.função>
main()
{
  <def.const.>
  <def.tipo>
  <declara var.>

  <comando>
}

```

FORTRAN

```
program <nome>;  
  <declara var.>  
  <def.const.>  
  <bloco>  
  <decl.subrot.>
```

Nota-se que algumas informações não são apresentadas, quando sob o contexto de certas linguagens, satisfazendo suas respectivas construções sintáticas. Por exemplo, a declaração de rótulos não aparece na representação do programa principal, quando sob o contexto da linguagem C e FORTRAN. A definição de tipos não aparece na representação do programa em FORTRAN. Outras informações são mostradas de modo diferente em cada uma das representações, como por exemplo, a declaração de variáveis. Isto é proporcionado através das informações descritas nos atributos apresentados anteriormente para o tipo de objeto "subrotina", nos ERP. Da representação na base de dados de uma subrotina principal é obtido apenas o que é relevante para a sua representação na linguagem corrente, além de apresentar as estruturas no lugar e formato especificado pela construção sintática desta linguagem.

Através das informações nos ERP, é possível que determinadas estruturas sintáticas inexistentes para uma linguagem sejam apresentadas por combinações de outras estruturas desta linguagem. Por exemplo, a estrutura de controle "do" (ver a Figura 3.1a e 3.1b) é apresentada sob o contexto das linguagens Pascal e FORTRAN através da combinação de estruturas destas linguagens, definidas pelas informações nos ERPPas e ERPFOR. Estas informações são descritas para os respectivos ERP do seguinte modo:

.ERPPas

tipo de objeto	tipo de atributo	atributos
do	do	SUBS repeat 0 INSE <bloco> 1 RELA "executa_blc" INSE until(not @); 0 INSE <condição> 1 RELA "verifica_cond" INSE <comando> -2 RELA segue FIM

.ERPC:

tipo de objeto	tipo de atributo	atributos
do	do	SUBS do 0 INSE <bloco> 1 RELA "executa_blc" INSE while(@); 0 INSE <condição> 1 RELA "verifica_cond" INSE <comando> -2 RELA segue FIM

.ERPFOR:

tipo de objeto	tipo de atributo	atributos
do	do	SUBS @ continue 0 INSE <rótulo> 1 RELA "rótulo_do_do" INSE <bloco> 0 RELA "executa_blc" INSE if(@) goto @ 0 INSE <condição> 1 RELA "verifica_cond" INSE <rótulo> 1 RELA "do_desvia_para" INSE endif -2 INSE <comando> -1 RELA segue FIM

A apresentação da estrutura "do" sob o contexto das linguagens de acordo com as informações descritas acima seria:

PASCAL

```
repeat
  <bloco>
  until(not <condição>);
<comando>
```

C

```
do
  <bloco>
  while(<condição>);
<comando>
```

FORTTRAN

```
<rótulo> continue  
  <bloco>  
  if (<condição>) goto <rótulo>  
  endif  
<comando>
```

Os lugares de inserção <bloco> e <condição> são comuns às representações da estrutura "do", portanto são mantidas, mas os demais itens são apresentados de modo que a estrutura seja exibida corretamente.

O lugar de inserção <comando> aparece nas representações das estruturas de modo a permitir a continuação da edição, caso seja necessário.

5.7 - Considerações Finais.

Como foi possível observar, através do armazenamento genérico das estruturas que descrevem um programa, obtêm-se as suas várias representações em cada uma das linguagens suportadas pelo Editor. Os programas são armazenados, segundo a concepção lógica da DLA e recuperados com o auxílio do ERP da linguagem corrente.

O PEGSS permite a adição de linguagens pertencentes a uma mesma abordagem na estrutura da base de dados. A DLA deverá ser atualizada e um novo ERP deverá ser adicionado. As atualizações na DLA devem ocorrer de modo a permitir que a construção sintática da nova linguagem passe a estar representada. Como a DLA já foi concebida, prevendo estas expansões, as atualizações deverão restringir-se apenas às estruturas inexistentes para as linguagens já suportadas. Por outro lado, um novo ERP deve ser introduzido na base de dados, onde devem estar especificadas informações sobre a sintaxe concreta da linguagem e de controle de recuperação de representações internas de programas.

Capítulo 6

EXPRESSÕES

6.1 - Considerações Iniciais.

Como já mencionado no Capítulo 3, toda expressão é introduzida no PEGSS pelo usuário. Isso é efetuado, digitando-se os caracteres que a descrevem, o que difere do modo de inserção das outras estruturas sintáticas em um programa. Neste Capítulo será apresentado o processo de edição de uma expressão, o modo como é armazenada e como se realiza a tradução de uma expressão escrita em uma linguagem para outra.

6.2 - Expressões em Geral.

A introdução de uma expressão pelo usuário durante o processo de edição é feita nos lugares de inserção tais como:

- .<atribuição> ,
- .<condição> ,
- .<constante> ,
- .<expr.> ,
- .<índice> ,
- .<lista expr.> e
- .<tam.> .

Com o cursor posicionado em qualquer um destes lugares de inserção e pressionando-se a tecla "return", será permitido que o programador possa inserir os caracteres que descrevem a expressão, através de uma janela na parte inferior do vídeo. Em

seguida, é efetuada a análise léxica, evidenciando-se a correção, é iniciado o processo de preparação da expressão para o seu armazenamento; caso existam erros léxicos, a expressão é desprezada, e uma nova deverá ser introduzida.

A inserção descrita acima também poderá ocorrer em lugares de inserção no programa que já possuam expressões anteriormente editadas, possibilitando a atualização destas expressões, ou a introdução de uma nova.

O processo de preparação de uma expressão para o seu posterior armazenamento, na base de dados, consiste em transformá-la em uma expressão genérica. Uma expressão genérica é a expressão na forma pós-fixa, obtida a partir de uma expressão infixa inserida pelo usuário, com seus operadores convertidos em operadores únicos para as linguagens suportadas pelo Editor, ou seja, em operadores genéricos. A representação genérica dos operadores e os operadores equivalentes nas linguagens são observados nas Figuras 6.1 e 6.2. Uma expressão é assim armazenada para possibilitar a sua respectiva representação nas outras linguagens do Editor.

PASCAL	C	FORTTRAN	Genérico	Significado
+	+	+	#1#	adição
-	-	-	#2#	subtração
*	*	*	#3#	multip.
/	/	/	#4#	divisão
div	/		#5#	div. inteira
mod	%		#6#	resto
		**	#7#	exponenc.
:=	=	=	#8#	atribuição
=	==	.EQ.	#9#	igualdade
<	<	.LT.	#10#	menor que
>	>	.GT.	#11#	maior que
>=,=>	>=,=>	.GE.	#12#,#13#	maior igual
<=,=<	<=,=<	.LE.	#14#,#15#	menor igual
<>	!=	.NE.	#16#	diferente
in			#17#	pertinência
and	&&	.AND.	#18#	e
or		.OR.	#19#	ou
		.EQV.	#20#	equivalência
		.NEQV.	#21#	nao equival.
	&		#22#	e bit-a-bit
			#23#	ou inclus.
	^		#24#	ou exclus.
	<<		#25#	desloc. esq.
	>>		#26#	desloc. dir.
	+=		#27#	atrib. adicao
	-=		#28#	atrib. subtr.
	*=		#29#	atrib. mult.
	/=		#30#	atrib. divis.
	%=		#31#	atrib. resto
	&=		#32#	atrib. conj.
	^=		#33#	atr.disj.exc.
	=		#34#	atr.disj.inc.
	<<=		#35#	atr.desl.esq.
	>>=		#36#	atr.desl.dir.
	->		#37#	apont. membro
.	.		#38#	espec. membro
[	[	(	#39#	inic. vetor
]	]	)	#40#	final. vetor
(	(	(	#41#	abre parent.
)	)	)	#42#	fecha paren.
,	,	,	#43#	virgula
..			#44#	sep. indices

Figura 6.1: Tabela de equivalência entre os operadores das linguagens Pascal, C e FORTRAN, e suas representações genéricas.

PASCAL	C	FORTTRAN	Genérico	Significado
not	!	.NOT.	#45#	negação
	-		#46#	complem. um
+	+	+	#47#	mais unário
-	-	-	#48#	menos unário
	&		#49#	endereço
^	*		#50#	apontad. end.
	++op		#51#	incr. prefixo
	--op		#52#	decr. prefixo
	op++		#53#	incr. posfixo
	op--		#54#	decr. posfixo
	(int)		#55#	cast
	(char)		#56#	cast
	(float)		#57#	cast
	(void)		#58#	cast
	sizeof(int)		#59#	sizeof
	sizeof(char)		#60#	sizeof
	sizeof(float)		#61#	sizeof
	sizeof(void)		#62#	sizeof
+	+	//	#63#	concatenac.
	?:		#64#	oper. cond.

Figura 6.2: Continuação da Tabela de equivalência entre os operadores das linguagens Pascal, C e FORTRAN.

Nota-se nestas figuras que alguns operadores de uma linguagem não são definidos em outras. Um exemplo é o operador "++" da linguagem C que não possui um operador equivalente nas demais linguagens.

Para se obter uma expressão genérica, é necessário que, após a sua inserção na forma infixa, sejam cumpridas mais duas etapas: transformá-la da forma infixa para a forma pós-fixa e transformar a expressão pós-fixa em uma pós-fixa genérica. Estas etapas são mostradas na Figura 6.3.

A obtenção da representação pós-fixa a partir de uma expressão infixa é feita obedecendo às prioridades entre os operadores da linguagem corrente. A expressão infixa pode conter operadores genéricos oriundos de transformações de uma linguagem para outra, sofridas pelo programa que a contém. Nestes casos, as prioridades dos operadores genéricos devem ser mantidas, de forma a não afetar a avaliação desejada da expressão. Isto é conseguido dando ao operador genérico uma prioridade variável de acordo com o

Figura 6.3: Etapas para a transformação de uma expressão infixa para a sua respectiva forma pós-fixa genérica.

operador ao qual é comparado durante a manipulação da pilha para a construção da expressão pós-fixa.

A prioridade entre os operadores de cada linguagem é representada nas Figuras 6.4 e 6.5. Também se mostrada, para cada uma das linguagens a prioridade de operadores das outras linguagens (operadores dentro de retângulos) em relação aos seus. Na realidade, no lugar desses operadores indefinidos para a linguagem deveria estar a sua representação genérica; apenas para uma melhor compreensão, neste texto são conservadas as suas representações originais. Como por exemplo, o operador "!=" não existe na linguagem Pascal, mas possui uma prioridade, quando comparado com operadores desta linguagem. Sua prioridade é igual a 1 se comparado aos demais operadores, mas tem prioridade igual a 7 se comparado com um dos operadores ':=' ou 'in'.

Para ilustrar o processo de preparação de uma expressão para o seu armazenamento, mostra-se a seguir, uma expressão infixa editada sob o contexto da linguagem Pascal e a sua representação, ao atingir as etapas descritas na Figura 6.2. Considere-se a expressão infixa editada sob o domínio da linguagem Pascal:

$$a := b + c * e[10 \text{ div } b] / f \quad (1)$$

A expressão (1) na forma pós-fixa fica:

$$a \ b \ c \ e[10 \ b \ \text{div} \ ] \ * \ f \ / \ + \ :=$$

A representação genérica da expressão (1) é:

$$a \ b \ c \ e \ \#39\# \ 10 \ b \ \#5\# \ \#40\# \ \#3\# \ f \ \#4\# \ \#1\# \ \#8\# \quad (2)$$

PASCAL

14	(,) , [,]	
13	^ , . , ->	^ , & , - , ++ , -- , cast , sizeof 13
12	**	
11	not ,	& , - , ++ , -- , cast , sizeof
10	* , / , div , mod ,	and , & , ^ , 11
9	+	- , or , >> , << 10 or , & , ^ , 10
8	<< , >>	
7	= , <> , < , > , <= , >= ,	
	:= , in ,	& , ^ , , .EQV. , .NEQV. 8
	:= , in ,	op= 7
6	&	
5	^	
4		
3	.EQV.	
2	.NEQV.	
1	op=	
0	,	

C

15	(,) , [,] , . , ->
14	**
13	! , & , - , ++ , -- , cast , sizeof , *
12	* , / , %
11	+ , -
10	<< , >>
9	< , > , <= , >= , in
8	== , != , in
7	&
6	^
5	
4	
3	.EQV.
2	.NEQV.
1	= , op= , in*
0	,

op pode ser + , - , * , / , % , & , ^ , | , << ou >>
 in* tem prioridade 1 em relação aos demais operadores.

Figura 6.4: Prioridades entre os operadores das linguagens Pascal e C.

FORTRAN	
16	(,) , ^ , . , -> , *
15	**
14	& , - , ++ , -- , cast , sizeof , ^
13	* , / , % , mod , div
12	+ , -
11	<< , >>
10	.EQ. , .NE. , .LT. , .LE. , .GT. , .GE. , in
9	.NOT. , & , - , ++ , -- , cast , sizeof , ^
8	&
7	^
6	
5	.AND.
4	.OR.
3	.EQV.
2	.NEQV.
1	= , op= , in*
0	,

onde op pode ser +, -, *, /, %, &, ^, |, << ou >>
in* tem prioridade 1 em relação aos demais operadores.

Figura 6.5: Prioridades entre os operadores da linguagem FORTRAN.

Após uma expressão ser armazenada na base de dados, é possível obtê-la sob o contexto de qualquer uma das linguagens suportadas pelo Editor. Na recuperação de uma expressão genérica, para a sua respectiva representação na linguagem corrente, é efetuado o processo inverso ao do seu armazenamento. Este processo inicia-se a partir da expressão na forma pós-fixa genérica e completa-se, passando pelas seguintes etapas: conversão dos operadores genéricos da expressão em operadores equivalentes da linguagem corrente e transformação da expressão pós-fixa na expressão infixa. A figura 6.6 mostra esta transformação.

Para ilustrar esta transformação mostram-se a seguir as etapas descritas acima para a expressão genérica (2) na obtenção de suas respectivas representações, nas linguagens FORTRAN e C.

Expressão genérica (2) armazenada na base de dados:

a b c e #39# 10 b #5# #40# #3# f #4# #1# #8# (2)

Figura 6.6: Etapas para a transformação de expressões genéricas em suas respectivas representações infixa na linguagem corrente.

Expressão (2) com os operadores genéricos convertidos para os operadores equivalentes da linguagem FORTRAN:

$$a \ b \ c \ e(\ 10 \ b \ \#5\# \ ) \ * \ f \ / \ + \ := \quad (3)$$

Expressão (3) transformada para a forma infixa:

$$a \ = \ b \ + \ c \ * \ e(\ 10 \ \#5\# \ b \ ) \ / \ f \quad (4)$$

A seguir mostra-se a representação da expressão (2) na Linguagem C. Expressão (2) com os operadores genéricos convertidos para os operadores equivalentes da linguagem C:

$$a \ b \ c \ e[\ 10 \ b \ / \ ] \ * \ f \ \#4\# \ + \ = \quad (5)$$

Expressão (5) transformada para a forma infixa:

$$a \ = \ b \ + \ c \ * \ e[\ 10 \ / \ b \ ] \ \#4\# \ f \quad (6)$$

Pode-se notar que nas expressões (4) e (6) aparecem operadores genéricos, pois estes não possuem operadores equivalentes nas linguagens destino das transformações, ficando a critério do programador manter estes operadores na expressão, ou alterá-los, utilizando-se de operadores da linguagem corrente. Deve-se notar também que a prioridade entre os operadores da expressão (1) foi mantida através dos parênteses introduzidos nas respectivas expressões (5) e (6), o que permite obter a mesma expressão, caso se deseje transformá-la para a sua respectiva representação, na linguagem em que foi escrita originalmente.

6.3 - Índices do Tipo de Dado Vetor.

As expressões que definem o índice de um vetor também são armazenadas de um modo genérico, a fim de que seja possível a sua representação nas linguagens suportadas pelo Editor. Pode ocorrer que um índice armazenado deste modo não tenha uma representação possível, em algumas linguagens.

Um índice também é uma expressão, e, portanto, é introduzido pelo programador do mesmo modo como o descrito na seção anterior.

Após a introdução da expressão que descreve um índice lexicamente correto, o Editor cumpre algumas etapas antes do seu armazenamento na base de dados. A primeira etapa é obter, a partir desta expressão, as subexpressões que definem os seus limites inferior e superior. Caso não seja possível, na linguagem corrente, obter algumas dessas subexpressões, o Editor se incumbirá de gerá-las. A segunda etapa é transformar cada uma das subexpressões para a forma pós-fixa e, posteriormente, converter os seus operadores em operadores genéricos, obtendo duas subexpressões genéricas. A última etapa é armazenar as subexpressões como atributos dos respectivos tipos "inf" e "sup" do objeto corrente de tipo "expressão".

Durante a primeira etapa, após uma análise da expressão, o Editor definirá a categoria do índice que determina se as subexpressões que descrevem o índice são representáveis ou não nas demais linguagens suportadas pelo Editor. As categorias possíveis são:

a. índice_g: define os índices cujas subexpressões inferior e superior são representáveis em qualquer linguagem suportada pelo Editor. Suas subexpressões são constituídas do seguinte modo:

- inferior: 0,
- superior: c1,

onde c1 é uma constante inteira ≥ 0 .

Estas subexpressões são obtidas das seguintes definições de índices editadas sob o domínio das linguagens:

- **Pascal:**

expressão: 0 .. c1

- **FORTTRAN:**

expressão: 0 : c1

- **C:**

expressão: c2,

onde $c2 = c1 + 1$.

São armazenadas como atributos dos tipos de atributos "inf" e "sup" nas seguintes formas genéricas, se escritas sob o contexto da linguagem:

- **Pascal e FORTRAN:**

"inf": 0

"sup": c1

- **C:**

"inf": 0,   gerada pelo Editor.

"sup": c1,   onde $c1 = c2 - 1$.

b. índice_{pc}: corresponde aos índices cujas subexpressões inferior e superior só permitem serem representadas sob o contexto das linguagens Pascal e C. Suas subexpressões possuem a seguinte forma para o Editor:

. inferior: 0

. superior: <expressão-1>

São obtidas das expressões editadas sob o domínio das linguagens:

- **Pascal:**

expressão: 0 .. <expressão-1>

- C:

expressão: < expressão-2 >

onde < expressão-1 > e < expressão-2 > são expressões infixas que não representam um número natural.

São armazenadas em "inf" e "sup" como:

- Pascal:

"inf": 0

"sup": < expressão-3 >

- C:

"inf": 0, gerada pelo Editor.

"sup": < expressão-4 > 1 #2#,

onde < expressão-3 > é a < expressão-1 > na forma pós-fixa genérica e a < expressão-4 > é a < expressão-2 > na forma pós-fixa genérica.

c. índice_pf: representam os índices cujas subexpressões inferior e superior permitem serem apresentadas sob o contexto das linguagens Pascal e FORTRAN. Suas subexpressões possuem a seguinte forma para o Editor:

- inferior: c1,

onde c1 é uma const. inteira não nula.

- superior: c2,

onde c2 é uma constante inteira e $c1 \leq c2$.

São obtidas das expressões editadas sob o domínio das linguagens:

- Pascal:

expressão: c1 .. c2

- **FORTTRAN:**

expressão: c1 : c2

São armazenadas em "inf" e "sup" como:

"inf": c1

"sup": c2

d. índice_p: corresponde aos índices cujas subexpressões inferior e superior permitem serem representadas somente no contexto da linguagem Pascal.

- inferior: <expressão-1>
- superior: <expressão-2> ,

onde <expressão-1> e <expressão-2> não representam números naturais.

São obtidas das expressões editadas sob o domínio da linguagem:

- **Pascal:**

expressão: <expressão-1> .. <expressão-2>

São armazenadas em "inf" e "sup" como:

"inf": <expressão-1'> ,

"sup": <expressão-2'> ,

onde <expressão-1'> é a expressão <expressão-1> na forma pós-fixa genérica e a <expressão-2'> é a <expressão-2> na forma pós-fixa genérica.

Uma vez que o índice é armazenado, na base de dados, de modo genérico este poderá ser representado em qualquer linguagem, desde que a categoria definida pelo Editor permita esta representação.

Durante o processo de recuperação de um índice da base de dados sob o contexto de uma linguagem, o Editor verifica se a categoria do índice permite a sua representação nesta linguagem. Caso afirmativo, o Editor prepara a expressão para ser exibida; caso contrário, o índice não será representado na linguagem corrente.

A preparação de um índice a ser mostrado consiste em analisar a expressão e decidir o que deverá ser exibido desta expressão. A expressão resultante terá os seus operadores genéricos convertidos em operadores da linguagem corrente e posteriormente convertidos para a forma infixa. Estas etapas finais são as mesmas mostradas na Figura 6.6.

Estas transformações, para a obtenção da expressão que compõe, o índice são apresentadas a seguir.

Considerando que as informações na base de dados, nos respectivos tipos de atributos "inf", "sup" e "categoria" do objeto corrente do tipo "expressão" sejam as seguintes:

1. "inf": 0
"sup": c1
"categoria": índice_g,

então, a representação destas informações, como uma definição de índice, são transformadas pelo Editor, sob o contexto das respectivas linguagens, do seguinte modo:

- **Pascal:**

índice: 0 .. c1

- **C:**

índice: c2, onde $c2 = c1 + 1$.

- **FORTRAN:**

índice: 0 : c1

2. "inf": 0
"sup": <expressão-1>
"categoria": índice_pc

- **Pascal:**

índice: 0 .. <expressão-1'>

- **C:**

índice: <expressão-1'> + 1 ou
<expressão-2'> ,

onde <expressão-1'> é a <expressão-1> na forma infixa ou <expressão-1'> = <expressão-2'> - 1.

3. "inf": c1
"sup": c2
"categoria": índice_pf

- **Pascal:**

índice: c1 .. c2

- **FORTTRAN:**

índice: c1 : c2

4. "inf": <expressão-1>
"sup": <expressão-2>
"categoria": índice_p

- **Pascal:**

índice: <expressão-1'> .. <expressão-2'> ,

onde <expressão-1'> e <expressão-2'> são, respectivamente, as <expressões-1> e <expressão-2> transformadas para a forma infixa.

Para ilustrar as etapas mostradas nas Figuras 6.3 e 6.6, para as expressões que descrevem índices são apresentados alguns exemplos de edição, armazenamento e representação destas expressões, na Figura 6.7.

Linguagem para inserção	Edição	Representação Genérica	Apresentação	Linguagem para Apresentação
Pascal	0..5	inf:0 sup:5 Categoria: índice_g	0..5	Pascal
C	6		6	C
FORTTRAN	0:5		0:5	FORTTRAN
Pascal	0..a	inf:0 sup:a Categoria: índice_pc	0..a	Pascal
C	a + 1		a + 1	C
			<índice>	FORTTRAN
Pascal	-1..5	inf:0 sup:5 Categoria: índice_pf	-1..5	Pascal
			<índice>	C
FORTTRAN	-1:5		-1:5	FORTTRAN

Figura 6.7: Exemplos de expressões que descrevem índices.

Portanto, nota-se que, através das categorias de um índice, é possível ao Editor determinar se a expressão é representável ou não, na linguagem corrente, e, caso isto seja possível, determina como deverá ser exibida, efetuando em algumas situações uma preparação, antes de serem mostradas.

6.4 - Expressões da Estrutura de Controle "FOR".

As estruturas de controle de fluxo, denominadas pelo PEGSS "for" (ver Figura 3.1a e 3.1b), consistem de três expressões: expressão inicial, expressão de parada e expressão de incremento. Entre as linguagens suportadas pelo Editor, algumas podem ser omitidas, ou não definidas em sua sintaxe. Para o caso da inexistência de uma destas expressões, o Editor gera uma expressão que a represente, de modo que a transformação para a estrutura equivalente, em outra linguagem, possa ser efetuada.

Um exemplo das expressões de uma estrutura "for" é apresentado abaixo, pelo seu respectivo gabarito na linguagem C:

```
for(<expr.1>; <expr.2>; <expr.3>)
    <bloco>
```

onde

<expr.1> permite a descrição da sua expressão inicial e

<expr.2> permite a descrição da sua expressão de parada e

<expr.3> permite a descrição da sua expressão de incremento.

Como ocorre para as expressões escritas através do PEGSS, a introdução de uma das expressões que compõem a estrutura "for" só é validada após constatada sua correção léxica; caso contrário, uma nova expressão deverá ser introduzida.

Nas expressões lexicamente corretas, é feita uma análise para determinar a categoria da estrutura "for" que, assim como a categoria de um índice, permite determinar se a estrutura é representável nas demais linguagens suportadas pelo Editor.

Após esta análise, o Editor pode transformar as expressões introduzidas em expressões equivalentes, a fim de que a expressão a ser armazenada seja a mais genérica possível, com a vantagem de serem representáveis nas outras linguagens. Estas transformações podem ocorrer por substituição de operadores, inserção de novos operadores, ou de subexpressões.

As etapas seguintes são a de conversão da expressão para a sua respectiva representação na forma pós-fixa, e a conversão de seus operadores em operadores genéricos, preparando a expressão, para que seja armazenada na base de dados (Figura 6.3). A seguir, serão descritos, para cada possível categoria de uma estrutura "for", os tipos de expressões que o caracterizam e como são armazenadas na base de dados.

a. for_g: categoria para as estruturas "for" cujas expressões são representáveis em qualquer das linguagens suportadas pelo Editor. Suas expressões, escritas sob o contexto de qualquer das linguagens suportadas pelo Editor, são transformadas para as seguintes representações genéricas:

- expressão inicial:

```
<var> <expressão-1> #8#
```

- expressão de parada:

`<var> <expressão-2> #14#` ou

`<var> <expressão-2> #15#`

- expressão de incremento:

`<var> <var> 1 #1# #8#`

onde `<var>` é a variável de controle da estrutura "for",

`<expressão-1>`, `<expressão-2>` são expressões pós-fixas genéricas,

o operador genérico `#8#` representa um símbolo de atribuição (por exemplo o `:=` no Pascal),

os operadores genéricos `#14#` e `#15#` representam respectivamente `"<="` e `"<"`, e

o operador genérico `#1#` é o operador de adição.

Esta estrutura, segundo a sintaxe da linguagem Pascal, não permite a descrição da expressão de incremento. Neste caso, o Editor gera uma expressão genérica de incremento de um, para que esta possa ser utilizada na transformação para as demais linguagens. Por exemplo, para a estrutura "for" em Pascal com as seguintes expressões definidas pelo programador:

```
for i := 1 to a + b do
    <bloco>                                     (1)
```

seriam armazenadas na base de dados as seguintes informações referentes à esta estrutura:

expressão inicial:

`i 1 #8#`

expressão de parada:

`i a b #1# #14#`

expressão de incremento:

`i i 1 #1# #8#`

categoria: forg

Estas informações, assim como para as todas as estruturas "for", encontram-se armazenadas na base de dados do seguinte modo:

expressão inicial: descrito como atributo para o tipo "descrição" de um objeto cujo tipo é "inicial",

expressão de parada: descrito como atributo do topo "descrição" de um objeto cujo tipo é "parada",

expressão de incremento: descrito como atributo para o tipo "descrição" de um objeto cujo tipo é "incremento",

categoria: especificada como atributo para o tipo "categoria" de um objeto cujo tipo é "for".

Um exemplo de como ficam armazenadas na base de dados uma estrutura "for" é mostrado na Figura 6.8, onde a estrutura representada é a definida em (1).

Figura 6.8: DRI de uma estrutura "for".

As estruturas "for" desta categoria, cujas expressões são escritas utilizando-se de operadores restritos à linguagem corrente, terão suas expressões transformadas para uma representação genérica entre as linguagens. Por exemplo, a estrutura mostrada a seguir editada sob a linguagem C:

```
for(i = 1; i <= a+b; i++)           (2)
    <bloco>
```

As informações armazenadas na base de dados sobre esta estrutura são exatamente as mesmas do exemplo anterior. A expressão de incremento contendo o operador "+" é transformada para uma expressão representável para as outras linguagens.

Um outro exemplo de estruturas desta categoria, escrito sob o contexto da linguagem C, cujas expressões são transformadas para que a representação nas outras linguagens seja facilitada é o seguinte:

