Giovanni Allan Buranello

Instrumentação de programas Delphi para implementação de critérios de teste estrutural

Marília - SP

Dezembro / 2006

Giovanni Allan Buranello

Instrumentação de programas Delphi para implementação de critérios de teste estrutural

Dissertação apresentada ao Programa de Mestrado em Ciência da Computação da Fundação Eurípides Soares da Rocha de Marília para a obtenção da qualificação no Mestrado em Ciência da Computação.

Orientador:
Prof^{o}. Dr^{o}. Márcio Eduardo Delamaro

MESTRADO EM CIÊNCIA DA COMPUTAÇÃO
PPGCC - PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
FUNDAÇÃO EURÍPIDES SOARES DA ROCHA

Marília - SP

Novembro / 2006

Agradecimentos

Dedico meus sinceros agradecimentos:

– ao professor doutor Márcio Eduardo Delamaro, pela orientação e incentivo;

– ao professor doutor Auri Marcelo Rizzo Vicenzi, pelo seu inestimável auxílio na conclusão deste trabalho;

– ao professor doutor Adenilso da Silva Simão, pela criação da Ferramenta IDeL e pelo apoio e incentivo;

– a Gustavo Rondina, que foi peça fundamental para realização deste trabalho;

– a Roque Maitino Neto, pelo apoio e amizade;

– à Fundação Eurípides de Marília;

– a todos os colegas do Mestrado em Ciência da Computação da UNIVEM Marília.

"Uma coisa só é impossível até que alguém
duvida e acaba provando o contrário".
Albert Einstein

Resumo

Os *softwares* têm sido cada vez mais utilizados em tarefas importantes, estando em evidência a procura de confiança nos mesmos.

Devido à quantidade de falhas e erros encontrados nos *softwares*, um grande esforço tem sido feito visando solucionar esse crescente problema. Para tanto, foram desenvolvidas diversas técnicas, critérios e ferramentas de teste para garantir a qualidade e confiança do produto.

Uma das fases do teste de software é a instrumentação, no qual são inseridos pontos de provas sobre o código de um programa feito em uma determinada linguagem de programação, afim de auxiliar o testador a reconhecer onde e o que foi executado.

Ferramentas de teste de software para linguagem de programação Delphi são escassas. Sendo assim, o presente trabalho foi desenvolvido no sentido de diminuir a escassez, e tem como objetivo o desenvolvimento de um instrumentador de código para a linguagem Delphi, com o auxílio da linguagem de descrição de instrumentação IDeL, que torna o processo de desenvolvimento de instrumentadores mais eficientes através da confecção de uma gramática e uma descrição para a instrumentação.

Palavras-chaves: Instrumentação, Delphi, IDel, IDelGen, Gramática, Yacc, lex, Teste de Software e Critério de Teste.

Abstract

Software systems have been more used in important tasks each day, in evidence, nowadays, is also the search for trust on them.

Because of the quantity of fails and errors found in these softwares, a great effort has been made, trying to solve this increasing problem. For such result, many technique, testing criteria and testing tools were developed to guarantee the quality of the product.

One of software testing phases is the instrumentation, where checking points are inserted into a program code written in a certain programming language, in such a way, to help the recognize where and what has been executed.

Delphi programming language software testing tools deficient. This way, the present dissertation was developed in a way to fulfill this deficiency, and has as an objective, the development of an instrumenter code for Delphi language, with the help of IDeL instrumentation description language, which makes the process of the development of intrumenters more efficient through the arrangement of grammar and a description for the instrumentation.

keyworks: Instrumentation, Delphi, IDel, IDelGen, Grammar, Yacc, lex, Software Testing e Testing criteria.

Sumário

Lista de Figuras

1 Introdução p. 13

1.1 Contexto . . . . p. 13

1.2 Motivação . . . . p. 14

1.3 Organização do trabalho . . . . p. 15

2 A Atividade de Teste p. 16

2.1 Técnicas de Teste de Software . . . . p. 17

2.1.1 Teste Funcional . . . . p. 17

2.1.2 Teste Baseado em Erro . . . . p. 18

2.1.3 Teste Estrutural . . . . p. 18

2.2 Fases da Atividade de Teste . . . . p. 21

2.2.1 Teste de Unidade . . . . p. 21

2.2.2 Teste de Integração . . . . p. 22

2.2.3 Teste de Sistema . . . . p. 23

2.3 Ferramentas de Teste Estrutural . . . . p. 23

2.3.1 Ferramenta POKE-TOOL . . . . p. 24

2.3.2 Ferramenta JaBUTi . . . . p. 25

2.3.3 Ferramenta Discover . . . . p. 28

2.4 Considerações Finais . . . . p. 30

3 Geradores de Compiladores p. 32

3.1 Principais Características de uma Gramática . . . . p. 32
3.2 Yacc e Lex . . . . p. 37
3.3 Considerações Finais . . . . p. 40

4 IDeL e IDeLGen p. 41

4.1 Principais Características . . . . p. 42
4.1.1 Aspecto Operacional . . . . p. 42
4.1.2 Estrutura do IDeL . . . . p. 43
4.1.2.1 Unidade de Identificação . . . . p. 43
4.1.2.2 Unidade de Processamento . . . . p. 43
4.1.2.3 Implementação . . . . p. 46
4.2 Execução e Exemplos . . . . p. 46
4.2.1 Exemplos . . . . p. 47
4.3 Considerações Finais . . . . p. 48

5 Instrumentador para a linguagem Delphi p. 51

5.1 Principais características da Linguagem Delphi . . . . p. 52
5.1.1 Controladores de fluxo . . . . p. 52
5.1.2 Exceções . . . . p. 55
5.2 Aspectos de implementação . . . . p. 55
5.2.1 Geração do arquivo delphi.idel . . . . p. 56
5.2.1.1 Identificação das Unidades . . . . p. 57
5.2.1.2 Processamento das Unidades . . . . p. 58
Passo FindFunction . . . . p. 59
Passo MarkStatements . . . . p. 60
Passo LinkStatement . . . . p. 61
Passo JoinStatement . . . . p. 62

Passo JoinToFunction . . . . p. 63

Passo MakeGraph . . . . p. 65

5.2.2 Seção de Implementação . . . . p. 73

5.3 Execução do Instrumentador . . . . p. 74

5.4 Considerações Finais . . . . p. 76

6 Estudo de Caso p. 79

6.1 Descrição do Programa . . . . p. 79

6.2 Tamanho do Programa . . . . p. 80

6.3 Tempo de execução . . . . p. 83

6.4 Considerações Finais . . . . p. 86

7 Conclusões e trabalhos futuros p. 91

7.1 Trabalhos Futuros . . . . p. 92

Referências p. 94

Apendice A - Gramática do Delphi (delphi.y) p. 96

Apêndice B - Gramática do Delphi (delphi.l) p. 128

Apêndice C - Descrição do Delphi na IDeL (delphi.idel) p. 133

Lista de Figuras

2.1 Grafo de Fluxo de Controle . . . . p. 19

2.2 Grafo de fluxo de controle . . . . p. 21

2.3 Opções disponíveis na ferramenta POKE-TOOL . . . . p. 25

2.4 Tela para criar uma sessão de teste na POKE-TOOL . . . . p. 25

2.5 Exemplo da tela JaBUTi, visualização pelo código-fonte . . . . p. 26

2.6 Grafo gerado pela JaBUTi no critérios Todos-Arcos-ei . . . . p. 27

2.7 JaBUTi - código bytecode . . . . p. 28

2.8 JaBUTi - Sumário de cobertura por critério . . . . p. 28

2.9 JaBUTi - Cobertura da classe por critério . . . . p. 28

2.10 JaBUTi - Cobertura por métodos . . . . p. 29

2.11 Tela de execução de teste na Ferramenta Discover . . . . p. 30

2.12 Pasta *Linker* do Delphi . . . . p. 30

3.1 Programa exemplo *while* . . . . p. 33

3.2 Árvore sintática *while* . . . . p. 33

3.3 Exemplo da gramática BNF . . . . p. 35

3.4 Programa exemplo - gramática BNF . . . . p. 35

3.5 Exemplo de uma produção . . . . p. 36

3.6 Exemplo de uma produção opcional . . . . p. 36

3.7 Exemplo de uma produção com repetição . . . . p. 36

3.8 Exemplo de uma produção com repetição . . . . p. 37

3.9 Gramática de uma declaração de variável em C simplificada . . . . . . p. 37

3.10 Arquivo com regras léxico . . . . p. 38

3.11 Arquivo sintático para a produção do analisador sintático . . . . p. 39

4.1 Execução IDeLgen . . . . p. 43

4.2 Regra de instrumentação do comando *while* . . . . p. 44

4.3 Grafo de como funciona a regra *while* . . . . p. 46

4.4 ***P*** - programateste.C . . . . p. 48

4.5 Arquivo programateste-instrumentado.C . . . . p. 48

4.6 Arquivo programateste.main.dot . . . . p. 49

4.7 Grafo gerado pela Ferramenta Graphviz . . . . p. 49

5.1 Geração do instrumentador idel.delphi . . . . p. 56

5.2 Seção de identificação das unidades do arquivo delphi.idel. . . . . p. 57

5.3 Árvore de padrões da produção :head. p. 58

5.4 Programa Exemplo. . . . . p. 59

5.5 Padrão FindFunction. . . . . p. 60

5.6 Grafo do Padrão FindFunction. . . . . p. 60

5.7 Árvore sintática fplist. . . . . p. 60

5.8 Padrão MarkSatatements. . . . . p. 61

5.9 Grafo do Padrão MarkStatements. . . . . p. 61

5.10 Padrão LinkStatementList. . . . . p. 62

5.11 Grafo do Padrão LinkStatement. . . . . p. 62

5.12 Padrão JoinStantemet. . . . . p. 63

5.13 Grafo do Passo JoinStatement. . . . . p. 63

5.14 Padrão JoinToFunction. . . . . p. 64

5.15 Grafo do Passo JoinToFunction. . . . . p. 64

5.16 Instrumentaçao do Passo JoinToFunction. . . . . p. 65

5.17 Padrão While. . . . . p. 66

5.18 Grafo Padrão While. . . . . p. 67

5.19 Instrumentação do padrão While. . . . . p. 67
5.20 Padrão Try2. . . . . p. 68
5.21 Grafo Padrão Try2. . . . . p. 68
5.22 Padrão if-then. . . . . p. 69
5.23 Grafo Padrão if-then. . . . . p. 69
5.24 Instrumentação do padrão If-Then . . . . p. 70
5.25 Passo IfThenElse. . . . . p. 70
5.26 Grafo Padrão if-then-else. . . . . p. 71
5.27 Instrumentação do padrão If-Then-else . . . . p. 71
5.28 Padrão RepeatUntil. . . . . p. 72
5.29 Grafo do Padrão RepeatUntil. . . . . p. 72
5.30 Código não instrumentado (a) código instrumentado (b) . . . . p. 73
5.31 Padrão ForTo. . . . . p. 73
5.32 Grafo do Padrão ForTo. . . . . p. 74
5.33 Código não instrumentado (a) código instrumentado (b). . . . . p. 75
5.34 Implementação do checkopoint before. . . . . p. 75
5.35 Programa exemplo função exittest. . . . . p. 76
5.36 Execução da IDeL. . . . . p. 76
5.37 Função exittest instrumentada . . . . p. 77
5.38 Grafo da função exittest. . . . . p. 77
5.39 Arquivo DOT da função exittest. . . . . p. 78
6.1 Exemplo de realce de contraste com o filtro Butterworth (NUNES, 2001) p. 80
6.2 Exemplo de equalizaçao de Histograma (NUNES, 2001) . . . . p. 81
6.3 Realce de contraste pela Curva Característica . . . . p. 85
6.4 Realce de Contraste pelos Coeficientes de Atenuação e Segmentação . . p. 85
6.5 Realce de Contraste pela Modificação do Histograma e Segmentação . . p. 88

6.6 Arquivo DOT . . . . p. 88

6.7 Grafo da Procedure Pacotelocal . . . . p. 89

6.8 Grafo da Procedure Pacotelocal Percorrido . . . . p. 90

1 *Introdução*

Neste capítulo serão apresentados o contexto, a motivação para realizar o trabalho e os objetivos a serem atingidos. No final do capítulo, é apresentada a estrutura para esta dissertação.

1.1 Contexto

A Engenharia de Software tem crescido tanto em importância, quanto em abrangência de atuação. Na década de 60, o software exercia papel secundário em detrimento ao hardware (PRESSMAN, 2001). Com a crescente necessidade de produtos mais confiáveis, o planejamento e o uso de técnicas passaram a fazer parte do desenvolvimento do software.

No tocante à abrangência de sua atuação, a engenharia de software passou a englobar atividades que dizem respeito não apenas à fase de construção do produto, mas também aos processos posteriores à sua implantação. A depuração, a visualização da informação e a análise de código são exemplos de atividades realizadas, muitas vezes após a implantação do sistema, mas que mantêm o mesmo nível de importância em relação aos processos preliminares da engenharia de sistemas.

O desenvolvimento de software envolve uma série de atividades humanas nas quais a possibilidade de inclusão de falhas e erros no projeto podem ser enormes e ocasionarem grande dano. Erros podem começar a surgir imperceptivelmente, logo no início do processo de desenvolvimento de um produto. Assim, a atividade de teste é de vital importância na garantia da qualidade do software, representando a última revisão da especificação, projeto e codificação (PRESSMAN, 2001).

A visibilidade crescente do software como um elemento do sistema e o custo associado à falha motivam esforços para a prática de testes em software(PRESSMAN, 2001) . Segundo Pressman (2001) não é raro que uma empresa de desenvolvimento de software consuma entre trinta e quarenta por cento do custo total do projeto em testes.

No intuito de reduzir tais custos, várias técnicas e critérios que auxiliam na condução e avaliação da atividade de teste têm sido desenvolvidos e aprimorados. A diferença entre estas técnicas está, basicamente, na origem da informação que é utilizada para avaliar ou construir conjuntos de casos de teste, possuindo cada técnica uma variedade de critérios para esse fim. Na técnica Funcional (caixa-preta), os requisitos e testes são obtidos a partir da especificação do produto; na técnica Estrutural (caixa branca), os requisitos são obtidos a partir da própria implementação do software. Finalmente, na técnica Baseada em Erros, os elementos requeridos para caracterizar a atividade de teste são baseados em erros comuns que podem ocorrer durante o desenvolvimento de software (VINCENZI, 1997).

1.2 Motivação

Para a aplicação de teste é necessária a existência de uma ferramenta de teste automatizada que o apóie. Pelo uso de ferramentas é possível obter maior eficácia e uma redução do esforço necessário para a realização do teste, bem como diminuir os erros que são causados pela intervenção humana nesta atividade.

A instrumentação é uma técnica frenqüentemente usada na engenharia de software por diferentes propósitos, por exemplo, programa para rastrear a execução, programa para analisar corbertura de códigos em teste de software; usada também na engenharia reversa. A instrumentação de programas é uma das atividades realizadas pelas ferramentas de teste. Este processo realiza a inserção de pontos de provas em lugares estratégicos, junto ao código fonte de uma linguagem de programação, sendo assim, é possível a execução do programa fazendo um mapeamento do que e onde foi executado (SIMAO et al., 2001).

Várias ferramentas têm sido desenvolvidas com o objetivo de realizar instrumentação de código. Particularmente, a IDeL e IDeLgen (SIMAO et al., 2001) foram construídas para serem integradas em um ambiente genérico de teste baseado em fluxo de dados.

O que impulsionou o desenvolvimento deste trabalho foi a falta de ferramentas de apoio ao teste estrutural para a linguagem de programação Delphi. Com base nas ferramentas IDel e IDeLgen, foi desenvolvida uma aplicação de suporte à técnica de teste estrutural, que tem como objetivo instrumentar códigos escritos em Delphi. A IDeL realiza instrumentação de códigos escritos em qualquer linguagem desde que, uma grámatica e uma descrição de como instrumentar seja passada como parâmetro para a ferramenta IDeLgen produzir o instrumentador para a linguagem em questão. Antes deste trabalho

a IDeL contava somente com a gramática e descrição de intrumentação para a linguagem de programação C. No entanto, foi possível a confecção de uma gramática para a linguagem Delphi e uma descrição de instrumentação para a mesma, contribuindo assim, com a ferramenta para atender mais uma linguagem de programação (SIMAO et al., 2001).

No Capítulo 4 serão abordadas as Ferramentas IDeL e IDeLgen em questão com maiores detalhes.

1.3 Organização do trabalho

O presente trabalho inicia-se com a introdução. Em seguida, no Capítulo 2, é descrito o teste de software, ressaltando suas várias técnicas, abordagens e fases. No Capítulo 3 é descrita a complexidade da confecção da uma gramática e também a adaptação da mesma nas ferramentas Yacc e Lex. No Capítulo 4 são descritas, detalhadamente, as ferramentas IDel e IDeLgen, com a apresentação de exemplos. No Capítulo 5 é apresentado o propósito deste trabalho e os exemplos de como foi confeccionada a descrição de intrumentação para linguagem Delphi. Já no Capítulo 6 é apresentado o estudo de caso, com o exemplo de um programa propositalmente grande para averiguação da ferramenta, escrita na linguagem Delphi. No Capítulo 7 são apresentadas as considerações finais e sugestões para futuros trabalhos. Nos apêndices são apresentados os arquivos confeccionados. Os Apêndices A e B apresentam os arquivos delphi.y e delphi.l, respectivamente, nesta ordem, eles formam a gramática da linguagem Delphi; em seguida o Apêndice C, que é o arquivo delphi.idel que contém regras de como instrumentar a linguagem Delphi.

2 A Atividade de Teste

Sistematicamente, com o passar dos anos, as empresas desenvolvedoras de software espalhadas pelo mundo têm investido recursos na missão de melhorar, em sentido amplo, a qualidade de seu produto final. Novas abordagens de análise, técnicas avançadas de programação e meios como teste têm sido usados em busca da qualidade. A engenharia de software evoluiu significativamente nas últimas décadas procurando estabelecer técnicas, critérios, métodos e ferramentas para produção de software. A crescente utilização de sistemas baseados em computação, em praticamente todas as áreas de atividade humana, provoca uma crescente procura por qualidade e confiança no software (SIMAO et al., 2001; VINCENZI, 1997).

Desenvolver softwares com qualidade não é uma tarefa fácil. Pesquisadores na área de Engenharia de Software têm trabalhado, desenvolvendo métodos, técnicas e ferramentas para propiciar o desenvolvimento de softwares com alta qualidade. Uma das atividades que ajuda a garantir a qualidade é o teste de software. Este tem por objetivo identificar possíveis erros introduzidos durante o processo de desenvolvimento. A atividade de teste consiste em executar um programa com o objetivo de encontrar erros, através dos seguintes passos: 1) construção de conjunto de casos de teste; 2) execução de um programa com esse conjunto de casos de teste; 3) análise do comportamento do programa para determinar se o mesmo está correto. Esses passos se repetem até que se tenha confiança de que o programa realiza o esperado com o mínimo de erros possível (MALDONADO et al., 2004).

Duas abordagens são utilizadas com o objetivo de encontrar formas econômicas e produtivas: a abordagem teórica e a abordagem empírica. Na abordagem teórica, procuram-se estabelecer propriedades e características dos critérios de teste, tais como, a eficácia de uma estratégia de teste ou a relação de inclusão entre os critérios. Entende-se por critério de teste a definição de quais propriedades precisam ser testadas para encontrar o maior número de erros possível. Na abordagem empírica, dados e estatísticas são coletados, os quais registram, por exemplo, a freqüência com que diferentes estratégias de teste revelam a presença de erros em uma determinada coleção de programas, fornecendo diretrizes para

a escolha entre os diversos critérios disponíveis (HOWDEN, 1986).

O processo de desenvolvimento de software abrange uma diversidade de atividades em que, apesar dos métodos, técnicas e ferramentas empregados, erros ainda podem ocorrer. Podem ocorrer desde o início do desenvolvimento, por exemplo, especificando erroneamente os requisitos do sistema, como também nos estágios finais pela inserção de defeitos no projeto ou na implementação do sistema (PRESSMAN, 2001).

2.1 Técnicas de Teste de Software

Os critérios de teste podem ser classificados basicamente em três técnicas: funcional, estrutural e baseada em erros.

Segundo Maldonado (1991),

> a diferença entre essas técnicas está na origem da informação utilizada para avaliar ou construir os conjuntos de casos de teste, sendo que cada uma delas possui um conjunto de critérios para esse fim.

No entanto, nenhuma destas técnicas de teste é suficiente para garantir a qualidade do teste. Na verdade, elas se complementam e devem ser aplicadas em conjunto a fim de assegurar um teste de boa qualidade (PRESSMAN, 2001).

2.1.1 Teste Funcional

No teste funcional (ou teste caixa-preta), os requisitos de teste são estabelecidos a partir das especificações do software, sua implementação não é necessariamente considerada. O objetivo do teste funcional é encontrar discrepâncias entre o comportamento atual do sistema e o descrito em sua especificação. Assim, consideram-se apenas as entradas, as saídas e o estado do programa e o testador não têm necessariamente acesso ao código fonte do software (ROCHA et al., 2004). O sistema é uma "caixa preta" cujo comportamento somente pode ser determinado estudando suas entradas e saídas relacionadas. O testador está preocupado somente com a funcionalidade, e não com a implementação do software (SOMMERVILLE, 2003). Pressman (2001) sustenta que o teste funcional tende a ser exercido no último estágio de teste, enquanto o teste estrutural é executado no inicio do processo de teste, porque o teste funcional tem o propósito diferente do estrutural.

Um problema com a técnica funcional é a dificuldade de quantificar a atividade de teste, visto que não se pode garantir que partes essenciais ou críticas do programa sejam

executadas (MALDONADO et al., 2004).

2.1.2 Teste Baseado em Erro

A técnica de teste baseado em erro utiliza informações sobre os erros mais freqüentes cometidos no processo de desenvolvimento de software para derivar os requisitos de teste. O destaque da técnica está nos erros que o programador ou projetista pode cometer durante o desenvolvimento e nas abordagens que podem ser usadas para detectar sua ocorrência (MALDONADO et al., 2004). Dois critérios de teste baseado em erro são:

- semeadura de erros: neste critério uma quantidade conhecida de erros é introduzida no programa. Após o teste, do total de erros encontrados, verificam-se quais são naturais e quais foram introduzidos. Usando de probabilidade estatística, o número de erros naturais ainda existentes no programa pode ser estimado;
- análise de mutantes: é um critério que utiliza um conjunto de programas ligeiramente modificados, obtidos a partir de um determinado programa P, para avaliar o quanto um conjunto de casos de teste T é adequado para o teste de P. O objetivo é encontrar um conjunto de casos de teste capaz de revelar as diferenças de comportamento existentes entre P e seus mutantes (DEMILLO; LIPTON; SAYWARD, 1978).

.

2.1.3 Teste Estrutural

A técnica de teste estrutural, conhecida como teste caixa branca (em oposição ao nome caixa preta), tem o mesmo objetivo das outras técnicas. A diferença encontra-se no fato de o teste estrutural levar em consideração a implementação do programa. A maioria dos critérios dessa técnica utiliza uma representação de programa conhecida como grafo de programa ou grafo de fluxo de controle, mostrada na Figura2.1 (MALDONADO et al., 2004).

Para que o grafo seja gerado, deve-se abstrair os nós e arestas a partir do código-fonte do programa. Cada nó do grafo, identificado na Figura-2.1 por um número, representa um bloco indivisível de comandos. Segundo Maldonado et al. (2004), cada bloco tem as seguintes características: uma vez que o primeiro comando do bloco é executado, todos

os demais são executados sequencialmente e não existe desvio de execução para nenhum comando dentro do bloco.

Assim, um desvio na execução seqüencial do código – originada através de um comando `if` ou `while`, por exemplo – acarretaria na criação de um novo nó no grafo. As arestas, representadas por uma seta, indicam o fluxo entre os nós. Pode-se observar na Figura 2.1 que, partindo do nó 1, o fluxo pode tomar dois caminhos possíveis: a aresta do nó 1 ao nó 3 ou a aresta do nó 1 ao nó 2. Tal fato explica-se pela presença do comando *if* no código.

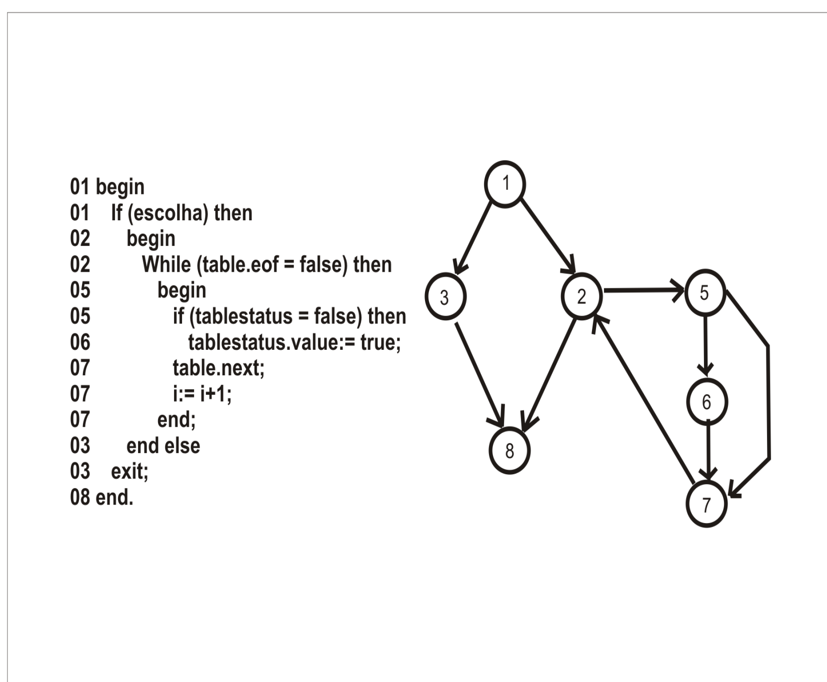

Figura 2.1: Grafo de Fluxo de Controle

Ntafos (1988) sustenta que a técnica de teste estrutural é provavelmente a mais usada, devido principalmente à simplicidade de aplicação. Além disso, a maioria dos testes estruturais não fornece orientações para a seleção de dados de teste num certo sub-domínio de caminhos de execução, e alguns erros só serão descobertos pelo teste se o caminho é executado com valores de um pequeno subdomínio do domínio maior. Mesmo com estes potenciais inconvenientes, mais e mais critérios de teste estrutural têm sido desenvolvidos e estudados, dentre eles os de Todos-Nós, Todos-Arcos e Todos-Caminhos. O critério de teste Todos-Nós requer que cada nó do grafo seja executado pelo menos uma vez, ou seja, que cada comando do programa seja executado uma vez pelo menos. O critério Todos-Arcos requer que cada aresta do grafo, ou seja, cada desvio de fluxo de controle do programa, seja exercitado pelo menos uma vez. O critério de teste Todos-Caminhos

requer que todos os caminhos possíveis caminho ou desvio de fluxo do programa sejam executados (MALDONADO et al., 2004; NTAFOS, 1988; VINCENZI, 1997).

Um exemplo demonstra de forma simplificada o uso destes critérios: considerando um programa como o da Figura 2.1 e considerando o critério Todos-Nós, supondo que (i) a condição de execução do comando `if` (nó 1) seja verdadeira (ou seja, a variável `escolha` seja verdadeira), (ii) a condição de execução do comando `while` (nó 2) seja verdadeira (ou seja, não esteja configurado o final de arquivo de `table`) e (iii) a condição de execução do comando `if` no nó 5 seja verdadeira (ou seja, a variável `tablestatus` contenha o valor falso), serão então exercitados os nós (1,2,5,6,7,2,8).

No entanto, se for criado outro caso de teste em que a variável `escolha` seja definida como falsa, serão exercitados os nós (1,3,8). Nesta situação, já está satisfeito o critério Todos-Nós, mas não os critérios Todos-Caminhos e Todos-Arcos. Erros podem ser encontrados mais facilmente com o critério de Todos-Caminhos, mas este critério exige muito esforço. Caso fosse possível utilizá-lo, seriam exercitados todos os critérios, ou seja, executando o critério Todos-Caminhos, automaticamente seriam exercitados Todos-Nós e Todos-Arcos.

Por volta da década de 1970, surgiram os critérios baseados em análise de fluxo de dados (HERMAN, 1976), que utilizam informações do fluxo de dados para derivar os requisitos de teste. Na Figura 2.2 é ilustrada este tipo de critério. Rapps e Weyuker (1982) propuseram o Grafo Def-uso que consiste em uma extensão do grafo de programa. Neste grafo, são exploradas as associações entre a definição e o uso das variáveis determinando os caminhos a serem exercitados. Quando a variável é usada em uma computação, diz-se que seu uso é computacional (c-uso); quando usada em uma condição, seu uso é predicativo (p-uso). Alguns critérios desta classe são: Todas-Definições (all-defs), Todos-Usos (all-uses), Todos-Du-Caminhos (all-du-paths), Todos-P-Usos (all-p-uses), Todos-P-Usos/Alguns-C-Usos (all-p-uses/some-c-uses) e Todos-C-Usos/Alguns-P-Usos (all-c-uses/some-p-uses) (VINCENZI, 1997).

Por exemplo, para exercitar a variavel `z` definida no nó 4 (Figura 2.2), de acordo com o critério Todas-Definições, poderiam ser executados um dos seguintes subcaminhos: (4,5,6); (4,5,7,8); e (4,5,7,9).

O teste estrutural é, em geral, aplicado a unidades de programa relativamente pequenas, como sub-rotina, ou às operações associadas com um objeto. Como o nome sugere, o testador pode analisar o código e utilizar conhecimentos sobre a estrutura de um componente, a fim de escolher os dados para teste. A análise do código pode ser utilizada para

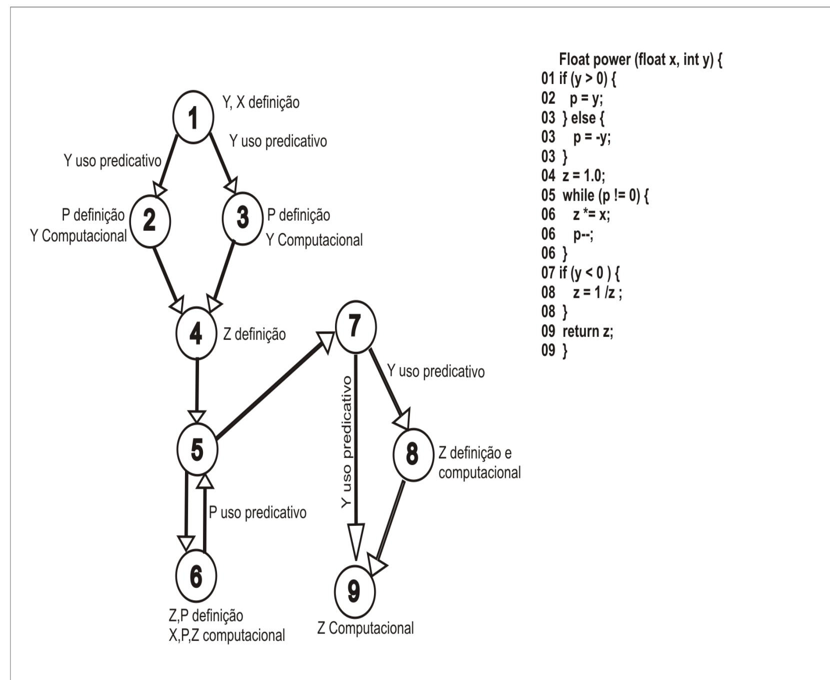

Figura 2.2: Grafo de fluxo de controle

levantar os casos de teste e também para descobrir quantos casos de teste são necessários para cobrir todas as definições e uso no programa, para que os mesmos sejam executados pelo menos uma vez durante o processo de teste (SOMMERVILLE, 2003).