```
for(i = 1; i < a+b; i++)           (3)
    <bloco>
```

Neste caso, a expressão de parada é transformada (somente para o caso da expressão de incremento possuir passo um) para a seguinte expressão equivalente:

$$i \leq a+b-1$$

Deste modo a expressão de parada genérica seria a seguinte:

$$i \leq a + b - 1$$

As outras expressões genéricas ficariam do mesmo modo que as descritas para a estrutura "for" em (1). Com isto, a estrutura (3) que não poderia ser facilmente transformada para uma representação, sob o contexto das linguagens Pascal e FORTRAN, passa a permitir a conversão desta expressão.

As expressões de parada e incremento de uma estrutura desta categoria, escrita sob o contexto da linguagem FORTRAN, são transformadas em expressões equivalentes e possíveis de serem representadas nas outras linguagens. Por exemplo:

```
do 10, i = 1, a + b, 1
  <bloco>
10 continue
```

(4)

As informações armazenadas na base de dados correspondentes a esta estrutura são exatamente as mesmas utilizadas para representar as estruturas (1) e (2).

Nota-se, com isto, que as expressões de parada e de incremento sofreram o acréscimo da variável de controle e de alguns operadores, tornando-as mais fáceis de serem representadas nas outras linguagens.

b. forfc: categoria definida para as estruturas "for" cujas expressões são representáveis apenas para as linguagens C e FORTRAN. Estas estruturas caracterizam-se por possuírem expressões de incremento diferente de um, inclusive incrementos negativos. Após a edição de uma estrutura deste subtipo, as informações armazenadas na base de dados serão as seguintes:

- expressão inicial:

`<var> <expressão-1> #8#`

- expressão de parada:

`<var> <expressão-2> #14# ou`

`<var> <expressão-2> #15# ou`

`<var> <expressão-2> #12# ou`

`<var> <expressão-2> #13#`

- expressão de incremento:

`<var> <var> <expressão-3> #1# #8# ou`

`<var> <var> <expressão-3> #2# #8#`

onde: `<var>` é a variável de controle da estrutura "for",

`<expressão-1>`, `<expressão-2>` e `<expressão-3>` são expressões pós-fixas genéricas,

o operador genérico `#8#` representa um símbolo de atribuição (por exemplo o "=" em C),

os operadores genéricos `#12#`, `#13#`, `#14#` e `#15#` representam respectivamente ">=", ">", "<=" e "<",

os operadores genéricos `#1#` e `#2#` são respectivamente os operadores de adição e subtração.

Um exemplo de estrutura "for" desta categoria, escrita sob o contexto da linguagem FORTRAN, seria:

```
do 10, i = c, a + b, -10
  <bloco>
10 continue
```

As informações armazenadas na base de dados sobre esta estrutura são:

- expressão inicial:

`i c #8#`

- expressão de parada:

i a b #1# #12#

- expressão de incremento:

i i 10 #2# #8#

- categoria: forfc

O Editor determinará a expressão de parada desta estrutura após a inserção da expressão de incremento, pois o operador #12# de sua expressão é obtido, sabendo-se que o incremento da estrutura é negativo.

Outro exemplo de estrutura "for" desta categoria é a estrutura abaixo escrita, sob o domínio da linguagem C:

```
for(i = 1; i <= a; i += b+c)
    <bloco>
```

As informações a respeito desta estrutura armazenadas na base de dados são:

- expressão inicial:

i 1 #8#

- expressão de parada:

i a #14#

- expresssão de incremento:

i i b c #1# #1# #8#

- categoria: forfc

A expressão de incremento é transformada para uma expressão equivalente, mas com operadores existentes nas outras linguagens.

c. forc: categoria definida para as estruturas "for" cujas expressões são representáveis, apenas sob o contexto da linguagem C. Estruturas desta categoria caracterizam-se por possuírem, em suas expressões, subexpressões separadas pelo operador ";", ou na expressão de parada, o operador relacional utilizado ser o ">"; ou alguma das suas expressões ter sido omitida na descrição da estrutura. As expressões que descrevem uma estrutura "for" desta categoria não sofrem qualquer alteração proporcionada pelo Editor, pois uma vez que suas expressões são representadas apenas sob o contexto da linguagem C, estas não necessitam de alterações que as tornem representáveis nas outras linguagens.

Exemplificando uma estrutura "for" desta categoria:

```
for(i=1,j=i;i<a,j<=b;i++,j++)  
    <bloco>
```

As informações armazenadas na base de dados sobre esta estrutura são:

- expressão inicial:

```
i 1 #8# j 1 #8# #43#
```

- expressão de parada:

```
i a #10# j b #14# #43#
```

- expressão de incremento:

```
i #51# j #51# #43#
```

- subtipo: forc

Outros exemplos de estruturas desta categoria seriam os seguintes:

```
for (j=i; j>b; j--)  
    <bloco>
```

e

```
for (; j<=b; j++)  
    <bloco>
```

No primeiro exemplo, a expressão de parada da estrutura não permite a representação imediata desta estrutura, para uma equivalente na linguagem FORTRAN,

e, na linguagem Pascal, isto seria impossível. No segundo exemplo, a omissão da expressão inicial dificulta a determinação de uma estrutura que a represente nas outras linguagens. Para estes dois casos, assim como o exemplo anterior a estes dois, suas expressões pós-fixas genéricas armazenadas na base de dados, são obtidas diretamente das expressões descritas na estruturas, sem nenhuma alteração.

d. forf: categoria definida para as estruturas "for" cujas expressões são representáveis apenas sob o contexto da linguagem FORTRAN. Suas expressões são caracterizadas por possuírem, como expressão de incremento, expressões que não representem constantes inteiras, pois, para expressões de incremento desta forma, não é possível, ao Editor, determinar a expressão de parada genérica.

Um exemplo de estrutura "for" deste subtipo escrito sob o contexto da linguagem FORTRAN seria:

```
do 10, i = c, a, d + e
  <bloco>
10 continue
```

As informações sobre esta estrutura que são armazenadas na base de dados:

- expressão inicial:
i c #8#
- expressão de parada:
a
- expressão de incremento:
d e #1#
- categoria: forf

As estruturas deste subtipo terão suas expressões armazenadas na base de dados, na forma pós-fixa genérica, sem que, antes, suas expressões originais tenham sofrido modificações. Não é possível para o Editor, pelo menos no atual estágio, que se encontra implementado, determinar uma expressão de parada genérica, a partir de uma expressão de incremento, escrita desta forma.

e. forp: categoria definida para as estruturas "for" cujas expressões são representáveis apenas sob o contexto da linguagem Pascal. Esta categoria é dada a uma estrutura "for", escrita na linguagem Pascal, que contenha, ao menos, uma de suas expressões (início ou parada) ainda não definida.

Por exemplo, a estrutura "for" cuja expressão de parada ainda não tenha sido inserida:

```
for i = 1 to <expr> do
    <bloco>
```

As informações correspondente a esta estrutura armazenadas na base de dados são:

- expressão inicial:
i 1 #8#
- expressão de parada:
inexistente
- expressão de incremento:
i i 1 #1# #8#
- categoria: forp

Para que uma estrutura "for", escrita sob o contexto de uma linguagem, seja representada em outra linguagem verifica-se, primeiramente, se a categoria desta estrutura permite tal representação. Depois, o Editor determinará se as expressões que compõem a estrutura devem sofrer algum tipo de alteração, antes de serem representadas na linguagem corrente. Portanto, o processo de apresentação de uma estrutura desta, armazenada na base de dados, depende da sua categoria e da linguagem corrente.

Para a apresentação deste processo mostram-se para cada categoria da estrutura, as situações tratadas pelo Editor, em cada uma das linguagens por ele suportadas. As informações armazenadas na base de dados, para cada categoria, são as mesmas apresentadas anteriormente nesta seção.

Para a categoria:

a. for: as estruturas desta categoria podem ser representadas em qualquer linguagem. Considerando a linguagem corrente como:

a.1 Pascal:

- expressão inicial:
    `<var> := <expressão-1'>`
- expressão de parada:
    `<expressão-2'>`

a.2 C:

- expressão inicial:
    `<var> = <expressão-1'>`
- expressão de parada:
    `<var> <= <expressão-2'>`
- expressão de incremento:
    `<var> = <var> + 1`

a.3 FORTRAN:

- expressão inicial:
    `<var> = <expressão-1'>`
- expressão de parada:
    `<expressão-2'>`

- expressão de incremento:

1

onde, <var> é a variável de controle da estrutura,
<expressão-1'> e <expressão-2'> são, respectivamente, a <expressão-1>
e <expressão-2> transformadas para a forma infixa na linguagem corrente.

Como para esta estrutura, na linguagem Pascal, a expressão de incremento não é descrita, o Editor despreza-a, mostrando, apenas, as expressões inicial e de parada. Para a apresentação da expressão de parada somente a <expressão-2> é transformada para a forma infixa, desprezando-se o restante da expressão genérica pós-fixada, que se encontra na base de dados.

A apresentação desta estrutura, na linguagem C, é proporcionada, transformando-se as expressões armazenadas na base de dados em expressões infixas, sem que seja efetuada qualquer alteração para que estas possam ser exibidas.

Sob o contexto da linguagem FORTRAN, a apresentação desta estrutura é efetuada para as expressões inicial e de parada, do mesmo modo que sob o contexto da linguagem Pascal. Para a exibição da sua expressão de incremento é obtida, apenas, a constante que define o incremento de um.

b. forfc: a transformação das estruturas desta categoria são possíveis apenas nas linguagens FORTRAN e C. Considera-se a linguagem corrente como:

b.1 C:

- expressão inicial:

<var> = <expressão-1'>

- expressão de parada:

<var> <= <expressão-2'> ou

<var> >= <expressão-2'>

- expressão de incremento:

$\langle \text{var} \rangle = \langle \text{var} \rangle + \langle \text{expressão-3'} \rangle$ ou

$\langle \text{var} \rangle = \langle \text{var} \rangle - \langle \text{expressão-3'} \rangle$

b.2 FORTRAN:

- expressão inicial:

$\langle \text{var} \rangle = \langle \text{expressão-1'} \rangle$

- expressão de parada:

$\langle \text{expressão-2'} \rangle$

- expressão de incremento:

$\langle \text{expressão-3'} \rangle$

- $\langle \text{expressão-3'} \rangle$

onde, $\langle \text{var} \rangle$ é a variável de controle da estrutura;

$\langle \text{expressão-1'} \rangle$, $\langle \text{expressão-2'} \rangle$ e $\langle \text{expressão-3'} \rangle$ são, respectivamente, as expressões $\langle \text{expressão-1} \rangle$, $\langle \text{expressão-2} \rangle$ e $\langle \text{expressão-3} \rangle$ transformadas para a forma infixa na linguagem corrente.

Sob o contexto da linguagem C, as expressões desta estrutura são obtidas diretamente das respectivas expressões genéricas que se encontram armazenadas na base de dados, obedecendo às etapas definidas na figura 6.6.

No entanto, sob o contexto da linguagem FORTRAN, as expressões de parada e incremento desta estrutura são obtidas através das subexpressões $\langle \text{expressão-2} \rangle$ e $\langle \text{expressão-3} \rangle$ armazenadas na base de dados, as quais devem obedecer também às etapas definidas na Figura 6.6, para as suas apresentações.

c. forc: suas expressões são obtidas apenas cumprindo-se as etapas descritas na Figura 6.6.

d. forf: mesmo que o anterior.

e. forp: mesmo que o anterior.

Para as estruturas cujas categorias estão descritas nos itens c, d e e não há alteração nas suas expressões, tanto nos seus armazenamentos, quanto para as suas recuperações. Isto porque a apresentação destas estruturas está restrita a apenas uma linguagem, respectivamente, Pascal, C e FORTRAN.

6.5 - Considerações Finais.

O tratamento de expressões é uma das tarefas mais complexas do PEGSS, pois, além de depender do tratamento dado ao restante da linguagem, possui uma diversidade de sintaxe e aplicação muito grande. O tratamento não levou em consideração a correção de tipo das variáveis e constantes. Embora essa análise possa ser feita, seguindo-se as mesmas regras adotadas pelos analisadores semânticos dos compiladores usuais das linguagens tratadas, o aumento da complexidade do Editor não caberia no propósito deste trabalho.

Por outro lado, a análise de expressões no contexto mais amplo de um programa, e não apenas de um comando, fica bastante facilitado, pelo enfoque adotado de se representar cada expressão de maneira mais genérica possível, ao mesmo tempo em que essa análise pode permitir um uso mais intenso de expressões genéricas. Por exemplo, alguns operadores que não possuem correspondente numa linguagem pode ser expresso por outros que existam. Mas para que essa prática seja completa, é necessário que mesmo nas linguagens onde um operador é reconhecido, a existência de uma sub-expressão que pode ser "compactada" em um operador deve ser reconhecida também. Isso corresponde, por exemplo a se poder intercambiar a expressão:

```
a = i ++;
```

na linguagem C para as expressões:

```
a := i;  
i := I + 1;
```

na linguagem Pascal, e vice-versa.

A extensão do PEGSS, para efetuar este tipo de análise, é viável, e por si só, forneceria um mecanismo inicial de otimização de código fonte, ou mesmo um mecanismo de padronização automática de formato de codificação e ou documentação de programas.

Capítulo 7

ASPECTOS GERAIS DA IMPLEMENTAÇÃO

7.1 - Considerações Iniciais.

Neste Capítulo são apresentados os principais módulos utilizados na implementação do PEGSS, especificadas as estruturas utilizadas pelo Sistema para a manipulação das hierarquias de programas, seus módulos e o funcionamento desses módulos. Também são descritos os principais módulos utilizados para a criação e recuperação de programas armazenados na base de dados e a manipulação das informações, por estes módulos.

A versão atual do PEGSS foi desenvolvida em computadores da linha IBM PC-XT, tendo como sistema operacional o MS-DOS e implementado em linguagem de programação "C". No entanto o Sistema foi especificado levando-se em consideração a continuação do seu desenvolvimento em estações de trabalho, sob o sistema operacional UNIX.

O PEGSS utiliza-se de rotinas que fazem parte da Interface de Programação de Aplicações (IPA) do GEO e são descritas em [GEO_91]. Essas rotinas permitem manipular as informações na base de dados, durante a fase de edição e recuperação de programas.

7.2 - Hierarquia.

Cada programa escrito, usando o PEGSS, possui uma estrutura hierárquica, para armazenar as informações a respeito da sua representação e localização na base de dados. Esta estrutura permite que um programa escrito em uma linguagem seja exibido em vídeo, onde a sua edição pode ser empreendida. Sobre esta estrutura são permitidas operações para criar, mostrar, editar e eliminar um programa.

7.2.1 - Definição dos elementos de uma hierarquia.

As estruturas hierárquicas são mantidas em memória, para agilizar o acesso interativo do usuário ao programa. Elas são montadas, baseadas também em informações armazenadas na memória, e, podem sofrer alterações, devido a atuação do usuário ao editar o programa. Estas alterações provocam atualizações nas informações a respeito do programa que encontram-se armazenadas na base de dados. Isto é concretizado através de informações obtidas na composição hierárquica do programa.

As informações necessárias para a manipulação de uma estrutura hierárquica, encontram-se em dois tipos de elementos:

a. Estrutura de controle da hierarquia - contém informações gerais sobre uma hierarquia, necessárias pelas rotinas no momento de sua manipulação. A definição desta estrutura é mostrada a seguir:

```
struct MH_Hierarquia{

    pal2    sentinela;
    struct MH_Texto *inicio,
                                *final,
                                *corrente;

    pal2    cor[4],
            nivelatual,
            lintopo,
            coltopo;

    codiobjeto co;
    enum defestado{
        ALTERADO,
        PREPARADO,
        PROCESSANDO
    } estado;

    struct MH_Texto *nivcorrente[MAXNIV+1];
```

onde os seus campos armazenam as seguintes informações:

- sentinela:** campo de segurança para garantir que os dados desta estrutura não encontram-se danificados;
- início:** indicador para o primeiro nó da hierarquia;
- final:** indicador para o último nó da hierarquia;
- corrente:** indicador para o nó corrente da hierarquia;
- cor[4]:** cores utilizadas para a exibição da janela e dos nós contidos em seu interior;
- nivelatual:** indica o nível do nó corrente da hierarquia;
- lintopo:** linha do nó que está no topo da janela;
- coltopo:** coluna do nó que está no pé da janela;
- co:** código do objeto que restringe a colônia que contém o programa na base de dados;
- estado:** estado atual de preparação da hierarquia cujos valores podem ser:
- **ALTERADO:** indica que a hierarquia não está preparada;
  - **PREPARADO:** indica que a hierarquia está pronta para processar;
  - **PROCESSANDO:** indica que a hierarquia já foi processada ao menos uma vez;
- nivcorrente:** pilha contendo os últimos nós de cada nível, utilizada como estrutura auxiliar para a execução de algumas funções.

b. Nós da hierarquia - Um nó armazena informações sobre cada elemento que compõem uma estrutura sintática (itens sintáticos) a ser apresentada em um programa, além das informações sobre as ligações com os outros nós da hierarquia, o local que o nó deverá ser exibido na janela e outras informações. A definição de um nó utilizada na implementação é descrita detalhadamente a seguir:

```
struct MH_Texto{
    pal2    sentinela;
    struct MH_Texto *previo,
                *proximo,
                *pai,
                *filho;

    cadeia cad;
    pal2    inclin,
            inccol,
            Idl,
            nivel,
            lin,
            colinic,
            col,
            podeditar,
            mostraprox;
    codiobjeto
        co;
}
```

onde cada um dos seus campos contém as seguintes informações:

sentinela:	campo de segurança para garantir que os dados deste nó não encontram-se danificados;
prévio:	indicador para o nó anterior a este neste nível;
próximo:	indicador para o próximo nó no mesmo nível;
pai:	indicador para o nó pai a este no nível imediatamente anterior;
filho:	indicador para o primeiro nó um nível depois;
cad:	contém a sequência de caracteres que descrevem o item armazenado neste nó;
inclin:	número de linhas a serem avançadas em relação à linha inicial do nó anterior, para o início da apresentação do item neste nó;
inccol:	número de colunas a serem avançadas em relação a coluna inicial determinada para o item deste nó durante a preparação da hierarquia;
idl:	identificador do item deste nó;
nível:	nível deste nó na hierarquia;

lin: linha inicial do item neste nó para a sua exibição na janela;
colinic: coluna inicial para apresentação do item contendo o caractere "@";
col: coluna inicial do item neste nó para a sua exibição na janela;
podeditar: indica se o item é editável;
mostraprox: indica se o item do próximo nó no mesmo nível que este deve ser mostrado em destaque;
co: código do objeto na base de dados representado neste nó.

O Apêndice A mostra o conjunto de todas as composições de nós, para os comandos da linguagem Pascal, representadas nesta estrutura hierárquica. Para facilitar o entendimento, nesse Apêndice é mostrada uma representação parcial dos ponteiros que estruturam cada uma destas composições.

7.2.2 - Operações básicas em hierarquias.

As rotinas implementadas para a manipulação de hierarquias de programas permitem que algumas operações sejam utilizadas pelo Editor, a fim de que as estruturas, nela representadas, possam ser mostradas, editadas e eliminadas, além da sua criação. Estas operações e as rotinas que as executam são as seguintes:

- Criar uma hierarquia de programa: `Mh_Cria()`;
- Incluir novos itens sintáticos: `Mh_Poetexto()`;
- Substituir um item no nó corrente: `Mh_Substexto()`;
- Preparar uma hierarquia para o processamento: `Mh_Prepara()`;
- Processar uma hierarquia: `Mh_Processa()`;
- Eliminar uma hierarquia: `Mh_Elimina()`.

Para poder inserir algum item sintático (nó) em uma hierarquia, esta deve ser previamente criada com o uso da rotina `Mh_Cria()`. Isto ocorre sem nenhum item (nó), cuja posterior incorporação devem ser feita, através de sucessivas chamadas da rotina `Mh_Poetexto()`. A Figura 7.1 mostra através das representações (1), (2) e (3) os modos possíveis de inserções em um hierarquia. Os possíveis lugares para a inserção dos nós estão

representados por nós pontilhados. O nó corrente, em uma inserção, é o posicionado por uma seta. Os itens podem ser inseridos como nós terminais da hierarquia (1), entre dois outros itens (nós) em níveis diferentes (2), entre dois outros itens (nós) em um mesmo nível e no final de qualquer nível anterior ao nível do item corrente (3).

Se um item deve ser alterado, isto é feito através da rotina `Mh_Subtexto()` que permite a troca das informações que o descrevem, pelas informações de um novo. As informações sobre o novo item que poderão substituir as antigas são:

- **id:** identificador do novo item;
- **cad:** sequência de caracteres que descreve o novo item;
- **co:** código do objeto que representa o novo item na base de dados;
- **inclin:** novo avanço de linha;
- **Inccol:** novo avanço de coluna;
- **edita:** indica se o novo item é editável ou não;
- **mostrap:** indica se o próximo item no mesmo nível deve ser mostrado em destaque;

Figura 7.1: Representação das possíveis inserções na hierarquia.

Como se pode notar, as informações sobre as ligações que envolvem o nó não são alteradas em uma substituição de itens, através desta rotina, o que mantém a posição do nó, na estrutura hierárquica.

Não foi prevista uma rotina para a eliminação de um nó da hierarquia, assim, se a alteração a ser efetuada for na estrutura da hierarquia, é necessário eliminar toda a hierarquia através da rotina `Mh_Elimina()`, e reconstruí-la novamente. Na atual implementação do Editor, a eliminação de um nó da hierarquia se efetua apenas em algumas situações pela rotina `leatributo()`, descrita na seção seguinte. Isto ocorre somente como auxílio na montagem da hierarquia de um programa, obtido da base de dados.

Uma vez pronta, a hierarquia deve ser preparada para o processamento, o que é se

faz através da rotina `Mh_Prepara()`. Nesta etapa, o item é preparado de acordo com os requisitos de endentação de linha e colunas, definidos pelo Editor. Também é levado em consideração se o número de caracteres que descrevem o item não ultrapassa o limite máximo de colunas possíveis de serem utilizados pelo item. Um item com caracteres "@" terá o seu tamanho verificado após serem substituídos pelos respectivos itens dos outros nós, como já citado na seção 4.4. Os trechos do item que ultrapassam o limite máximo de colunas serão alocados nas linhas seguintes.

O processamento de uma hierarquia é efetuado pela rotina `Mh_Processa()` que mostra cada item da hierarquia na janela do Editor, de acordo com as suas linhas e colunas iniciais, definidas pelo `Mh_Prepara()`. Os caracteres dos itens que pertencem à estrutura sintática corrente são mostrados, em destaque, e os demais, no modo normal. Durante o processamento da hierarquia são permitidas ao usuário a "navegação" e a edição sobre a hierarquia. Através das ligações entre os nós da hierarquia é possível "navegar" de um nó para outro.

7.3 - Criação e Recuperação de Programas .

7.3.1 - Criação de Programas.

Durante o processo de criação de um programa, algumas informações são atualizadas, simultaneamente, tanto na hierarquia que o representa, quanto na base de dados, mais especificamente na colônia que contém as informações sobre este programa. Na hierarquia, são adicionados ou substituídos nós, contendo informações sobre o novo gabarito ou expressão. Tais informações são compostas, entre outras, pelos itens sintáticos que descrevem o gabarito na linguagem corrente e pelo código do objeto que o representa. Na base de dados, são criados os objetos, relacionamentos e atributos que representam as informações descritas no gabarito ou expressão.

Um programa é criado a partir da introdução do nome do arquivo que o conterà. Esta informação é armazenada na base de dados, através da criação de um objeto do tipo "arquivo", cujo nome é o mesmo do programa introduzido pelo programador. Este objeto restringirá uma colônia do tipo "Arquivo" contendo todas as informações a respeito deste programa, na base de dados. Portanto, existirá, na base de dados, uma colônia para cada programa criado. A Figura 7.2 mostra um exemplo da representação das colônias dos

programas "prog1" e "prog2", armazenados na base de dados.

Como, para o Editor, todo arquivo (programa) contém uma subrotina principal, esta também deverá ser armazenada, inicialmente, na base de dados. Isto ocorre através da criação de um objeto do tipo "principal", além de um relacionamento criado, associando os objetos de tipo "arquivo" e "principal".

Figura 7.2: Representação das colônias de programas na base de dados.

Todo programa criado sob o domínio do PEGSS terá inicialmente algumas informações iniciais, colocadas na base de dados, pelo Editor. Por exemplo, estas informações estão representadas no Diagrama de Representação de Instâncias (DRI) da Figura 7.3 para o arquivo (programa) de nome "teste".

No início de criação de um programa, além destas informações que são tratadas pelo Editor, são também criadas, em memória, as informações para a concepção inicial da hierarquia que irá representá-lo.

Figura 7.3: DRI sobre as informações que são inicialmente criadas pelo PEGSS para um arquivo/programa de nome "teste".

Os primeiros nós da hierarquia contêm os itens sintáticos que compõem o gabarito inicial de uma programa e o código do objeto de tipo "principal", além de outras informações, utilizadas pelo Editor. Considerando que o arquivo (programa) "teste" esteja sendo editado sob o contexto da linguagem Pascal, a sua hierarquia inicial a mostrada na Figura 7.4., onde col1 é o código do objeto de tipo "principal".

O processamento desta hierarquia permitirá que os itens sintáticos que compõem o gabarito inicial de um programa, na linguagem Pascal, seja apresentado no formato:

```
program <nome>;
  <declaração>
  <decl.subrot.>
  <bloco>
```


As rotinas do Editor que determinam quais informações devem ser introduzidas, na hierarquia e na base de dados, para o gabarito inicial de um programa qualquer, ou para cada requisição feita pelo programador de inserção de gabarito, ou para as expressões inseridas, são as seguintes:

Figura 7.4: Hierarquia inicial do arquivo/programa "teste".

- `editapas()`: trata da edição de programas sob o contexto da linguagem Pascal.
- `editac()`: trata da edição de programas sob o contexto da linguagem C.
- `editafor()`: trata da edição de programas sob o contexto da linguagem FORTRAN.

Cada uma destas rotinas é constituída pelo conjunto de manipuladores para cada comando da linguagem correspondente. Um trecho de uma destas rotinas é apresentado a seguir:

```

pal2 editapas(item, con, codH)
  pal2 item;          /* num. do lugar de inserção requisitado */
  codiobjetocon;      /*codigo do objetodonó corrente          */
  hierarquia codH;    /* hierarquia do programa corrente      */
  {
    pal2 ret;
    defcadeia(tipo,    MAXCAD);
    defcadeia(subtipo, MAXCAD);
    defcadeia(relac,   MAXCAD);

    switch(item){

      case INICIO_ED:
        /* item que inicia um programa em Pascal */
        kdef(tipo, MAXCAD, "subrotina");
    }
  }

```

```

    kdef(subtipo, MAXCAD, "principal");
    ret = edcriaobj(tipo, subtipo, &cocl, &erro);
    if (ret != 0) mensagen(5, erro);
    kdef(relac, MAXCAD, "contem_prog");
    ret = edcriarel(con, relac, cocl, &erro);
    if (ret != 0) mensagen(5, erro);
    editem(INS, 46, cocl, 1, codH);/* program @;    */
    editem(INS, 44, cocl, 1, codH);/* <nome>        */
    editem(INS, 13, cocl, 0, codH);/* <declaraç.>   */
    editem(INS, 17, cocl, 0, codH);/* <decl.sub.>  */
    editem(INS, 03, cocl, 0, codH);/* <bloco>     */
    break;

case 13: /* <declaração> */
    editem(INS, 42, con, 0, codH);/* label @;      */
    editem(INS, 16, con, 1, codH);/* <decl.rot.> */
    editem(INS, 12, con,-1, codH);/* const       */
    editem(INS, 22, con, 1, codH);/* <decl.con.> */
    editem(INS, 57, con,-1, codH);/* type        */

    editem(INS, 24, con, 1, codH);/* <def.tipo>  */
    editem(INS, 60, con,-1, codH);/* var         */
    editem(INS, 19, con, 1, codH);/* <decl.var.> */
    break;

.
.
.

} /* fim switch */
return(0);
} /* fim editapas */

```

Neste trecho, é representado apenas o item que inicia um programa, na linguagem Pascal e o item que permite inserir as definições e declarações, em uma subrotina. Todos os outros itens que correspondem a lugares de inserção dos gabaritos da linguagem Pascal também são tratados por esta rotina.

As rotinas requisitadas pelo `editapas()`, `editac()` ou `editafor()`, tais como

edcriaobj(), edcriarel() e editem() é que tratam as informações, na base de dados e hierarquia.

A rotina editem() trata da inserção ou substituição de uma nó na hierarquia que representa o programa escrito. Esta rotina é apresentada abaixo:

```
#include <stxpas.h> /* define os itens do Pascal      */
#include <stxc.h>    /* define os itens do C          */
#include <stxfor.h> /* define os itens do FORTRAN     */
#include <hrot.h>    /* declarações para a manipulação de */
                  /* hierarquias                          */

pal2 editem(modo, item, coc, incrnivel, codH)
pal2 modo; /* INS:inserção ou SUB:substituição do nó */
pal2 item; /* número do lugar de inserção repres. no nó */
codiobjeto co; /* código do objeto representado no nó */
pal2 incrnivel; /* increm. de nível do nó na hierarquia */
hierarquia codH; /* hierarquia que representa o programa */
{
    defcadeia(nomeitem, MAXCAD);
    byte nivel, ret;

    switch(LING){

        case PASCAL:
            aux      = item_pas[item].nome;
            inclin   = item_pas[item].inclin;
            inccol   = item_pas[item].inccol;
            edita    = item_pas[item].edita;
            mostrap  = item_pas[item].mostrap;
            break;

        case C:
            aux      = item_c[item].nome;
            inclin   = item_c[item].inclin;
            inccol   = item_c[item].inccol;
            edita    = item_c[item].edita;
            mostrap  = item_c[item].mostrap;
            break;
```

```
case FORTRAN:
    aux      = item_for[item].nome;
    inclin   = item_for[item].inclin;
    inccol   = item_for[item].inccol;
    edita    = item_for[item].edita;
    mostrap  = item_for[item].mostrap;
    break;

} /* fim do switch /
kdef(nomeitem,MAXITEM," "); nomeitem[1]--;
kcopp( kcvf(aux), 1, aux[1], nomeitem, 1);
nivel = codH->nivelatual + incrnivel;

if (modo == SUB) /* subst. nó na hierarquia */
    ret = edsubtx(nomeitem,item,co,inclin,inccol,edita,
                  mostrap,codH);
else /* insere nó na hierarquia */
    ret = Mh_Poetexto(nomeitem,item,co,inclin,inccol,
                      edita,mostrap,nivel,codH);
if (ret != 0){
    Mh_Trataerro(ret);
    return(1);
}
return(0);
} /* fim do editem */
```

As rotinas `edsubtx()`, `Mh_Poetexto()` e `Mh_Trataerro()` são rotinas para a manipulação de hierarquias e têm como função, respectivamente, substituir um nó, colocar um novo nó e emitir mensagens de erro.

Algumas das rotinas que tratam das informações a serem armazenadas, na base de dados, são mostradas a seguir.

A rotina `edcriaobj()` permite criar um objeto, na base de dados, através dos seus nome de tipo e subtipo. O código do objeto criado é retornado por essa rotina.

```
#include <mro.h>

pal2 edcriaobj(nometp, nomesub, coc, erro)
cadeia nometp; /* nome do tipo do objeto a ser criado */
cadeia nomesub; /* nome do subtipo do objeto */
codiobjeto *coc; /* código do objeto criado */
pal2 *erro; /* num. do erro obtido */
{
    coditipo ctatipo,
            ctas,
            cto;

    byte ret;
    union B_valores valor;
    defcadeia(cad, MAXCAD);

    *erro = CodATR("-.tipo", &ctatipo);
    if (*erro != 0) return(1);
    if ((*erro = CodOBJ(nometp, &cto)) != 0) return(1);

    /* cria um obj. sem ident. */
    if ((*erro = CNOBJ(cto)) != 0)
        return(1);
    if (kcompv(nomesub, "nenhum") != 0){ /* existe subtipo */
        if ((*erro = CodATR(nomesub, &ctas)) != 0)
            return(1);
        itok( (pal2) ctas, cad);
        if ((*erro = GTPRO(cad, ctatipo, &valor)) != 0)
            return(1);
        if ((*erro = CTPRO(valor, ctatipo)) != 0)
            return(1);
    }
    return(0);
} /* fim do edcriaobj */
```

As rotinas seguintes fazem parte da Interface de Programação de Aplicações (IPA) do GEO:

- CodATR: obtém o código do tipo de atributo,
- CodOBJ: obtém o código do objeto,
- CNOBJ: cria um objeto sem identificador,
- GTPRO: converte uma cadeia em atributo e
- CTPRO: coloca um atributo no objeto corrente,

A rotina a seguir permite colocar um relacionamento entre os objetos origem e destino, especificados através de seus códigos:

```
#include <mro.h>

pal2 edcriarel(coo, nomemod, cod, erro)
codiobjeto coo; /* codigo do objeto origem */
cadeia nomemod; /* nome da modalidade */
codiobjeto cod; /* codigo do objeto destino */
pal2 *erro;      /* erro na base de dados */
{
    coditipo ctmod,
            ctas,
            cto;

    if ((*erro = PNOBJ(coo)) != 0) return(1);
    if ((*erro = CodMOD(nomemod, &ctmod)) != 0) return(1);
    if ((*erro = CTREL(ctmod, cod)) != 0) return(1);
    return(0);
} /* fim do edcriarel */
```

As rotinas também fazem parte do IPA do GEO:

- PNOBJ: posiciona em um objeto através do seu código,
- CodMOD: obtém o código da modalidade e
- CTREL: associa o objeto corrente ao objeto destino através da modalidade especificada,

A maioria das expressões escritas sob o domínio do PEGSS são armazenadas como atributo de tipo "descrição", em um objeto de tipo "expressão". O tamanho máximo destas expressões é de (4*MAXCAD) caracteres, ou seja, 4 atributos do tipo "descrição", onde cada

atributo tem tamanho máximo de MAXCAD caracteres. Este tamanho máximo, para uma expressão, é limitado durante a inserção da expressão.

A rotina que permite a inserção de expressões deste tipo, na base de dados, é mostrada abaixo:

```
#include <mro.h>

pal2 edinsprop(tipo, valor, co, erro)
cadeia tipo; /* nome do tipo do atributo */
cadeia valor; /* atributo a ser inserido */
codiobjeto co; /* cod. do obj. que contem o tipo de atrib. */
pal2 *erro; /* erro na base de dados */
{
    coditipo cta;
    byte res,
        numval,
        i, j;
    defcadeia(cad, MAXCAD);
    union B_valores valorp;

    if ((*erro = CodATR(tipo, &cta)) != 0) return(1);
    if ((*erro = PNOBJ(co)) != 0) return(1);
    numval = valor[1] / MAXCAD;
    res = 0;
    if (numval == 0) numval++;
    else if (numval >= 1){
        res = valor[1] % MAXCAD;
        if (res != 0) numval++;
    }
    else return(1);
    i = 1;
    j = MAXCAD;
    while (numval >= 1){
        cad[1] = 0;
        kcopp(valor, i, j, cad, 1);
        GTPRO(cad, cta, &valorp);
        if ((*erro = CTPRO(valorp, cta)) != 0) return(1);
    }
}
```

```
    numval--;  
    i = j + 1;  
    if (numval == 1) j += res;  
    else j += MAXCAD;  
  }  
  return(0);  
} /* fim do edinsprop */
```

7.3.2 - Recuperação de Programas.

Um programa escrito sob o domínio do Editor, em uma linguagem, pode ter a sua representação sob o contexto de outra linguagem, entre as suportadas pelo Editor, graças ao modo único em que é armazenado, definido pela concepção lógica da DLA, e por seu modo de recuperação.

As informações de realização da leitura e das informações do programa a serem recuperadas são obtidas na base de dados através dos Esquemas de Recuperação de Programas (ERP), que se compõem em um Esquema para cada linguagem, suportada pelo Editor. Na atual versão do PEGSS, estes Esquemas são ERPPas, ERPC e ERPFOR. Simultaneamente, são inseridas, na hierarquia do programa, informações que lhe permitem representá-lo no vídeo e informações que permitem associar cada uma das suas estruturas com as suas respectivas informações, na base de dados.

Os objetos das colônias que contêm os ERP possuem as informações, obedecendo à construção sintática de cada linguagem, e à forma de leitura, na base de dados, dos objetos de mesmo tipo, na colônia que contém as informações de um programa. Logo, para todo tipo de objeto que aparece na colônia que contém o programa existe um objeto de mesmo tipo na colônia que contém o ERP da linguagem corrente que permite a sua recuperação. Estas informações, como já apresentado na seção 5.6, são atributos do tipo "lebase" e, para os tipos de objetos que possuam subtipos, estas informações são complementadas com os atributos do seu tipo de atributo cujo nome é o mesmo que o atributo que define o subtipo.

A seguir, é descrito como é efetuada a recuperação de um programa através do PEGSS. Para isto é utilizado um trecho do arquivo "teste" armazenado na base de dados, onde apenas as informações sobre uma estrutura "ifte" é destacada. Também é utilizado o ERPPas, pois a recuperação do programa ocorre sob o contexto da linguagem Pascal. A

Figura 7.5 mostra parte dos conteúdos das colônias utilizadas para a leitura do arquivo "teste".

Figura 7.5: Representação das colônias contendo o ERPPas e o programa "teste".

As informações armazenadas nos tipos de atributos "lebase" e "ifte" do comando "c1" na colônia que contém o ERPPas são apresentadas mais detalhadamente seguir:

tipo do objeto	tipo do atributo	atributos
comando	lebase	TIPO FIM
ifte	ifte	SUBS if(0) 0 INSE <condição> 1 RELA verifica_cond INSE then 0 INSE <bloco> 1 RELA executa_caso_v INSE else -1 INSE <bloco> 1 RELA executa_caso_f INSE <comando> -2 RELA segue FIM

O tipo de objeto "comando" possui outros tipos de atributos como "ift", "while", "do", etc, que não estão sendo apresentados neste exemplo. A descrição completa das informações contidas nos Esquemas de Recuperação de Programas das linguagens suportadas pelo PEGSS é encontrada em [Val_92]. No Apêndice C é mostrado apenas as informações do ERPPas.

A leitura de um programa é efetuada pela rotina `leatributo()`, que recebe do Editor as informações a respeito do Esquema de Recuperação a ser utilizado, através do código do objeto que restringe a colônia que contém o respectivo ERP e as informações do programa a ser lido, através do código do objeto que restringe a colônia que o contém. Outras informações também são passadas inicialmente para esta rotina, como o tipo de atributo do objeto que restringe a colônia com o ERP, no caso "lebase", e um indicador para a hierarquia que armazenará a representação do programa que deve ser lido. Estas informações são passadas pelo Editor, através dos parâmetros definidos para esta rotina, como pode ser verificado na sua descrição abaixo:

```
#include <mro.h>
#include <hrot.h>
#include <edicao.h>
#include <expr.h>

pal2 leatributo (ctapc, coerp, coprog, coc, ctoc, codH, erro)
coditipo ctapc; /*código do atributo propriedade corrente*/
codiobjeto coerp, /*código do objeto que restringe a colô-*/
                /*nia com o ERP. */
coprog,          /*código do objeto que restringe a colô-*/
                /*nia que contém o programa. */
coc;             /*código do objeto corrente. */
coditipo ctoc;   /*código do tipo de objeto corrente. */
hierarquia codH; /*hierarquia que contém o programa. */
pal2          *erro; /*indicação de erro. */
```

Utilizando-se das duas colônias, representadas na Figura 7.5, a rotina `leatributo()` vai obtendo as informações de como ler e o que ler do programa. Por exemplo, no momento em que esta rotina estiver recuperando a estrutura "ifte", após ter obtido as informações sobre outras estruturas representadas na base de dados, as informações passadas, através dos seus parâmetros são:

ctapc:	código do tipo de atributo "lebase";
coerp:	código do objeto que contém o ERPPas;
coprog:	código do objeto que contém o programa "teste";
coc:	código do objeto de tipo "comando" (c2) na colônia que contém o programa;
ctoc:	código do tipo de objeto "comando".

Em seguida é ativada a colônia com o ERPPas e inicia-se a leitura das informações no tipo de atributo "lebase" do tipo de objeto "comando". No seu primeiro atributo, está especificada a instrução TIPO que determina, para rotina `leatributo()`, a leitura do atributo de tipo "tipo" do objeto corrente na colônia que contém o programa, no caso o objeto de código "c2". Neste atributo está especificado que o subtipo do objeto "c2" é o definido pelo atributo "ifte" e, portanto, a rotina deve voltar à colônia com o ERPPas e

obter as informações de recuperação de objetos deste subtipo. Tais informações estão descritas como atributos do tipo "ifte" do objeto de mesmo tipo que o posicionado na colônia que contém o programa, no caso o tipo "comando". Estes atributos descrevem instruções que determinam quais relacionamentos do objeto de código "c2" devem ser posicionados para obter as informações sobre a condição e os blocos da estrutura "ifte" e, conforme estas informações vão sendo obtidas, sua forma de colocação na hierarquia que representa o programa "teste". A cada instrução RELA, encontrada em alguns destes atributos, é determinada uma chamada recursiva da rotina `leatributo()` que inicia o processo de obtenção das informações, no tipo de atributo "lebase", para um outro tipo de objeto. Ao final da leitura dos atributos do tipo "ifte", na colônia com o ERPPas, a rotina volta, para terminar a leitura dos atributos do tipo "lebase", ainda nesta colônia, cuja instrução FIM finaliza a obtenção das informações da estrutura representada nestes objetos, na base de dados. A partir daí a rotina volta ao ponto em que se encontrava antes de efetuar a chamada recursiva de `leatributo()`, com os parâmetros descritos no início deste exemplo, permitindo a continuação da recuperação de informações de outras estruturas do programa, caso existam.

Quando esta rotina termina a recuperação de um programa, tem-se uma composição hierárquica de itens sintáticos que representam o programa na linguagem corrente, pronta para ser processada, e exibida na janela do Editor.

A rotina `leatributo()`, além das instruções citadas no exemplo dado acima (TIPO, RELA e FIM), trata as outras instruções que foram descritas na seção 5.6, como INSE, SUBS, AVAN, ELIM, CONC, ATRI, etc.

7.4 - Expressões.

Todas as expressões introduzidas sob o controle do PEGSS passam inicialmente por uma análise léxica para, posteriormente, serem transformadas em uma representação pós-fixa genérica, que a deixa pronta para o armazenamento na base de dados. As etapas pelas quais uma expressão passa são mostradas na Figura 6.3. As expressões que descrevem um índice na definição de um vetor e as que descrevem uma estrutura "for" passam por uma etapa a mais que as outras, antes de serem armazenadas na base de dados, onde é feita uma análise, para determinar as suas respectivas categorias.

7.4.1 - Expressões em geral.

Após a introdução pelo usuário de uma expressão, é feita a sua análise léxica. Verifica-se se os seus operandos e operadores estão definidos para a linguagem corrente, ou se são operadores genéricos. Uma vez que a expressão esteja léxicamente correta, esta é transformada para a sua respectiva representação pós-fixa. As rotinas que efetuam estas operações são:

- `infpos_pas()`: determina que seja efetuada a análise léxica da expressão infixa, sob o contexto da linguagem Pascal, para posteriormente transformá-la na sua representação pós-fixa.
- `infpos_c()`: determina que seja efetuada a análise léxica da expressão infixa, sob o contexto da linguagem C, para posteriormente transformá-la na sua representação pós-fixa.
- `infpos_for()`: determina que seja efetuada a análise léxica da expressão infixa, sob o contexto da linguagem FORTRAN, para posteriormente transformá-la na sua representação pós-fixa.

A expressão pós-fixa obtida através dessas rotinas possui os seus operadores escritos, segundo a linguagem corrente, ou na forma genérica. Para que seja possível o armazenamento desta expressão, seus operadores devem ser todos transformados para a sua respectiva representação genérica. Isto é efetuado pela rotina `posgenerica()` que faz esta transformação, independente da linguagem em que estão escritos.

O processo inverso, descrito pela Figura 6.6, onde a expressão na forma pós-fixa genérica, que está armazenada na base de dados, deve ser representada na forma infixa, segundo a linguagem corrente, é proporcionado pelas rotinas `genericapos()` e `posinf()`. Estas rotinas tem as seguintes funções:

- `genericapos()`: transforma uma expressão na forma pós-fixa genérica na sua respectiva representação pós-fixa, segundo a linguagem corrente;

- `posinf()`: transforma uma expressão na forma pós-fixa, onde os seus operadores são definidos na linguagem corrente, ou estão na forma genérica, para a sua respectiva representação infixa.

7.5 - Considerações Finais.

Esta versão do Editor suporta apenas um subconjunto das linguagens Pascal, C e FORTRAN. Para que o suporte do PEGSS seja estendido para as demais estruturas sintáticas, devem ser adicionados no módulo de tratamento da edição os manipuladores para estas novas estruturas, além da sintaxe concreta que permite sua exibição.

O PEGSS permite a inserção de novas linguagens, além das contempladas pela atual versão. Sob o aspecto da implementação tratada nas sub-seções anteriores, essas adições provocariam o acréscimo de outros módulos: o módulo para o tratamento da edição sob o contexto da nova linguagem; descrição da sua sintaxe concreta, através do seu respectivo arquivo cabeçalho; e os módulos para o tratamento de suas expressões, os quais devem passar a considerar os operadores da nova linguagem.

O sistema de computação utilizado para a implementação deste trabalho foi:

- .Computador PC - AT 286 (IBM),
- .Sistema Operacional MS-DOS,
- .Linguagem de programação "C",
- .Compilador Aztec v.2.02.

O tamanho do projeto implementado foi o seguinte:

- .Número de linhas de código fonte: 14.240 linhas,
- .Sistema no modo executável:
  - ..EGSPas - 268,4 Kbytes,
  - ..EGSC - 271,2 Kbytes,
  - ..EGSFOR - 271,9 Kbytes.

O sistema foi dividido em três para facilitar este trabalho. Cada um deles suporta a edição e recuperação de programas para as respectivas linguagens: Pascal, C e FORTRAN.

Capítulo 8

CONCLUSÕES, DECISÕES DE PROJETO E FUTURAS PESQUISAS

8.1 - Conclusões.

A utilização de uma base de dados GEO facilitou a implementação dos recursos oferecidos pelo PEGSS como o suporte a múltiplas linguagens e à verificação da semântica estática. O MRO mostrou-se capaz de representar as informações referentes à construção sintática das linguagens, além de oferecer, através do conceito de colônias, a possibilidade de se efetuar a verificação da semântica estática.

Através do MRO foi possível descrever a representação genérica dos códigos. Isso capacita o PEGSS a utilizar um conjunto de recursos, tais como "browsers", analisadores, otimizadores de códigos, de validação, geradores de banco de testes, geradores intrínsecos de códigos para qualquer linguagem, e outros. Estes recursos compõem-se em ferramentas de auxílio à fase de codificação do ciclo de desenvolvimento de software. Além disso, o GEO oferece recursos como: acesso multiusuário e controle de concorrência, que permitem a implementação de recursos de desenvolvimento de software em equipe, facilitando a comunicação entre elas. Essas facilidades oferecidas permitem concluir pela adequação do GEO em aplicações de Engenharia de Software.

A estrutura hierárquica utilizada mostrou-se capaz de suportar a edição sensível à sintaxe das linguagens suportadas pelo Editor. Através dela pôde-se armazenar a representação de programas escritos em qualquer das linguagens suportadas pelo Editor. Essa estrutura também permite que programas sejam construídos no modo "top-down", pela inserção de gabaritos e expressões. O que não se pode concluir sobre esta estrutura é sua capacidade no apoio à compilação e depuração, pois módulos equivalentes não foram construídos.

No trabalho para capacitar o PEGSS ao suporte homogêneo de múltiplas linguagens, concluiu-se que uma parcela significativa de estruturas sintáticas possuem representação nas linguagens suportadas, embora, para atingir esta amplitude, algumas estruturas devam sofrer um tratamento antes de serem armazenadas. Por exemplo, existem estruturas que exigem uma ampla análise para encontrar sua representação única, podendo ter suas expressões transformadas para atingir a maior generalidade possível. Por outro lado, esse recurso pode ser útil como ferramenta de integração e comunicação entre equipes de trabalho que utilizam-se de linguagens diferentes durante a fase de implementação do Projeto.

Na atual versão do PEGSS, a inserção de novas linguagens provoca alterações em algumas das rotinas que compõem sua implementação. Para que o Editor seja totalmente independente de linguagem, as informações sobre a sintaxe concreta e sobre o controle da edição devem ser armazenadas na base de dados. Deste modo, a adição de linguagens pertencentes a um mesmo paradigma ocorreria de maneira imediata.

A implementação deste projeto permitiu a obtenção de alguns subprodutos:

- a representação interna genérica de programas;
- a representação homogênea de códigos para posterior pré-processamento;
- o Editor que gera códigos sintaticamente corretos e
- as rotinas para tratamento de textos hierárquicos.

8.2 - Decisões de Projeto.

8.2.1 - Definição da Abrangência do PEGSS.

Quando iniciou-se este trabalho, não se tinha uma noção completa dos problemas que poderiam surgir, tanto no que se refere à capacidade do MRO de representar as construções sintáticas das linguagens, quando na extensão que se poderia alcançar nas traduções de programas escritos em uma linguagem para outra. Por outro lado, sabia-se que os programas poderiam ser representados em uma estrutura hierárquica, e que, a partir dela, poder-se-iam obter alguns recursos de suporte à edição sensível à sintaxe.

Os primeiros esforços na busca de subsídios para a especificação deste trabalho se fizeram através do levantamento dos ambientes existentes na literatura, onde não encontrou-se nenhum com a capacidade de representar um mesmo programa através de mais de uma linguagem do mesmo paradigma e que também oferecesse suporte à edição sensível à sintaxe.

A partir daí, começou-se a se definir um conjunto de estruturas sintáticas equivalentes entre as linguagens Pascal, C e FORTRAN. Havia um consenso de que estas linguagens deveriam ser as primeiras a serem suportadas pelo Editor, devido à: grande utilização destas linguagens e ao fato de pertencerem a um mesmo paradigma, o que implicaria a facilidade de inserção de novas linguagens com a mesma abordagem.

Através das representações equivalentes das estruturas começaram-se a definir as representações genéricas destas estruturas, que foram então descritas utilizando-se o MRO, que definiu a DLA.

8.2.2 - Definição da DLA.

Durante o processo de modelagem da DLA alguns fatores tiveram que ser considerados e praticamente direcionaram todas as decisões tomadas nesta fase. Entre estes fatores os principais são os seguintes:

.Especificar as informações sobre as estruturas sintáticas do modo mais genérico possível, para que possam ser representadas sob o contexto de qualquer linguagem.

.Para a representação genérica de uma estrutura devem ser consideradas todas as associações possíveis em cada uma das linguagens. Portanto, uma estrutura genérica deve conservar todas as classes sintáticas das linguagens em que é representável;

.Compor os tipos de objetos e seus relacionamentos de modo que o processo de recuperação das estruturas neles representados possam ser efetuado sob o contexto de qualquer linguagem.

Esses fatores estão fortemente relacionados, pois a edição, a tradução e a recuperação efetuam-se em torno das informações representadas na DLA, isto é, essas três

etapas se interrelacionam através desta modelagem. Isto significa que qualquer alteração na DLA se propaga para as especificações definidas sobre ela.

Com isto, e considerando os propósitos deste trabalho, decidiu-se restringir o conjunto das estruturas sintáticas para um subconjunto representativo das linguagens a serem suportadas, pois a complexidade das informações que devem ser controladas aumenta a cada nova estrutura a ser suportada.

Por outro lado, em algumas ocasiões teve-se que decidir entre obedecer fielmente, ou não, as construções sintáticas das linguagens, em benefício da obtenção do máximo de estruturas traduzidas. Um exemplo desta situação ocorre quando subrotinas definidas no escopo de outra sobrotina, em Pascal, devem ser traduzidas para as linguagens C ou FORTRAN. Neste caso, optou-se em traduzir as subrotinas pertencentes apenas ao escopo global do programa, omitindo as definidas no escopo de outras. Deste modo, para este Protótipo, seguiu-se o padrão de que seriam sempre consideradas as construções sintáticas que podem efetivamente ser representadas nas linguagens, deixando para as futuras implementações a adaptação de situações que não são perfeitamente intercambiáveis, tal como representar rotinas não globais em Pascal para o escopo global de C ou FORTRAN.