2.2 Fases da Atividade de Teste

Segundo Pressman (2001), a estratégia convencional de teste prima pela realização de teste de forma incremental, ou seja, inicia-se pelo teste de unidade, em seguida o de integração e finalmente o teste do sistema como um todo. A menor unidade de teste é o sub-programa (módulo, procedimento ou componente) e, via de regra, tais partes do programa são testadas em primeiro lugar e individualmente. Em seguida, elas são integradas e testes de integração são executados, para que, no final, o sistema todo seja testado.

2.2.1 Teste de Unidade

O teste de unidade concentra-se na verificação da menor unidade de projeto de software: método ou classe. Usando a descrição do projeto como guia, caminhos de controle importantes são testados para descobrir erros dentro das fronteiras do módulo. No caso

de programas orientados a objeto, uma classe pode conter várias operações diferentes, e uma operação pode existir como parte de várias classes diferentes. O significado de teste de unidade muda e não se pode mais testar uma única operação isoladamente, como no teste de unidade convencional (PRESSMAN, 2001).

Alguns autores entendem que a classe é a menor unidade no contexto de software orientado a Objeto (MCGREGOR; SYKES, 2001), sendo que o teste de unidade poderia envolver o teste intra-método, inter-método e intra-classe e o teste de integração que corresponde ao teste inter-classe. Na Tabela 2.1, são listados os tipos de teste de software OO que podem ser aplicados.

Tabela 2.1: Relação entre Menor Método

	Menor Unidade: Método
Unidade	Intra-método
Intregração	Inter-método, Intra-classe e Inter-classe
Sistema	Toda aplicação
	Menor Unidade: Classe
Unidade	Intra-método, Inter-método e Intra-classe
Integração	Inter-classe
Sistema	Toda aplicação

Mcgregor e Sykes (2001) afirmam que os módulos ou classes são geralmente testados através da criação de um *driver*[1] de teste que cria instâncias da classe e também um ambiente apropriado para tais instâncias. O *driver* envia uma ou mais mensagens para uma instância, conforme especificado pelo caso de teste. Em seguida, checa o resultado daquelas mensagens baseado num valor de resposta, altera a instância e um ou mais parâmetros da mensagem. O *driver* de teste geralmente executa a exclusão de quaisquer instâncias que tenha criado se a linguagem, tal como o C++, possui alocação de memória gerenciada pelo programador. Um *stub* é a unidade que substitui, na hora do teste, uma outra unidade chamada pela unidade que está sendo testada. Em geral, um *stub* simula o comportamento da unidade chamada.

2.2.2 Teste de Integração

O teste de integração é uma técnica sistemática para a construção da estrutura do programa, realizando-se, ao mesmo tempo, teste para descobrir erros associados nas interações das unidades. O objetivo é, a partir dos módulos testados no nível de unidade,

[1]Um driver é geralmente um trecho de código usado para fornecer valores de entrada a uma classe ou módulos a serem testados.

construir a estrutura do programa que foi determinada pelo projeto. Existem dois tipos de integração: a não-incremental e a incremental. Na integração não incremental, todos os módulos são combinados antecipadamente e o programa é testado como um todo. Na integração incremental, o programa é construído e testado em pequenos segmentos, nos quais os erros são mais fáceis de serem corrigidos.

As técnicas de projeto de casos de teste funcional são as mais utilizadas durante esta fase. No entanto, iniciativas de utilização de critérios usados no teste de unidade para o teste de integração são encontradas na literatura, tais como a extensão de critérios baseados em fluxo de dados e critérios baseados na análise de mutantes (DELAMARO, 1997; PRESSMAN, 2001).

Existem também diferenças na abordagem de teste de integração entre programas convencionais e orientados a objeto. A integração no paradigma de orientação a objeto pode ser vista como a interação entre os objetos em um programa. Tal teste torna-se indispensável ao se levar em conta que uma instância de uma classe pode não conter falhas, mas se os serviços daquela instância não forem usados corretamente pelos outros componentes do programa então este contém falha(s). Desta forma, a correta interação — ou colaboração dos objetos em um programa, é crítica para a forma correta do programa (MCGREGOR; SYKES, 2001).

2.2.3 Teste de Sistema

Depois do teste de unidade e integração, o software foi integrado, o sistema funciona como um todo, são realizados os testes de sistema. O objetivo é assegurar que o software e os demais elementos que compõem o sistema, tais como, hardware e banco de dados, combinem-se adequadamente e que o desempenho geral desejado seja obtido.

2.3 Ferramentas de Teste Estrutural

O objetivo de ferramentas de teste é sempre voltado para custo e mão-de-obra. Com ferramentas de teste é possível apoiar-se em critérios e precisão, tornando muito mais baixo o custo do teste e mais rápido.

A disponibilidade de ferramentas de teste permite uma evolução de tecnologia para as indústrias, fator indispensável para a produção de software de alta qualidade. Tais ferramentas auxiliam pesquisadores e alunos de Engenharia de Software a adquirir os

conceitos básicos e experiência na comparação, seleção e estabelecimento de estratégias de teste (PRESSMAN, 2001).

Várias ferramentas foram desenvolvidas para esta finalidade: a Ferramenta POKE-TOOL (MALDONADO; CHAIM; JINO, 1991) que apóia a aplicação dos critérios Potenciais-Usos e também de outros critérios estruturais como Todos-Nós e Todos-Arcos; a ferramenta Discover que apóia o teste estrutural de unidade para a Linguagem Delphi; a Ferramenta Jabuti (VINCENZI et al., 2003) que trabalha com Java bytecode em teste estrutural também; a ferramenta Proteum (DELAMARO, 1993) que apóia o teste de mutação para programas C, e outras. A seguir serão apresentadas algumas ferramentas de suporte ao teste estrutural.

2.3.1 Ferramenta POKE-TOOL

O objetivo inicial da Ferramenta POKE-TOOL (**PO**tencial uses criteria **TOOL** for program testing) foi testar software trabalhando com o critério Potencial-Uso(PU) (MALDONADO, 1991). Este critério requer basicamente que para todo nó i e para toda variável x, para a qual existe uma definição em i, que pelo menos um caminho livre de definição com relação à variável x do nó i para todo nó e para todo arco possível de ser alcançado a partir de i por um caminho livre de definição de x seja exercitado. (MALDONADO et al., 2004). A ferramenta contém dois outros critérios também: Todos-Nós e Todos-Arcos. Na Figura 2.3 é mostrada a tela principal da ferramenta e as opções disponíveis.

Maldonado, Chaim e Jino (1991) criaram esta ferramenta motivados neste critério, sendo ela desenvolvida com o recurso multi-linguagem, podendo ser adaptada para quaisquer linguagens.

A Ferramenta POKE-TOOL é orientada à sessão de trabalho, e é utilizada para as atividades envolvendo um teste. O teste pode ser realizado em etapas, permitindo ao usuário encerrar o teste de um programa, bem como retomá-lo a partir de onde este foi interrompido. Basicamente, o usuário entra com o programa a ser testado, com o conjunto de casos de teste e seleciona todos ou alguns dos critérios disponíveis (Figura 2.4). Como saída, a ferramenta fornece ao usuário o conjunto de arcos, o Grafo Def obtido do programa em teste, o programa instrumentado para teste, o conjunto de associações necessárias para satisfazer o critério selecionado e o conjunto de associações ainda não exercitadas. O conjunto de arcos primitivos consiste de arcos que uma vez executados garantem a execução de todos os demais arcos do grafo de programa (MALDONADO; CHAIM; JINO, 1991; MALDONADO et al., 2004).

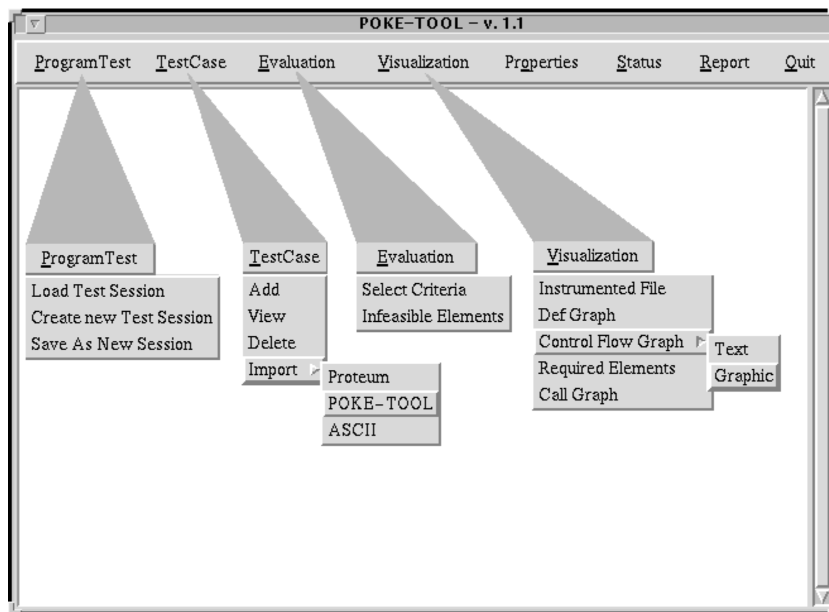

Figura 2.3: Opções disponíveis na ferramenta POKE-TOOL

Na Figura 2.4 é mostrada a criação de uma sessão de teste de um programa utilizando todos os critérios disponíveis.

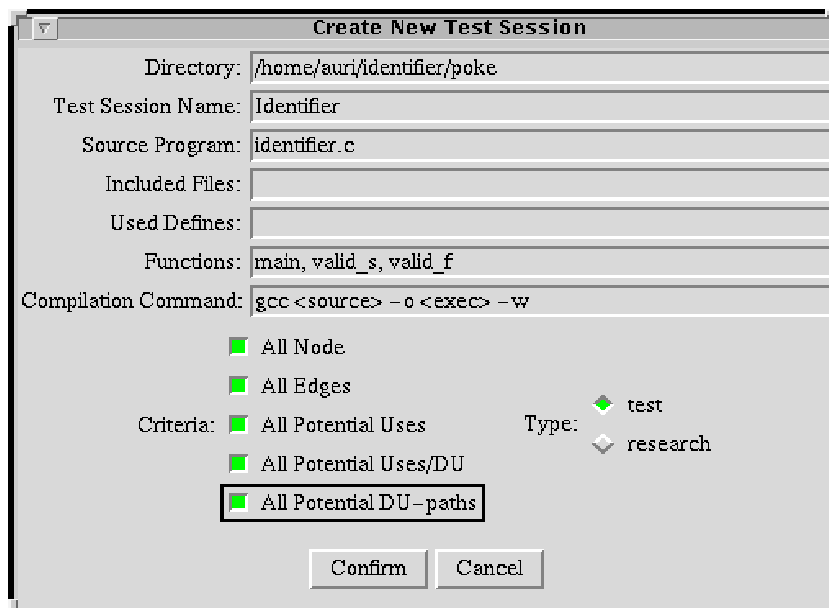

Figura 2.4: Tela para criar uma sessão de teste na POKE-TOOL

Com os relatórios gerados pela POKE-TOOL, o testador poderá avaliar de forma mais abrangente o andamento da atividade de teste, em diferentes aspectos, dependendo dos critérios desejados.

2.3.2 Ferramenta JaBUTi

A Ferramenta JaBUTi (**Ja**va **B**ytecode **U**nderstanding **T**est**i**ng) visa a ser um ambiente completo para o entendimento e teste de programas e componentes Java. JaBUTi fornece ao testador diferentes critérios de testes estruturais para a análise de cobertura.

A JaBUTi implementa atualmente critérios de teste intra-métodos sendo quatro de fluxo de controle (Todos-Nós-ei, Todos-Nós-ed, Todas-Arestas-ei, Todas-Arestas-ed), e quatro critérios de fluxo de dados (Todos-Usos-ei, Todos-Usos-ed, Todos-Pot-Uso-ei, Todos-Pot-Uso-ed). Observe que os pares Todos-Nós-ei, Todos-Nós-ed, Todas-Arestas-ei e Todas-Arestas-ed compõem os critérios Todos-Nós e Todos-Arcos, respectivamente; da mesma forma ocorre com os Todos-Usos-ei e Todos-Usos-ed que compõem o critério Todos-Usos, a diferença entre os critérios com ei e ed, é independente de exceção e dependente de exceção respectivamente. A JaBUTi realiza a análise de cobertura instrumentando um arquivo .class; após isso, ela coleta informações de cobertura durante a execução do programa e determina se cada um dos métodos de todas as classes foram testados de acordo com os critérios de teste disponíveis (VINCENZI et al., 2003).

Uma das características do software JaBUTi é a fácil visualização de blocos de códigos que são marcados com um peso, que facilitam a geração de casos de teste, de modo a maximizar a cobertura em relação aos critérios de teste. Estes blocos mostram de forma inteligente qual o requisito de teste que, se coberto, aumentaria de forma consideravel a cobertura em relação ao critério considerado, como mostrado na Figura 2.5.

A Ferramenta JaBUTi permite que os requisitos de teste de cada um de seus critérios possam ser visualizados no bytecode, código fonte e no grafo de cada método de cada uma das classes em teste. São associadas diferentes cores para indicar seus pesos. Podem ser vistas nas Figuras 2.5 , 2.6 e 2.7 cada uma destas representações.

Um exemplo, na Figura 2.7, mostra a cobertura de um determinado programa, em que a parte coberta indica o valor do peso como 0. Deste modo o testador é auxiliado, sabendo que executou um caso de teste e que, de certa forma, executou aqueles blocos de comando.

Pode ser importante avaliar a cobertura de todo projeto em relação a cada um dos critérios de teste. Esta informação pode ajudar o testador a decidir se cada critério já atingiu seu objetivo. Caso tenha atingido, um critério de teste mais forte pode ser utilizado para continuar a evolução do conjunto de teste, por exemplo, passar do critério Todos-Nós, para o critério Todas-Arestas. A ferramenta JaBUTi gera este tipo de re-

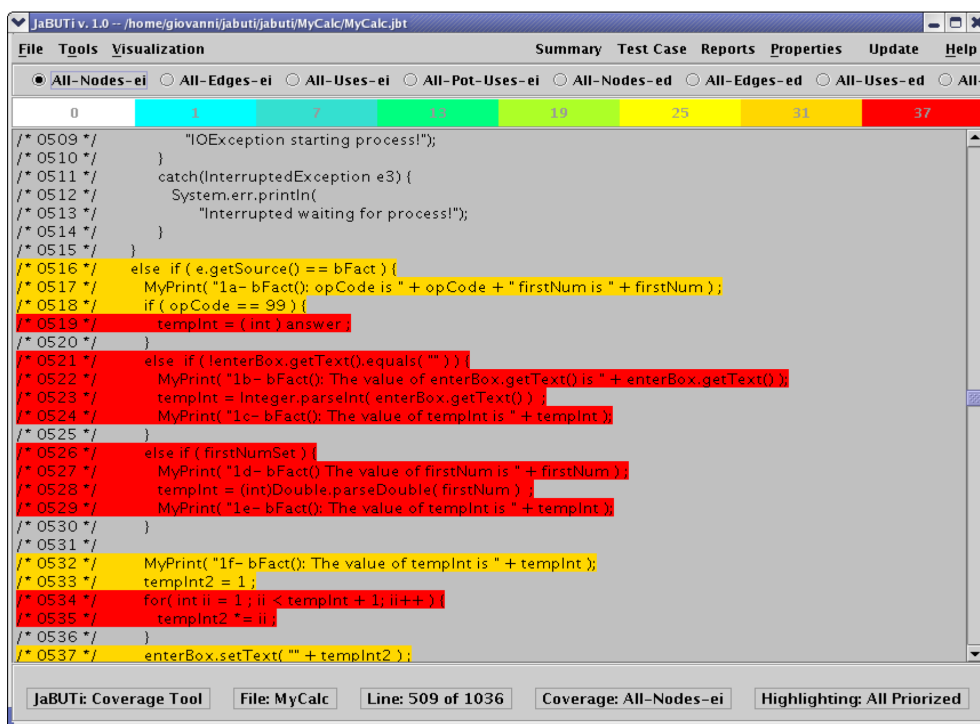

Figura 2.5: Exemplo da tela JaBUTi, visualização pelo código-fonte

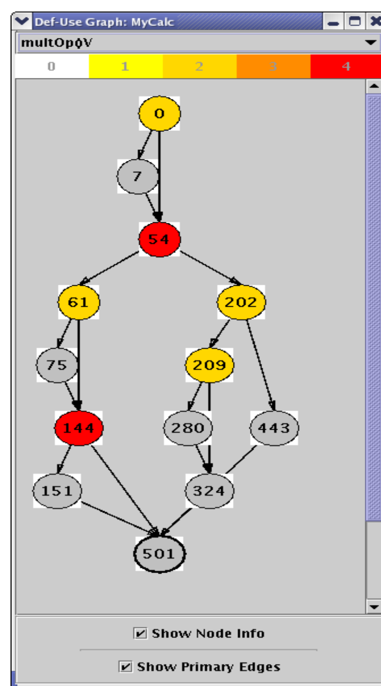

Figura 2.6: Grafo gerado pela JaBUTi no critérios Todos-Arcos-ei

latório considerando os oitos critérios de testes estruturais implementados : Todos-Nós-ei, Todas-Aresta-ei, Todos-Usos-ei, Todos-Nós-ed, Todas-Arestas-ed, Todos-Usos-ed, Todos-Pot-Uso-ei e Todos-Pot-Uso-ed. As Figuras 2.8, 2.9, 2.10 ilustram esse tipo de relatório

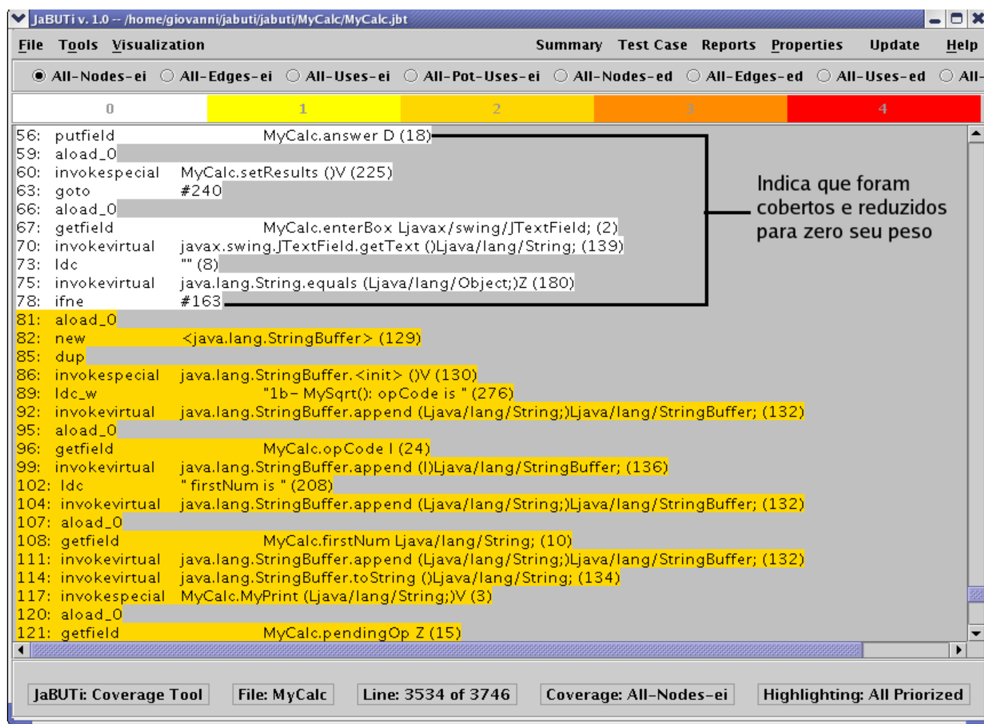

Figura 2.7: JaBUTi - código bytecode

(VINCENZI et al., 2003).

Overall Coverage Summary by Criterion

Testing Criterion	Coverage	Percentage
All-Nodes-ei	177 of 212	83%
All-Nodes-ed	0 of 4	0%
All-Edges-ei	208 of 284	73%
All-Edges-ed	0 of 8	0%
All-Uses-ei	458 of 659	69%
All-Uses-ed	0 of 4	0%
All-Pot-Uses-ei	796 of 1357	58%
All-Pot-Uses-ed	0 of 44	0%

Figura 2.8: JaBUTi - Sumário de cobertura por critério

All-Edges-ei Coverage per Class File

Class File Names	Coverage	Percentage
MyCalc	208 of 284	73%

Figura 2.9: JaBUTi - Cobertura da classe por critério

A JaBUTi pode também gerar relatórios mais completos como por exemplo, exportar

MyCalc actionPerformed(Ljava/awt...	101 of 129	
MyCalc checkFirstNum()Z	5 of 8	
MyCalc convertNumsToDouble()V	4 of 6	
MyCalc divOp()V	18 of 26	69%
MyCalc main(Ljava/lang/String;)V	0 of 0	
MyCalc multOp()V	16 of 18	88%
MyCalc opCodeMethod()V	17 of 20	85%
MyCalc plusOp()V	14 of 16	87%
MyCalc powerOp()V	5 of 14	
MyCalc setResults()V	0 of 0	
MyCalc setup()V	0 of 0	
MyCalc subOp()V	12 of 16	75%
MyCalc sysExit()V	0 of 0	

Figura 2.10: JaBUTi - Cobertura por métodos

relatórios para html com todos os dados de cobertura. Gerar informações de cobertura de cada critério, e em cada caso de teste individualmente. Também é gerado neste mesmo relatório, para cada critério, todos que foram cobertos e todos que não foram cobertos ainda.

2.3.3 Ferramenta Discover

A Discover é uma ferramenta que permite testar programas desenvolvidos na linguagem de programação Delphi. Usando uma interface gráfica, ela auxilia o testador de forma prática na cobertura do código. A Discover trabalha com teste de unidades somente para plataforma Windows.(CYAMON SOFTWARE, 2005)

Ao contrário das outras ferramentas apresentadas neste trabalho, a Discover somente trabalha com o Delphi, ou melhor, não há uma forma de introduzir outra linguagem nesta ferramenta. Não possui também um método que gere um grafo de controle ou de fluxo de dados. Ela apenas auxilia o testador diretamente pelo código-fonte, mostrando os códigos que foram testados através dos símbolos χ e $\surd$. A ferramenta baseia-se somente no critério Todos-Nós e no teste estrutural (caixa branca) (CYAMON SOFTWARE, 2005).

De acordo com a Figura 2.11, a Discover marca os trechos de código cobertos com o símbolo $\surd$ e, os não cobertos, com o símbolo χ. Do lado esquerdo da Figura 2.11 ficam as unidades do programa em teste. Do lado direito, situa-se o código-fonte de uma das unidades escolhidas para a análise de cobertura. A ferramenta disponibiliza relatórios que mostram todas as unidades e suas respectivas coberturas, tamanho, número de rotinas, além de quantidades de rotinas com 0% e 100% de cobertura.

Para que a Discover possa fazer o teste de um programa, primeiro o *Map file* do Delphi

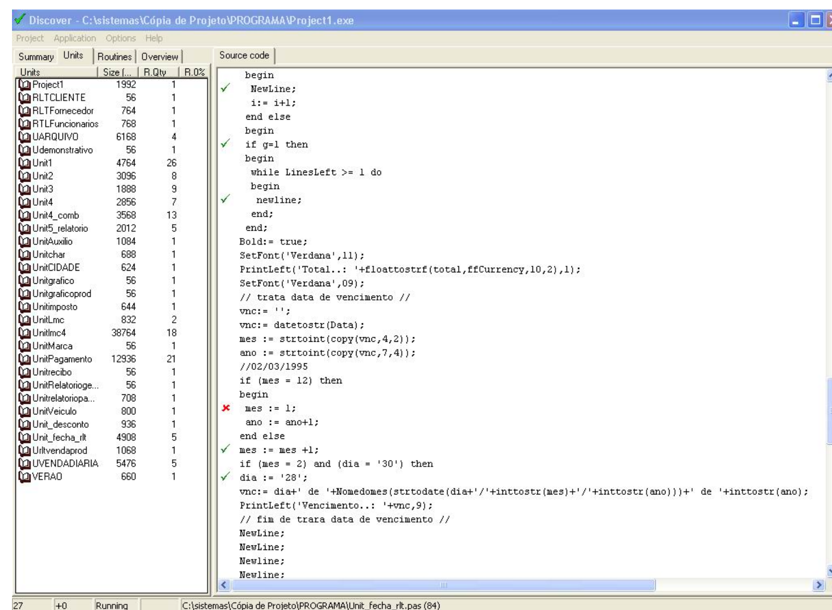

Figura 2.11: Tela de execução de teste na Ferramenta Discover

deve ser gerado durante a compilação do programa a ser testado. Este *Map file* gera um arquivo que armazena uma lista de segmentos, endereços de inicialização do programa, avisos ou erros produzidos durante a compilação do código executável e uma lista de símbolos públicos ordenada alfabeticamente, incluindo o endereço, tamanho em bytes, nome, grupo do segmento e a informação do módulo do programa. Para isso, é necessário configurar algumas opções no Delphi a partir do menu *project*, *Options*; na pasta *linker*, marcar a opção *Detailed*, conforme a Figura 2.12.

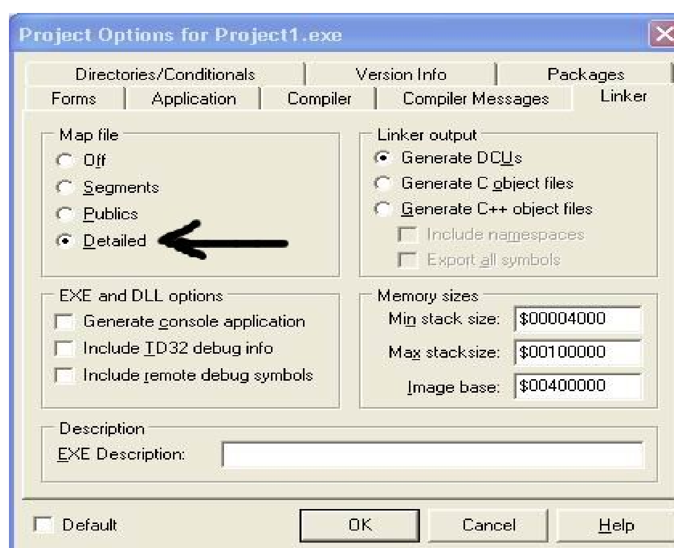

Figura 2.12: Pasta *Linker* do Delphi

2.4 Considerações Finais

Neste capítulo foi dada uma visão geral a respeito da atividade de teste e sua importância no processo de desenvolvimento de software. Além disso, foi fornecida uma descrição de diversas fases das principais técnicas e critérios de teste de software. Foram também apresentados exemplos das ferramentas POKE-TOLL, JaBUTi e Discover, descrevendo suas funcionalidades e aplicações no teste de software.

No próximo capítulo, será abordada a confecção da gramática de uma linguagem de programação, mostrará também as adaptações necessárias para ser utilizadas pelas ferramentas Lex e Yacc no propósito de gerar um analisador sintático.

3 Geradores de Compiladores

Este capítulo aborda o conceito de gramáticas visando a geração de compiladores. Será exemplificado o funcionamento da gramática BNF (Backus-Naur Form) e considerações sobre o funcionamento das ferramentas Yacc (*Yet Another Compiler-Compiler*) e Lex (*A Lexical Analyzer Generator*).

A gramática de uma linguagem de programação serve para determinar regras sintáticas e semânticas para a mesma, de forma que esta linguagem de programação seja escrita apoiada nesta gramática.

3.1 Principais Características de uma Gramática

As gramáticas sintáticas são dispositivos finitos que são usados freqüentemente para descrever linguagens infinitas. Quase todas as linguagens de programação são caracterizadas por uma gramática. Certamente, a gramática é parte da definição da linguagem. As gramáticas podem ser classificadas baseadas no tipo de produções que possuem. Uma importante classe é a gramática livre de contexto. São simples, mas bastante utilizadas pela maioria das linguagens de programação. Além disso, os algoritmos para reconhecê-las são computacionalmente tratáveis (SIMAO et al., 2001).

Uma árvore sintática representa a estrutura de um programa. Um instrumentador pode usar esta estrutura para derivar o grafo de um programa. Na Figura 3.2 é mostrado um exemplo de árvore sintática de um pequeno programa ilustrado na Figura 3.1, que é definida por uma gramática BNF abaixo:

$\langle S \rangle \rightarrow \langle W \rangle$

$\langle S \rangle \rightarrow \langle IF \rangle$

$\langle S \rangle \rightarrow$ ‘break’ ‘;’

$\langle S \rangle \rightarrow \langle ID \rangle$ ‘=’ $\langle E \rangle$ ‘;’

$\langle W \rangle \rightarrow$ ‘while’ ‘(’ $\langle E \rangle$ ‘)’ $\langle S \rangle$

$\langle IF \rangle \rightarrow$ ‘if’ ‘(’ $\langle E \rangle$ ‘)’ $\langle S \rangle$ ‘else’ $\langle S \rangle$

$\langle E \rangle \rightarrow \langle ID \rangle$ ‘>=’ $\langle C \rangle$

$\langle E \rangle \rightarrow \langle ID \rangle$ ‘+’ $\langle ID \rangle$

$\langle E \rangle \rightarrow \langle ID \rangle$ ‘(’ $\langle E \rangle$ ‘)’

$\langle E \rangle \rightarrow \langle E \rangle$ ‘,’ $\langle E \rangle$

$\langle ID \rangle \rightarrow$ identifier

$\langle C \rangle \rightarrow$ integer

Gramática BNF usada para escrever o programa da Figura 3.1

```
WHILE (A >= 1)
  IF (B >= 0)
   BREAK;
   ELSE
  A = A - B;
```

Figura 3.1: Programa exemplo *while*

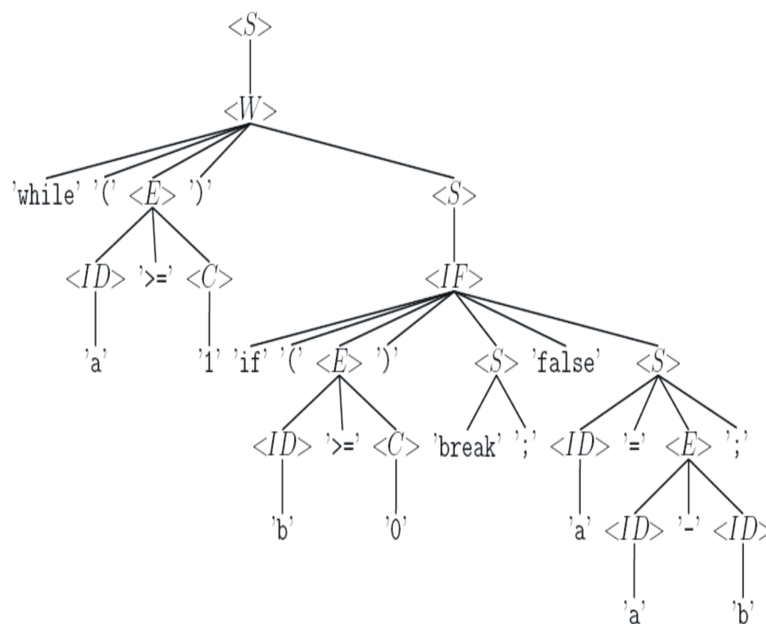

Figura 3.2: Árvore sintática *while*

Aho, Sethi e Ullman (1986) afirmam que muitas construções de linguagens de programação possuem uma estrutura inerentemente recursiva que pode ser identificada por gramática livre de contexto. Por exemplo, poder-se-ia ter um enunciado condicional definido por uma regra tal como:

se $S1$ e $S2$ são enunciados e E é uma expressão, então
"if E then $S1$ else $S2$" é um enunciado.