8.2.3 - Esquemas de Recuperação.

Paralelamente à definição da DLA especificaram-se os Esquemas de Recuperação de Programas para cada uma das linguagens. Decidiu-se que, não só as informações sobre como recuperar o programa, mas também sobre como exibir cada uma das suas estruturas, deveriam estar armazenadas na base de dados e em um mesmo local. A tal decisão deve-se: à necessidade de investigar a habilidade do GEO em recuperar estas informações, já que mais de uma colônia deve ser manipulada durante esta etapa (a colônia contendo as informações do programa e a colônia com ERP da linguagem corrente) e de comparar o modo em que foram dispostas estas informações com a outra possibilidade utilizada na fase de edição é descrita na sub-seção 8.6.

Esta última razão visa estudar a possibilidade de se ter o Editor totalmente independente das linguagens por ele suportadas.

Assim, para que novas linguagens sejam suportadas, deve-se inserir na base de dados o seu ERP. Essas informações viabilizam a tradução de programas para uma nova linguagem.

8.2.4 - Hierarquia.

A escolha da estrutura hierárquica apresentada no Capítulo 4 para representação intermediária dos programas baseou-se na sua eficiência em relação a algumas necessidades para a exibição do programa, como por exemplo: a apresentação do programa de forma indentada, facilidade na representação da "navegação" sobre suas estruturas e a possibilidade de representação dos nós pertencentes a uma estrutura, por sua vez representada em um nó, através dos nós nos níveis seguintes.

Apesar das facilidades oferecidas por esta estrutura, ainda se desconhece a sua capacidade para o suporte à execução e depuração. A utilização das árvores de sintaxe abstrata pelos ambientes orientados para estruturas tem se mostrado eficiente. No entanto, neste protótipo não foram construídos os módulos de execução e depuração do programa armazenado em representações genéricas.

A única dificuldade encontrada na utilização desta estrutura surgiu da necessidade de adaptar, em algumas situações, a sintaxe concreta para que as estruturas representadas nos nós da hierarquia pudessem ser exibidas através da sobreposição de textos. Isto foi resolvido através da utilização do caractere "@" no texto de um nó, como indicador de que, durante o processamento da hierarquia, o texto do seu nó filho deverá substituí-lo.

8.2.5 - Expressões.

Nos Ambientes descritos no Capítulo 2, as expressões são descritas obedecendo a sua construção sintática e inseridas de modo interativo. Obtém-se alguns recursos da representação interna das expressões, como a representação gráfica da sua árvore de sintaxe. As expressões no PEGSS são inseridas através da descrição de caractere por caractere, em uma janela fora do texto do programa. A opção por esse modo deu-se pela agilidade oferecida ao usuário durante a sua descrição, apesar da inserção ainda não ser interativa, e também, pela restrição imposta pela possibilidade de terem-se operadores genéricos nas expressões, o que dificulta a utilização de uma árvore de sintaxe para a sua representação.

8.2.6 - Sintaxe Concreta e informações para a edição.

As informações que compõem a sintaxe concreta e as que determinam as ações do Editor durante o processo de construção de programas ficam à disposição na memória. Como foi descrito no Capítulo 7, tais informações estão definidas, respectivamente, através de arquivos cabeçalhos e rotinas de controle de edição.

Os fatores determinantes para que estas informações fossem assim definidas ao invés de serem armazenadas na base de dados, foram os seguintes:

primeiro: a necessidade de investigar a evolução da implementação do Protótipo EGSS com as informações dispostas deste modo, já que as informações sobre a recuperação de programas foram definidas na base de dados e

segundo: oferecer maior agilidade ao Editor pois, durante o processo de edição, o número de acessos na base de dados é grande.

Com isto, para capacitar o Editor a suportar novas linguagens é necessário, considerando apenas o aspecto da edição, especificar a sua sintaxe abstrata através de um arquivo cabeçalho e definir uma nova rotina para controlar a edição, sob o contexto da nova linguagem.

8.3 - Futuras Pesquisas.

8.3.1 - Tornar o PEGSS totalmente independente de linguagem.

O fato de ter-se optado pela definição das informações que descrevem a sintaxe concreta, como descrito na sub-seção 8.6, tornou menos prática a capacidade do PEGSS em dar suporte a novas linguagens. As extensões necessárias para viabilizar a inserção de novas linguagens atigem algumas rotinas do Editor utilizadas para a tradução de programas, além da implementação de outras que permitem a edição sob o contexto destas linguagens. A redefinição do Editor, para suportar a especificação da sintaxe concreta utilizando-se as instruções de controle da edição armazenadas na base de dados, o capacitaria à geração

de suporte a novas linguagens sem que sejam alteradas quaisquer rotinas do Editor.

8.3.2 - Ampliar a verificação automática de escopo para os identificadores através da DLA.

Na atual implementação do PEGSS não se faz uma análise da semântica estática por completo, ou seja, os escopos dos identificadores não são totalmente verificados durante a edição de um programa. Apenas emitem-se mensagens avisando o usuário se tal identificador já foi ou não declarado no contexto todo do programa, incluindo o contexto de suas subrotinas, não distinguindo se o nome é local a uma subrotina ou global ao programa. Isto inclui também os tipos de dados definidos pelos usuários.

Apesar da não utilização de uma árvore de sintaxe abstrata para esta verificação parcial dos identificadores, como é efetuado pelos ambientes descritos no Capítulo 2, o PEGSS pode ampliar esta verificação tornando-a completa apenas alterando-se a especificação da DLA, pois é através da estrutura utilizada pelo Modelo de Representação de Objetos (MRO) que são obtidas estas informações.

Atualmente a DLA descreve as subrotinas de subtipo "principal", "procedimento" e "função" pertencentes ao tipo de colônia "Arquivo". Com isto, os identificadores representados nos tipos de objetos "nome_var", "nome_subrot", "ident_tipo_novo", etc, são únicos em toda uma colônia do tipo "Arquivo" e não em uma subrotina. Para que isto ocorra o tipo de objeto "subrotina" deverá restringir um outro tipo de colônia diferente de "Arquivo". Deste modo, todos os objetos que representem identificadores de uma subrotina serão únicos apenas para o tipo de colônia constricta pelo objeto que representa tal subrotina, definindo o escopo para estes identificadores. Assim, por exemplo, quando em uma declaração de variável for utilizado um tipo que ainda não foi definido, o Editor poderá emitir uma mensagem identificando tal fato, verificando esta indefinição no escopo local ou global da subrotina corrente.

Uma outra vantagem decorrente desta alteração na DLA é livrar o usuário de ter que especificar a expressão de retorno através da sua escolha em um menu de opções, quando sob o contexto das linguagens Pascal e FORTRAN (ver sub-seção 3.4.2), permitindo que o usuário descreva esta expressão como um simples comando de atribuição.

8.3.3 - Conclusões sobre a implementação.

Alguns recursos não foram concluídos nesta atual versão do PEGSS, tais como a eliminação de estruturas durante o processo de edição de um programa e a análise sintática das expressões descritas pelo usuário. Além disso, é necessário definir o suporte para as estruturas sintáticas complementares às definidas no Protótipo EGSS, a fim de que o usuário possa ter, em uma primeira etapa, uma ferramenta completa de edição sensível à sintaxe para as linguagens Pascal, C e FORTRAN.

O processo de eliminação de estruturas é muito parecido com o de inclusão, pois devem ser efetuadas atualizações na hierarquia que representa o programa e na base de dados, especificamente na colônia de objetos que o representa. Alguns detalhes devem ser considerados. Por exemplo, se uma estrutura "ifte" for eliminada, serão também eliminadas as estruturas definidas em seus lugares de inserções como "<condição>" e "<bloco>" e assim por diante, o que implica eliminar o nó da hierarquia com o texto "if(@)" e todos os nós ligados a ele nos níveis seguintes. Simultaneamente, os objetos que representam as estruturas eliminadas deverão ser também eliminados, além da criação de relacionamentos entre os objetos que restaram a fim de representar corretamente a nova estrutura do programa na base de dados.

A análise sintática deve ser efetuada assim que o usuário terminar de introduzir a expressão. Considerando que uma expressão pode ter sido obtida pela tradução de um programa escrito em outra linguagem, poderá ocorrer que nela existam operadores genéricos que não possuam representação na linguagem corrente. A princípio, sem que um estudo mais profundo tenha sido efetuado, não tem sentido fazer a análise sintática de uma expressão que contenha operadores não definidos para a linguagem corrente. Portanto, o mais razoável seria desprezar a análise para as expressões com esta característica, assumindo que foram validadas no contexto da linguagem onde foram originalmente inseridas. Para as expressões que acabaram de ser criadas o analisador sintático se incumbiria desta tarefa.

Pelo modo como estão representadas as expressões, segundo a DLA, cada uma delas contém a informação do seu tipo. Por exemplo, uma expressão que descreve a condição de uma estrutura "while" é armazenada em um tipo de objeto "expressão" cujo atributo para "modo" é "condição". Aproveitando-se a informação de que esta expressão é do tipo condição, o analisador sintático poderá restringir sua verificação apenas para a sintaxe de uma condição, o que agilizará esta etapa para todos os tipos de expressões.

8.3.4 - Criação de novas ferramentas.

O PEGSS deverá ser transportado para uma estação de trabalho SUN e aí desenvolvido, aproveitando-se os recursos oferecidos por estas máquinas. O objetivo é fazer com que o Editor passe a ser uma ferramenta de um Ambiente de Programação ou, até mesmo, de um Ambiente de Desenvolvimento de Software com suporte a gerenciamento de projetos e controle de versão. Além disso, alguns recursos devem ser adicionados, como por exemplo:

- . Suporte à documentação de programas - Documentação automática proporcionada pelo Editor ou introduzida pelo usuário no corpo do programa durante a edição. Geração de relatórios de apoio à documentação externa de programas, nos quais um conjunto de informações a respeito do programa poderia ser obtido na base de dados.
- . Recursos que viabilizem a edição de textos em qualquer ponto do programa, não apenas nos lugares de inserção dos gabaritos.
- . Representação do programa em diferentes níveis de detalhes.
- . Representações gráficas do programa a partir da sua representação interna, como por exemplo, a apresentação do programa utilizando o Diagrama de Nassi-Schneiderman e recursos de programação visual.
- . Auxílio à exibição dos tipos já definidos durante a edição.
- . Suporte à verificação da semântica dinâmica.
- . Suporte à depuração de programas durante a execução - Recursos que permitam acompanhar o fluxo dos dados durante a execução do programa através de representações gráficas como o Diagrama de Controle de Fluxo. Recursos que permitam acompanhar as estruturas executadas através da representação textual do programa ("tracing"). Auxílio à definição de pontos de parada no texto do programa durante a execução. Suporte ao acompanhamento e edição dos valores assumidos pelas estruturas de dados

definidas no programa durante a execução.

. Suporte ao gerenciamento de projetos e controle de versões.

.Alguns recursos podem ser utilizados, beneficiando-se das características da base, como por exemplo:

.compartilhamento de código (conceito de objeto complexo da abordagem de Orientação a Objetos),

.acesso multiusuário,

.controle de concorrência,

.além de outras informações de suporte às fases de desenvolvimento de Projetos.

Estas ferramentas devem ser oferecidas sem que sejam esquecidos fatores como integração, capacidade de estendê-las ou substituí-las, interface amigável e comunicação com usuário.

Referências Bibliográficas

- [Amb_85] V.Ambriola, et alli, "Automatic Generation of Execution Tools in a GANDALF environment", Journal of Systems and Software, Vol. 5, No. 2, pp. 155-172, Maio 1985.
- [Bia_92] Mauro Biajiz, "MRO* - Uma Atualização do Modelo de Representação de Objetos e uma Abordagem Funcional", Dissertação de Mestrado, ICMSC-USP, 1992.
- [Dar_87] S.A. Dart, et alli, "Software development environments", IEEE Computer, Vol. 20, No. 11, pp. 18-28, Novembro 1987.
- [Don_75] V. Donzeau-Gouge, et alli, "A structure oriented program editor: a first step towards computer assisted programming", INRIA Tech. Rep., No.114, Abril 1975.
- [Don_84] V. Donzeau-Gouge, et alli, "Programming environments based on structured editors: The MENTOR experience", D. R. Barstow et alli (eds.), Interactive Programming Environments, McGraw-Hill, pp.128-140, 1984.
- [Ell_85] R. J. Ellison, Barbara J. Staudt, "The Evolution of the GANDALF System", Journal of Systems and Software, Vol. 5, No. 2, pp. 107-120, Maio 1985.
- [Fis_84] C. N. Ficher, et alli, "The POE Language-Based Editor Project", in Proc. ACM SIGSOFT/SIGPLAN Soft. Eng. Simp. on Pract. Soft. Devel. Envir., Abril 1984.

- [GEO_91] **"Manual de Referência do GEreenciador de Objetos"**, Documento de Trabalho do Grupo de Base de Dados e Engenharia de Software do ICMSC, Outubro 1991.
- [Hab_86] A. N. Habermann, et alli, **"Gandalf: Software Development Environments"**, IEEE Trans. Soft. Eng., Vol. 12, No. 12, pp. 1117-127, Dezembro 1986.
- [Han_84] W. J. Hansen, **"User Engineering Principles for Interactive Environments"**, D. R. Barstow et alli (eds.), Interactive Programming Environments, McGraw-Hill, pp. 217-231, 1984.
- [Kai_85] Gail E. Kaiser, Elaine Kant, **"Incremental Parsing without a Parser"**, Journal of Systems and Software, Vol. 5, No. 2, pp. 121-140, Maio 1985.
- [Med_81] R. Medina-Mora e P. Feiler, **"An Incremental programming environment"**, IEEE Trans. Software Eng., Vol. 7, pp. 472-482, Setembro 1981.