A Backus-Naur Form (BNF) foi criada (em meados da década de 1950) em um esforço conjunto de John Backus e Noam Chomsky. A BNF é o método mais popular para descrever concisamente a sintaxe de uma linguagem de programação.

As ferramentas usadas neste trabalho adotaram o uso da gramática BNF que é uma metalinguagem[1] que representa a gramática livre de contexto. A nova notação foi ligeiramente alterada um pouco mais tarde para se descrever a linguagem ALGOL-60 por Peter Naur. Este método revisado para descrição de sintaxe passou a ser conhecido como Forma Backus-Naur ou, simplesmente, BNF (Backus-Naur Form). Este tipo de gramática simplificadamente consiste em terminais, não-terminais, um símbolo de partida e produções.

Uma gramática BNF é composta por um conjunto finito de regras que definem uma linguagem de programação. Além disso, BNF usa abstrações para representar estruturas sintáticas. As abstrações na descrição BNF são freqüentemente chamadas de símbolos não-terminais, ou simplesmente não-terminais. A representação para os nomes das abstrações (não-terminais) em BNF é um nome cercado pelos símbolos de menor e maior $< e >$. Os itens léxicos da linguagem, também conhecidos como lexemas ou tokens, são representados por nomes em negrito. Os itens léxicos são freqüentemente chamados de terminais. Os terminais são símbolos básicos a partir dos quais as cadeias são formadas. A palavra "*token*" será um sinônimo de "terminal". No contexto de linguagem de programação, cada uma das palavras chaves **if**, **then**, **else** e **begin** é um terminal, e todas as palavras que são reservadas da linguagem de programação como operadores lógicos, aritméticos entre outros.

Os não-terminais são variáveis sintáticas que denotam cadeias de caracteres. Os não-terminais definem conjuntos de cadeias que auxiliam a definição da linguagem gerada pela gramática e também impõem uma estrutura hierárquica na linguagem que é útil tanto para a análise sintática quanto para a tradução.

[1]Uma metalinguagem é uma linguagem que é usada para descrever uma outra linguagem. BNF é uma metalinguagem para linguagens de programação

Uma regra BNF tem sempre um não-terminal em seu lado esquerdo, e composto por terminais e/ou não terminais em seu lado direito. O símbolo `::=` é usado com o sentido de "é definido por" e une o lado esquerdo ao direito da regra. O símbolo `|` é usado com o significado de "ou" e é usado para não se precisar escrever o lado esquerdo repetidas vezes. Na Figura 3.3 é exemplificada esta gramática e na Figura 3.4 é mostrado um exemplo de sua produção (AHO; SETHI; ULLMAN, 1986).

```
<programa> ::= begin <lista_sentenças> end

<lista_sentenças> ::= <sentença>

          | <sentença> ; <lista_sentenças>

<sentença> ::= <variável> := <expressão>

<variável> ::= A | B | C | D | E

<expressão> ::= <variável> + <variável>

          | <variável> - <variável>

          | <variável>
```

Figura 3.3: Exemplo da gramática BNF

Como se pode observar na Figura 3.3, os não-terminais da gramática BNF são 5 (programa, lista-sentenças, sentença, variável e expressão). Os terminais são 11 ($begin, end,$; , := , $A, B, C, D, E, +e-$). Um programa válido nesta linguagem é o que se encontra na Figura 3.4.

```
Begin
  A:=B + C;
  B:=B - D;
  A:=B
End
```

Figura 3.4: Programa exemplo - gramática BNF

A Forma de Backus-Naur (BNF) nos permite que o lado direito das produções possua

alguns operadores para facilitar a escrita da gramática. Operadores estes apresentados a seguir:

Seleção - Entre o parêntese e separado por um | podem ser utilizados na produção para optar por uma produção ou outra como mostra a Figura 3.5

exemplo $\langle digito \rangle \rightarrow (1|2)x$ esta produção resulta em:
$\langle digito \rangle \Rightarrow 1x$ ou
$\langle digito \rangle \Rightarrow 2x$

Figura 3.5: Exemplo de uma produção

Opcional - Como ilustra a Figura 3.6, são colchetes e o que estiver dentro deles possibilitam a opção de aplicar na produção ou não.

exemplo $\langle digito \rangle \rightarrow (1|2)[x]$ essa produção tem como válida o seguinte:
$\langle digito \rangle \Rightarrow 1$ ou
$\langle digito \rangle \Rightarrow 2$ ou
$\langle digito \rangle \Rightarrow 1x$ ou
$\langle digito \rangle \Rightarrow 2x$

Figura 3.6: Exemplo de uma produção opcional

Repetição de 0 ou mais vezes - O que estiver entre parêntese pode ser usado com um $*$ indicando que a produção que está dentro do parêntese pode se repetir zero ou mais vezes, representada pela Figura 3.7.

exemplo $\langle digito \rangle \rightarrow (1|2)^*x$ a produção resultaria no seguinte:
$\langle digito \rangle \Rightarrow x$ ou
$\langle digito \rangle \Rightarrow 1x$ ou
$\langle digito \rangle \Rightarrow 2x$ ou
$\langle digito \rangle \Rightarrow 11111x$ ou
$\langle digito \rangle \Rightarrow 222222x$ ou
$\langle digito \rangle \Rightarrow 1212122x$

Figura 3.7: Exemplo de uma produção com repetição

Repetição de 1 ou mais vezes - O que estiver entre parêntese pode ser usado com um $+$ inidicando que a produção que está dentro do parêntese pode se repetir uma ou mais vezes, como mostra a Figura 3.8.

exemplo $\langle digito \rangle \rightarrow (1|2)^{+}x$ a produção resultaria no seguinte:
$\langle digito \rangle \Rightarrow 1x$ ou
$\langle digito \rangle \Rightarrow 2x$ ou
$\langle digito \rangle \Rightarrow 11111x$ ou
$\langle digito \rangle \Rightarrow 222222x$ ou
$\langle digito \rangle \Rightarrow 1212122x$

Figura 3.8: Exemplo de uma produção com repetição

3.2 Yacc e Lex

Lex e Yacc são ferramentas desenvolvidas para escrever compiladores e interpretadores, no entanto são usadas para vários outros tipos de aplicações.

As Ferramentas IDel E IDelGen usadas neste trabalho, utilizam as ferramentas Lex e Yacc para fazer um analisador sintático para linguagem de programação fornecida a ela. Mas para que isso seja possível há necessidade do desenvolvimento de uma gramática para o analisador sintático (Yacc) e o léxico (Lex).

Como exemplo, pode-se tomar uma gramática de uma declaração de variável em C simplificada mostrada pela Figura 3.9

```
<dec>::=<tipo> <var> {, <var>}';'
<tipo>::=[signed|unsigned] (int|char)
<var>::={*} identificador {'[' numero ']'}
<letra> := a|b|c|d|e|f|g|h|i|j|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digito> := 0|1|2|3|4|5|6|7|8|9
<identificador> ::= <letra> { <letra>|<digito>|'_' }
<numero> ::= digito { digito }
```

Figura 3.9: Gramática de uma declaração de variável em C simplificada

Gramática simplificada no formato BNF

Esta gramática no formato BNF tem as seguintes características:

Símbolos terminais: signed, unsigned, int, char, '[', ']', ',' Símbolos não-terminais: <dec>, <tipo>, <var>, <identificador>, <numero> Símbolo inicial: <dec>. As seqüências válidas para ela são, por exemplo:

```
int a,**b[3][4];
unsigned char *p;
signed char str[100];
```

As Ferramentas Lex e Yacc trabalham juntas. O analisador léxico produzido pela ferramenta Lex separa o texto de entrada em vários símbolos, também conhecido como *tokens* e repassa para o analisador sintático. O analisador sintático produzido pela ferramenta Yacc "consome" *tokens* com o objetivo de construir uma frase válida segundo uma determinada gramática.

Para a gramática descrita acima o arquivo léxico(3.10) ficaria assim:

```
01 DIGITO   [0-9]
02 LETRA    [a-zA-Z_]
03 %%
04 [signed|unsigned] (int|char) { RETURN(TIPO); }
05 {LETRA}({LETRA}|{DIGITO})*   { RETURN(IDENTIFICADOR); }
06 {DIGITO}{DIGITO}*         { RETURN(NUMERO); }
07 "*"+             { RETURN(ASTERISCO); }
08 ";"              { RETURN(PONTOEVIRGULA); }
09 ","              { RETURN(VIRGULA); }
10 "["              { RETURN(ABRECOLCHETES); }
11 "]"              { RETURN(FECHACOLCHETES); }
12 %%
```

Figura 3.10: Arquivo com regras léxico

Exemplo de um arquivo léxico

O arquivo léxico, conforme citado anteriormente, tem a tarefa de reconhecer os *tokens*. Em outras palavras, trata-se de um *scanner* que busca os padrões escritos em forma de expressão regular. Neste arquivo, nas linhas 01 e 02, é possível observar duas definições para digito e letra, que serão usados na seções iniciadas por `"%%"`, que se estende da linha 04 a 10.

Na linha 4, observa-se uma expressão regular exigindo que para validar o retorno de "TIPO", seja encontrado um "signed", um "unsigned" ou nenhum deles, mas que tenha um "int" ou um "char" ao menos.

Na linha 05 vê-se outra expressão regular, formada por uma letra e mais uma combinação de letras ou números. Na linha 5 também existe uma expressão regular que combina um número ou um número mais uma combinação de números. Pode-se tomar como exemplo a expressão `{NUMERO}{NUMERO}*`. O asterisco usado no segundo "NUMERO" significa que pode haver mais de uma repetição de número ou nenhuma. No entanto, tem-se o primeiro "NUMERO" forçando que um número pelo menos seja reconhecido.

O arquivo sintático para a produção do analisador sintático(3.11) ficaria assim:

```
01 %token IDENTIFICADOR
02 %token NUMERO
03 %token ASTETISCO
04 %token PONTOEVIRGULA
05 %token VIRGULA
06 %token ABRECOLCHETES
07 %token FECHACOLCHETES
08 %token TIPO
09 %%
10 dec  :TIPO varlist PONTOEVIRGULA
11  ;
12
13 var  : ASTERISCO var_decl
14  | var_decl
15  ;
16
17 var_decl: IDENTIFICADOR
18  | INDETIFICADOR array_index
19  ;
20
21 array_index : ABRECOLCHETE NUMERO FECHACOLCHETE
22     | array_index ABRECOLCHETE NUMERO FECHACOLCHETE
23         ;
24
25 varlist : var
26  | varlist VIRGULA var
27  ;
28
29 %%
```

Figura 3.11: Arquivo sintático para a produção do analisador sintático

Arquivo sintático

Deve-se observar que a gramática BNF é escrita de forma única, ou seja, para que a gramática seja analisada sintaticamente é preciso construir um analisador léxico como mostrado anteriormente e um analisador sintático. Na verdade, o analisador léxico vai servir como um scanner que busca padrões escritos em forma de expressão regular e envia para o analisador sintático que, por sua vez, vai formar frases e validar de acordo com os padrões descritos no arquivo sintático acima.

O arquivo sintático define os *tokens* em letras maiúsculas e as regras em letras minúsculas somente para fácil vizualização, não é regra do gerador de analisador sintático,

ou seja, os terminais em letra maiúscula e os não-terminais em minúsculas. Observa-se que as linhas 01 a 08 servem para informar quais são os *tokens* que serão usados. Nas linhas 10 a 28 estão presentes as regras da gramática. Pode-se observar na linha 10 uma regra chamada "dec" (declaração), cuja satisfação se dá através de três argumentos válidos. Um "TIPO", "varlist", "PONTOEVIRGULA", "TIPO" e "PONTOEVIRGULA" são retornados direto pelo analisador léxico. O "varlist" é uma regra cujas características encontram-se na Linha 25 da Figura 3.11.

3.3 Considerações Finais

Neste capítulo foram apresentadas técnicas de construções de gramática, que foram utilizadas para a confecção da gramática da linguagem Delphi. Este capítulo é importante para a compreensão da complexidade da construção da gramática utilizada neste trabalho. A gramática Delphi foi muito trabalhosa de se construir, a própria linguagem Delphi tem uma gramática no padrão BNF, mas não se encontra completa, muitas pesquisas e testes foram realizado para o termino desta gramática. Este capitulo também ressalta sobre as ferramentas Yacc e Lex, que são utilizadas para geração do instrumentado Delphi juntamente com a ferramenta IDeLGen. No próximo capítulo serão apresentadas detalhadamente as ferramentas IDeLGen e IDeL que seu propósito é gerar um instrumentador para uma determinada linguagem de programação, e este trabalho utiliza estas ferramentas para gerar um instrumentador para Linguagem Delphi.

4 IDeL e IDeLGen

Várias atividades na engenharia de software inserem-se no contexto da análise de um programa: engenharia reversa, análise de performance, visualização de programas e depuração. Instrumentação é uma técnica freqüentemente usada na engenharia de software para estes diferentes propósitos.

Na atividade de teste de software, os grafos de programas são indicados para manipulação e análise. Para a sua análise, é freqüente a necessidade de uma visão mais detalhada de um programa em execução. Pode-se querer saber, por exemplo, qual caminho foi percorrido. Para este propósito, é necessária a inserção de comandos no código do programa sem mudar a funcionalidade do mesmo, mas inserindo informações como *log* para percorrê-lo.

Um programa de teste pode usar estes *logs* para registrar a informação de execução, ou seja, qual nó, aresta, arcos, definição e uso foram alcançados. Pode-se desejar também o valor de uma variável, no momento em que o programa executa determinado nó.

Simao et al. (2001) propuseram uma instrumentação orientada à metalinguagem, chamada IDeL[1], projetada para suportar ambas tarefas de instrumentação: a derivação estrutural do produto e a inclusão de instruções de comando. Esta metalinguagem deve ser instanciada fornecendo-se uma gramática livre de contexto para uma linguagem específica. Para promover seu uso prático, IDeL é também suportada por um sistema, conhecido como IDelgen[2], que pode ser visto como um gerador da aplicação feito sob medida para processo de instrumentação, facilitando o reuso e a evolução da instrumentação.

[1]**I**nstrumentation **De**scription **L**anguage
[2]**I**nstrumentation **De**scription **L**anguage **Gen**erator

4.1 Principais Características

A fim de projetar um mecanismo abstrato para instrumentar programas, é necessário criar um arquivo em que as regras de instrumentação próprias da linguagem sejam descritas. Através deste arquivo, o programa poderá ser instrumentado para determinada linguagem cujas regras estejam nele descritas.

Simao et al. (2001) decidiram usar as árvores sintáticas como formato intermediário. Primeiramente, programas escritos na maioria das linguagens podem (com mais ou menos esforço) ser traduzidos em uma árvore sintática e as técnicas para isso são bem estabelecidas e bem definidas na literatura. Em segundo lugar, usando conceitos do paradigma da programação transformacional (exemplificado por linguagens tais como a TXL e Refine), é possivel definir métodos para manusear a árvore sintática (SIMAO et al., 2001).

4.1.1 Aspecto Operacional

Uma vez selecionada uma gramática, um programa nesta gramática e uma descrição de instrumentação para a mesma, podem ser gerados os grafos e instrumentação do programa. Para esta tarefa foi desenvolvido o IDeLgen, que tem como objetivo gerar um programa de instrumentação específico de uma gramática. Na Figura 4.1 é mostrada a execução do IDeLgen, onde observa-se que, para o IDeLgen gerar o programa IDel.grm, será necessária uma gramática específica para o programa que será instrumentado, deste ponto em diante, o programa deve ser executado com uma descrição de instrumentação chamado de *desc* e um programa *P*, gerando finalmente os grafos e a instrumentação.(SIMAO et al., 2001)

A gramática é composta por dois arquivos, um deles contém expressões léxicas, que geralmente é representado com a extensão ".l". O outro arquivo é descrito por uma gramática livre de contexto para análise sintática. Ela é representada pela extensão ".y", como se pode ver na Figura 4.1. A Idel funciona da seguinte maneira: dado uma gramática .l e .y, a Idel produz um programa chamado IDeL.grm. Este por sua vez receberá um código escrito na linguagem em questão e um esquema de como será instrumentado, como representado na Figura 4.1 com o nome de *desc*. Desta forma, quando executado, são gerados os grafos e o programa instrumentado.

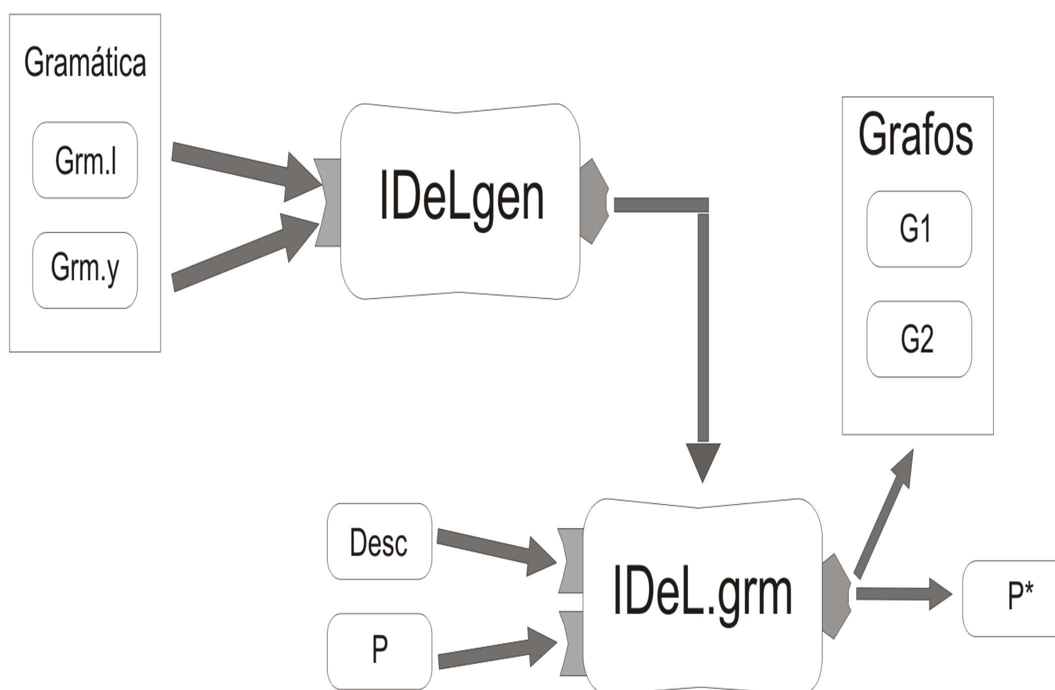

Figura 4.1: Execução IDeLgen

4.1.2 Estrutura do IDeL

A instrumentação na IDeL está dividida em três partes principais: Unidade de identificação, Unidade de processamento e Implementação.

4.1.2.1 Unidade de Identificação

A primeira etapa da instrumentação dedica-se à identificação das unidades de alto nível no programa a ser instrumentado. O código é analisado e são extraídas as unidades, que podem ser funções, métodos ou procedimentos, dependendo da linguagem. Na instrumentação, as unidades são caracterizadas por uma lista de padrões. Padrões são como dados para pesquisa que procuram funções, métodos ou procedimentos. Toda vez que um destes padrões for encontrado, a unidade será processada, ou seja, após encontrado um padrão a unidade de processamento descrita abaixo é acionada (SIMAO et al., 2001).

4.1.2.2 Unidade de Processamento

A Unidade de processamento é a parte principal de um instrumentador, ela é responsável por definir, produzir o grafo de programa e adicionar marcas especiais à lista de inserção dos nós da árvore sintática quando necessário. Estas marcas indicam que tipo

de transformação deverá ser feita na árvore sintática, de modo a inserir pontas de provas. A unidade de processamento é dividida dentro de uma seqüência de etapas de processo. Esta consiste de uma seqüencia de regras para instrumentação. Por exemplo, regras que decidirão como tratar comandos como *while*, *if* e outros.

Dependendo do tipo de instrumentação desejada, pode ser necessário alterar o programa para monitorar sua execução. Tal alteração depende da estrutura do programa e da semântica da linguagem. Na maioria das vezes, as alterações são realizadas antes ou depois de uma determinada estrutura, e estão relacionadas com um nó da árvore em particular. Para isso, o instrumentador mantém em cada nó da árvore uma tabela que contém as alterações que devem ser feita nos componentes. Na última fase da instrumentação a árvore é percorrida e cada alteração é realizada baseada nas informações presentes na tabela de implementação de cada um dos nós.

Simao et al. (2001) criaram regras no programa IDeL que é composto de sete seções: definição de nome, declaração de meta-variável, definição de padrão, geração de nós, definição da topologia do grafo, atribuição e inserções de marcas.

Estas regras servem para determinar como será gerado um grafo e instrumentado o código fonte da linguagem de programação fornecida. Elas estão contidas no arquivo *desc* (Figura 4.1) usado pela IDeL. Este arquivo pode ser alterado para que haja compatibilidade às varias linguagens de programação existentes.

```
1   rule While
2   var
3       :e as [E]
4       :s as [S]
5   pattern
6       [S<while ( :e ) :s>]
7   declare node $control
8   graph
9       $begin    -> $control
10      $control -> $begin:s
11      $end:s    -> $control
12      $control -> $end
13  assignment
14      assign $break:s      to $end
15      assign $continue:s to $control
16      assign $puse:e       to $control
17  instrument
18      add checkpointBefore $control to :e
19      add checkpointBefore $begin:s to :s
20      add checkpointAfter  $end      to  self
21  end rule
```

Figura 4.2: Regra de instrumentação do comando *while*

Pode-se ter uma idéia destas regras de acordo com a regra de instrumentação mostrada pela Figura 4.2. Para se entender melhor como funcionam estas regras de instrumentação, observa-se na linha 1, que é uma definição do nome, que neste caso, define o nome da regra

como “While”. Isto tem somente a finalidade de documentar, e não tem nenhum impacto sobre a semântica da regra. A linha 2 até a 4 é a seção da declaração de meta-variáveis.

As linhas 5 e 6 fazem parte da seção da “definição de padrão”, a que as sub-árvores da árvore sintática devem combinar para que esta regra de instrumentação seja aplicada, conforme as meta-variáveis. Neste padrão, as meta-variáveis :e e :s são unificadas a expressão de controle e ao corpo do while respectivamente. A linha 7 é a seção de declaração do nó. Neste exemplo, ela (a tarefa) cria um novo nó, adiciona-o ao grafo e atribui ao nome simbólico `$control` no mapeamento de nó do grafo correspondente à sub-árvore que combina com o padrão.

Para entender como (e porquê) a instrumentação funciona, é necessário primeiro explicar algumas coisas que são assumidas neste padrão (ou modelo). Todo comando tem dois nomes simbólicos: `$begin` e `$end`. Estes são, respectivamente, os nós logo antes e logo depois do comando. Na verdade, estes nomes simbólicos assumidos são fornecidos por passos de processamento anteriores, responsáveis por criar e atribuir valor a eles.

As linhas 8 a 12 referem-se à seção da definição da topologia do grafo, na qual declaram as arestas no grafo. Neste exemplo, a linha 9 cria a aresta que liga o nó atribuído a `$begin` ao o nó do grafo atribuído a `$control`. A linha 10 cria a aresta que liga o nó atribuído a `$control` ao nó atribuído a `$begin` na sub-árvore unificada para a meta-variável :s.

A linha 14 atribui o nome simbólico `$break:s` ao nó do grafo `$end`, isto significa que quando o instrumentador está analisando os comandos na sub-árvore :s, uma referência ao nó “break” será direcionada ao nó após o *while*. Esta forma é utilizada no comando *break* na linguagem C.

A linha 16 atribui o nome simbólico `$puse:e` ao nó do grafo atribuído a `$control`. As linhas 17 a 20 são a seção da atribuição e inserções de *logs*. Nesta seção, a regra da instrumentação adiciona marcas especiais (comandos log) às listas da inserção, a fim de apontar onde as indicações do registro devem ser inseridas (SIMAO et al., 2001).

A Figura 4.3 ilustra como a regra *while* funciona. Os nomes simbólicos `$begin`, `$begin:s`, `$end:s` e `$end` são atribuídos aos nós apropriados antes de aplicar esta regra. O nó e as arestas que foram criadas por esta regra são destacados com linhas tracejadas, e os novos nomes simbólicos estão em itálico (SIMAO et al., 2001).

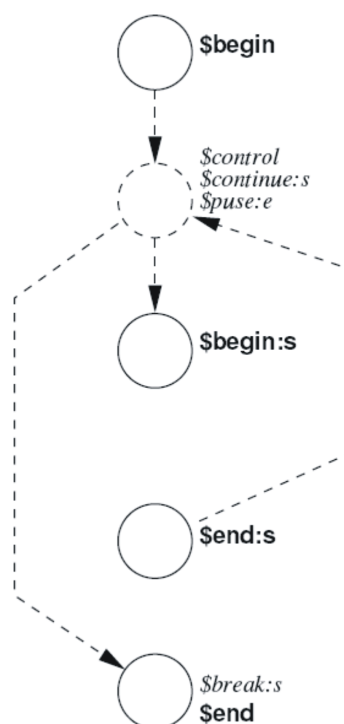

Figura 4.3: Grafo de como funciona a regra *while*

4.1.2.3 Implementação

Na unidade de processamento, as declarações *instrument* são abstratas. A instrumentação pode variar de linguagem para linguagem. A implementação dá uma sintaxe concreta para a declaração *instrument* e define na unidade de processamento como aplicar então a linguagem em questão. Nesta seção, é definido como o programa deve ser mudado a fim de introduzir os comandos (SIMAO et al., 2001).

4.2 Execução e Exemplos

A instrumentação de um programa na IDeL pode ser realizada em uma linguagem qualquer, desde que uma gramática livre de contexto e uma descrição de instrumentação estejam de acordo com esta linguagem. Esta descrição de instrumentação são regras criadas para que se gere a instrumentação e o grafo de acordo com a flexibilidade de cada linguagem.

4.2.1 Exemplos

O comando abaixo gera o aplicativo (instrumentador) com o nome de idel.C, recebe o parâmetro "C" e espera encontrar os dois arquivos "C.l" e "C.y" como foi exemplificado na Seção 4.1.1.

```
comando: idelgen C
```

Dado um programa $\boldsymbol{P}$, que é uma simples calculadora de peso ideal mostrado na Figura 4.4, pode-se executar na IDeL, usando o seguinte comando:

```
comando: ./IDeL.C -p <Programa> -i IDeL.grm -g <Nome do Programa>
 -o <Nome do arquivo instrumentada>
```

O comando "*IDeL.C*", na linha acima, é o aplicativo que foi gerado pelo IDeLgen (instrumentador) com as devidas gramáticas que, por sua vez, irá receber os parâmetros para a instrumentação. A opção -i aponta a descrição IDeL do instrumentador. A opção -p serve para indicar qual será o arquivo a ser instrumentado. A opção -g define o nome base que será utilizado para gerar os arquivos do tipo "DOT" posteriormente utilizado pela Ferramenta Graphiz[3] para gerar as figuras dos grafos e a opção -o informa o arquivo final instrumentado. O arquivo *IDeL.grm* acima são as descrições para gerar os grafos (Seção 4.1.2.2). Logo abaixo é mostrado um exemplo com o programa *programateste.C*.

```
exemplo: ./IDeL.C -p programateste.C -i IDeL.grm -g programateste
 -o programateste_instrumentado.C
```

Na verdade, a execução é bastante simples. Passando como argumento o nome dos dois arquivos, o instrumentador realiza a tarefa. São gerados dois arquivos: o programateste-instrumentado.C e o programateste.main.dot. O arquivo programa-instrumentado.C (Figura 4.5) é o programa instrumentado, o arquivo.main.dot (Figura 4.6) é o grafo. Conforme mostra a Figura 4.5, pode-se observar as *pontas de prova* de intrumentação indicadas pelas flechas. A ferramenta IDeL gera um grafo em modo texto (Figura 4.6) que é usado na Ferramenta ´´Graphviz"para produção de um grafo em forma de figura. O grafo do programa representado pela Figura 4.6 é passado para a Ferramenta Graphviz que produz um grafo mostrado pela Figura 4.7.

[3] *Graph Visualization Software* - é uma ferramenta que faz o layout e renderiza um grafo fornecido no forma de texto.

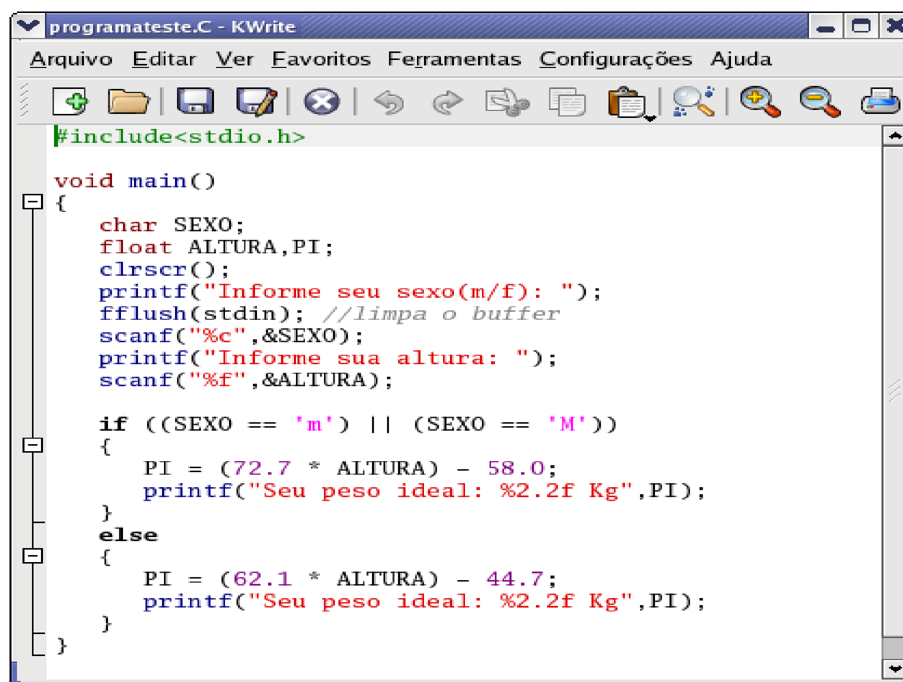

Figura 4.4: ***P*** - programateste.C

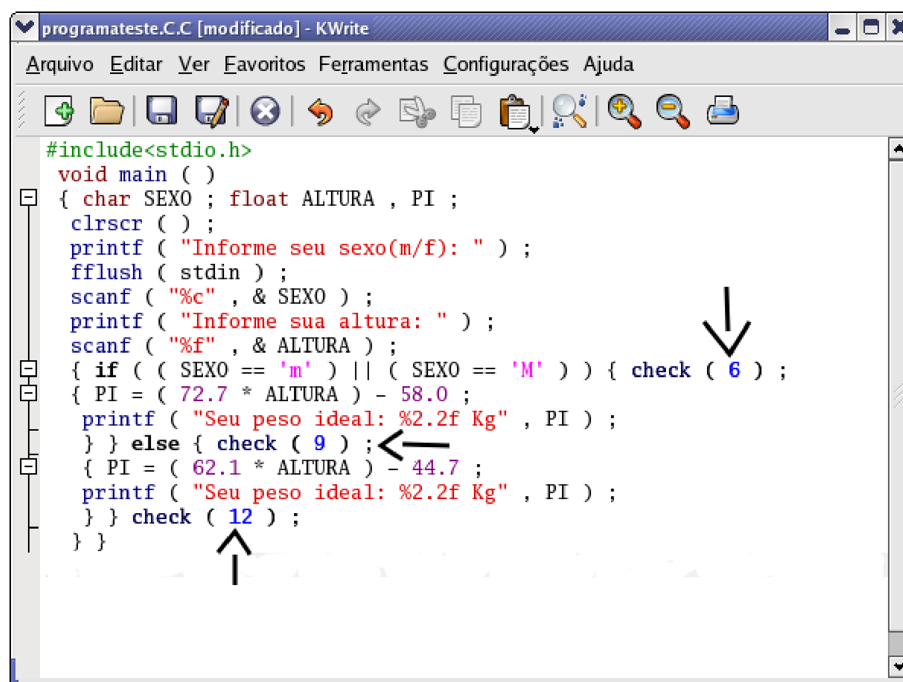

Figura 4.5: Arquivo programateste-instrumentado.C

4.3 Considerações Finais

Neste capítulo foi dada uma visão geral a respeito das Ferramentas IDeL e IDeLgen. Como visto, as ferramentas apresentam condições necessárias para gerar um instrumenta-

programateste.main.dot - KWrite

Arquivo Editar Ver Favoritos Ferramentas Configurações Ajuda

```
digraph gfc {
node [shape = doublecircle] 12;
node [shape = circle] 1;
/* cusage of clrscr at 1 */
node [shape = circle] 2;
/* cusage of stdin at 2 */
node [shape = circle] 3;
/* definition of SEXO at 3 */
node [shape = circle] 4;
/* definition of ALTURA at 4 */
node [shape = circle] 5;
/* pusage of SEXO at 5 */
/* pusage of SEXO at 5 */
node [shape = circle] 8;
node [shape = circle] 6;
/* definition of PI at 6 */
/* cusage of ALTURA at 6 */
node [shape = circle] 7;
/* cusage of PI at 7 */
node [shape = circle] 11;
node [shape = circle] 9;
/* definition of PI at 9 */
/* cusage of ALTURA at 9 */
node [shape = circle] 10;
/* cusage of PI at 10 */
1 -> 2;
2 -> 3;
3 -> 4;
4 -> 5;
5 -> 6;
5 -> 9;
8 -> 12;
11 -> 12;
6 -> 7;
7 -> 8;
9 -> 10;
10 -> 11;
 }
```

Figura 4.6: Arquivo programateste.main.dot

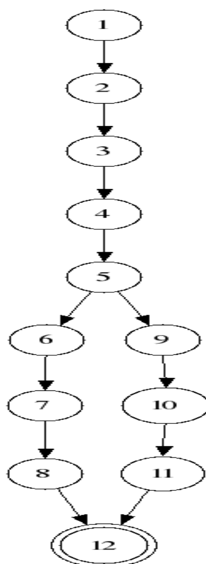

Figura 4.7: Grafo gerado pela Ferramenta Graphviz

dor competente para qualquer tipo de linguagem de programação, sendo possível descrever como será inserido pontas de provas no código da linguagem em questão, pois cada linguagem de programação tem caracteriscas diferentes. Este capítulo apresenta também

exemplos da funcionalidade e aplicação das Ferramentas IDeL e IDeLGen.

No próximo capítulo as Ferramentas IDel e IDeLgen serão utilizadas na geração de um instrumentador para linguagem de programação Delphi, discutindo sobre a confecção dos artefatos necessários na produção do instrumentador.

5 Instrumentador para a linguagem Delphi

Conforme visto nos capítulos anteriores, o principal objetivo da atividade de teste é aumentar a confiabilidade e garantir a qualidade de um produto para que este seja liberado com o mínimo de defeitos possível.

Procurando reduzir o custo e o tempo associado à atividade de teste, bem como aumentar a sua eficácia, várias técnicas e critérios de teste têm sido propostos. Devido ao aspecto complementar destas técnicas, o testador deve aplicá-las em conjunto para obter um teste de boa qualidade. O desenvolvimento de ferramentas de teste também contribui para a redução dos custos associados a esta atividade, além de evitar que erros sejam introduzidos pela intervenção humana durante a realização dos testes. Embora haja uma variedade de programas para teste de software, surge a necessidade de uma ferramenta para teste de software em Delphi, que hoje é largamente usada para produção de software. Por isso, o propósito deste trabalho é produzir um mecanismo que instrumente o código fonte e gere o grafo. Com o auxílio das Ferramentas IDeL e IDeLGen isso se tornou possível através da construção de uma gramática e uma descrição de instrumentação requerida para a geração de um instrumentador.

Neste capítulo são mostradas as principais características de um instrumentador para a Linguagem Delphi utilizando as Ferramentas IDeL e IDeLGen é apresentado também características da Linguagem Delphi. Este capítulo inicia com as principais características da linguagem de programação Delphi. Em seguinda, inicia a confecção do instrumentador.

Para que a IDeL instrumente o programa em Delphi há necessidade de se passar por duas etapas:

- Preparação e construção da gramática do Delphi;
- Adequação da descrição de instrumentação da Ferramenta IDeL.

5.1 Principais características da Linguagem Delphi

Delphi é uma ferramenta predominantemente visual, embora deva-se codificar para que o programa possa fazer o que se deseja dele.

A linguagem que está por trás dos programas Delphi é a Object Pascal, derivada da Linguagem Pascal, criada por Niklaus Wirth. Originalmente, ela era apenas uma linguagem estruturada. Com o tempo a Borland criou o Turbo Pascal e, a partir da Versão 5.5, criou extensões orientadas a objeto.

A Linguagem Pascal não é sensível a caixa, isto é, não faz diferença entre letra maiúscula e minúscula. As linhas podem ser quebradas em qualquer lugar (menos no meio de uma cadeia de caracteres).

O Delphi permite três tipos de comentários:

- “`{`” a chave indica que inicia um comentário com várias linhas até que haja outra chave “`}`” fechando então o comentário.
- “(*” o abre parêntese mais o asterístico é outra forma de comentar várias linhas e termina com asterístico mais fecha parentese “*)”.
- “`//`” é um tipo de comentário que funciona somente para uma unica linha, se houver quebra de linha ele não satisfaz mais”.

A Linguagem Pascal que é usada no Delphi é fortemente tipada, isto é, para se usar uma variável deve-se declará-la, determinando o tipo a ela associado. Por exemplo:

```
var
  varint   :  Integer;
```

Esta declaração cria uma variavel do tipo inteira.

5.1.1 Controladores de fluxo

Toda linguagem de programação tem a necessidade de controlar o fluxo. A Linguagem Delphi tem uma série de comandos que faz isso. Por exemplo:

if..then..else: Este comando permite desviar o fluxo, após a palavra chave *if* vem uma expressão que, se verdadeira, executa o comando após o *then* e, se for falsa, executa

o comando após o *else*. O *else* é opcional e, se não for utilizado, não executará nada de especial caso a expressão seja falsa:

```
if a = 3 then
 b := 5
else
 c := 10
```

Nota-se que não há ; antes do *else*, pois a linguagem determina que haja um ; para separar uma instrução de outra instrução, o comando `b := 5 else c := 10` forma uma unica instrução. O *begin* e o *end* são opcionais no exemplo acima, mas no exemplo abaixo não, o *begin* e o *end* servem para separar blocos de comando. Se não forem utilizados entende-se que somente um comando depois do *then* e um comando depois do *else* sejam executados.

```
if a = 3 then
begin
 b := 5;
 c := 10
end else
begin
 c := 12;
 b := 4
end;
```

Case: Várias vezes é necessário usar seqüência muito grande de teste, o Delphi tem um comando que permite substituir esta seqüência, quando se quer testar uma variável para verificar se ela é um determinado número (ou tipo ordinal – caracteres, enumerados, inteiro): o *case* tem a estrutura semelhante à seguinte:

```
case a of
 1: ...;
 2: ...;
 3: ...;
 else ...;
```

For: Quando for preciso executar uma mesma ação repetidas vezes, pode-se usar um dos comandos de repetição disponíveis no Delphi.

O *for* é usado quando se sabe com antecedência o número de vezes que a operação será executada. Sua sintaxe é:

```
for i := 1 to 10 do
...
```

i é uma variável definida anteriormente, que guarda um contador que mostra quantas vezes o comando foi executado. Ele pode aumentar de valor, como mostrado acima, ou diminuir de valor, como em:

```
for i := 10 downto 1 do
...
```

Neste caso, usa-se a palavra-chave *downto* para indicar o decréscimo d variável.

While: O *for* permite incrementos no contador de 1 ou mais de 1, não possibilitando testes de condições mais complexas. Uma repetição mais flexível é o *while*. Sua sintaxe é:

```
while <condição> do
```

`<condição>` pode ser qualquer expressão boleana desejada. Por exemplo:

```
j := 0;
while j < 100 do begin
   // executa algum procedimento
   j := j+2;
end;
```

Repeat: A repetição *while* pode não ser executada nenhuma vez, caso a condição de teste não seja satisfeita, porque o programa nem entra na repetição. Quando se quer obrigar a execução do procedimento de repetição ao menos uma vez e testar a condição ao final, pode-se usar o *repeat*:

```
repeat
...
until <condição>;
```

Este loop é executado pelo menos uma vez e repetindo até que a condição seja verdadeira.

5.1.2 Exceções

No Delphi 1 foram introduzidas as exceções. Quando ocorre algo que não deveria acontecer ou uma condição de erro, é levantada uma exceção. O fluxo do programa se interrompe, sendo capturado por um manipulador de exceções, que trata o erro, exemplo:

```
if num = 0 then
 raise exception.create('Número inválido');
num := 3000/num;
```

Inicialmente, testa-se a variável *num* para verificar se ela é igual a 0. Se for, é levantada uma exceção. Neste caso, a linha seguinte não é executada.

Normalmente, as exceções são tratadas no manipulador padrão do delphi, que mostra um quadro com a mensagem. Porém, há casos em que se deseja um tratamento especial de exceção. Quando isso é necessário, usa-se um bloco *try ... except*, como mostrado a seguir:

```
try
   if num = 0 then
     raise exception.create('Número inválido');
   num := 5000 / num;
except
   showmessage('O Numero não pode ser 0');
end;
```

5.2 Aspectos de implementação

Para a geração de um instrumentador são necessários na IDeLGen dois arquivos. A IDeLGen aceita a entrada de arquivos com as extensões **.l**(Lex) e **.y**(Yacc), os dois formam uma gramática. Estes arquivos são produzidos conforme as ferramentas Lex e Yacc requerem. Na Figura 4.1 é mostrada a implantação e execução da IDeLGen.

Na Figura 5.1 é mostrada na prática como executar o IDeLGen e gerar um instrumentador, no comando simples da Figura 5.1 o "Delphi" representa os arquivos "delphi.l" e "delphi.y"

```
giovanni@server:~/delphi$ idelgen delphi
Building tree constructor
#
#
#
Building operator parser
#
#
#
Builting idel.delphi
Done.
```

Figura 5.1: Geração do instrumentador idel.delphi

5.2.1 Geração do arquivo delphi.idel

A descrição do instrumentador para a Linguagem Delphi está contida no arquivo "delphi.idel". Este arquivo é dividido em três seções: (i) Identificação das unidades, (ii) Processamentos das unidades e (iii) Implementação. O exemplo a seguir mostra um esquema de como o arquivo "delphi.idel" é estruturado.

```
Instrumenter <nome_do_instrumentador>

## Parte 1
unit
# Identificação da unidade
end unit

## Parte 2
setp <nome_do_passo>

pattern <nome_do_padrao>
   # Definição do padrão
end pattern

## Parte 3
implementation

implement
   # implementação
```

```
end implement

end instrument
```

5.2.1.1 Identificação das Unidades

A unidade é uma estrutura de alto nível que contém o código e a lógica do programa. Nos programas Delphi existem três tipos de unidade: função, procedimento e o corpo do programa principal. No arquivo "delphi.idel" encontra-se a seção da identificação de unidades, que visa encontrar estas construções da Linguagem Delphi. A partir de então, pode-se construir as árvores sintáticas. Para cada unidade encontrada será gerado um novo grafo. A unidade em questão encontrará funções e procedimentos e, a partir de cada um deles, será então gerado um grafo.

Uma limitação da IDeL é a não capacidade de reconhecer vários tipos de unidades. No caso da Linguagem C ou Java isto não representaria problemas, pois toda a lógica do programa está localizada dentro de funções e métodos, respectivamente. Para resolver este problema, optou-se por generalizar na gramática do Delphi função e procedimento,

```
 1 unit
 2 var
 3 .       :head   as  [name_impl]
 4 .       :name   as  [func_or_proc_or_method_name]
 5 .       :type   as  [proc_or_func_fptype]
 6 .       :pars   as  [fp_list]
 7 .       :decl   as  [impl_decl_sect_list]
 8 .       :ss     as  [stmt_list]
 9 named by
10 .       :name
11 match
12 .       [main_impl< :head :name :pars :type ; :decl begin :ss end ; >]
13 end unit
```

Figura 5.2: Seção de identificação das unidades do arquivo delphi.idel.

Na Figura 5.2 é mostrada a seção de identificação de unidades no arquivo "delphi.idel". A palavra **unit** é reservada, da linha 2 a 8 são declaradas as meta-variáveis que são utilizadas na linha 12. Estas metas variáveis casam com as regras gramaticais definidas na gramática pelo arquivo "delphi.y". As linhas 9 e 10 definem o nome pelo qual a unidade vai ser identificada, que neste caso se casa com a regra da gramática "`[func_or_proc_or_method_name]`" que é o próprio nome da função ou procedimento.

As linhas 11 e 12 constituem a seção de comparação e contêm o padrão que deve ser

encontrado na árvore sintática para que uma unidade seja identificada. Esse padrão está de acordo com as produções existentes na gramática Delphi, conforme pode ser observada na árvore sintática de padrões na Figura 5.3, que corresponde à produção `main_impl` da gramática. Quando esse padrão for identificado na árvore sintática do programa, a fase do processamento das unidades se inicia.

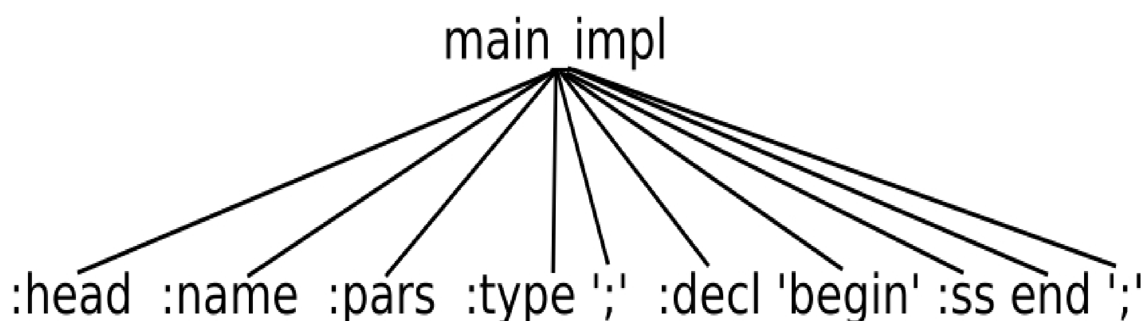

Figura 5.3: Árvore de padrões da produção :head.

5.2.1.2 Processamento das Unidades

A seção de processamento das unidades é composta de zero ou mais passos iniciados com a palavra reservada *step*, os nomes dos passos não são palavras reservadas da ferramenta. Cada passo contém uma série de padrões que são testados e aplicados na árvore sintática caso seja possível.

Seguem os passos utilizados para o instrumentador de Delphi:

- Passo FindFunction - Responsável por encontrar uma unidade, ou seja, uma função ou um procedimento;
- Passo MarkStatements - Este passo tem por objetivo encontrar, dentro de cada unidade, seus comandos;
- Passo LinkStatement - Tem o propósito de determinar os nós iniciais e finais de cada lista de comando.
- Passo JoinStatement - Este passo cuida da ligação dos nós do grafo, ele cria arestas que liga nós de um bloco de comando ao outro.
- Passo JoinToFunction - Este passo, além de cuidar dos nós do grafo, também adiciona duas pontas de provas afim de identificar no código os números dos nós logo

antes do nó `$init` e logo depois do nó `$exit` que são os nós do inicio da função e do fim da função respectivamente, e faz alteração para a instrumentação.

- Passo MakeGraph - Este passo é responsável pela instrumentação e a construção do grafo das principais construções da linguagem Delphi.

A Figura 5.4 servirá como exemplo para o entendimento dos passos da Ferramenta IDeL.

```
procedure comparar( var a,b :integer);
 begin

    while (a < 10) do
       b:= b + 1;

    if (b > 10) then
       b := a

 end;
```

Figura 5.4: Programa Exemplo.

Passo FindFunction

Assim que a unidade for identificada, o primeiro passo é o *FindFunction* que tem como finalidade procurar cada função ou procedimento e criar dois nós o `$init` e `$exit`. A linha 1 da Figura 5.5 define o nome do passo como "FindFunction". Este passo é composto apenas pelo padrão Function, cuja declaração é iniciada na linha 3. As linhas 4 a 10 são as declarações das meta-variáveis que são usadas para encontrar o padrão dado pela linha 12. As linhas 13 e 14 declaram os nós `$init` e `$exit` mostrado na Figura 5.6. As linhas 15 e 16 correspondem a uma seção de atribuição de nomes simbólicos. A atribuição da linha 16 serve para indicar que ocorre uma definição dos parâmetros formais da função no nó `$init`, ou seja, ele está associando o nó "$parameterdefinition" unificado com a meta-variável ":par", que é atribuída pela regra `fp_list` da gramática do Delphi. Ela representa os parâmetros de uma função ou procedimento da linguagem Delphi. A figura 5.7 mostra uma árvore sintática de uma produção como **(var keys : Word)**, isso tudo é ligado ao nó simbólico $init.

No caso do procedimento "comparar"da Figura 5.4, as variáveis "a"e "b"são associadas ao nó `$init` a fim de marcar as atribuições das variáveis.

```
 1 step FindFunction
 2
 3 pattern Function
 4 var
 5       :head   as  [name_impl]
 6       :name   as  [func_or_proc_or_method_name]
 7       :type   as  [proc_or_func_fptype]
 8       :pars   as  [fp_list]
 9       :decl   as  [impl_decl_sect_list]
10       :ss     as  [stmt_list]
11 match
12       [main_impl< :head :name :pars :type ; :decl begin :ss end ; >]
13 declare node $init
14 declare node $exit
15 assignment
16       assign $parameterdefinition:pars to $init
17 end pattern
```

Figura 5.5: Padrão FindFunction.

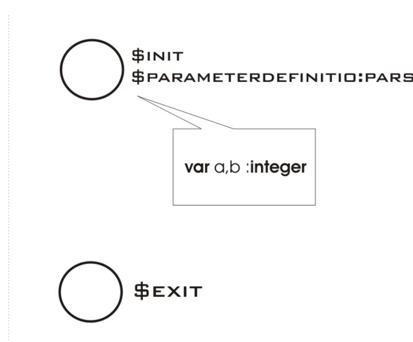

Figura 5.6: Grafo do Padrão FindFunction.

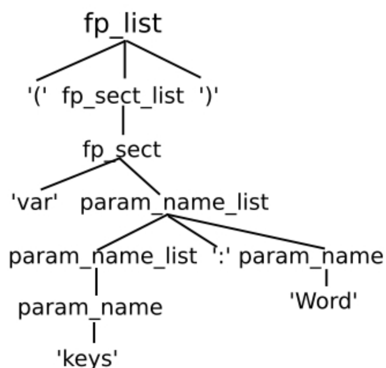

Figura 5.7: Árvore sintática fplist.

Passo MarkStatements

O próximo passo a ser aplicado na árvore é o MarkStatements (Figura 5.8). Ele é composto pelo padrão FooStatement, declarado na linha 3. Esse padrão tem o objetivo de encontrar todos os comandos presentes na árvore sintática de acordo com a linha 7, e declarar um nó `$begin` e um nó `$end` (linhas 8 e 9), que correspondem ao início e ao fim

do comando, respectivamente. A Figura 5.9 exemplifica a ação das linhas 8 e 9 da Figura 5.8.

```
 1 step MarkStatements
 2
 3 pattern FooStatement
 4 var
 5 .       :s      as  [unlabelled_stmt]
 6 match
 7 .       [stmt< :s >]
 8 declare node $begin
 9 declare node $end
10 end pattern
```

Figura 5.8: Padrão MarkSatatements.

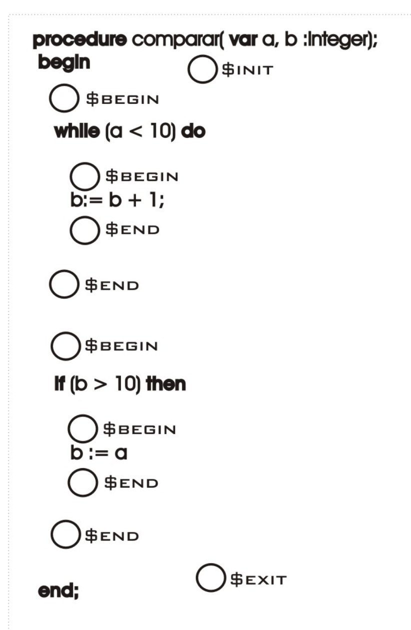

Figura 5.9: Grafo do Padrão MarkStatements.

Passo LinkStatement

Após o passo MarkStatements, o passo LinkStatement percorre a árvore com o intuito de determinar os nós iniciais e finais de cada lista de comandos a partir dos nós iniciais e finais de cada comando individual. Como pode ser visto na Figura 5.10, esse passo é composto pelos padrões Statement (linha 3) e List (linha 13).

As linhas 9,10,22 e 23 dos dois padrões ligam os nós `$begin` e `$end` com os `$begin:s` e `$end:s` dos padrões encontrados, ou seja, este passo determina o nó final e inicial de cada lista de comandos. As linhas 19 e 20 são responsáveis pela topologia do grafo, cria uma

```
 1 step LinkStatementList BT
 2
 3 pattern Statement
 4 var
 5 .       :s      as  [stmt]
 6 match
 7 .       [stmt_list< :s >]
 8 assignment
 9 .       assign $begin   to  $begin:s
10 .       assign $end     to  $end:s
11 end pattern
12
13 pattern List
14 var
15 .       :s      as  [stmt]
16 .       :ss     as  [stmt_list]
17 match
18 .       [stmt_list< :ss ; :s >]
19 graph
20 .       $end:ss ->  $begin:s
21 assignment
22 .       assign $begin   to  $begin:ss
23 .       assign $end     to  $end:s
24 end pattern
```

Figura 5.10: Padrão LinkStatementList.

aresta que parte do último nó da lista de comandos e vai até o primeiro nó do próximo comando. Na Figura 5.11 é mostrado como é construido o grafo neste passo.

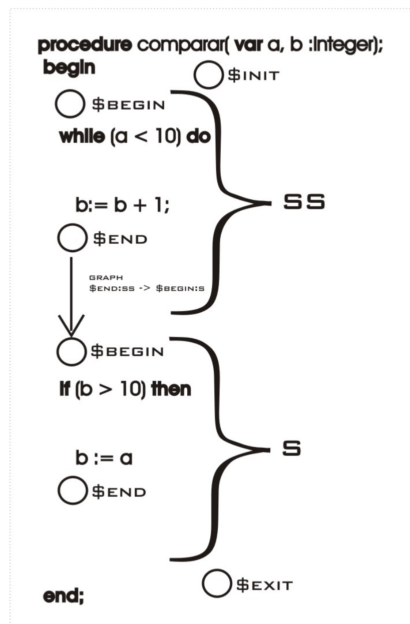

Figura 5.11: Grafo do Padrão LinkStatement.

Passo JoinStatement

O passo JoinStatement (Figura 5.12) possui apenas o padrão Join (Linha 3), que cria duas arestas no grafo, uma partindo do nó inicial do comando e chegando no nó inicial da

lista de comandos, e uma partindo do nó final da lista de comandos e chegando no nó final do comando como mostra a Figura 5.13. Dessa forma, quando uma lista de comandos for encontrada na árvore sintática (Linhas 6 e 7), ela será conectada ao grafo.

```
 1 step JoinStatement
 2
 3 pattern Join
 4 var
 5 .        :ss     as  [stmt_list]
 6 match
 7 .        [stmt< begin :ss end >]
 8 graph
 9 .        $begin  ->  $begin:ss
10 .        $end:ss ->  $end
11 end pattern
```

Figura 5.12: Padrão JoinStantemet.

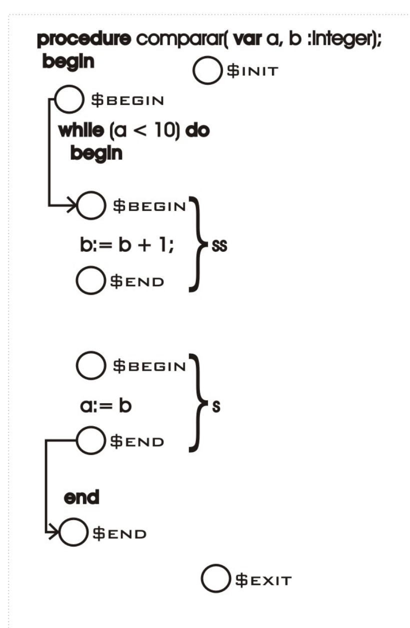

Figura 5.13: Grafo do Passo JoinStatement.

Passo JoinToFunction

O passo JoinToFunction mostrado na Figura 5.14 utiliza apenas o padrão Function1 (Linha 3) que conecta o nó inicial da unidade (`$init`) com o primeiro nó da lista *statement* (`$begin:ss`) (Linhas 13 - 15), e também, o último nó da lista *statement* (`$end:ss`) com o nó final da função (`$exit`) exemplificado na Figura 5.15. A seção de instrumentação (Linhas 16 a 18) marca na tabela de implementação dos nós `$init` e `$exit` que serão realizadas duas alterações no código associado a esses nós, uma logo antes do nó `$init` e outra logo depois do nó `$exit` mostrado na Figura 5.16, afim de adicionar pontas de

provas para que quando executado seja possível identificar onde e quais comandos foram executados.

```
 1 step JoinToFunction
 2
 3 pattern Function1
 4 var
 5         :head    as  [name_impl]
 6         :name    as  [func_or_proc_or_method_name]
 7         :type    as  [proc_or_func_fptype]
 8         :pars    as  [fp_list]
 9         :decl    as  [impl_decl_sect_list]
10         :ss      as  [stmt_list]
11 match
12         [main_impl< :head :name :pars :type ; :decl begin :ss end ; >]
13 graph
14         $init    ->  $begin:ss
15         $end:ss  ->  $exit
16 instrument
17         add init        $init   before  self
18         add exit        $exit   after   self
19 end pattern
```

Figura 5.14: Padrão JoinToFunction.

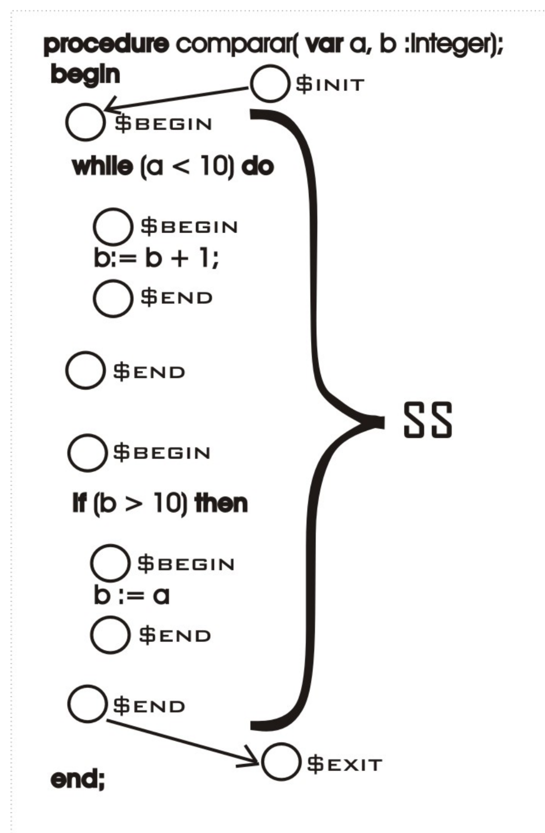

Figura 5.15: Grafo do Passo JoinToFunction.

```
procedure comparar( var a,b:integer);
 begin
   var tracefile : textfile;

     function check ( n : integer ) : boolean;
       begin
         writeln ( tracefile , n ) ;
         check := true ;
       end ;

     function checkfor ( n : integer ) : integer ;
       begin
          writeln ( tracefile , n ) ;
          checkfor := 0 ;
       end ;
     begin
       assignfile ( tracefile ,'comparar.trace.tc');
     if fileexists ('delphi.trace.tc') then
       reset (tracefile)
       else
        rewrite ( tracefile ) ;

     append ( tracefile ) ;
    begin
     try
     while (a < 10) do
       b:= b + 1;
    if (b > 10) then
       b := a
      writeln ( tracefile , '----------------' ) ;
      finally closefile ( tracefile )
     end
 end;
```


Figura 5.16: Instrumentaçao do Passo JoinToFunction.

Passo MakeGraph

O passo MakeGraph é responsável pela instrumentação das principais construções da linguagem Delphi, como comandos de repetição e comandos condicionais. Por ser um passo extenso, apenas os principais padrões serão detalhados neste trabalho. Todo o passo pode ser encontrado na íntegra no Apêndice C.

Comando de Repetição While

Pode-se compreender o padrão while observando a Figura 5.17. Na Linha 1, que é uma definição do nome, que neste caso, define o nome da regra como "While". Isto tem somente a finalidade de documentar, e não tem nenhum impacto sobre a semântica da regra. Da Linha 2 até a 4 é a seção da declaração de meta-variáveis, que são atribuídas de acordo com a gramática do Delphi (Apêndice A). As Linhas 5 e 6 fazem parte da seção da "definição de padrão", as sub-árvores da árvore sintática devem combinar para que esta regra de instrumentação seja aplicada, conforme as meta-variáveis. Neste padrão, as meta-variáveis :e e :s são unificadas à expressão de controle e ao corpo do while respectivamente.

A Linha 7 é a seção de declaração do nó. Neste exemplo, ele (o padrão) cria um novo nó, adiciona-o ao grafo e atribui ao nome simbólico `$control` no mapa de nó do grafo

correspondente à sub-árvore que combina com o padrão.

Todo comando tem dois nomes simbólicos: `$begin` e `$end` criados no passo MarkStatements. Estes são, respectivamente, os nós antes e depois do comando. As Linhas 8 a 12 fazem parte da seção *graph*, no qual declara as arestas no grafo. A Linha 9 cria a aresta que liga o nó `$begin` ao o nó do grafo `$control`. A Linha 10, cria a aresta que liga o nó `$control` ao o nó `$begin` unificada com a meta-variável :s. A Linha 14 atribui o nome simbólico `$raise:s` ao nó do grafo atribuído com o nome `$exit`, isto significa que quando o instrumentador está analisando os comandos em :s, uma referência ao nó "`$raise`" é feita ao nó após o while. Na Linha 17, atribui-se o nome simbólico `$pusage:e` ao nó do grafo atribuído a `$control`. Conseqüentemente, toda a regra da instrumentação aplicada a :e pode referir a `$pusage`. As Linhas 17 a 20 são a seção da atribuição e inserções de pontas de provas. Nesta seção, a regra da instrumentação adiciona pontas de provas (comandos), a fim de indicar onde e o que foi exercitado por um determinado caso de teste.

```
 1 pattern While
 2 var
 3         :e       as  [expr]
 4         :s       as  [stmt]
 5 match
 6         [stmt< while :e do :s >]
 7 declare node $control
 8 graph
 9         $begin          ->  $control
10         $control        ->  $begin:s
11         $end:s          ->  $control
12         $control        ->  $end
13 assignment
14         assign $raise:s          to  $exit
15         assign $break:s          to  $end
16         assign $definition:e     to  $control
17         assign $pusage:e         to  $control
18 instrument
19         add checkpoint  $begin          before  self
20         add checkpoint  $begin:s        before  :s
21         add checkpoint  $end            after   self
22         add checkpoint  $control        before  :e
23 end pattern
```

Figura 5.17: Padrão While.