- [Not_85] David Notkin, **"The GANDALF Project"**, The Journal of Systems and Software, Vol. 5, No. 2, pp. 91-106, Maio 1985.
- [Pre_87] R. S. Pressman, **"Software Engineering; A Practitioner's Approach"**, 2.ed. New York, Ed. McGraw-Hill Book Company, 1987.

- [Rei_84] Steven P. Reiss, "Graphical Program Development with PECAN Program Development Systems", Proc. ACM SIGSOFT/SIGPLAN Soft. Eng. Simp. on Pract. Soft. Devel. Envir., pp. 30-41, Abril 1984.
- [Rei_85] Steven P. Reiss, "PECAN: Program Development Systems that support multiple views", IEEE Trans. on soft. Eng., Vol. 11, No. 3, pp. 276-285, Março 1985.
- [Rep_84] Thomas Rep, Tim Teitelbaum, "The Synthesizer Generator", in Proc. ACM SIGSOFT/SIGPLAN Soft. Eng. Simp. on Pract. Soft. Devel. Envir., pp. 30-41, Abril 1984.
- [Tei_81] Tim Teitelbaun, et alli, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", D. R. Barstow et alli (eds.), Interactive Programming Environments, McGraw-Hill, pp. 97-116, 1984.
- [Tei_81] Tim Teitelbaun, et alli, "The Why and Wherefore of the Cornell Program Synthesizer", Proc. ACM SIGPLAN/SIGOA Symp. Test Manipulation, pp. 8-16, Junho 1981.
- [Tra-86] Caetano Traina Júnior, "Máquina e Modelo de Dados Dedicados para Aplicações de Engenharia", Tese de Doutorado, IFQSC-USP, Dezembro 1986.
- [Tra_88] Caetano Traina Jr., e Jan Frans Willem Slaets, "Um Modelo de Representação de Objetos", in Anais do 3o. Simpósio Brasileiro de Banco de Dados, Recife, pp. 227-242, Março de 1988.

-
- [Tra_91] Caetano Traina Jr., "GEO: Um Sistema de Gerenciamento de Base de Dados Orientado a Objetos - Estado Atual de Desenvolvimento e Implementação", in Anais do 6o. Simpósio Brasileiro de Banco de Dados, Manaus, pp. 193-207, Maio 1991.
- [Ugg_90] D. Uggioni, et alli, "Genese: Um Editor Genérico Dirigido pela Sintaxe", in Anais do 4o. Simpósio Brasileiro de Engenharia de Software, Águas de São Pedro, pp. 261-262, Outubro de 1990.
- [Val_92] Carlos R. Valêncio, "Documentação do Projeto de Construção do PEGSS", Notas Internas do ICMSC-USP, Novembro 1992.

APÊNDICE A

Composição Hierárquica dos Gabaritos da Linguagem Pascal

Considerações:

São apresentadas as subhierarquias que compõem cada um dos gabaritos utilizados para a representação intermediária das estruturas sintáticas da linguagem Pascal. Para facilitar esta apresentação são omitidos algumas ligações dos nós. As composições hierárquicas dos gabaritos das linguagens C e FORTRAN são encontradas em [Val_92].

C01 : subrotina (principal)

C01 : subrotina (principal ou procedimento ou funcao)

<decl.rotulo>	C01

C01 : subrotina ou decl_rotulo

C02 : decl_rotulo

<define const.>	C01

C01 : subrotina ou def_const

C02 : def_const

<define tipo>	C01

C01 : subrotina ou def_tipo

C02 : def_tipo

C03 : tipo_de_dado (var)

<declara var.>	C01

C01 : subrotina ou declaracao (var)
 C02 : declaracao (var)
 C04 : tipo_de_dado

<decl. subrot.>	C01

C01 : subrotina (procedimento ou funcao)

C02 : declaracao (param)

C03 : tipo_de_dado (param)

C01 : subrotina (principal)

C02 : bloco (inicio)

C01 : subrotina (proc. ou funcao)

C02 : bloco (subrot)

<bloco> (3)	C01

C01 : comando (ifte ou ifte_r)
C02 : bloco (composto (1))

<bloco> (4)	C01

C01 : comando (ift, ift_r, ifte, ifte_r, while, while_r, do, do_r, for ou for_r)
C02 : bloco (composto (2))

<bloco> (5)	C01

<comando>	C02

C01 : comando (repeat ou repeat_r)
C02 : bloco (composto (3))

C01 : comando ou bloco (composto)

ift:

C02 : comando (ift)

ifte:

C02 : comando (ifte)

goto:

C02 : comando (goto)

atrib:

C02 : comando (atrib)

cham_sub:

C02 : comando (cham_sub)

for:

C02 : comando (for)

repeat:

C02 : comando (repeat)

while:

C02 : comando (while)

do:

C02 : comando (do)

ifa:

C02 : comando (ifa)

if1:

C02 : comando (if1)

C01 : comando ou bloco (composto)

ifte_r:

C02 : comando (ifte_r)

ift_r:

C02 : comando (ift_r)

goto_r:

C02 : comando (goto_r)

atrib_r:

C02 : comando (atrib_r)

cham_sub_r:

C02 : comando (cham_sub_r)

for_r:

C02 : comando (cham_sub_r)

repeat_r:

while_r:

do_r:

ifa_r:

C02 : comando (ifa_r)

ifl_r:

C02 : comando (ifl_r)

<rotulo> (1)	C01

<rotulo> (2)	C02

identificador de rotulo	C03

C01 : comando (goto ou goto_r)

C02 : comando com rotulo

C03 : rotulo

(1) : rotulo de desvio

(2) : rotulo de comando

<nome>	C01

identificador	C02

C01 :

. subrotina (principal ou funcao ou procedimento),
. def_const,
. def_tipo,
. declaracao (var),
. declaracao (param),
. comando (cham_sub ou cham_sub_r).

C02 :

. nome_subrot,
. nome_const,
. ident_tipo_novo,
. nome_var,
. nome_parametro,
. nome_subrot.

<tipo> (1)	C01

identificador de tipo	C02

C01 : tipo_de_dado (param)

C02 : tipo_basico ou ident_tipo_novo

<tipo> (2)	C01

identificador de tipo	C02

C01 : tipo_de_dado (var)

C02 : tipo_basico ou ident_tipo_novo

Vetor:

C02 : array
C03 : tipo_de_dado (var)

Registro:

C02 : registro

C02 : array ou indice
C03 : indice

C01 : def_const
C02 : expressao (constante)

<atribuicao>	C01

expressao	C02

C01 : comando (atrib. ou atrib_r)

C02 : expressao (atribuicao ou retorno)

<indice>	C01

expressao	C02

C01 : indice ou array

C02 : expressao (indice (indice_g ou indice_p ou
indice_pc ou indice_pf))

<lista expr.>	C01

expressao	C02

C01 : comando (cham_sub ou cham_sub_r)

C02 : expressao (lista_expr)

<expr.> (1)	C01

expressao	C02

C01 : comando (for ou for_r)

C02 : expressao (inicio)

<expr.> (2)	C01

expressao	C02

C01 : comando (for ou for_r)

C02 : expressao (parada)

<expr.> (3)	C01

expressao	C02

C01 : comando (ifa ou ifa_r)

C02 : expressao (expr_aritm)

<condicao>	C01

expressao	C02

C01 : comando (ift, ifte, repeat, while, do, ifl, ift_r, ifte_r, repeat_r, while_r, do_r, ifl_r)

C02 : expressao (condicao)

identificador de tipo	C01

integer	C02

real	C02

char	C02

boolean	C02

tipo definido pelo usuario	C03

C01 : tipo_de_dado (var ou param)

C02 : tipo_basico

C03 : ident_tipo_novo

APÊNDICE B

Definição da Linguagem Abstrata

Considerações:

É mostrada a concepção lógica da DLA através de representações utilizando o Diagrama de Representação de Objetos (DRO). São descritas para cada linguagem (Pascal, C e FORTRAN) apenas o que lhe é relevante sobre a DLA. Portanto, da figura Pascal-1 até Pascal-13 mostra-se a parte da DLA tratada pelo Editor quando se está sob o contexto da linguagem Pascal. O mesmo ocorre para as linguagens C e FORTRAN.

APÊNDICE C

Esquema de Recuperação de Programa da Linguagem Pascal

Considerações:

Mostra-se as informações utilizadas pelo Editor para a recuperação e apresentação de programas na linguagem Pascal, tais informações compõem o ERPPas. Os ERP para as linguagens C e FORTRAN são encontradas em [Val_92].

tipo do objeto	tipo do atributo	atributos	Pascal
arquivo	lebase	RELA contém_prog FIM	
subrotina	lebase	TIPO FIM	
principal	principal	INSE program @; 0 INSE <nome> 1 RELA nome_da_subrot INSE label @; 0 INSE <decl.rótulo> 1 RELA decl_rótulos INSE const -1 INSE <define const.> 1 RELA define_const INSE type -1 INSE <define tipo> 1 RELA define_tipo INSE var -1 INSE <declara var.> 1 RELA decl_var_glob INSE <decl.subrot.> -1 RELA decl_global INSE <bloco> 0 RELA contém_blc_inic FIM	
			1

tipo do objeto	tipo do atributo	atributos	Pascal
procedi- mento	procedi- mento	SUBS procedure @(0); 0 INSE <nome> 1 RELA nome_da_subrot INSE <decl.parâm.> 1 RELA declara_parâmetros INSE label @; -1 INSE <decl.rótulo> 1 RELA decl_rótulos INSE const -1 INSE <define const.> 1 RELA define_const INSE type -1 INSE <define tipo> 1 RELA define_tipo INSE var -1 INSE <declara var.> 1 RELA decl_var_loc INSE <decl.subrot.> -1 RELA decl_local INSE <bloco> 0 RELA proc_contém_blc FIM	
			2

tipo do objeto	tipo do atributo	atributos	Pascal
função	função	SUBS function @(@):@; 0 INSE <nome> 1 RELA nome_da_subrot INSE <decl.parâm.> 1 RELA declara_parâmetros RELA retorna_tipo INSE label @; -1 INSE <decl.rótulo> 1 RELA decl_rótulos INSE const -1 INSE <define const.> 1 RELA define_const INSE type -1 INSE <define tipo> 1 RELA define_tipo INSE var -1 INSE <declara var.> 1 RELA decl_var_loc INSE <decl.subrot.> -1 RELA decl_local INSE <bloco> 0 RELA func_contém_blc FIM	
			3

tipo do objeto	tipo do atributo	atributos	Pascal
nome_subrot	lebase	IDEN FIM	
decl_rótulo	lebase	SUBS @,@ 0 INSE <rótulo> 1 RELA cria_rótulo INSE <decl.rótulo> 1 RELA prox_decl_rót. FIM	
def_const	lebase	SUBS @=@; 0 INSE <nome> 1 RELA nome_da_const INSE <constante> 1 RELA forma_const INSE <define const.>-2 RELA prox_def_const FIM	
def_tipo	lebase	SUBS @=@; 0 INSE <nome> 1 RELA nome_do_def_novo RELA contém_tipo INSE <define tipo> -1 RELA prox_def_tipo FIM	
rótulo	lebase	IDEN FIM	
ident_tipo_novo	lebase	IDEN FIM	
nome_var	lebase	IDEN FIM	
			4

tipo do objeto	tipo do atributo	atributos	Pascal
declaração	lebase	TIPO FIM	
variável	var	SUBS @: @; 0 INSE <nome> 1 RELA nome_da_var RELA tem_tipo_de_dado INSE <declara var.>-1 RELA prox_decl FIM	
parâmetro	parâm	SUBS @: @; @ 0 INSE <nome> 1 RELA nome_do_par RELA tem_tipo INSE <decl.parâm.>100 RELA prox_decl FIM	
decl_subrot_glob	lebase	TIPO FIM	
proc_glob	procedi- mento	RELA decl_proc_glob RELA prox_decl_glob FIM	
func_glob	função	RELA decl_func_glob RELA prox_decl_glob FIM	
nome_const	lebase	IDEN FIM	
			5

tipo do objeto	tipo do atributo	atributos	Pascal
bloco	lebase	TIPO FIM	
início	início	SUBS begin 0 INSE <comando> 1 RELA e_composto_por INSE end. -1 FIM	
subrot	subrot	SUBS begin 0 INSE <comando> 1 RELA e_composto_por INSE end; -1 FIM	
composto	composto	ATRI categoria FIM	
1	1	SUBS begin 0 INSE <comando> 1 RELA e_composto_por INSE end -1 FIM	
2	2	SUBS begin 0 INSE <comando> 1 RELA e_composto_por INSE end; -1 FIM	
3	3	INSE <comando> 1 RELA e_composto_por FIM	
			6

tipo do objeto	tipo do atributo	atributos	Pascal
decl_subrot_loc	lebase	TIPO FIM	
proc_loc	procedi- mento	RELA decl_proc_loc RELA prox_decl_loc FIM	
func_loc	função	RELA decl_func_loc RELA prox_decl_loc FIM	
tipo_de_dado	lebase	TIPO FIM	
dado_var	var	INSE <tipo> 1 RELA é_tipo_registro RELA é_tipo_array RELA é_tipo_novo RELA é_tipo_básico FIM	
dado_parâm	parâm	INSE <tipo> 1 RELA é_tipo_novo RELA é_tipo_básico FIM	
nome_parâmetro	lebase	IDEN FIM	
tipo_básico	lebase	IDEN FIM	
			7

tipo do objeto	tipo do atributo	atributos	Pascal
registro	lebase	SUBS record 0 INSE <declara var.> 1 RELA declara_campos INSE end; 0 FIM	
array	lebase	SUBS array[@] of @ 0 INSE <indices> 1 RELA possui_indice RELA possui_tipo FIM	
indice	lebase	SUBS @, @ 0 INSE <indice> 1 RELA forma_indice INSE <indices> 1 RELA prox_ind FIM	
expressão	lebase	ATRI modo FIM	
atribuição	modo	ATRI descrição FIM	
condição	modo	ATRI descrição FIM	
constante	modo	ATRI descrição FIM	
lista_expr	modo	ATRI descrição FIM	
início	modo	ATRI descrição FIM	
parada	modo	ATRI descrição FIM	
			8

tipo do objeto	tipo do atributo	atributos	Pascal
incremento	modo	ATRI descrição FIM	
retorno	modo	ATRI descrição FIM	
expr_aritm	modo	ATRI descrição FIM	
ativa	modo	ATRI descrição FIM	
expr_ind	modo	ATRI categoria ATRI inf ATRI sup FIM	
comando	lebase	TIPO FIM	
ift	ift	SUBS if(θ) 0 INSE <condição> 1 RELA verifica_cond INSE then 0 INSE <bloco> 1 RELA executa_caso_v INSE <comando> -2 RELA segue FIM	
ifl	ifl	SUBS if(θ) 0 INSE <condição> 1 RELA verifica_cond INSE then 0 INSE <comando> 1 RELA possui_cmd INSE <comando> -2 RELA segue FIM	
goto	goto	SUBS goto θ; 0 INSE <rótulo> 1 RELA desvia_para INSE <comando> -1 RELA segue FIM	
			9

tipo do objeto	tipo do atributo	atributos	Pascal
ifte	ifte	SUBS if(②) 0 INSE <condição> 1 RELA verifica_cond INSE then 0 INSE <bloco> 1 RELA executa_caso_v INSE else -1 INSE <bloco> 1 RELA executa_caso_f INSE <comando> -2 RELA segue FIM	
cham_sub	cham_sub	SUBS ②(②); 0 INSE <nome> 1 RELA nome_cham_sub INSE <lista expr.> 1 RELA possui_lis_expr INSE <comando> -2 RELA segue FIM	
ifa	ifa	SUBS IF(②=0) 0 INSE <expr.> 1 RELA verifica_expr_arit INSE goto ②; 0 INSE <rótulo> 1 RELA desvia_rot1 INSE else if(②>0) -1 INSE <expr.> 1 RELA verifica_expr_arit INSE goto ②; 0 INSE <rótulo> 1 RELA desvia_rot2 INSE else -1 INSE goto ②; 1 INSE <rótulo> 1 RELA desvia_rot3 INSE <comando> -4 RELA segue FIM	
			10

tipo do objeto	tipo do atributo	atributos	Pascal
for	for	SUBS for @ to @ do 0 INSE <expr.> 1 RELA expr_inicial INSE <expr.> 1 RELA expr_parada INSE <bloco> -1 RELA executa_bloco INSE <comando> -1 RELA segue FIM	
repeat	repeat	SUBS repeat 0 INSE <bloco> 1 RELA executa_blc INSE until(@); 0 INSE <condição> 1 RELA verifica_cond INSE <comando> -2 RELA segue FIM	
while	while	SUBS while(@)do 0 INSE <condição> 1 RELA verifica_cond INSE <bloco> 0 RELA executa_caso_v INSE <comando> -1 RELA segue FIM	
do	do	SUBS repeat 0 INSE <bloco> 1 RELA executa_blc INSE until(not @); 0 INSE <condição> 1 RELA verifica_cond INSE <comando> -2 RELA segue FIM	
atrib	atrib	SUBS @; 0 INSE <atribuição> 1 RELA formada_por_expr INSE <comando> -1 RELA segue FIM	
			11

tipo do objeto	tipo do atributo	atributos	Pascal
ift_r	ift_r	SUBS @:if(%) 0 INSE <rótulo> 1 RELA possui_rótulo INSE <condição> 1 RELA verifica_cond INSE then 0 INSE <bloco> 1 RELA executa_caso_v INSE <comando> -3 RELA segue FIM	
ifte_r	ifte_r	SUBS @:if(%) 0 INSE <rótulo> 1 RELA possui_rótulo INSE <condição> 1 RELA verifica_cond INSE then 0 INSE <bloco> 1 RELA executa_caso_v INSE else -1 INSE <bloco> 1 RELA executa_caso_f INSE <comando> -2 RELA segue FIM	
goto_r	goto_r	SUBS @:goto @; 0 INSE <rótulo> 1 RELA possui_rótulo INSE <rótulo> 1 RELA desvia_para INSE <comando> -1 RELA segue FIM	
atrib_r	atrib_r	SUBS @:@; 0 INSE <rótulo> 1 RELA possui_rótulo INSE <atribuição> 1 RELA formada_por_expr INSE <comando> -1 RELA segue FIM	
			12

tipo do objeto	tipo do atributo	atributos	Pascal
cham_sub_r	cham_sub_r	SUBS @:@(@); 0 INSE <rótulo> 1 RELA possui_rótulo INSE <nome> 1 RELA nome_cham_sub INSE <lista expr.> 1 RELA possui_lis_expr INSE <comando> -2 RELA segue FIM	
ifl_r	ifl_r	SUBS @:if(@) 0 INSE <rótulo> 1 RELA possui_rótulo INSE <condição> 1 RELA verifica_cond INSE then 0 INSE <comando> 1 RELA possui_cmd INSE <comando> -3 RELA segue FIM	
ifa_r	ifa_r	SUBS @:if(@=0) 0 INSE <rótulo> 1 RELA possui_rótulo INSE <expr.> 1 RELA verifica_expr_arit INSE goto @; 0 INSE <rótulo> 1 RELA desvia_rot1 INSE else if(@>0) -1 INSE <expr.> 1 RELA verifica_expr_arit INSE goto @; 0 INSE <rótulo> 1 RELA desvia_rot2 INSE else -1 INSE goto @; 1 INSE <rótulo> 1 RELA desvia_rot3 INSE <comando> -5 RELA segue FIM	
			14