O grafo gerado pelas Linhas 9 a 12 da Figura 5.17 é mostrado na Figura 5.18. Um nó `$control` é declarado na linha 7 para controlar o fluxo do comando while, ele receberá as variáveis de definição e de uso predicativo, as declarações das variáveis são denominadas como variáveis de definição, as variáveis que são utilizadas para testes lógicos em linguagem de programação são denominadas variáveis de uso predicativo.

As linhas 19 a 22 do padrão While mostrada pela Figura 5.17 fazem alterações no código do programa. Como mostrada na Figura 5.19, cada linha identificada na figura é responsável por uma modificação, com estas modificações que inserem pontas de provas

será possível após a execução do programa gerar um arquivo trace para verificação do fluxo do programa.

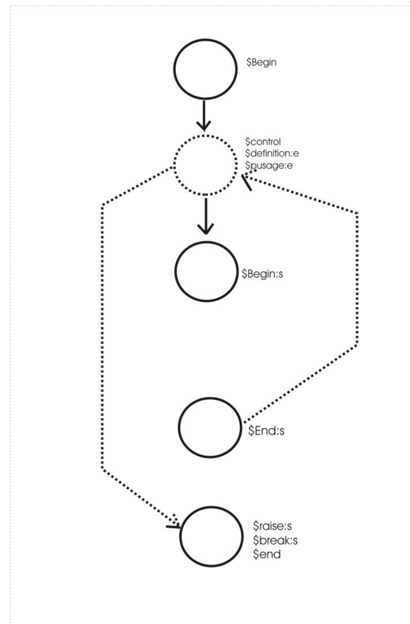

Figura 5.18: Grafo Padrão While.

```
begin
  writeln ( tracefile , 1 ) ;   } linha 19
   begin
     while check(2) and ((a < 10)) do   } linha 22
      begin
       writeln ( tracefile , 3 ) ;   } linha 20
       b:= b + 1;
      end ;
     writeln ( tracefile , 4 )   } linha 21
   end
end ;
```

Figura 5.19: Instrumentação do padrão While.

A seguir são apresentados os padrões **Padrão Try2**

Um outro padrão é o Try2 mostrado pela Figura 5.20 , nas Linhas 3 e 4 as metas variáveis são atribuidas como uma lista de comando. Assim quando é encontrado o padrão da Linha 6, o grafo é modificado, devido às Linhas 8, 9 e 10 como mostrada na Figura 5.21. Após isso, na Linha 12 atribui o nó `$raise:s1` ao nó `$begin:s2`.

```
1  pattern Try2
2  var
3          :s1     as  [stmt_list]
4          :s2     as  [stmt_list]
5  match
6          [stmt< try :s1 finally :s2 end >]
7  graph
8          $begin          ->  $begin:s1
9          $end:s1         ->  $begin:s2
10         $end:s2         ->  $end
11 assignment
12         assign $raise:s1       to  $begin:s2
13 end pattern
14
```

Figura 5.20: Padrão Try2.

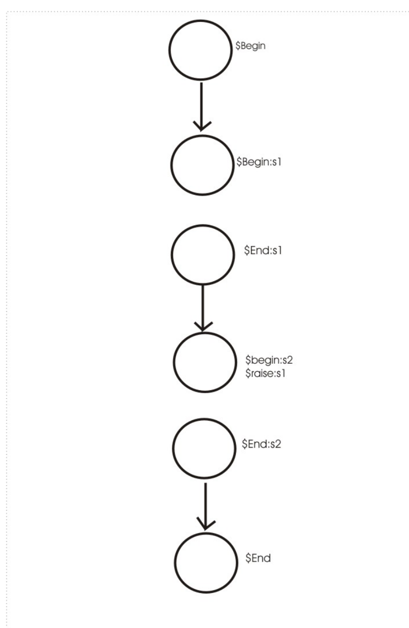

Figura 5.21: Grafo Padrão Try2.

Padrão if-then

O padrão if-then mostrado na Figura 5.22 encontra a produção do comando if-then pela expressão da linha 6. As variáveis ":e" e ":s" que são as expressões e os comandos de cada if-then respectivamente, estão declaradas nas linhas 3 e 4. As linhas 9, 10 e 11 são responsáveis pela construção do grafo mostrado na Figura 5.23. As linhas 14 e 15 são atribuidos as definições das variáveis e uso predicativo junto ao nó `$begin`. As linhas 17 a 20 fazem alteração no código como mostrado na Figura 5.24.

Padrão if-then-else

Neste padrão, observado na Figura 5.25 as Linhas 3 a 5 atribuem as metavariáveis de acordo com a gramática do Delphi. A Linha 7 é responsavel pelo padrão procurado quando percorrido na unidade. As Linhas de 9 a 12 fazem parte da construção dos nós do grafo

```
1  pattern IfThen
2  var
3        :e      as  [expr]
4        :s      as  [stmt]
5  match
6        [stmt< if :e then :s >]
7  declare node $foo
8  graph
9        $begin  ->  $end
10       $begin  ->  $begin:s
11       $end:s  ->  $end
12 assignment
13       assign $raise:s          to  $exit
14       assign $definition:e     to  $begin
15       assign $pusage:e         to  $begin
16 instrument
17       add checkpoint  $begin        before  :e
18       add checkpoint  $begin        before  self
19       add checkpoint  $begin:s      before  :s
20       add checkpoint  $end          after   self
21 end pattern
```

Figura 5.22: Padrão if-then.

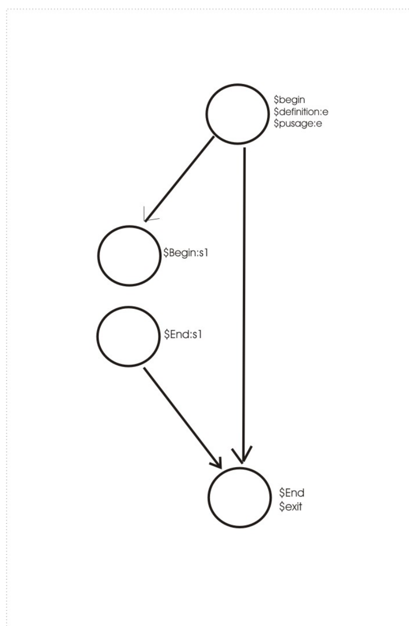

Figura 5.23: Grafo Padrão if-then.

um exemplo mostra a Figura 5.26. As Linhas 19 a 23 são instruções de intrumentação no código fonte do Delphi, ou seja, ela indica como inserir pontas de provas no código mostrado pela Figura 5.27.

Padrão RepeatUntil

O padrão RepeatUntil mostrado pela Figura 5.28 encontra o comando "repeat" pela produção da Linha 6. As Linhas 9 a 12 são responsáveis pela produção do grafo mostrada na Figura 5.29. A Linha 9 cria uma aresta que parte do nó `$begin` ao nó `$begin:s`, outra aresta é criada do nó `$end:ss` ao nó `$control` pela Linha 10, na linha 11 uma aresta

```
begin
 writeln ( tracefile , 1 ) ; } linha 18
      if check ( 2 ) and ( ( b > 10 ) ) then } linha 17
        begin
          writeln ( tracefile , 3 ) ; } linha 19
          b:= a
        end;
      writeln ( tracefile , 4 ) } linha 20
end;
```

Figura 5.24: Instrumentação do padrão If-Then .

```
1 pattern IfThenElse
2 var
3         :e      as  [expr]
4         :s1     as  [stmt]
5         :s2     as  [stmt]
6 match
7         [stmt< if :e then :s1 else :s2 >]
8 graph
9         $begin  ->  $begin:s1
10        $begin  ->  $begin:s2
11        $end:s1 ->  $end
12        $end:s2 ->  $end
13 assignment
14        assign $raise:s1        to  $exit
15        assign $raise:s2        to  $exit
16        assign $definition:e    to  $begin
17        assign $pusage:e        to  $begin
18 instrument
19        add checkpoint  $begin          before  :e
20        add checkpoint  $begin          before  self
21        add checkpoint  $begin:s1       before  :s1
22        add checkpoint  $begin:s2       before  :s2
23        add checkpoint  $end            after   self
24 end pattern
```

Figura 5.25: Passo IfThenElse.

parte do nó `$control` e chega ao nó `$begin:ss` e por fim mais uma aresta parte do nó `$control` para o nó `$end`.

Na seção *instrument* iniciada pela Linha 19 da Figura 5.28, são inseridas pontas de provas no código, de maneira que é possível saber quais comandos foram executados. A Figura 5.30(a) é revelado o código sem instrumentar e na Figura 5.30(b) é apresentado o programa instrumentado.

Padrão ForTo

O padrão ForTo é utilizado quando o instrumentador encontra um comando “for” no código. Esse comando é encontrado pela escrita da Linha 9 da seção *match* da Figura 5.31, para a produção dessa linha foram usadas as variáveis das Linhas 3 a 7 que são atribuidas pelos padrões da gramática do Delphi, na Figura 5.33 é mostrado que cada variável recebe como valor. As Linhas 10 a 12 declaram nós que serão utilizados na construção do grafo

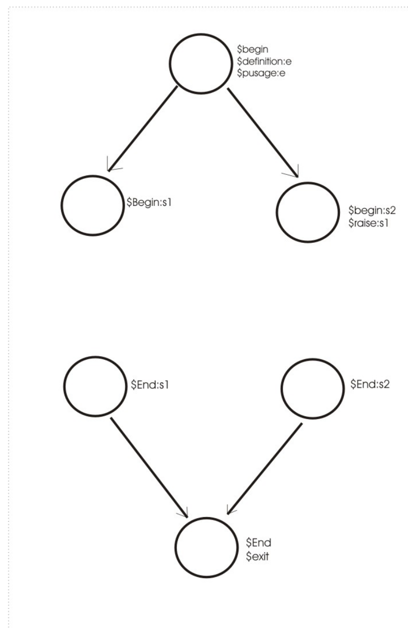

Figura 5.26: Grafo Padrão if-then-else.

```
begin
 writeln ( tracefile , 1 ) ; } linha 20
      if check ( 2 ) and ( ( b > 10 ) ) then } linha 19
        begin
          writeln ( tracefile , 3 ) ;} linha 21
          b:= a
        end else
        begin
          writeln ( tracefile , 4 ) ;} linha 22
          a := a + b
        end ;
        writeln ( tracefile , 5 )} linha 23
end;
```

Figura 5.27: Instrumentação do padrão If-Then-else .

e na instrumentação.

As Linhas 14 a 19 ligam os nós através de arestas a Figura 5.32 mostra a construção desta seção. As Linhas 21 a 27 atribuem os valores como definição de variáveis, uso predicativo de variáveis e uso computacional de variáveis, afim de atender diversos tipos de critérios de teste. Um deles é o critério de teste Todos-Usos, tem como objetivo que todas as associações entre uma definição de variável e seu uso predicativo e uso computacional sejam exercitadas pelos casos de teste, através de pelo menos um caminho livre de definição, ou seja, um caminho cujo a variável não é redefinida.

```
 1 pattern RepeatUntil
 2 var
 3         :e      as  [expr]
 4         :ss     as  [stmt_list]
 5 match
 6         [stmt< repeat :ss until :e >]
 7 declare node $control
 8 graph
 9         $begin           ->  $begin:ss
10         $end:ss          ->  $control
11         $control         ->  $begin:ss
12         $control         ->  $end
13 assignment
14         assign $raise:ss        to  $exit
15         assign $break:ss        to  $end
16         assign $definition:e    to  $control
17         assign $pusage:e        to  $control
18 instrument
19         add checkpoint  $begin          before  self
20         add checkpoint  $begin:ss       before  :ss
21         add checkpoint  $end            after   self
22         add checkpoint  $control        before  :e
23 end pattern
```

Figura 5.28: Padrão RepeatUntil.

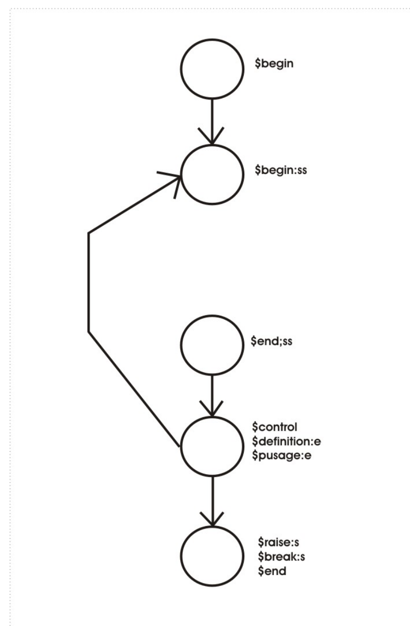

Figura 5.29: Grafo do Padrão RepeatUntil.

As Linhas 29 a 32 integram a seção de instrumentação. Estes comandos fazem modificação no código inserindo pontas de provas, a Figura 5.33(a) é um exemplo desta instrumentação, apresenta cada linha desta seção, comparando com a Figura 5.33(b) é visível a alteração que a seção de instrumentação faz com o código.

```
repeat
 begin
  calcula ( valor )
 end
until ( i < 10 ) ;
```

(a)

```
begin
 writeln ( tracefile , 1 ) ;    } linha 19
 begin
  repeat
   begin
    writeln ( tracefile , 2 ) ;    } linha 20
    begin
     calcula ( valor )
    end
   end
  until check ( 3 ) and ( ( i < 10 ) ) ;    } linha 22
 writeln ( tracefile , 4 )    } linha 21
 end
end;
```

(b)

Figura 5.30: Código não instrumentado (a) código instrumentado (b)

```
1  pattern ForTo
2  var
3          :v      as  [variable_reference]
4          :at     as  [attrib_sign]
5          :einit  as  [expr]
6          :e      as  [expr]
7          :s      as  [stmt]
8  match
9          [stmt< for :v :at :einit to :e do :s >]
10 declare node $control
11 declare node $initialization
12 declare node $increment
13 graph
14         $begin            ->  $initialization
15         $initialization   ->  $control
16         $control          ->  $begin:s
17         $control          ->  $end
18         $end:s            ->  $increment
19         $increment        ->  $control
20 assignment
21         assign $raise:s            to  $exit
22         assign $definition:v       to  $initialization
23         assign $cusage:einit       to  $initialization
24         assign $cusage:e           to  $increment
25         assign $definition:v       to  $increment
26         assign $pusage:v           to  $control
27         assign $break:s            to  $end
28 instrument
29         add checkpointfor       $control        before  :e
30         add checkpoint  $begin          before  self
31         add checkpoint  $begin:s        before  :s
32         add checkpoint  $end            after   self
33 end pattern
```

Figura 5.31: Padrão ForTo.

5.2.2 Seção de Implementação

A implementação consiste em adicionar pontas de provas para as declarações *instrument* localizadas na seção de processamento das unidades. Essa seção é composta por

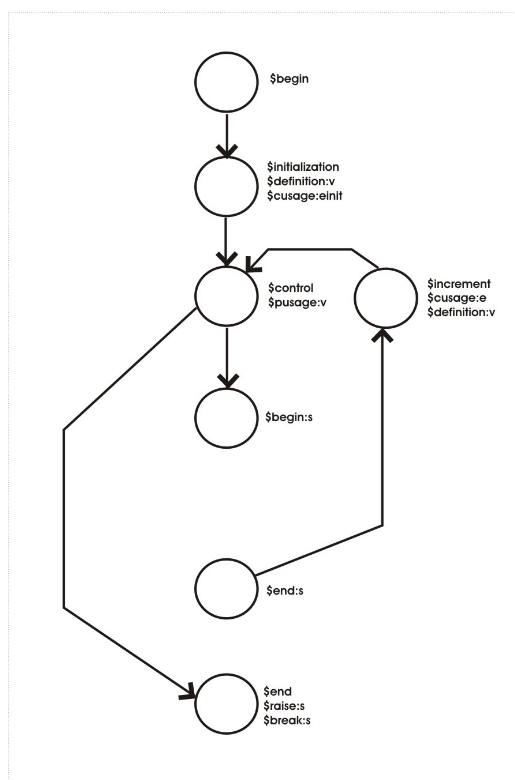

Figura 5.32: Grafo do Padrão ForTo.

uma lista de declarações **implement** que contém as ações que devem ser tomadas para alterar o programa.

Por exemplo, a implementação da declaração `instrument` presente na Linha 19 da Figura 5.17 está sendo utilizada para alterar o programa de forma que inclua um *checkpoint* antes de uma lista de comandos. Isso é implementado na seção de **implement**, Conforme mostrada na Figura 5.34. Esta implementação escreve no programa instrumentado o que é descrito na Linha 9. A meta-variável :n representa o número do nó controlado pela IDeL. O código alterado insere o comando "writeln" para escrever no arquivo "TraceFile" o número do nó. Quando da execução do programa instrumentado, o arquivo será gerado com os números dos nós criados na execução.

5.3 Execução do Instrumentador

Como exemplo de uma instrumentação foi usado um programa simples criado propositalmente para testar as regras descritas no arquivo "delphi.idel". Na Figura 5.35 é representado o teste da função "exittest" do arquivo "funcoes.pas". Depois de gerado o instrumentador com os arquivos "delphi.l" e "delphi.y" utiliza-se as regras descritas do

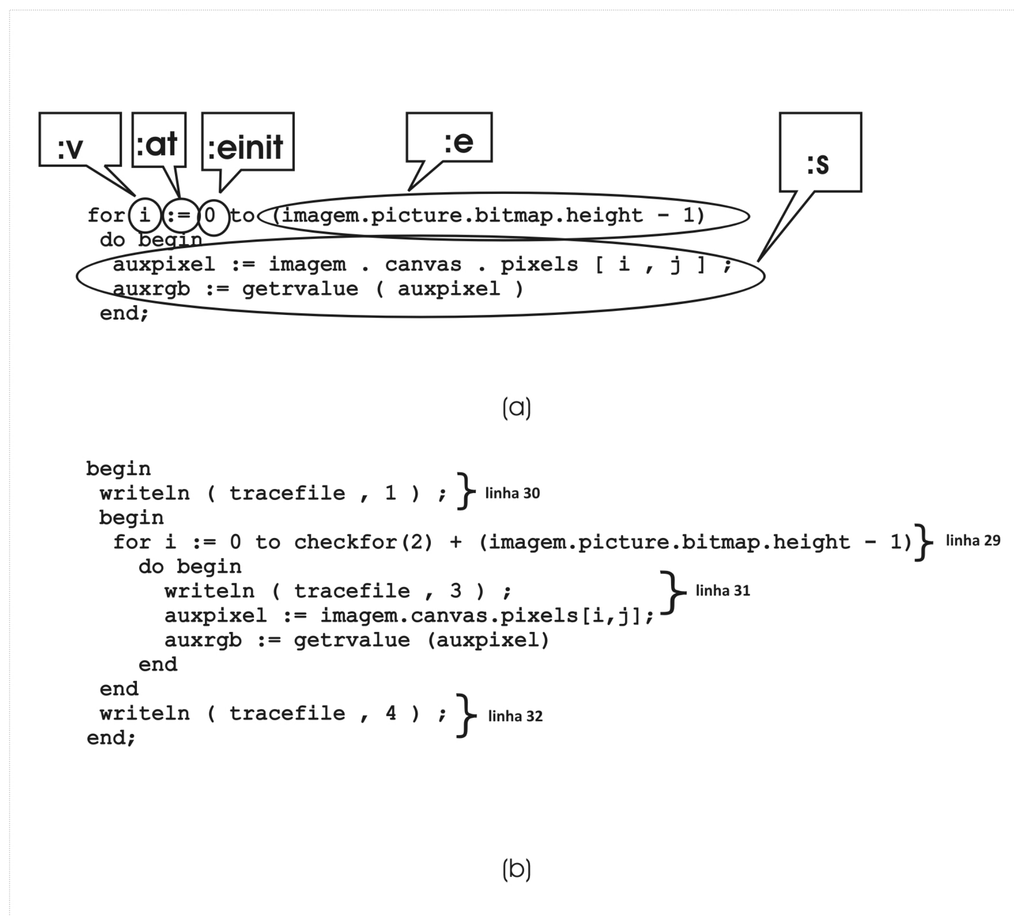

Figura 5.33: Código não instrumentado (a) código instrumentado (b).

```
 1 implement
 2 var
 3 .       :s.      as  [stmt_list]
 4 .       :n.      as  [constant]
 5 checkpoint $node before
 6 .       [stmt_list< :s >]
 7 binding :n to node $node
 8 as
 9 .       [stmt< begin writeln(TraceFile , :n ); begin :s end end >]
10 end implement
```

Figura 5.34: Implementação do checkopoint before.

arquivo "delphi.idel" sobre o programa.

A execução é bem simples, no comando utilizado para instrumentar o arquivo "funcoes.pas", a opção -p indica qual será o arquivo a ser instrumentado. A opção -i aponta a descrição IDeL do instrumentador. A opção -g define o nome base que será utilizado nos arquivos DOT gerados e a opção -o informa o nome do arquivo final instrumentado, esta

```
function exittest(a : integer; b : integer) : integer;
begin
        while a >= 1 do
         begin
                if b >= 0 then
                        exit
                else
                        a := a + b
         end;
        if b > 0 then
                b := b + 1
        else
                b := b - 1;

        exittest := 0
end;
```

Figura 5.35: Programa exemplo função exittest.

execuçao é exemplificada pela Figura 5.36.

```
 1 giovanni@notbook:~/delphi$ ./idel.delphi -p funcoes.pas -i delphi.idel -g funcoes -o funcoes.inst.pas
 2 Main-tg.cpp
 3 Options::getOption() OK
 4 processing source...
 5 0x40237180
 6 0x40237180
 7 processing instrumenter...
 8 running ...
 9 generating funcoes.power.dot
10 generating funcoes.exittest.dot
11 generating funcoes.whiletest.dot
12 generating funcoes.raisetest.dot
13 generating funcoes.repeatuntiltest.dot
14 generating funcoes.fortotest.dot
15 generating funcoes.fordowntotest.dot
16 generating funcoes.foobar.dot
17 generating funcoes.casetest.dot
18 generating funcoes.trytest.dot
19 generating funcoes.finallytest.dot
20 generating funcoes.TForm1.Button1Click.dot
21 done.
```

Figura 5.36: Execução da IDeL.

Como resultado, obtém-se a função instrumentada no arquivo funcoes.inst.pas mostrada na Figura 5.37.

O grafo da Figura 5.38 refere-se ao programa da Figura 5.35, o arquivo que é responsável pela geração do grafo é o da Figura 5.39

5.4 Considerações Finais

Neste capítulo foram apresentados os principais passos da ferramenta para instrumentação e seus principais padrões. Estes padrões foram adicionados conforme a necessidade da Linguagem Delphi, adaptando-se à instrumentação existente para Linguagem C. No entanto, existe grande diferença na sintaxe da linguagem C para linguagem Delphi, considerando que o Delphi é uma linguagem orientada a objetos. É válido dizer que tanto

```
function exittest ( a : integer ; b : integer ) : integer ;
var
 TraceFile : TextFile ;
 function check ( n : integer ) : boolean ;
begin
 writeln ( TraceFile , n ) ;
 check := true ;
end ;
function checkfor ( n : integer ) : integer ;
begin
 writeln ( tracefile , n ) ;
 checkfor := 0 ;
end ;
begin
 assignfile ( tracefile , 'exittest.trace.tc' ) ;
 if fileexists ( 'exittest.trace.tc' ) then reset ( tracefile )
else
 rewrite ( tracefile ) ;
 append ( tracefile ) ;
begin
try
begin
 writeln ( TraceFile , 1 ) ;
begin
 while check ( 2 ) and ( a >= 1 ) do
begin
 writeln ( TraceFile , 3 ) ;
begin
begin
 writeln ( TraceFile , 3 ) ;
begin
if check ( 3 ) and ( b >= 0 ) then
begin
 writeln ( TraceFile , 4 ) ;
 exit
end
else
begin
 writeln ( TraceFile , 5 ) ;
 a := a + b
end ;
 writeln ( TraceFile , 5 )
end
end ;
begin
 writeln ( TraceFile , 6 ) ;
begin
 if check ( 6 ) and ( b > 0 ) then
begin
 writeln ( TraceFile , 7 ) ;
 b := b + 1
end
else
begin
 writeln ( TraceFile , 8 ) ;
 b := b - 1
end ;
 writeln ( TraceFile , 9 )
end
end ;
 exittest := 0 ;
 finally
 close ( TraceFile )
 end
end
end ;
```

Figura 5.37: Função exittest instrumentada

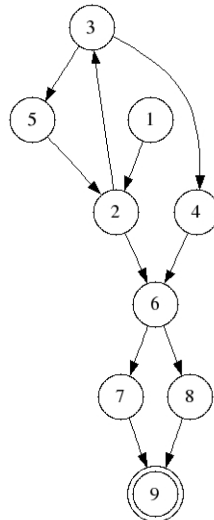

Figura 5.38: Grafo da função exittest.

a gramática, como a descrição da IDeL foram complexas em sua confecção. Este capítulo é importante para o entendimento deste trabalho, porque explica como é instrumentada a Linguagem Delphi, ou seja, como são usados as pontas de provas que o instrumentador insere em lugares estratégicos para o entendimento da execução do programa. Por exemplo, para saber se a expressão do comando "for" foi verdadeira, é necessário adicionar uma função chamada checkfor (Figura 5.33(b)), que somada com a expressão já existente não causará nenhum efeito além do que a própria expressão original causa.

```
 1 digraph gfc {
 2 node [shape = doublecircle] 9;
 3 node [shape = circle] 1;
 4 node [shape = circle] 5;
 5 node [shape = circle] 3;
 6 /* pusage of b at 3 */
 7 node [shape = circle] 4;
 8 node [shape = circle] 6;
 9 /* pusage of b at 6 */
10 node [shape = circle] 7;
11 node [shape = circle] 8;
12 node [shape = circle] 2;
13 /* pusage of a at 2 */
14 /* cusage of a at 2 */
15 /* cusage of b at 2 */
16 1 -> 2;
17 2 -> 3;
18 5 -> 2;
19 2 -> 6;
20 3 -> 4;
21 3 -> 5;
22 4 -> 6;
23 6 -> 7;
24 6 -> 8;
25 7 -> 9;
26 8 -> 9;
27  }
```

Figura 5.39: Arquivo DOT da função exittest.

A confecção da descrição da IDeL foi complexa devido à variedade de comandos que a linguagem Delphi contém. Neste capítulo foram exemplificadas algumas delas, e o arquivo na íntegra está no Apêndice C. No próximo capítulo será apresentado um estudo de casos com o instrumentador de Delphi nele, são feitas algumas medições relevantes sobre o tamanho do arquivo e o tempo de execução do programa instrumentado e do programa original sem instrumentação.

6 Estudo de Caso

Neste capítulo, são exemplificados casos de instrumentação para Linguagem Delphi, comparando o tamanho do arquivo compilado instrumentado com o tamanho do arquivo compilado não instrumentado, também aborda sobre o tempo de execução. Estes testes foram feitos com um software cujo seu código em Delphi é consideravelmente grande para a verificação do instrumentador, entre estes casos de testes são exemplificados o processo de instrumentação.

6.1 Descrição do Programa

Foi utilizado como estudo de caso, um software desenvolvido na linguagem de programação Delphi, que foi tese de doutorado, defendido pela Dra. Fátima Marques, na USP- São Carlos, cujo o tema é “Realce de Contraste de Mamas Densas Para Detecção de Clusters de microcalcificação mamária” (NUNES, 2001).

Este software possibilita a identificação precoce de microcalcificação mamária em imagens de baixo contraste, não detectadas em processamento convencional antes do realce de contraste.

O software usa técnicas para realce de contraste neste programa que são:

- Filtro Butterworth
- Equalização de histograma.
- Função logarítmica.
- Função de potenciação.
- Filtro passa-alta.
- Correção gama.

Todas estas técnicas são para realce de contraste. A técnica de filtro Butterworth que aplicada em uma imagem como a Figura 6.1(a) resulta na Figura 6.1(b) (NUNES, 2001).

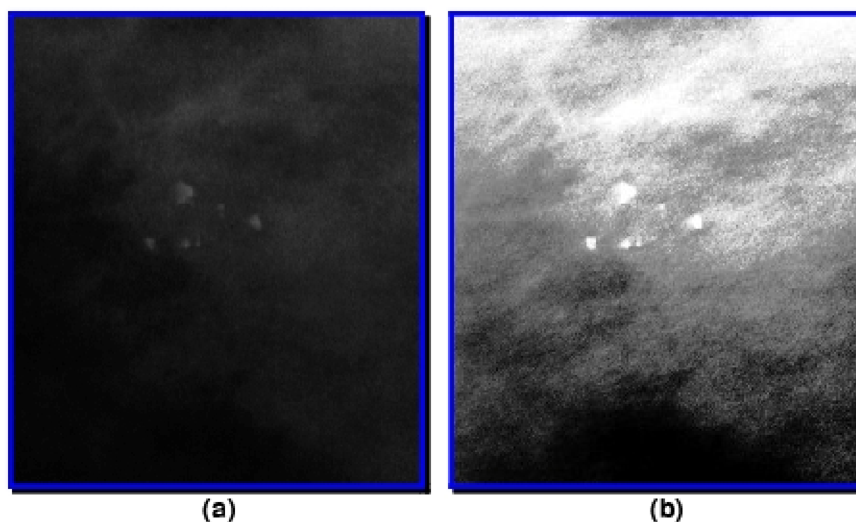

Figura 6.1: Exemplo de realce de contraste com o filtro Butterworth (NUNES, 2001)

Outra técnica utilizada é a de equalização de histograma. Ela tem a finalidade de obter um histograma uniforme, através do espalhamento da distribuição dos níveis de cinza ao longo de toda a escala de resolução de contraste. Na Figura 6.2 é exemplificada esta técnica (NUNES, 2001).

A linguagem de programação Delphi foi utilizada por Nunes (2001) em sua tese, o qual é propositalmente utilizado neste trabalho por revelar extensos códigos de programação.

Nas próximas seções serão feitas algumas análises considerando-se características do código instrumentado em comparação ao programa original.

6.2 Tamanho do Programa

Programas feitos em Delphi são divididos em unidades, cada unidade geralmente representada por uma *unit* e um arquivo com extensão ".pas". Uma unidade em Delphi pode conter apenas funções e procedimentos para dar suporte ao desenvolvimento do software.

A maior unidade do programa de realce de contraste de mamas densas para detecção de clusters de microcalcificação mamária é o arquivo *UImagem.pas*, ele contém 5388 linhas entre programação e comentários. Esta unidade é responsável por quase toda a

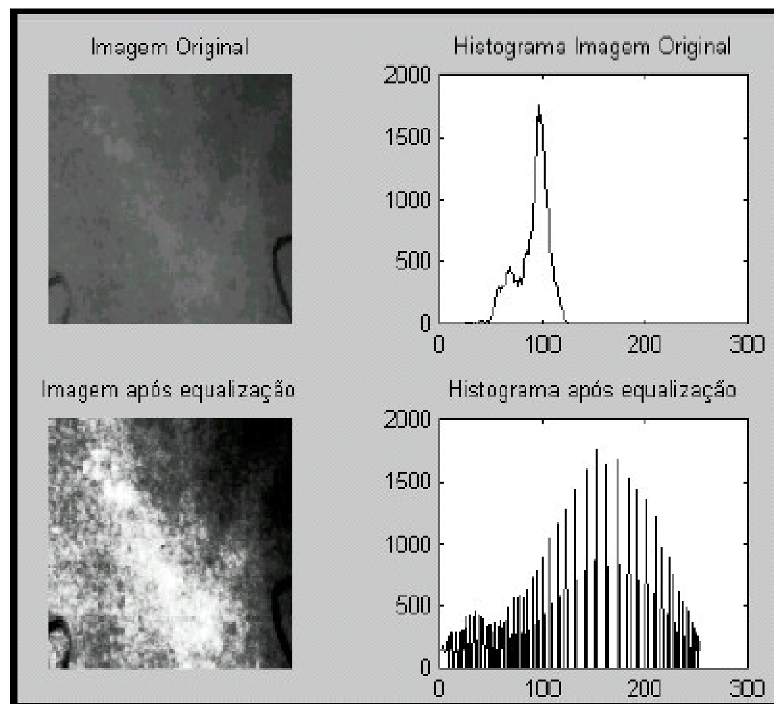

Figura 6.2: Exemplo de equalizaçao de Histograma (NUNES, 2001)

compilação realizada pelo software. Haja vista o software instrumentado não necessitar de uma boa legibilidade, ou seja, o desenvolvedor não terá que ler ou entender o código instrumentado, o instrumentador gera em apenas uma única linha o código instrumentado, impossibilitando assim a comparação de quantidades de linhas do instrumentado com o original.

Comparando o executável do programa não instrumentado que tem 1.231.360kb, ou seja 1.2mb, com o programa instrumentado que tem 1.320.448kb ou 1.3mb, nota-se que o programa instrumentado aumentou em 89.088kb ou seja, teve um aumento de apenas 7,2%. Estes 7,2% não significa que o resultado de tempo de compilação ou de execução sejam parecidos percentualmente.

Exemplo da execução de instrumentação da UImagem.pas:

```
giovanni@notbook:~/delphi$ ./idel.delphi -p UImagem.pas -i delphi.idel -g teste -o teste.instru.pas
Main-tg.cpp
0x40233920
0x40233920
Options::getOption() OK
processing source...
processing instrumenter...
running ...
generating teste.tfimagem.salvaimagem.dot
generating teste.tfimagem.carregaimagem.dot
```

```
generating teste.tfimagem.formclose.dot
generating teste.tfimagem.formpaint.dot
generating teste.tfimagem.segmentao1click.dot
generating teste.tfimagem.transfapclick.dot
generating teste.tfimagem.clusterclick.dot
generating teste.tfimagem.crescimentorclick.dot
generating teste.tfimagem.redimensionarclick.dot
generating teste.tfimagem.transformacao_ap.dot
generating teste.tfimagem.maxvalorrgb.dot
generating teste.tfimagem.minvalorrgb.dot
generating teste.tfimagem.mostrarelogioinicio.dot
generating teste.tfimagem.terminarelogio.dot
generating teste.tfimagem.clusterizacao.dot
generating teste.tfimagem.min.dot
generating teste.tfimagem.max.dot
generating teste.tfimagem.redimens.dot
generating teste.tfimagem.imagemmousedown.dot
generating teste.tfimagem.imagemmousemove.dot
generating teste.tfimagem.imagemmouseup.dot
generating teste.tfimagem.filtragemmedianaclick.dot
generating teste.tfimagem.colar1click.dot
generating teste.tfimagem.recortar1click.dot
generating teste.tfimagem.copiar1click.dot
generating teste.tfimagem.crescimento.dot
generating teste.tfimagem.iniciapaleta.dot
generating teste.tfimagem.localizacluster.dot
generating teste.tfimagem.marcacluster.dot
generating teste.tfimagem.segmenta.dot
generating teste.tfimagem.iniciaimgaux.dot
generating teste.tfimagem.aplicar_boxrim.dot
generating teste.tfimagem.thresholdglobal.dot
generating teste.tfimagem.morfologiamatematicabinaria.dot
generating teste.tfimagem.thresholdlocal.dot
generating teste.thresholdpixel.dot
generating teste.tfimagem.histograma1click.dot
generating teste.tfimagem.regiesdeinteresseclick.dot
generating teste.tfimagem.regiaointeresse.dot
generating teste.tfimagem.formclick.dot
generating teste.tfimagem.histograma2.dot
generating teste.tfimagem.realce2.dot
generating teste.tfimagem.realceparcial.dot
generating teste.tfimagem.segmenta2.dot
generating teste.tfimagem.passaalta.dot
generating teste.tfimagem.aplicar_boxrim2.dot
generating teste.tfimagem.thresholdglobal2.dot
generating teste.tfimagem.morfologiamatematicabinaria2.dot
generating teste.tfimagem.thresholdlocal2.dot
generating teste.thresholdpixel2.dot
generating teste.tfimagem.pacoteimagens.dot
generating teste.tfimagem.pacotelocal.dot
generating teste.histogramalocal.dot
generating teste.tfimagem.realcelocal.dot
generating teste.tfimagem.realceatenuacaolocal.dot
generating teste.tfimagem.juntaimagens.dot
```

```
generating teste.tfimagem.clusterizacao2.dot
generating teste.tfimagem.transformacao_ap2.dot
generating teste.tfimagem.procurareg.dot
done.
```

6.3 Tempo de execução

Este estudo de caso mostra a execução do programa levando em consideração o tempo de execução do programa que está instrumentado comparado ao mesmo programa no estado original. Foram realizadas trinta execuções de teste em cada uma das três rotinas: realce de contraste pela curva característica, realce de contraste pelos coeficientes de atenuação e segmentação e realce de contraste pela modificação do histograma e segmentação. Para este estudo de caso, é muito importante saber com precisão o tempo de execução destas rotinas. Para isso foi necessário modificar o código-fonte do programa para que estas rotinas fossem executadas automaticamente. Um componente TTimer do Delphi foi utilizado para esta tarefa, manipulando as opções do programa de forma idêntica para cada um dos trinta casos de teste e gerando um arquivo com o tempo de execução de cada rotina. Na Figura 6.3 é mostrada a execução desta rotina.

O caso número 1 que processa as imagens para o realce de contraste pela curva característica conforme a Tabela 6.1 foram necesários 16,05 segundos para realçar o contraste de 55 figuras, com o software instrumentado foi preciso 20,72 segundos para este processo, ou seja, 4,67 segundos ou 29,09% a mais que o original.

No segundo caso, foi analisado a execução de uma rotina do sistema um pouco mais pesada, que é para realce de contraste pelos coeficientes de atenuação e segmentação exemplificado pela tela da execução pela Figura 6.4. Neste caso, foram necessários, em média, 77,96 segundos usando o programa original contra 152,12 segundos do programa instrumentado, ou seja, um aumento de 74,16 segundos a mais nesta execução. Na Tabela 6.1 são apresentados estes resultados.

E no terceiro caso de teste, o realce de contraste pela modificação do histograma e segmentação, é mais processado que o segundo, ele leva 89,05 segundos, ou seja 1 minuto e 29 segundos para ser realizado pelo programa original contra 155,92 segundos (2 minutos e 36 segundos) pelo programa instrumentado, este aumento é significativo devido a sua maior atividade de processamento. Mostrada na Figura 6.5. Na Tabela 6.3 são mostrados estes resultados.

Contudo, estes três casos de testes realizados, executaram as seguintes rotinas da

Tabela 6.1: Tempo de execução de Realce de Contraste pela Curva Característica

Número	**Tempo não instrumentado**	**Tempo instrumentado**
01	16,20	20,77
02	16,00	20,77
03	16,01	20,72
04	16,06	22,71
05	15,95	21,42
06	16,06	22,77
07	16,07	20,27
08	15,94	20,57
09	16,11	20,24
10	15,96	20,21
11	16,03	20,27
12	16,03	20,57
13	16,08	20,24
14	16,00	20,21
15	16,07	17,43
16	16,02	20,19
17	16,12	20,24
18	16,09	20,17
19	16,39	20,24
20	16,05	21,81
21	15,96	20,59
22	15,94	22,81
23	16,11	21,43
24	16,01	22,77
25	16,04	20,12
26	15,94	21,10
27	16,01	20,14
28	16,01	20,25
29	16,07	20,24
30	16,05	20,21
Média	**16,05 segundos**	**20,72 segundos**
Desv.Pad.	**0,09 segundos**	**1,08 segundos**

unidade de imagem gerando o arquivo trace com a extensão tc para cada arquivo:

- histograma
- carregaimagem
- pacotelocal
- formclick
- formpaint
- histograma2
- maxcalorrgb
- minvalorrgb
- pacoteimagens
- procurareg
- realce2
- realceparcial
- salvaimagem

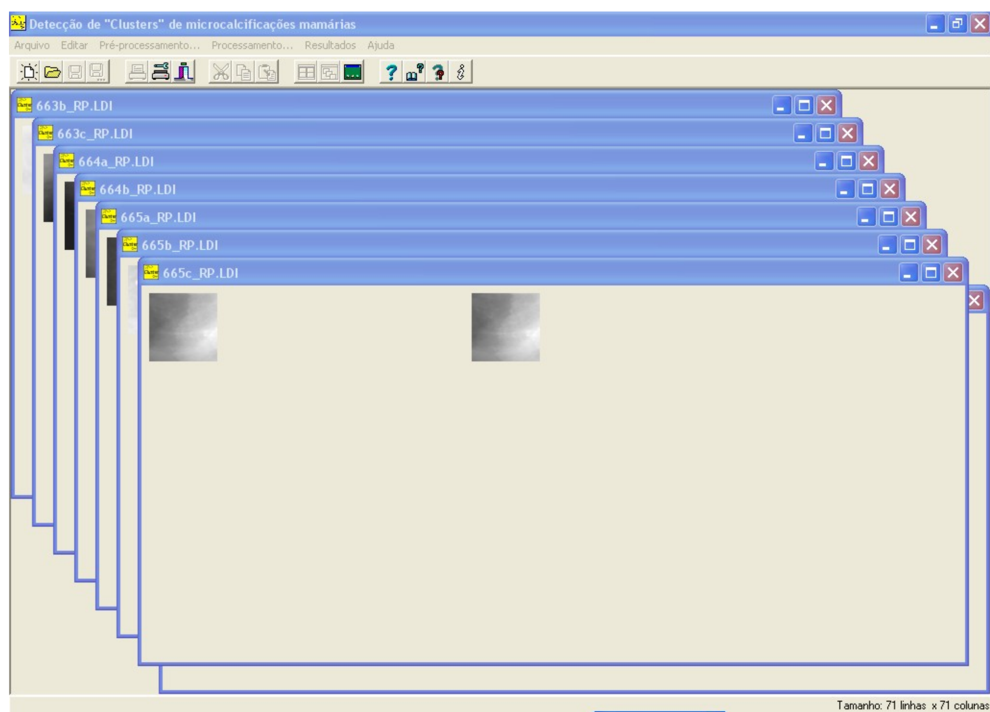

Figura 6.3: Realce de contraste pela Curva Característica

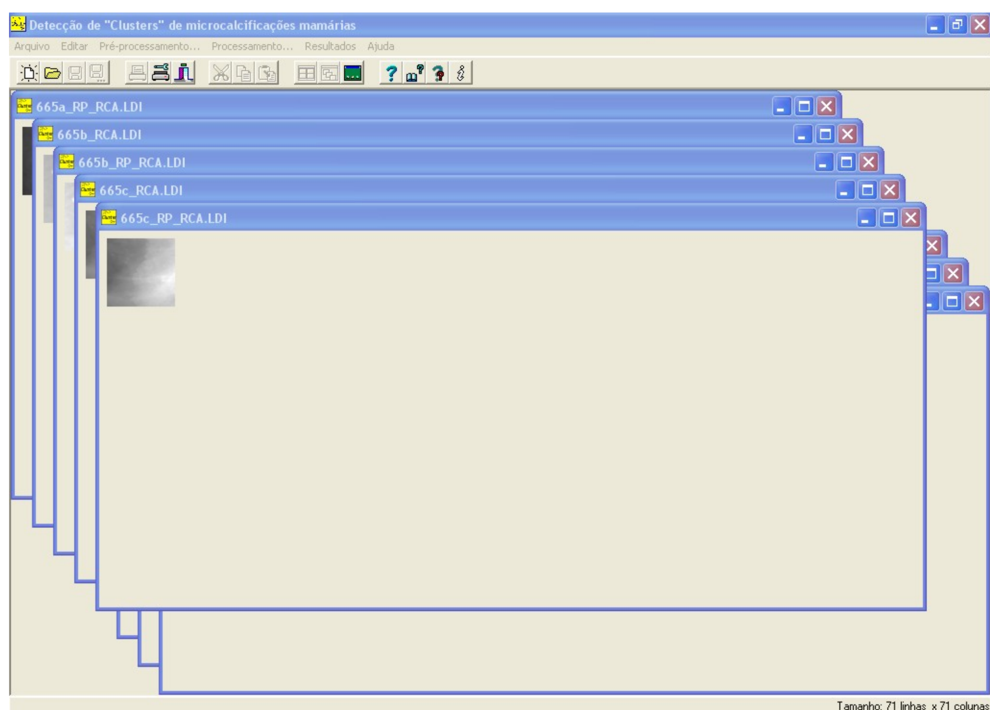

Figura 6.4: Realce de Contraste pelos Coeficientes de Atenuação e Segmentação

A unidade *pacotelocal* que será exemplificada a seguir foi executada no primeiro caso de teste, e a ferramenta de instrumentação gerou um grafo desta rotina, como mostrada na Figura 6.7. Depois de executado, ela gera um arquivo trace com os números dos nós

Tabela 6.2: Tempo de execução do Realce de Contraste pelos Coeficientes de Atenuação e Segmentação

Número	Tempo não instrumentado	Tempo instrumentado
01	78,89	144,22
02	78,02	164,91
03	77,60	157,34
04	77,45	147,23
05	77,71	159,73
06	77,76	158,45
07	78,24	153,69
08	77,72	147,32
09	77,72	152,25
10	78,16	158,45
11	77,82	153,69
12	77,80	147,32
13	77,85	152,25
14	77,84	151,22
15	78,00	147,72
16	77,87	136,59
17	77,67	150,41
18	78,08	135,77
19	78,00	157,90
20	78,68	158,19
21	78,10	157,34
22	77,91	147,25
23	77,81	159,73
24	78,06	158,45
25	77,89	153,89
26	78,38	146,82
27	77,88	152,25
28	77,84	147,32
29	77,90	152,25
30	77,94	153,69
Média	**77,96 segundos**	**152,12 segundos**
Desv.Pad.	**0,30 segundos**	**6,63 segundos**

cobertos.

Um arquivo “dot” mostrado pela Figura 6.6 , especifica como deve ser gerado o grafo em forma de figura pela Ferramenta Graphviz.

Uma das seqüências observadas de sua execução foi a dos nós 1, 2, 3, 4, 5, 7, 9, 11, 13, 15, 17, 20, 21, 22, 23, 24, 25, 27, 29, 32, 33, 35, 36, 35, 37, 32, 33, 34, 20 e volta para o nó 21 fazendo este mesmo caminho várias vezes até sair do procedimento. Na Figura 6.8 são representados os nós executados. Vale lembrar que a pintura escura dos nós mostrados na figura 6.8 não foi feita pelo programa, é apenas uma ilustração para vizualizar quais nós foram executados.

6.4 Considerações Finais

Neste capítulo foi apresentado o estudo de caso, apresentando o tamanho do arquivo executável e o tempo de execução. Foi exemplificado em um programa proposital-

Tabela 6.3: Tempo de execução do Realce de Contraste pela Modificação do Histograma e Segmentação

Número	Tempo não instrumentado	Tempo instrumentado
01	92,12	152,07
02	88,44	150,75
03	98,71	157,91
04	88,40	161,89
05	88,38	151,66
06	88,73	166,81
07	88,95	152,43
08	88,32	151,66
09	88,54	151,72
10	88,87	166,81
11	88,43	152,43
12	88,50	151,66
13	88,43	151,72
14	88,55	152,68
15	88,65	168,33
16	88,59	152,54
17	88,47	165,31
18	88,55	152,63
19	88,51	164,88
20	89,88	152,73
21	88,69	158,11
22	88,49	161,84
23	88,35	151,85
24	88,39	166,81
25	88,70	152,12
26	88,53	151,68
27	88,47	150,82
28	88,58	151,66
29	88,60	151,72
30	88,71	152,43
Média	**89,05 segundos**	**155,92 segundos**
Desv.Pad.	**1,96 segundos**	**6,10 segundos**

mente escrito na linguagem Delphi com 5.388 linhas de programação, para averiguação da gramática e a descrição do instrumentador. O aumento de tempo consideravelmente extense é devido a quantidade de código que é processado nos casos de teste, pois o processamento de imagens utiliza muitos laços ou comandos de repetição. Mesmo assim, o tempo que o programa demorou a mais para a execução das rotinas revela-se insignificante comparado com o propósito da ferramenta, mesmo porque, o software instrumentado só será utilizado para encontrar/rastrear erros, ou seja, auxiliar o testador de software a encontrar erros. No próximo capítulo serão apresentadas as considerações finais, as quais ressaltarão a importância do teste de software e descreverão o propósito deste trabalho, além disso oferecendo sugestões para trabalhos futuros.

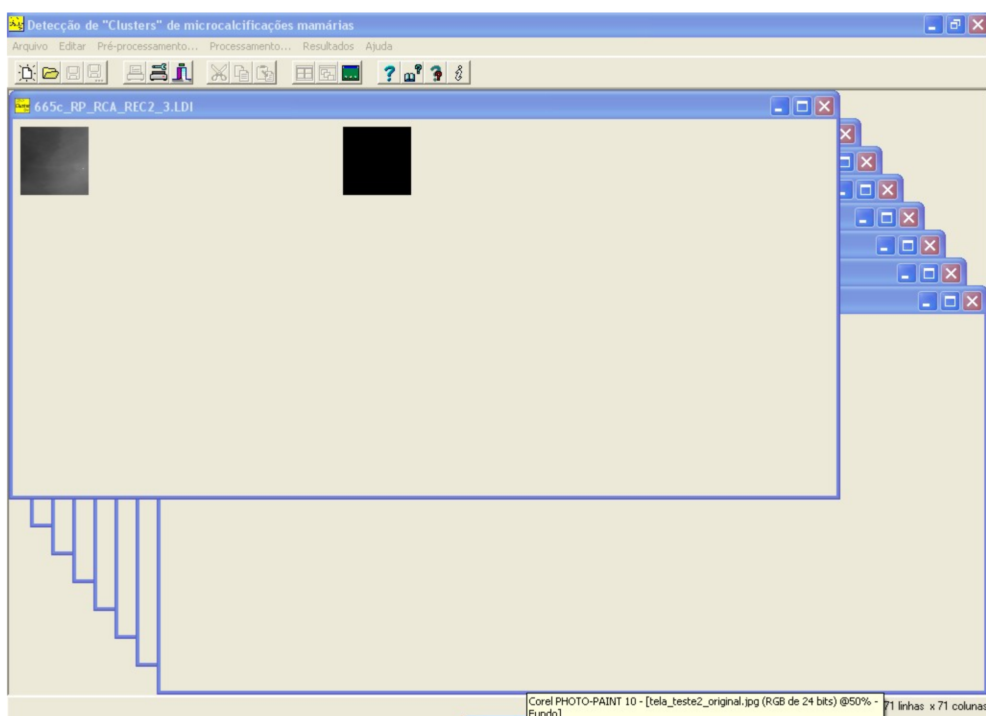

Figura 6.5: Realce de Contraste pela Modificação do Histograma e Segmentação

```
digraph gfc {
node [shape = doublecircle] 31;
node [shape = circle] 1;
/* pusage of icancelapacote at 1 */
node [shape = circle] 19;
node [shape = circle] 2;
/* pusage of lexecmh at 2 */
/* pusage of lexecca at 2 */
node [shape = circle] 18;
node [shape = circle] 3;
/* pusage of lexecca at 3 */
node [shape = circle] 4;
node [shape = circle] 5;
/* pusage of lexecmh at 5 */
node [shape = circle] 6;
node [shape = circle] 7;
/* pusage of lexecsegorig at 7 */
/* pusage of lexecsegrealce at 7 */
node [shape = circle] 8;
node [shape = circle] 9;
/* pusage of lexectap at 9 */
node [shape = circle] 10;
node [shape = circle] 11;
/* pusage of lexecdc at 11 */
node [shape = circle] 12;
node [shape = circle] 13;
/* pusage of lexeccurva at 13 */
node [shape = circle] 14;
node [shape = circle] 15;
node [shape = circle] 16;
node [shape = circle] 21;
/* pusage of lusaporcfibra at 21 */
node [shape = circle] 24;
node [shape = circle] 22;
/* pusage of nporcfibra at 22 */
node [shape = circle] 23;
node [shape = circle] 25;
/* pusage of lexecsegorig at 25 */
node [shape = circle] 26;
node [shape = circle] 27;
/* pusage of lexecmh at 27 */
/* pusage of lexecca at 27 */
/* pusage of lexecfp at 27 */
/* pusage of lexectap at 27 */
node [shape = circle] 28;
node [shape = circle] 30;
node [shape = circle] 20;
/* cusage of vetornomes[nindicepacote] at 20 */
/* cusage of nindicepacote at 20 */
/* cusage of nomearq at 20 */
/* cusage of round(procurareg(tabporc,arquivooriginal)*100) at 20 */
/* cusage of procurareg(tabporc,arquivooriginal) at 20 */
/* cusage of tabporc at 20 */
/* cusage of arquivooriginal at 20 */
/* cusage of fpacote.caporcfibra.value at 20 */
/* cusage of nporcfibra at 20 */
/* cusage of vetornomes[nindicepacote] at 20 */
/* cusage of nindicepacote at 20 */
/* cusage of nomearq at 20 */
/* cusage of nomearq at 20 */
/* cusage of vetornomes[nindicepacote] at 20 */
/* cusage of nindicepacote at 20 */
/* cusage of nomearq at 20 */
node [shape = circle] 17;
node [shape = circle] 34;
/* cusage of quantimagens at 34 */
node [shape = circle] 32;
node [shape = circle] 29;
node [shape = circle] 37;
/* cusage of grdauximagem.rowcount at 37 */
node [shape = circle] 35;
/* cusage of grdimagem.cells[j,i] at 35 */
/* cusage of j at 35 */
/* cusage of i at 35 */
node [shape = circle] 33;
node [shape = circle] 36;
/* cusage of grdauximagem.colcount at 36 */
30 -> 31; 11 -> 13;
11 -> 12; 12 -> 13;
13 -> 15; 13 -> 14;
14 -> 15; 15 -> 16;
16 -> 17; 15 -> 16;
16 -> 17; 17 -> 20;
20 -> 21; 20 -> 30;
34 -> 20; 21 -> 25;
21 -> 22; 24 -> 25;
22 -> 24; 22 -> 23;
23 -> 24; 25 -> 27;
25 -> 26; 26 -> 27;
27 -> 29; 27 -> 28;
28 -> 29; 29 -> 32;
32 -> 33; 32 -> 34;
37 -> 32; 33 -> 35;
35 -> 36; 35 -> 37;
36 -> 35; 1 -> 2;
1 -> 19;  19 -> 31;
2 -> 3;   2 -> 18;
18 -> 31; 3 -> 5;
3 -> 4;   4 -> 5;
5 -> 7;   5 -> 6;
6 -> 7;   7 -> 9;
7 -> 8;   8 -> 9;
9 -> 11;  9 -> 10;
10 -> 11;
```

Figura 6.6: Arquivo DOT

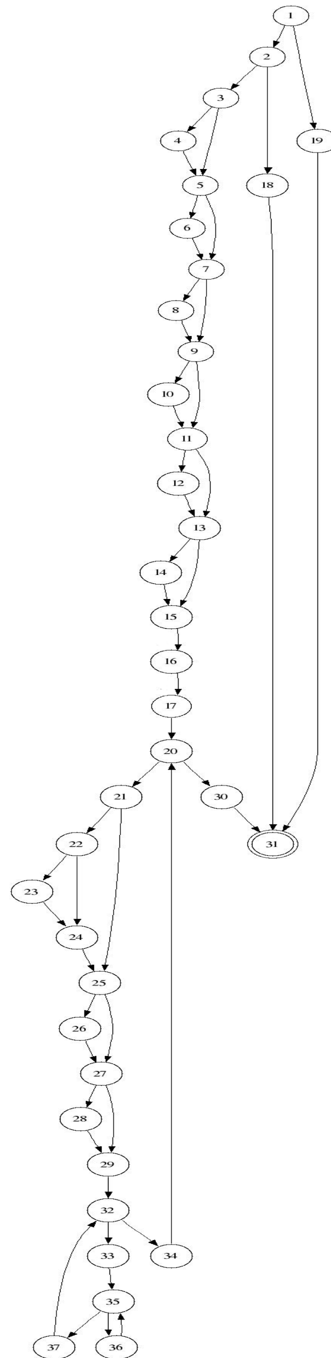

Figura 6.7: Grafo da Procedure Pacotelocal

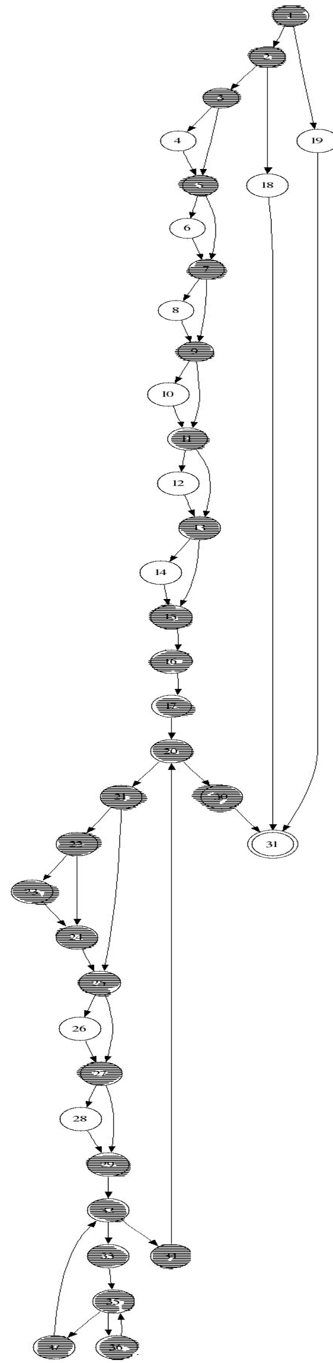

Figura 6.8: Grafo da Procedure Pacotelocal Percorrido

7 Conclusões e trabalhos futuros

A linguagem de programação Delphi é muito utilizada por ser uma linguagem fácil e familiar como é a linguagem de programação Pascal. A visibilidade crescente do software como um elemento do sistema e o custo associado à falha motivam esforços para a prática de testes de software. Atividades agregadas sobre o nome Garantia de Qualidade de Software tem sido introduzida ao longo do processo de desenvolvimento, e a falta de ferramentas de apoio ao teste estrutural para a linguagem de programação Delphi foi o que impulsionou este trabalho. Encontra-se no mercado apenas uma ferramentas de apoio ao teste estrutural para Linguagem Delphi, que atende apenas o critério Todos Nós. Com este trabalho além deste critério Todos Nós será possível utilizar o critério Todos os Caminhos que não é comum em ferramentas de teste de software, e também critério Todos os Usos que dificilmente encontra-se em ferramentas de teste.

O processo de instrumentação é de fundamental importância em diversas áreas da Engenharia de Software, tornando-se mais confiável e preciso quando realizado com o auxílio de um software instrumentador com propósito específico. O instrumentador Delphi foi desenvolvido com o apoio da linguagem de descrição de instrumentação IDeL e seu respectivo compilador IDeLGen. Mesmo com pesquisas e revisões bibliográficas, não foi possível encontrar uma representação BNF completa da gramática do Delphi. Assim, seu desenvolvimento foi parte integrante deste trabalho.

A gramática do Delphi foi desenvolvida para os programas Lex (analisador léxico) e Yacc (analisador sintático). No entanto, foram desenvolvidos dois arquivos: o delphi.y (Apêndice A), com 320 produções e o delphi.l (Apêndice B) que reconhece 147 tokens (terminais). O analisador léxico reconhece o token e repassa para o analisador sintático realizar a análise sintática. O Lex e Yacc, responsáveis pela construção do analisador léxico e sintático respectivamente, foram usados pela ferramenta IDeLGen para gerar o instrumentador.

A obtenção da gramática apresentou considerável dificuldade e se revelou de suma

importância no contexto do projeto. Conforme visto no Capítulo 3, a complexidade para confeccionar uma gramática é grande, tendo em vista que a Linguagem Delphi possui uma gramática mais complexa do que as Linguagens "C" ou "Pascal". Outros fatores de complexidades são a sua flexibilidade na programação estruturada e seu atendimento a vários tipos de componentes visuais. Foi despendido um esforço muito grande para a confecção da descrição da IDeL, pois a falta de documentação para a confecção dos requisitos requeridos por ela dificultou muito o trabalho.

O único documento disponível para consulta sobre os aspectos operacionais do sistema é um relatório técnico (SIMAO et al., 2001) que, apesar de grande utilidade para este trabalho, não abrange toda a complexidade do gerador de instrumentador. Uma linha errada no arquivo de descrição causa um erro complicadíssimo de ser rastreado. Baseado no arquivo de descrição, de como instrumentar a Linguagem "C", foram alteradas e adicionadas várias outras regras para atender à sintaxe da Linguagem Delphi.

Por fim, tendo em vista que a linguagem de programação Delphi é largamente utilizada, a possibilidade de instrumentar códigos Delphi é de suma importância, pois com a confecção da gramática e a descrição de instrumentação, junto com as ferramentas IDeL e IDeLGen é possível atender a esta necessidade. Assim este trabalho contribui com a instrumentação e teste para a linguagem Delphi.

7.1 Trabalhos Futuros

Ao longo da confecção deste projeto foi possível identificar melhorias que envolvem a alteração dos sistemas IDeL e IDeLGen. Um dos problemas encontrados foi na descrição de instrumentação da IDeL. A ferramenta permite encontrar apenas uma única unidade de identificação, unidade esta, que encontra cada função ou procedimento do programa Delphi para a instrumentação. Seria de muita utilidade se fosse possível identificar várias unidades. A solução encontrada para resolver este problema foi generalizar a função e procedimento direto na gramática do Delphi (arquivo delphi.y Apêndice A - Pagina 96), possibilitando a escrita de apenas uma unidade.

Outra melhoria envolve a alteração na gramática da descrição de como instrumentar (arquivo delphi.idel Apêndice C - Página 133), para que seja possível especificar a criação de nós e arestas diferenciados, já que a IDeL utiliza o programa "Graphviz" para gerar os grafos. Este programa aceita inúmeros tipos de arestas e nós. É de grande importância a possibilidade de saber quais códigos fazem parte de qual nó, possibilitando ao testador

uma fácil escolha para caso de teste.

Uma interface gráfica poderia ser construída para manipular toda a ferramenta. Esta ferramenta também poderia levantar os requisitos de teste para a cobertura de vários critérios e ainda obter vários tipos de relatórios de testes. Em um trabalho futuro, pode-se utilizar o instrumentador gerado para linguagem Delphi deste projeto e confeccionar uma ferramenta visual que oriente o testador em vários critérios existentes em teste de software, exemplo a JaBUTi apresentado no Capítulo 2 Página 25

Referências

AHO, A. V.; SETHI, R.; ULLMAN, J. D. *Compiladores Princípios, Técnicas e Ferramentas*. Massachusetts, 01867, USA: LTC-Livros Técnicos e científicos Editora S.A, 1986.

CYAMON SOFTWARE. *Discover for Delphi*. 2005. Discover Ferramenta de teste para Delphi. Disponível em: <http://www.cyamon.com/discover1.htm>. Acesso em: 27 mar. 2005.

DELAMARO, M. E. *Proteum: Um ambiente de teste baseado na análise de mutantes*. Dissertação (Mestrado) — Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo, Outubro 1993.

DELAMARO, M. E. Mutação de interface: Um critério de adequação inter-procedimental para o teste de integração. 1997.

DEMILLO, R.; LIPTON, R.; SAYWARD, F. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, v. 11, p. 34–41, 1978.

HARROLD, M. J.; ROTHERNEL, G. Performing data flow testing on classes. *Australian Comtuter Journal*, p. 92–96, 1994.

HERMAN, P. M. A data flow analysis approach to program testing. *Australian Computer Journal*, 1976.

HOWDEN, W. E. *Functional program testing and analysis*. New York, NY, USA: McGraw-Hill, Inc., 1986. ISBN 0-070-30550-1.

KUNG, D. C.; HSIA, P. Object orientado integration testing. *In Third IEEE International High-Assurance Systems Engineering Symposium*, p. 411–426, Nov 1998.

MALDONADO, J. C. *Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software*. Tese (Doutorado) — DCA/FEE/UNICAMP, Campinas, SP, Brasil, Julho 1991.

MALDONADO, J. C. et al. *Introdução ao teste de software*. São Carlos, Janeiro 2004. Instituto de Ciências Matemáticas e de Computação.

MALDONADO, J. C.; CHAIM, M. L.; JINO, M. A brief view of potential uses criteria and of poke-tool testing tool. Instituto de Ciências Matemáticas e de Computação da Universidade de São Paulo. 1991.

MCGREGOR, J. D.; SYKES, D. A. A pratical guide to testing object-oriented. Addison-Wesley. 2001.

NTAFOS, S. C. A comparison of some structural testing strategies. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 14, n. 6, p. 868–874, 1988. ISSN 0098-5589.

NUNES, F. L. S. *Investigações em processamento de imagens mamográficas para auxílio ao diagnóstico de mamas densas.* Tese (Doutorado) — IFSC/USP, S.Carlos (SP), 2001.

PRESSMAN, R. S. *Software Engineering: A Practitioner's Approach.* New York, NY, USA: McGraw-Hill Higher Education, 2001. ISBN 0072496681.

RAPPS, S.; WEYUKER, E. J. Data flow analysis techniques for test data selection. In: *ICSE '82: Proceedings of the 6th international conference on Software engineering.* [S.l.]: IEEE Computer Society Press, 1982. p. 272–278.

ROCHA, A. D. et al. *Teste Funcional: Uma abordagem Auxiliada por Aspectos.* São Carlos, SP, Brasil, 2004.

SIMAO, A. S. et al. *A Language for the Description of Program Instrumentation and Automatic Generation of Instrumenters.* São Carlos, 2001. 23 p.

SOMMERVILLE, I. *Software Engineering.* New York, NY, USA: Addison Wesley, 2003. ISBN 8588639076.

VINCENZI, A. M. R. *Subsídios para o Estabelecimento de Estratégias de Teste Baseadas na Técnica de Mutação.* Dissertação (Mestrado) — Instituto de Ciências Matemáticas de São Carlos - USP, São Carlos, abril 1997.

VINCENZI, A. M. R. et al. JaBUTi: A coverage analysis tool for Java programs. In: *XVII SBES – Simpósio Brasileiro de Engenharia de Software*. Manaus, AM, Brasil: [s.n.], 2003. p. 79–84.

Apendice A - Gramática do Delphi (delphi.y)

```
%token ABSOLUTE
%token ABSTRACT
%token AND
%token ANPARSANT
%token ARRAY
%token AS
%token ASM
%token ASSEMBLER
%token AT
%token ATSIGN
%token ATTRIB
%token AUTOMATED
%token _BEGIN
%token CASE
%token CDECL
%token CIRC
%token CLASS
%token CLOSEBRACKET
%token CLOSECURL
%token CLOSEPAR
%token COLON
%token COMMA
%token CONST
%token CONSTRUCTOR
%token CONTAINS
%token DEFAULT
%token DESTRUCTOR
%token DIFF
%token DISPID
%token DISPINTERFACE
%token DIV
%token DO
%token DOT
%token DOTDOT
%token DOWNTO
%token DYNAMIC
%token ELSE
%token END
%token EQUAL
%token EXPORT
%token EXPORTS
%token EXTERNAL
```

```
%token EXIT
%token FAR
%token _FILE
%token FINALIZATION
%token FINALLY
%token FOR
%token FORWARD
%token FUNCTION
%token GE
%token GOTO
%token GT
%token HALT
%token IDENTIFIER
%token IF
%token IMMCHAR_ID
%token IMPLEMENTATION
%token IMPLEMENTS
%token IN
%token INDEX
%token INHERITED
%token INITIALIZATION
%token INLINE
%token INTERFACE
%token INTERRUPT
%token IS
%token LABEL
%token LE
%token LIBRARY
%token LT
%token MESSAGE
%token MINUS
%token MOD
%token NAME
%token NEAR
%token NIL
%token NODEFAULT
%token NOT
%token OBJECT
%token OF
%token ON
%token OPENBRACKET
%token OPENCURL
%token OPENPAR
%token OR
%token OUT
%token OVERLOAD
%token OVERRIDE
%token PACKAGE
%token PACKED
%token PASCAL
%token PLUS
%token PRIVATE
%token PROCEDURE
%token PROGRAM
```

```
%token PROPERTY
%token PROTECTED
%token PUBLIC
%token PUBLISHED
%token RAISE
%token READ
%token RECORD
%token REGISTER
%token REINTRODUCE
%token REPEAT
%token REQUIRES
%token RESIDENT
%token RESOURCESTRING
%token SAFECALL
%token SEMICOLON
%token SET
%token SHL
%token SHR
%token SLASH
%token STAR
%token STARSTAR
%token STDCALL
%token STORED
%token STRING
%token STRING_CONST
%token THEN
%token THREADVAR
%token TILDE
%token TO
%token TOP
%token TRY
%token TYPE
%token UNIT
%token UNSIGNED_INTEGER
%token UNSIGNED_REAL
%token UNTIL
%token USES
%token VAR
%token VIRTUAL
%token WHILE
%token WITH
%token XOR
%token EXCEPT
%token ARRAY_RANGE

%%
program
        : program_file
        | unit_file
        | library_file
        | package_file
        ;

package_file
```

```
        : PACKAGE IDENTIFIER SEMICOLON
          requires_clause
          contains_clause
        ;

requires_clause
        :
        | REQUIRES requires_units_list SEMICOLON
        ;

requires_units_list
        : requires_units_list COMMA IDENTIFIER
        | IDENTIFIER
        ;

contains_clause
        :
        | CONTAINS contains_units_list SEMICOLON
        ;

contains_units_list
        : contains_units_list COMMA contains_unit_name
        | contains_unit_name
        ;

contains_unit_name
        : IDENTIFIER
        | IDENTIFIER  keyword_in string
        ;

library_file
        : library_heading main_uses_clause
          library_block DOT
        ;

library_heading
        : LIBRARY IDENTIFIER SEMICOLON
        ;

library_block
        : library_impl_decl_sect_list compound_stmt
        ;

library_impl_decl_sect_list
        :
        | library_impl_decl_sect_list library_impl_decl_sect
        ;

library_impl_decl_sect
        : impl_decl_sect
        | export_clause
        ;

export_clause
```

```
        :
| EXPORTS exports_list SEMICOLON
        ;

exports_list
        : exports_entry
        | exports_list COMMA exports_entry
        ;

exports_entry
        : exports_identifier exports_index exports_name exports_resident
        ;

exports_identifier
        : qualified_identifier
        ;

exports_index
        :
        | INDEX integer_const
        ;

exports_name
        :
        | NAME string_const_2
        ;

exports_resident
        :
        | RESIDENT
        ;

program_file
        : program_heading main_uses_clause
          program_block DOT
        ;

program_heading
        :
        | PROGRAM IDENTIFIER program_heading_2
        ;

program_heading_2
        : SEMICOLON
        | OPENPAR program_param_list CLOSEPAR SEMICOLON
        ;

program_param_list
        : IDENTIFIER
        | program_param_list COMMA IDENTIFIER
        ;

program_block
        : program_decl_sect_list compound_stmt
```

```
        ;

program_decl_sect_list
:
| program_decl_sect_list impl_decl_sect
| program_decl_sect_list export_clause
;

uses_clause
        :
        | USES used_units_list SEMICOLON
        ;

used_units_list
        : used_units_list COMMA IDENTIFIER
        | IDENTIFIER
        ;

main_uses_clause
        :
        | USES main_used_units_list SEMICOLON
        ;

main_used_units_list
        : main_used_units_list COMMA main_used_unit_name
        | main_used_unit_name
        ;

main_used_unit_name
        : IDENTIFIER
        | IDENTIFIER  keyword_in string
        ;

unit_file
        : unit_heading interface_part implementation_part initialization_part DOT
;

unit_heading
        : UNIT IDENTIFIER SEMICOLON
        ;

interface_part
        : INTERFACE uses_clause int_decl_sect_list
        ;

implementation_part
: IMPLEMENTATION uses_clause impl_decl_sect_list export_clause
        ;

initialization_part
        : compound_stmt
        | new_initialization_part END
        | new_initialization_part new_finalization_part END
        | END
```

```
        ;

new_initialization_part
        : INITIALIZATION stmt_list
        ;

new_finalization_part
        : FINALIZATION stmt_list
        ;

impl_decl_sect_list
        :
        | impl_decl_sect_list impl_decl_sect
        ;

impl_decl_sect
        : block_decl_sect
        | threadvar_decl_sect
        | constructor_impl
        | destructor_impl
        ;

block_decl_sect
        : const_type_var_decl
        | label_decl_sect
        | main_impl
        ;

const_type_var_decl
        : const_decl_sect
        | type_decl_sect
        | var_decl_sect
        ;

label_decl_sect
        : LABEL label_list SEMICOLON
        ;

label_list
        : label
        | label_list COMMA label
        ;

label
        : IDENTIFIER
        ;

const_decl_sect
        : const_decl_sect_old
        | const_decl_sect_resourcestring
        ;

const_decl_sect_old
        : CONST const_decl
```

```
        | const_decl_sect_old const_decl
        ;

const_decl_sect_resourcestring
        : RESOURCESTRING const_decl_resourcestring
        | const_decl_sect_resourcestring const_decl_resourcestring
        ;

const_decl_resourcestring
        : IDENTIFIER EQUAL res_string
        ;

res_string
        : qualified_identifier
        | string
        | res_string PLUS qualified_identifier
        | res_string PLUS string
        ;

type_decl_sect
        : TYPE type_decl
        | type_decl_sect type_decl
        ;

var_decl_sect
        : VAR var_decl
        | var_decl_sect var_decl
        ;

threadvar_decl_sect
        : THREADVAR threadvar_decl
        | threadvar_decl_sect threadvar_decl
        ;

int_decl_sect_list
        :
        | int_decl_sect_list int_decl_sect
        ;

int_decl_sect
        : const_type_var_decl
        | threadvar_decl_sect
        | proc_heading
        | func_heading
        ;

const_decl
        : const_name EQUAL const SEMICOLON
        | const_name COLON type EQUAL typed_const SEMICOLON
        ;

const_name
: IDENTIFIER
;
```

```
const   : const_simple_expr
        | const_simple_expr const_relop const_simple_expr
        ;

const_relop
        : EQUAL
        | DIFF
        | LT
        | GT
        | LE
        | GE
        | keyword_in
        ;

const_simple_expr
        : const_term
        | const_simple_expr add_op const_term
        ;

const_term
        : const_factor
        | const_term const_mulop const_factor
        ;

const_mulop
: STAR
| SLASH
| DIV
| MOD
| SHL
| SHR
| AND
        ;

const_factor
        : const_variable
        | unsigned_number
        | string
        | NIL
        | ATSIGN const_factor
        | OPENPAR const CLOSEPAR
        | NOT const_factor
        | sign const_factor
        | set_const
        ;

const_variable
        : qualified_identifier
        | const_variable const_variable_2
        ;

const_variable_2
        : OPENBRACKET const_expr_list CLOSEBRACKET
```

```
        | DOT IDENTIFIER
        | CIRC
        | OPENPAR const_func_expr_list CLOSEPAR
        ;

const_expr_list
        : const
        | const_expr_list COMMA const
        ;

const_func_expr_list
        : const
        | const COMMA const
        ;

const_elem_list
        :
        | const_elem_list1
        ;

const_elem_list1
        : const_elem
        | const_elem_list1 COMMA const_elem
        ;

const_elem
        : const
        | const DOTDOT const
        ;

unsigned_number
        : unsigend_integer
        | UNSIGNED_REAL
        ;

sign
        : PLUS
        | MINUS
        ;

typed_const
        : const
        | array_const
        | record_const
        | OPENPAR CLOSEPAR
        ;

array_const
        : OPENPAR typed_const_list CLOSEPAR
        ;

typed_const_list
        : typed_const COMMA typed_const
        | typed_const_list COMMA typed_const
```

```
        ;

record_const
        : OPENPAR const_field_list CLOSEPAR
        ;

const_field_list
        : const_field
        | const_field_list SEMICOLON
        ;

const_field
        : const_field_name COLON typed_const
        ;

const_field_name
        : IDENTIFIER
        ;

set_const
        : OPENBRACKET const_elem_list CLOSEBRACKET
        ;

type_decl
        : IDENTIFIER EQUAL type_decl_type SEMICOLON
        ;

type_decl_type
        : type
        | TYPE type
        | object_type
        ;

type    : simple_type
        | pointer_type
        | structured_type
        | string_type
        | procedural_type
        ;

simple_type
        : simple_type_ident
        | const_simple_expr DOTDOT const_simple_expr
        | OPENPAR enumeration_id_list CLOSEPAR
        ;

simple_type_ident
        : qualified_identifier
        ;

enumeration_id_list
        : enumeration_id COMMA enumeration_id
        | enumeration_id_list COMMA enumeration_id
        ;
```

```
enumeration_id
: IDENTIFIER
;

pointer_type
        : CIRC IDENTIFIER
        ;

structured_type
        : unpacked_structured_type
        | PACKED unpacked_structured_type
        ;

unpacked_structured_type
        : array_type
        | record_type
        | set_type
        | file_type
        ;

array_type
        : ARRAY array_index_decl OF type
        ;

array_index_decl
:
| OPENBRACKET simple_type_list CLOSEBRACKET
;

simple_type_list
        : simple_type
        | simple_type_list COMMA simple_type
| ARRAY_RANGE
        ;

record_type
        : RECORD field_list END
        ;

field_list
        :
        | fixed_part
        | variant_part
        | fixed_part_2 SEMICOLON variant_part
        ;

fixed_part
        : fixed_part_2
        | fixed_part_2 SEMICOLON
        ;

fixed_part_2
        : record_section
```

```
        | fixed_part_2 SEMICOLON record_section
        ;

record_section
        : record_section_id_list COLON type
        ;

record_section_id_list
        : record_section_id
        | record_section_id_list COMMA record_section_id
        ;

record_section_id
: IDENTIFIER
;

variant_part
        : CASE tag_field OF variant_list
        ;

tag_field
        : tag_field_typename
        | tag_field_name COMMA tag_field_typename
        ;

tag_field_name
: IDENTIFIER
;

tag_field_typename
: qualified_identifier
;

variant_list
        : variant_list_2
        | variant_list_2 SEMICOLON
        ;

variant_list_2
        : variant
        | variant_list_2 SEMICOLON variant
        ;

variant : case_tag_list COLON OPENPAR variant_field_list CLOSEPAR
        ;

variant_field_list
        : field_list
        ;

case_tag_list
        : const
        | case_tag_list COMMA const
        ;
```

```
set_type
        : SET OF simple_type
        ;

file_type
        : _FILE OF type
        | _FILE
        ;

string_type
        : STRING
        | STRING OPENBRACKET const CLOSEBRACKET
        ;

procedural_type
        : procedural_type_0 proctype_calling_conv
        ;

proctype_calling_conv
        :
        | calling_convention
        ;

procedural_type_0
        : procedural_type_decl
        | procedural_type_decl OF OBJECT
        ;

procedural_type_decl
        : PROCEDURE fp_list
| FUNCTION fp_list COLON fptype
        ;

object_type
        : new_object_type
        | old_object_type
        | it_interface_type
        ;

old_object_type
        : OBJECT oot_successor oot_component_list oot_privat_list END
        ;

oot_privat_list
        :
        | PRIVATE oot_component_list
        ;

oot_component_list
        :
        | oot_field_list
        | oot_field_list oot_method_list
        | oot_method_list
```

```
        ;

oot_successor
        : OPENPAR oot_typeidentifier CLOSEPAR
        ;

oot_typeidentifier
        : qualified_identifier
        ;

oot_field_list
        : oot_field
        | oot_field_list oot_field
        ;

oot_field
        : oot_id_list COLON type SEMICOLON
        ;

oot_id_list
        : oot_field_identifier
        | oot_id_list COMMA oot_field_identifier
        ;

oot_field_identifier
        : IDENTIFIER
        ;

oot_method_list
        : oot_method
        | oot_method_list oot_method
        ;

oot_method
        : oot_method_head
        | oot_method_head VIRTUAL SEMICOLON
        ;

oot_method_head
        : proc_heading
        | func_heading
        | oot_constructor_head
        | oot_destructor_head
        ;

oot_constructor_head
        : CONSTRUCTOR IDENTIFIER fp_list SEMICOLON
        ;

oot_destructor_head
        : DESTRUCTOR IDENTIFIER fp_list SEMICOLON
        ;

new_object_type
```

```
        : not_class_reference_type
        | not_object_type
        ;

not_class_reference_type
        : CLASS OF not_class_type_identifier
        ;

not_class_type_identifier
        : IDENTIFIER
        ;

not_object_type
        : CLASS
        | CLASS not_heritage
        | CLASS not_component_list_seq END
        | CLASS not_heritage not_component_list_seq END
        ;

not_component_list_seq
        : not_component_list
        | not_component_list_seq not_visibility_specifier not_component_list
        ;

not_heritage
        : OPENPAR not_object_type_identifier CLOSEPAR
        ;

not_object_type_identifier
        : qualified_identifier
        | qualified_identifier COMMA it_interface_list
        ;

it_interface_list
: it_qualified_interfacename
| it_interface_list COMMA it_qualified_interfacename
;

not_visibility_specifier
        : PUBLISHED
        | PUBLIC
        | PROTECTED
        | PRIVATE
        | AUTOMATED
        ;

not_component_list
        :
        | not_component_list_1
        | not_component_list_2
        | not_component_list_1 not_component_list_2
        ;

not_component_list_1
```

```
        : not_field_definition
        | not_component_list_1 not_field_definition
;

not_component_list_2
        : not_method_definition
        | not_property_definition
        | not_component_list_2 not_method_definition
        | not_component_list_2 not_property_definition
        ;

not_field_definition
        : not_field_identifier_list COLON type SEMICOLON
        ;

not_field_identifier_list
        : not_field_identifier
        | not_field_identifier_list COMMA not_field_identifier
        ;

not_field_identifier
        : IDENTIFIER
        ;

not_method_definition
        : not_method_heading not_method_directives
        ;

not_method_heading
        : CLASS proc_heading
        | CLASS func_heading
        | func_heading
        | proc_heading
        | it_method_attribution
        | not_constructor_heading_decl
        | not_destructor_heading_decl
        ;

not_constructor_heading_decl
        : CONSTRUCTOR IDENTIFIER fp_list SEMICOLON
        ;

not_destructor_heading_decl
        : DESTRUCTOR IDENTIFIER fp_list SEMICOLON
        ;

not_method_directives
        : not_method_directives_0
        | not_method_directives_0 not_dispid_specifier
        ;

not_method_directives_0
        : not_method_directives_reintroduce
          not_method_directives_overload
```

```
          not_method_directives_1
        ;

not_method_directives_reintroduce
        :
        | REINTRODUCE SEMICOLON
;

not_method_directives_overload
        :
        | OVERLOAD SEMICOLON
;

not_method_directives_1
        :
        | VIRTUAL SEMICOLON not_method_directives_2
        | DYNAMIC SEMICOLON not_method_directives_2
        | MESSAGE  integer_const SEMICOLON not_method_directives_2
        | OVERRIDE SEMICOLON
        ;

not_method_directives_2
        : not_method_directives_cdecl not_method_directives_export
          not_method_directives_abstract
        ;

not_method_directives_cdecl
        :
        | calling_convention SEMICOLON
        ;

not_method_directives_export
        :
        | EXPORT SEMICOLON
        ;

not_method_directives_abstract
        :
        | ABSTRACT SEMICOLON
        ;

integer_const
        : const_simple_expr
        ;

not_property_definition
        : PROPERTY IDENTIFIER not_property_interface
          not_property_specifiers SEMICOLON not_array_defaultproperty
        ;

not_array_defaultproperty
        :
        | DEFAULT SEMICOLON
        ;
```

```
not_property_interface
        :
        | not_property_parameter_list COLON not_type_identifier
          not_property_interface_index
        ;

not_type_identifier
        : qualified_identifier
        | STRING
        ;

not_property_interface_index
        :
        | INDEX integer_const
        ;

not_property_parameter_list
        :
        | OPENBRACKET not_parameter_decl_list CLOSEBRACKET
        ;

not_parameter_decl_list
        : not_parameter_decl
        | not_parameter_decl_list SEMICOLON not_parameter_decl
        ;

not_parameter_decl
        : not_parameter_name_list COLON fptype
        ;

not_parameter_name_list
        : not_parameter_name
        | not_parameter_name_list COMMA not_parameter_name
        ;

not_parameter_name
        : IDENTIFIER
        ;

not_property_specifiers_0
: not_read_specifier
        ;

not_property_specifiers
        : not_property_specifiers_0  not_stored_specifier
          not_default_specifier not_implements_specifier
        | not_property_specifiers_0  not_dispid_specifier
        ;

not_dispid_specifier
        : DISPID integer_const
        ;
```

```
not_default_specifier
        :
        | DEFAULT const
        | NODEFAULT
        ;

not_stored_specifier
        :
        | STORED not_stored_bool_const
        ;

not_stored_bool_const
        : const
        ;

not_read_specifier
        :
        | READ not_field_or_method
        ;

not_field_or_method
        : IDENTIFIER
        ;

not_implements_specifier
:
| IMPLEMENTS not_interfacename_list
;

not_interfacename_list
: not_interfacename
| not_interfacename_list COMMA not_interfacename
;

not_interfacename
: IDENTIFIER
;

optGUID
:
| OPENBRACKET string CLOSEBRACKET
;

it_interface_type
: it_dispinterface_type
| it_normalinterface_type
;

it_normalinterface_type
: INTERFACE SEMICOLON
| INTERFACE it_heritage optGUID it_interface_elementlist END SEMICOLON
| INTERFACE it_heritage optGUID END SEMICOLON
;
```

```
it_heritage
:
| OPENPAR it_qualified_interfacename CLOSEPAR
;

it_dispinterface_type
: DISPINTERFACE SEMICOLON
| DISPINTERFACE optGUID it_interface_elementlist END SEMICOLON
;

it_qualified_interfacename
: qualified_identifier
;

it_interface_elementlist
: it_method_or_property
| it_interface_elementlist it_method_or_property
;

it_method_or_property
        : it_method_definition
        | it_property_definition
;

it_method_definition
: it_method_heading it_method_directives
        ;

it_method_heading
        : func_heading
        | proc_heading
        | it_method_attribution
        | it_constructor_heading_decl
        | it_destructor_heading_decl
        ;

it_constructor_heading_decl
        : not_constructor_heading_decl
        ;

it_destructor_heading_decl
        : not_destructor_heading_decl
        ;

it_method_attribution
: FUNCTION it_interfacename DOT IDENTIFIER EQUAL IDENTIFIER SEMICOLON
| PROCEDURE  it_interfacename DOT IDENTIFIER DOT IDENTIFIER SEMICOLON
;

it_interfacename
: IDENTIFIER
;

it_method_directives
```

```
: not_dispid_specifier
;

it_property_definition
: not_property_definition
;

var_decl
        : var_name_list COLON type absolute_clause_or_init_var SEMICOLON
        ;

threadvar_decl
        : var_name_list COLON type SEMICOLON
        ;

var_name_list
        : var_name
        | var_name_list COMMA var_name
        ;

var_name
: IDENTIFIER
;

absolute_clause_or_init_var
        : absolute_clause
        | EQUAL typed_const
        ;

absolute_clause
        :
        | ABSOLUTE unsigend_integer COLON unsigend_integer
        | ABSOLUTE unsigend_integer
        | ABSOLUTE declared_var_name
        ;

declared_var_name
        : qualified_identifier
        ;

constructor_impl
        : not_constructor_heading not_constructor_block_decl
        ;

destructor_impl
        : not_destructor_heading not_constructor_block_decl
        ;

not_constructor_heading
        : CONSTRUCTOR ot_qualified_identifier fp_list SEMICOLON
        ;

not_destructor_heading
        : DESTRUCTOR ot_qualified_identifier fp_list SEMICOLON
```

```
        ;

ot_qualified_identifier
        : IDENTIFIER DOT IDENTIFIER
        ;

not_constructor_block_decl
        : block
        | external_directr
        | asm_block
        ;
name_impl
        : PROCEDURE
| FUNCTION
;

main_impl
        : proc_impl
| func_impl
;

proc_impl
        : name_impl func_or_proc_or_method_head proc_block
        ;

func_impl
        : name_impl func_or_proc_or_method_head func_block
        ;

proc_or_func_fptype
:
| COLON fptype
;

func_or_proc_or_method_head
: func_or_proc_or_method_name fp_list  proc_or_func_fptype SEMICOLON
;

func_or_proc_or_method_name
: ot_qualified_identifier
        | IDENTIFIER
;

proc_heading
        : PROCEDURE proc_or_func_head
        ;

func_heading
        : FUNCTION proc_or_func_head
        ;

proc_or_func_head
: proc_or_func_name fp_list proc_or_func_fptype SEMICOLON
;
```

```
proc_or_func_name
        : IDENTIFIER
        ;

proc_block
        : proc_block_prefix proc_block_decl
        | inline_directr SEMICOLON
        ;

proc_block_prefix
        : func_block_prefix
        | INTERRUPT SEMICOLON
        ;

func_block
        : func_block_prefix func_block_overload proc_block_decl
        | inline_directr SEMICOLON
        ;

func_block_prefix
        :
        | NEAR SEMICOLON
        | FAR SEMICOLON
        | EXPORT SEMICOLON
        | calling_convention SEMICOLON
        ;

calling_convention
        : CDECL
        | REGISTER
        | STDCALL
        | SAFECALL
        | PASCAL
        ;

proc_block_decl
        : block
        | external_directr
        | asm_block
        | FORWARD SEMICOLON
        ;

func_block_overload
:
| OVERLOAD
;

external_directr
        : EXTERNAL SEMICOLON
        | EXTERNAL string_const_2 external_directr_2
        ;

external_directr_2
```

```
        : SEMICOLON
        | NAME string_const_2 SEMICOLON
        | INDEX external_directr_3 SEMICOLON
        ;

external_directr_3
        : unsigend_integer
        ;

string_const_2
        : const_simple_expr
        ;

asm_block
: ASSEMBLER SEMICOLON impl_decl_sect_list asm_stmt SEMICOLON
;

inline_directr
: INLINE OPENPAR inline_element CLOSEPAR
;

inline_element
        : inline_param
        | inline_element SLASH inline_param
        ;

inline_param
        : GT inline_const
        | inline_param_variable
        ;

inline_param_variable
        : variable_reference
        | inline_param_variable sign inline_const
        ;

inline_const
        : const_factor
        ;

block
: impl_decl_sect_list compound_stmt SEMICOLON
        ;

fp_list
:
        | OPENPAR fp_sect_list CLOSEPAR
| OPENPAR CLOSEPAR
        ;

fp_sect_list
        : fp_sect
        | fp_sect_list SEMICOLON fp_sect
        ;
```

```
fp_sect
: keyword_in param_name_list fp_sect_typedef fp_sect_const
        | VAR param_name_list fp_sect_typedef
        | CONST param_name_list  fp_sect_typedef fp_sect_const
        | OUT param_name_list fp_sect_typedef
   | VAR param_name_list
| CONST param_name_list
| OUT param_name_list
        ;

keyword_in
:
| IN
;

fp_sect_typedef
: COLON fptype_new
;

fp_sect_const
:
| EQUAL const
;

param_name_list
        : param_name
        | param_name_list COMMA param_name
        ;

param_name
: IDENTIFIER
        ;

fptype
        : func_res_type_name
        | STRING
        ;

fptype_new
        : fptype
        | ARRAY OF fptype
        | ARRAY OF CONST
        ;

func_res_type_name
        : qualified_identifier
        ;

stmt
: unlabelled_stmt
| label COLON unlabelled_stmt
;
```

```
unlabelled_stmt
        :
        | attribution
        | proc_call
        | goto_stmt
        | compound_stmt
        | if_stmt
        | case_stmt
        | repeat_stmt
        | while_stmt
        | for_stmt
        | with_stmt
        | asm_stmt
        | inline_directr
        | inherited_stmt
        | try_stmt
        | raise_stmt
| exit_stmt
| halt_stmt
        ;

exit_stmt
: EXIT
;

attribution
        : variable_reference attrib_sign expr
        ;

proc_call
        : variable_reference
        ;

goto_stmt
        : GOTO label
        ;

compound_stmt
        : _BEGIN stmt_list END
        ;

stmt_list
        : stmt
        | stmt_list SEMICOLON stmt
        ;

if_stmt
: IF expr if_then_else_branch
        ;

if_then_else_branch
        : THEN then_branch
        | THEN then_branch ELSE else_branch
        ;
```

```
then_branch
        : stmt
        ;

else_branch
        : stmt
        ;

case_stmt
        : CASE expr OF case_list else_case END
        ;

case_list
        : case
        | case_list SEMICOLON case
        ;

case    :
        | case_label_list COLON stmt
        ;

case_label_list
        : case_label
        | case_label_list COMMA case_label
        ;

case_label
        : const
        | const DOTDOT const
        ;

else_case
        :
        | ELSE stmt_list
        ;

repeat_stmt
        : REPEAT stmt_list UNTIL expr
        ;

while_stmt
        : WHILE expr DO stmt
        ;

for_stmt
        : FOR attribution TO expr DO stmt
| FOR attribution DOWNTO expr DO stmt
        ;

with_stmt
        : WITH variable_list DO stmt
| WITH OPENPAR variable_list CLOSEPAR DO stmt
        ;
```

```
variable_list
        : variable_reference
        | variable_list COMMA variable_reference
        ;

inherited_stmt
        : INHERITED
| INHERITED proc_call
        ;

try_stmt
        : TRY stmt_list try_stmt_2
        ;

try_stmt_2
        : FINALLY stmt_list END
        | EXCEPT exception_block END
        ;

exception_block
        : exception_handler_list exception_block_else_branch
        | exception_handler_list SEMICOLON exception_block_else_branch
        | stmt_list
        ;

exception_handler_list
        : exception_handler
        | exception_handler_list SEMICOLON exception_handler
        ;

exception_block_else_branch
        :
        | ELSE stmt_list
        ;

exception_handler
        : ON exception_identifier DO stmt
        ;

exception_identifier
        : exception_class_type_identifier
        | exception_variable COLON exception_class_type_identifier
        ;

exception_class_type_identifier
        : qualified_identifier
        ;

exception_variable
        : IDENTIFIER
        ;

raise_stmt
```

```
        : RAISE
        | RAISE variable_reference
        | RAISE variable_reference AT raise_at_stmt_2
        ;

raise_at_stmt_2
        : variable_reference
        | NIL
        ;

variable_reference
        : variable_reference_at variable
        | variable
        ;

variable_reference_at
        : ATSIGN
        | variable_reference_at ATSIGN
        ;

variable
        : qualified_identifier
        | INHERITED IDENTIFIER
        | variable variable_2
        ;

variable_2
        : OPENBRACKET expr_list CLOSEBRACKET
        | DOT IDENTIFIER
        | CIRC
        | OPENPAR expr_list CLOSEPAR
        ;

expr_list
        :
        | expr_list2
        ;

expr_list2
        : expr
        | expr_list2 COMMA expr
| expr_list2 DOT expr
        ;

expr : simple_expr
        | simple_expr rel_op simple_expr
        | simple_expr COLON simple_expr
        | simple_expr COLON simple_expr COLON simple_expr
;

rel_op
: const_relop
| IS
        ;
```

```
simple_expr
        : term
        | simple_expr add_op term
        ;

add_op
: PLUS
| MINUS
| OR
| XOR
        ;

term
: factor
        | term mul_op factor
        ;
mul_op
: const_mulop
| AS
        ;

factor
: variable_reference
        | unsigned_number
        | string
        | OPENBRACKET elem_list CLOSEBRACKET
        | NIL
        | NOT factor
| parenth_factor
        | sign factor
;

parenth_factor
: OPENPAR expr CLOSEPAR
| parenth_factor CIRC
;

elem_list
        :
        | elem_list1
        ;

elem_list1
        : elem
        | elem_list1 COMMA elem
        ;

elem
: expr
        | expr DOTDOT expr
        ;

qualified_identifier
```

```
        : UNIT DOT IDENTIFIER
        | IDENTIFIER
        ;

asm_stmt
        : ASM END SEMICOLON
        ;

halt_stmt
: HALT
;

constant
: UNSIGNED_INTEGER
;

unsigend_integer
: constant
;

attrib_sign
: ATTRIB
;

string
: STRING_CONST
        | CIRC IMMCHAR_ID
;
%%
```

Apêndice B - Gramática do Delphi (delphi.l)

```
A [aA]
B [bB]
C [cC]
D [dD]
E [eE]
F [fF]
G [gG]
H [hH]
I [iI]
J [jJ]
K [kK]
L [lL]
M [mM]
N [nN]
O [oO]
P [pP]
Q [qQ]
R [rR]
S [sS]
T [tT]
U [uU]
V [vV]
W [wW]
X [xX]
Y [yY]
Z [zZ]
DIGIT [0-9]
LETTER [a-zA-Z_]
HEX [a-fA-F0-9]
EXP [Ee][+-]?{D}+
NQUOTE [^']

%{
#include <stdio.h>

static void count();
%}

%%
"//".*"\n" { }
"{".*"}" { }
[{}]* { }
```

```
{A}{N}{D} { count(); RETURN(AND); }
{A}{R}{R}{A}{Y} { count(); RETURN(ARRAY); }
{A}{B}{S}{O}{L}{U}{T}{E} { count(); RETURN(ABSOLUTE); }
{A}{B}{S}{T}{R}{A}{C}{T} { count(); RETURN(ABSTRACT); }
"<>" { count(); RETURN(DIFF); }
{C}{A}{S}{E} { count(); RETURN(CASE); }
{B}{E}{G}{I}{N} { count(); RETURN(_BEGIN); }
{C}{L}{A}{S}{S} { count(); RETURN(CLASS); }
{C}{O}{N}{S}{T} { count(); RETURN(CONST); }
{C}{O}{N}{S}{T}{R}{U}{C}{T}{O}{R} { count(); RETURN(CONSTRUCTOR); }
{A}{S} { count(); RETURN(AS); }
{A}{S}{N} { count(); RETURN(ASM); }
{A}{S}{S}{E}{M}{B}{L}{E}{R} { count(); RETURN(ASSEMBLER); }
{A}{T} { count(); RETURN(AT); }
{A}{U}{T}{O}{M}{A}{T}{E}{D} { count(); RETURN(AUTOMATED); }
{C}{D}{E}{C}{L} { count(); RETURN(CDECL); }
{C}{O}{N}{T}{A}{I}{N}{S} { count(); RETURN(CONTAINS); }
{D}{E}{F}{A}{U}{L}{T} { count(); RETURN(DEFAULT); }
{D}{E}{S}{T}{R}{U}{C}{T}{O}{R}  { count(); RETURN(DESTRUCTOR); }
{D}{I}{S}{P}{I}{D} { count(); RETURN(DISPID); }
{D}{I}{S}{P}{I}{N}{T}{E}{R}{F}{A}{C}{E} { count(); RETURN(DISPINTERFACE); }
{D}{I}{V} { count(); RETURN(DIV); }
{D}{O} { count(); RETURN(DO); }
{D}{O}{W}{N}{T}{O} { count(); RETURN(DOWNTO); }
{D}{Y}{N}{A}{M}{I}{C} { count(); RETURN(DYNAMIC); }
{E}{L}{S}{E} { count(); RETURN(ELSE); }
{E}{N}{D} { count(); RETURN(END); }
{E}{X}{P}{O}{R}{T} { count(); RETURN(EXPORT); }
{E}{X}{I}{T} { count(); RETURN(EXIT); }
{E}{X}{P}{O}{R}{T}{S} { count(); RETURN(EXPORTS); }
{E}{X}{T}{E}{R}{N}{A}{L} { count(); RETURN(EXTERNAL); }
{E}{X}{C}{E}{P}{T} { count(); RETURN(EXCEPT); }
{F}{A}{R} { count(); RETURN(FAR); }
{F}{I}{L}{E} { count(); RETURN(_FILE); }
{F}{I}{N}{A}{L}{I}{Z}{A}{T}{I}{O}{N}  { count(); RETURN(FINALIZATION); }
{F}{I}{N}{A}{L}{L}{Y} { count(); RETURN(FINALLY); }
{F}{O}{R} { count(); RETURN(FOR); }
{F}{O}{R}{W}{A}{R}{D} { count(); RETURN(FORWARD); }
{F}{U}{N}{C}{T}{I}{O}{N} { count(); RETURN(FUNCTION); }
{G}{O}{T}{O} { count(); RETURN(GOTO); }
{H}{A}{L}{T} { count(); RETURN(HALT); }
{I}{F} { count(); RETURN(IF); }
{I}{M}{P}{L}{E}{M}{E}{N}{T}{S}  { count(); RETURN(IMPLEMENTS); }
{I}{N} { count(); RETURN(IN); }
{I}{N}{D}{E}{X} { count(); RETURN(INDEX); }
{I}{N}{H}{E}{R}{I}{T}{E}{D}  { count(); RETURN(INHERITED); }
{I}{N}{L}{I}{N}{E} { count(); RETURN(INLINE); }
{I}{N}{T}{E}{R}{F}{A}{C}{E} { count(); RETURN(INTERFACE); }
{I}{N}{T}{E}{R}{R}{U}{P}{T} { count(); RETURN(INTERRUPT); }
{I}{S} { count(); RETURN(IS); }
{L}{A}{B}{E}{L} { count(); RETURN(LABEL); }
{L}{I}{B}{R}{A}{R}{Y} { count(); RETURN(LIBRARY); }
{M}{E}{S}{S}{A}{G}{E} { count(); RETURN(MESSAGE); }
{M}{O}{D} { count(); RETURN(MOD); }
```

```
{N}{A}{M}{E} { count(); RETURN(NAME); }
{N}{E}{A}{R} { count(); RETURN(NEAR); }
{N}{I}{L} { count(); RETURN(NIL); }
{N}{O}{D}{E}{F}{A}{U}{L}{T}  { count(); RETURN(NODEFAULT); }
{N}{O}{T} { count(); RETURN(NOT); }
{O}{B}{J}{E}{C}{T} { count(); RETURN(OBJECT); }
{O}{F} { count(); RETURN(OF); }
{O}{N} { count(); RETURN(ON); }
{O}{R} { count(); RETURN(OR); }
{O}{U}{T} { count(); RETURN(OUT); }
{O}{V}{E}{R}{L}{O}{A}{D} { count(); RETURN(OVERLOAD); }
{O}{V}{E}{R}{R}{I}{D}{E} { count(); RETURN(OVERRIDE); }
{P}{A}{C}{K}{A}{G}{E} { count(); RETURN(PACKAGE); }
{P}{A}{C}{K}{E}{D} { count(); RETURN(PACKED); }
{P}{A}{S}{C}{A}{L} { count(); RETURN(PASCAL); }
{P}{R}{I}{V}{A}{T}{E} { count(); RETURN(PRIVATE); }
{P}{R}{O}{C}{E}{D}{U}{R}{E}  { count(); RETURN(PROCEDURE); }
{P}{R}{O}{G}{R}{A}{M} { count(); RETURN(PROGRAM); }
{P}{R}{O}{P}{E}{R}{T}{Y} { count(); RETURN(PROPERTY); }
{P}{R}{O}{T}{E}{C}{T}{E}{D}  { count(); RETURN(PROTECTED); }
{P}{U}{B}{L}{I}{C} { count(); RETURN(PUBLIC); }
{P}{U}{B}{L}{I}{S}{H}{E}{D}  { count(); RETURN(PUBLISHED); }
{R}{A}{I}{S}{E} { count(); RETURN(RAISE); }
{R}{E}{A}{D} { count(); RETURN(READ); }
{R}{E}{C}{O}{R}{D} { count(); RETURN(RECORD); }
{R}{E}{G}{I}{S}{T}{E}{R} { count(); RETURN(REGISTER); }
{R}{E}{I}{N}{T}{R}{O}{D}{U}{C}{E} { count(); RETURN(REINTRODUCE); }
{R}{E}{P}{E}{A}{T} { count(); RETURN(REPEAT); }
{R}{E}{Q}{U}{I}{R}{E}{S} { count(); RETURN(REQUIRES); }
{R}{E}{S}{I}{D}{E}{N}{T} { count(); RETURN(RESIDENT); }
{S}{A}{F}{E}{C}{A}{L}{L} { count(); RETURN(SAFECALL); }
{S}{E}{T} { count(); RETURN(SET); }
{S}{H}{L} { count(); RETURN(SHL); }
{S}{H}{R} { count(); RETURN(SHR); }
{S}{T}{D}{C}{A}{L}{L} { count(); RETURN(STDCALL); }
{S}{T}{O}{R}{E}{D} { count(); RETURN(STORED); }
{S}{T}{R}{I}{N}{G} { count(); RETURN(STRING); }
{T}{H}{E}{N} { count(); RETURN(THEN); }
{T}{H}{R}{E}{A}{D}{V}{A}{R}  { count(); RETURN(THREADVAR); }
{T}{O} { count(); RETURN(TO); }
{T}{R}{Y} { count(); RETURN(TRY); }
{T}{Y}{P}{E} { count(); RETURN(TYPE); }
{U}{N}{I}{T} { count(); RETURN(UNIT); }
{U}{N}{T}{I}{L} { count(); RETURN(UNTIL); }
{U}{S}{E}{S} { count(); RETURN(USES); }
{V}{A}{R} { count(); RETURN(VAR); }
{V}{I}{R}{T}{U}{A}{L} { count(); RETURN(VIRTUAL); }
{W}{H}{I}{L}{E} { count(); RETURN(WHILE); }
{W}{I}{T}{H} { count(); RETURN(WITH); }
{X}{O}{R} { count(); RETURN(XOR); }
{I}{N}{I}{T}{I}{A}{L}{I}{Z}{A}{T}{I}{O}{N} { count(); RETURN(INITIALIZATION); }
{I}{M}{P}{L}{E}{M}{E}{N}{T}{A}{T}{I}{O}{N} { count(); RETURN(IMPLEMENTATION); }
{R}{E}{S}{O}{U}{R}{C}{E}{S}{T}{R}{I}{N}{G} { count(); RETURN(RESOURCESTRING); }
":=" { count(); RETURN(ATTRIB); }
```

```
".." { count(); RETURN(DOTDOT); }
"." { count(); RETURN(DOT); }
"-" { count(); RETURN(MINUS); }
"+" { count(); RETURN(PLUS); }
"*" { count(); RETURN(STAR); }
"/" { count(); RETURN(SLASH); }
"^" { count(); RETURN(CIRC); }
"@" { count(); RETURN(ATSIGN); }
"<" { count(); RETURN(LT); }
">" { count(); RETURN(GT); }
"<=" { count(); RETURN(LE); }
">=" { count(); RETURN(GE); }
"(" { count(); RETURN(OPENPAR); }
")" { count(); RETURN(CLOSEPAR); }
("["|"<:") { count(); RETURN(OPENBRACKET); }
("]"|":>") { count(); RETURN(CLOSEBRACKET); }
";" { count(); RETURN(SEMICOLON); }
"," { count(); RETURN(COMMA); }
":" { count(); RETURN(COLON); }
"=" { count(); RETURN(EQUAL); }
{LETTER}({LETTER}|{DIGIT})* { count(); RETURN(IDENTIFIER); }
(#[0-9]+)+ { count(); RETURN(STRING_CONST); }
'({NQUOTE}|'')+'{ count(); RETURN(STRING_CONST); }
\'(\\.|[^\\'])*\'{ count(); RETURN(STRING_CONST); }
{DIGIT}+\.\.{DIGIT}+ { count(); RETURN(ARRAY_RANGE); }
0[xX]{HEX}+ { count(); RETURN(UNSIGNED_INTEGER); }
0{DIGIT}+ { count(); RETURN(UNSIGNED_INTEGER); }
{DIGIT}+ { count(); RETURN(UNSIGNED_INTEGER); }
{DIGIT}+{EXP} { count(); RETURN(UNSIGNED_REAL); }
{DIGIT}*"."{DIGIT}+({EXP})? { count(); RETURN(UNSIGNED_REAL); }
{DIGIT}+"."{DIGIT}*({EXP})? { count(); RETURN(UNSIGNED_REAL); }
[\n] { count(); }
[ \t\v\f] { count(); }
. { }
%%

static int column = 0;

#define DEBUG
#include "Debug.h"

void count()
{
int i;

if(yytext != NULL)
{
for (i = 0; yytext[i] != '\0'; i++)
{
if (yytext[i] == '\n')
{
column = 0;
}
else if (yytext[i] == '\t')
```

```
{
column += 8 - (column % 8);
}
else
{
column++;
}
}
}
}
```

Apêndice C - Descrição do Delphi na IDeL (delphi.idel)

```
###############################################
##  delphi.idel                              ##
##                                           ##
##  Instrumentador para Delphi/ObjectPascal  ##
##                                           ##
##                                           ##
##    Giovanni Buranello                     ##
##                                           ##
##                              AGOSTO/2006  ##
###############################################

## Especificação IDeL de um instrumentador para as linguagens
## Delphi e Object Pascal.

instrumenter delphi

###############################################
#### Secao 1) Identificacao da Unidade     ####
###############################################

## Cada função é uma unidade.
unit
var
:head as  [name_impl]
:name as  [func_or_proc_or_method_name]
:type as  [proc_or_func_fptype]
:pars as  [fp_list]
:decl as  [impl_decl_sect_list]
:ss as  [stmt_list]
named by
:name
match
[main_impl< :head :name :pars :type ; :decl begin :ss end ; >]
end unit

###############################################
#### Secao 2) Processamento da Unidade     ####
###############################################

## Este passo encontra cada função e criar um nó chamado init
## e um nó chamado exit para cada uma, que são respectivamente
## os nós inicial e final.
```

```
step FindFunction

pattern Function
var
:head as  [name_impl]
:name as  [func_or_proc_or_method_name]
:type as  [proc_or_func_fptype]
:pars as  [fp_list]
:decl as  [impl_decl_sect_list]
:ss as  [stmt_list]
match
[main_impl< :head :name :pars :type ; :decl begin :ss end ; >]
declare node $init
declare node $exit
assignment
assign $parameterdefinition:pars to $init
end pattern

## Este passo cria um nó chamado begin e um nó chamado end para
## cada statement encontrado dentro da unidade.
step MarkStatements

pattern FooStatement
var
:s as  [unlabelled_stmt]
match
[stmt< :s >]
declare node $begin
declare node $end
end pattern

## Este passo determina o nó final e inicial de cada lista de statements
## a partir dos nó final e inicial das subárvores individuais.
step LinkStatementList BT

pattern Statement
var
:s as  [stmt]
match
[stmt_list< :s >]
assignment
assign $begin to  $begin:s
assign $end to  $end:s
end pattern

pattern List
var
:s as  [stmt]
:ss as  [stmt_list]
match
[stmt_list< :ss ; :s >]
graph
$end:ss ->  $begin:s
assignment
```

```
assign $begin to  $begin:ss
assign $end to  $end:s
end pattern

## Este passo conecta o nó final do statement anterior com o nó
## inicial do próximo statement.
step JoinStatement

pattern Join
var
:ss as  [stmt_list]
match
[stmt< begin :ss end >]
graph
$begin ->  $begin:ss
$end:ss ->  $end
end pattern

## Este passo conecta o nó inicial de uma função (init) com o
## primeiro nó da statement list, e também o último nó da statement
## list com o nó final da função (exit).
step JoinToFunction

pattern Function1
var
:head as  [name_impl]
:name as  [func_or_proc_or_method_name]
:type as  [proc_or_func_fptype]
:pars as  [fp_list]
:decl as  [impl_decl_sect_list]
:ss as  [stmt_list]
match
[main_impl< :head :name :pars :type ; :decl begin :ss end ; >]
graph
$init ->  $begin:ss
$end:ss ->  $exit
instrument
add init $init before self
add exit $exit after self
end pattern

## Este passo processa cada statement de acordo com sua semântica
## e gerar os respectivos grafos e tabelas de implementações.
step MakeGraph

## Comando Raise, serve para lancar uma excessao.
pattern Raise
var
:r as  [raise_stmt]
match
[stmt< :r >]
graph
$begin ->  $raise
end pattern
```

```
## Tratamento de excessao!

pattern ExceptionBlock1
var
:ss as  [stmt_list]
match
[exception_block< :ss >]
graph
$begin ->  $begin:ss
$end:ss ->  $end
end pattern

pattern ExceptionBlock2
var
:el as  [exception_handler_list]
match
[exception_block< :el >]
graph
$begin ->  $begin:el
$end:el ->  $end
end pattern

pattern ExceptionBlock3
var
:el as  [exception_handler_list]
:ss as  [stmt_list]
match
[exception_block< :el else :ss >]
graph
$begin ->  $begin:el
$begin ->  $begin:ss
$end:el ->  $end
$end:ss ->  $end
end pattern

pattern ExceptionBlock4
var
:el as  [exception_handler_list]
:ss as  [stmt_list]
match
[exception_block< :el ; else :ss >]
graph
$begin ->  $begin:el
$begin ->  $begin:ss
$end:el ->  $end
$end:ss ->  $end
end pattern

## Quando ocorre um raise dentro do try, vai direto
## para o nó de tratamento de excessao.
pattern Try1
var
:s as  [stmt_list]
```

```
:eb as  [exception_block]
match
[stmt< try :s except :eb end >]
graph
$begin ->  $begin:s
$begin ->  $begin:eb
  $end:s ->  $end
$end:eb ->  $end
assignment
assign  $raise:s to  $begin:eb
end pattern

## Try com finally, que sempre eh executado.
pattern Try2
var
:s1 as  [stmt_list]
:s2 as  [stmt_list]
match
[stmt< try :s1 finally :s2 end >]
graph
$begin ->  $begin:s1
$end:s1 ->  $begin:s2
$end:s2 ->  $end
assignment
assign  $raise:s1 to  $begin:s2
end pattern

## Comando exit, sai de um laço.
pattern Exit
var
:e as  [exit_stmt]
match
[stmt< :e >]
graph
$begin ->  $exit
end pattern

## Comando halt, sai da funcao.
pattern Halt
var
:h as  [halt_stmt]
match
[stmt< :h >]
graph
$begin ->  $exit
assignment
assign $begin to $exit
instrument
add halt $begin before self
add checkpoint $begin before self
end pattern

## Chamada de procedimento ou função.
pattern ProcCall
```

```
var
:p as  [proc_call]
match
[stmt< :p >]
graph
$begin ->  $end
end pattern

## Comando asm, permite usar código assembler.
pattern Asm
var
:a as  [asm_stmt]
match
[stmt< :a >]
graph
$begin ->  $end
end pattern

## Comando Inline, permite usar código assembler.
pattern InlineDirectr
var
:i as  [inline_directr]
match
[stmt< :i >]
graph
$begin ->  $end
end pattern

## Comando Inherited
pattern Inherited
var
:i as  [inherited_stmt]
match
[stmt< :i >]
graph
$begin ->  $end
end pattern

## Comando If...Then
pattern IfThen
var
:e as  [expr]
:s as  [stmt]
match
[stmt< if :e then :s >]
declare node $foo
graph
$begin ->  $end
$begin ->  $begin:s
$end:s ->  $end
assignment
assign $raise:s to  $exit
assign $definition:e to  $begin
assign $pusage:e to  $begin
```

```
instrument
add checkpoint $begin before :e
add checkpoint $begin before self
add checkpoint $begin:s before :s
add checkpoint $end after self
end pattern

## Comando If...Then...Else
pattern IfThenElse
var
        :e as  [expr]
:s1 as  [stmt]
:s2 as  [stmt]
match
[stmt< if :e then :s1 else :s2 >]
graph
$begin ->  $begin:s1
$begin ->  $begin:s2
$end:s1 ->  $end
$end:s2 ->  $end
assignment
assign $raise:s1 to  $exit
assign $raise:s2 to  $exit
assign $definition:e to  $begin
assign $pusage:e to  $begin
instrument
add checkpoint $begin before :e
add checkpoint $begin before self
add checkpoint $begin:s1 before :s1
add checkpoint $begin:s2 before :s2
add checkpoint $end after self
end pattern

## Comando While...Do
pattern While
var
:e as  [expr]
:s as  [stmt]
match
[stmt< while :e do :s >]
declare node $control
graph
$begin ->  $control
$control ->  $begin:s
$end:s ->  $control
$control ->  $end
assignment
assign $raise:s to  $exit
assign $break:s to  $end
assign $definition:e to  $control
assign $pusage:e to  $control
instrument
add checkpoint $begin before self
add checkpoint $begin:s before :s
```

```
add checkpoint $end after self
add checkpoint $control before :e
end pattern

## Comando Repeat...Until
pattern RepeatUntil
var
:e as  [expr]
:ss as  [stmt_list]
match
[stmt< repeat :ss until :e >]
declare node $control
graph
$begin ->  $begin:ss
$end:ss ->  $control
$control ->  $begin:ss
$control ->  $end
assignment
assign $raise:ss to  $exit
assign $break:ss to  $end
assign $definition:e to  $control
assign $pusage:e to  $control
instrument
add checkpoint $begin before self
add checkpoint $begin:ss before :ss
add checkpoint $end after self
add checkpoint $control before :e
end pattern

## Comando For...To
pattern ForTo
var
:v as  [variable_reference]
:at as  [attrib_sign]
:einit as  [expr]
:e as  [expr]
:s as  [stmt]
match
[stmt< for :v :at :einit to :e do :s >]
declare node $control
declare node $initialization
declare node $increment
graph
$begin ->  $initialization
$initialization ->  $control
$control ->  $begin:s
$control ->  $end
$end:s ->  $increment
$increment ->  $control
assignment
assign $raise:s to  $exit
assign $definition:v to  $initialization
assign $cusage:einit to  $initialization
assign $cusage:e to  $increment
```

```
assign $definition:v to  $increment
assign $pusage:v to  $control
assign $break:s to  $end
instrument
add checkpointfor $control before :e
add checkpoint $begin before self
add checkpoint $begin:s before :s
add checkpoint $end after self
end pattern

## Comando For...Downto
pattern ForDownto
var
:v as  [variable_reference]
:at as  [attrib_sign]
:einit as  [expr]
:e as  [expr]
:s as  [stmt]
match
[stmt< for :v :at :einit downto :e do :s >]
declare node $control
declare node $initialization
declare node $increment
graph
$begin ->  $initialization
$initialization ->  $control
$control ->  $begin:s
$control ->  $end
$end:s ->  $increment
$increment ->  $control
assignment
assign $raise:s to  $exit
assign $definition:v to  $initialization
assign $cusage:einit to  $initialization
assign $cusage:e to  $increment
assign $definition:v to  $increment
assign $pusage:v to  $control
assign $break:s to  $end
instrument
add checkpointfor $control before :e
add checkpoint $begin before self
add checkpoint $begin:s before :s
add checkpoint $end after self
end pattern

## Comando With sem parênteses
pattern With1
var
:vl as  [variable_list]
:s as  [stmt]
match
[stmt< with :vl do :s >]
graph
$begin ->  $begin:s
```

```
$end:s -> $end
assignment
assign $raise:s to $exit
instrument
add checkpoint $begin before self
add checkpoint $end after self
end pattern

## Comando With com parênteses
pattern With2
var
:vl as [variable_list]
:s as [stmt]
match
[stmt< with ( :vl ) do :s >]
graph
$begin -> $begin:s
$end:s -> $end
assignment
assign $raise:s to $exit
instrument
add checkpoint $begin before self
add checkpoint $end after self
end pattern

## Comando Case
pattern CaseStatement
var
:e as [expr]
:cl as [case_list]
:s as [stmt_list]
match
[stmt< case :e of :cl else :s end >]
graph
$begin -> $begin:cl
$begin -> $begin:s
$end:cl -> $end
$end:s -> $end
assignment
assign $raise:s to $exit
assign $pusage:e to $begin
instrument
add checkpoint $begin before self
add checkpoint $begin:s before :s
add checkpoint $end after self
end pattern

## Lista de casos dentro do comando Case.
pattern CaseList
var
:cl as [case_list]
match
[case_list< :cl >]
graph
```

```
$begin -> $begin:cl
$end:cl -> $end
end pattern

## Cada caso dentro da lista de casos do comando Case.
pattern Case
var
:s as [stmt]
:la as [case_label_list]
match
[case< :la : :s >]
graph
$begin -> $begin:s
$end:s -> $end
assignment
assign $raise:s to $exit
instrument
add checkpoint $begin:s before :s
end pattern

## Este passo encontra expressões que nao são usadas de forma
## predicativa e conecta os seus nós begin e end.
step Expressions

## As expressões utilizadas nas atribuicões não são usadas
## de forma predicativa, apenas uso computacional.
pattern Expression1
var
:v as [variable_reference]
:at as [attrib_sign]
:e as [expr]
match
[stmt< :v :at :e >]
graph
$begin -> $end
assignment
assign $definition:v to $begin
assign $cusage:e to $control
end pattern

## Este passo marca o uso e definições de variáveis.
step Marks TB

## Ignora os identificadores que são utilizados como nome
## de funções ou procedimentos
pattern SkipFunction
var
:v as [variable]
:args as [expr_list]
match
[proc_call< :v ( :args ) >]
assignment
assign $cusage:v to $null
assign $pusage:v to $null
```

```
end pattern

## Marca a definição de uma variável em uma expressão de
## atribuição.
pattern MarkExpression
var
:v as  [variable_reference]
:at as  [attrib_sign]
:e as  [expr]
match
[attribution< :v :at :e >]
graph
mark definition of :v at $definition
assignment
assign $cusage:v to  $null
assign $pusage:v to  $null
end pattern

## Marca a dereferenciação de uma variável.
pattern MarkDeref
var
:v as  [variable]
match
[variable< :v ^ >]
graph
mark definition of :v at $derefdefinition
assignment
assign $cusage:v to  $null
assign $pusage:v to  $null
end pattern

## Marca o uso de uma variável.
pattern MarkUsage
var
:v as  [variable]
match
[variable_reference< :v >]
graph
mark cusage of :v at  $cusage
mark pusage of :v at  $pusage
mark definition of :v at  $parameterdefinition
end pattern

###############################################
#### Secao 3) Implementacao                ####
###############################################
implementation

## Insere um checkpoint antes de uma expressao
implement
var
:e as [expr]
:n as  [constant]
checkpoint $node before
```

```
[expr< :e >]
binding :n to node $node
as
[expr< check( :n ) and ( :e ) >]
end implement



implement
var
:e as [expr]
:n as  [constant]
checkpointfor $node before
[expr< :e >]
binding :n to node $node
as
[expr< checkfor( :n ) + ( :e ) >]
end implement




## Insere um checkpoint depois de um statement
implement
var
:s as  [stmt]
:n as  [constant]
checkpoint $node after
[stmt< :s >]
binding :n to node $node
as
[stmt< begin :s ; writeln ( TraceFile , :n ) end >]
end implement

## Insere um checkpoint antes de um statement
implement
var
:s as  [stmt]
:n as  [constant]
checkpoint $node before
[stmt< :s >]
binding :n to node $node
as
[stmt< begin writeln(TraceFile, :n); :s end >]
end implement

## Insere um checkpoint depois de uma lista de statements
implement
var
:s as  [stmt_list]
:n as  [constant]
checkpoint $node after
[stmt_list< :s >]
binding :n to node $node
```

```
as
[stmt< begin :s; begin writeln(TraceFile, :n) end end >]
end implement

## Insere um checkpoint antes de uma lista de statements
implement
var
:s as  [stmt_list]
:n as  [constant]
checkpoint $node before
[stmt_list< :s >]
binding :n to node $node
as
[stmt< begin writeln(TraceFile , :n ); begin :s end end >]
end implement

## Fecha o arquivo antes de um halt.
implement
var
:h as  [halt_stmt]
halt $node before
[stmt< :h >]
as
[stmt< begin closefile(TraceFile) ; :h end>]
end implement

## Abre um arquivo de trace assim que comeca uma funcao e
## define a função check, que serve para gravar no arquivo
## de trace.
implement
var
:head as  [name_impl]
:name as  [func_or_proc_or_method_name]
:type as  [proc_or_func_fptype]
:pars as  [fp_list]
:decl as  [impl_decl_sect_list]
:ss as  [stmt_list]
:file as  [string]
:n as  [constant]
init $init before
[main_impl< :head :name :pars :type ; :decl begin :ss end ; >]
binding :n to node $init
binding :file to literal [string< '[:name].trace.tc' >]
as
[main_impl<:head :name :pars :type ; :decl
var TraceFile : TextFile;

function check(n : integer) : boolean;
begin
writeln(TraceFile, n);
check := true;
end;

function checkfor(n : integer) : integer;
```

```
begin
writeln(TraceFile, n);
checkfor := 0;
end;

begin
assignfile(TraceFile, :file);
if fileexists( :file ) then
  reset ( TraceFile )
else
  rewrite ( TraceFile );
append(TraceFile);
begin
:ss
end
end ;>]
end implement

## Fecha o arquivo assim que termina a funcao
implement
var
:head as  [name_impl]
:name as  [func_or_proc_or_method_name]
:type as  [proc_or_func_fptype]
:pars as  [fp_list]
:decl as  [impl_decl_sect_list]
:ss as  [stmt_list]
:file as  [string]
:n as  [constant]
exit $exit after
[main_impl< :head :name :pars :type ; :decl
var TraceFile : TextFile;

function check(n : integer) : boolean;
begin
writeln(TraceFile, n);
check := true;
end;

function checkfor(n : integer) : integer;
begin
writeln(TraceFile, n);
checkfor := 0;
end;

begin
assignfile(TraceFile, :file);
if fileexists( :file ) then
  reset ( TraceFile )
else
  rewrite ( TraceFile );
append(TraceFile);
begin
:ss
```

```
end
end ; >]
binding :n to node $exit
binding :file to literal [string< '[:name].trace.tc' >]
as
[main_impl< :head :name :pars :type ; :decl

var TraceFile : TextFile;

function check(n : integer) : boolean;
begin
writeln(TraceFile, n);
check := true;
end;

function checkfor(n : integer) : integer;
begin
writeln(TraceFile, n);
checkfor := 0;
end;

begin
assignfile(TraceFile, :file);
if fileexists( :file ) then
  reset ( TraceFile )
else
  rewrite ( TraceFile );
append(TraceFile);
begin
:ss ;
writeln(TraceFile, '----------------');
closefile(TraceFile)
end
end ;>]
end implement

end instrumenter
```