FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO “EURÍPIDES DE MARÍLIA” – UNIVEM
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GUSTAVO GARCIA RONDINA

INSTRUMENTADOR DE CÓDIGO PARA A LINGUAGEM DELPHI

MARÍLIA
2005

GUSTAVO GARCIA RONDINA

INSTRUMENTADOR DE CÓDIGO PARA A LINGUAGEM DELPHI

Monografia apresentada como Trabalho de Conclusão de Curso ao Curso de Bacharelado em Ciência da Computação, mantido pela Fundação de Ensino Eurípides Soares da Rocha, para a obtenção do título de Bacharel em Ciência da Computação. (Área de Concentração: Engenharia de Software).

Orientador:
Prof. Dr. Marcio Eduardo Delamaro

MARÍLIA
2005

RONDINA, Gustavo Garcia

Instrumentador de código para a linguagem Delphi / Gustavo Garcia Rondina; orientador: Prof. Dr. Marcio Eduardo Delamaro. Marília, SP: [s.n.], 2005.

122 f.

Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília – Fundação de Ensino Eurípides Soares da Rocha.

1. Engenharia de Software 2. Teste e Manutenção

CDD: 005.10287

GUSTAVO GARCIA RONDINA

INSTRUMENTADOR DE CÓDIGO PARA A LINGUAGEM DELPHI

Banca examinadora do trabalho de conclusão de curso apresentado ao Curso de Bacharelado em Ciência da Computação da UNIVEM/F.E.E.S.R., para a obtenção do Título Bacharel em Ciência da Computação. Área de Concentração: Engenharia de Software.

Resultado: _____

ORIENTADOR: Prof. Dr. _____

1º EXAMINADOR: _____

2º EXAMINADOR: _____

Marília, 30 de Novembro de 2005.

Aos meus Pais,
que compreendem e apóiam meus anseios acadêmicos.

AGRADECIMENTOS

Agradeço primeiramente aos meus pais e irmãos, por me oferecem todo o suporte e afeto necessários para que eu atingisse mais um importante objetivo. Sem eles nada disso teria sido possível.

Sou imensamente grato também ao Professor Marcio Delamaro, não somente pelos quatro anos de excelente orientação, mas principalmente pela amizade e compreensão nos momentos difíceis. Ele é um exemplo que sempre carregarei comigo, onde quer que eu vá.

Agradeço aos meus bons e velhos amigos Rodrigo Araújo, Marcelo Rossi, Luiz Raphael e Alison Barros, com os quais compartilhei ótimos momentos ao longo da graduação. Juntos aprendemos e crescemos muito, pessoal e profissionalmente.

Não posso deixar de expressar a gratidão que tenho com a 5º turma da Ciência da Computação. Nos quatro anos que se passaram nós rimos muito, rimos até mesmo dos nossos próprios infortúnios. Tive a chance de conhecer pessoas muito peculiares (muitas vezes excêntricas) e muito fiéis, amigos para se levar por toda uma vida.

Finalmente, Agradeço a todos os professores do curso de Bacharelado em Ciência da Computação do UNIVEM por sempre enxergarem em mim um potencial maior do que eu acredito possuir e me fazerem crer que basta lutar muito para atingir um objetivo, por mais distante e impossível que ele pareça.

"I do not fear computers. I fear the lack of them."
Isaac Asimov

RONDINA, Gustavo Garcia. **Instrumentador de código para a linguagem Delphi.** 2005. 122 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2005.

RESUMO

Diversas atividades da Engenharia de Software exigem uma análise detalhada do código fonte dos programas enquanto os mesmos são executados. A técnica Estrutural de teste de Software, por exemplo, é baseada em critérios que avaliam a cobertura dos programas testados para garantir que cada instrução do programa é executada pelo menos uma vez. Para que isso seja possível é necessário submter o programa à um processo chamado instrumentação, responsável pela alteração do programa. Essa alteração grava informações relevantes em um arquivo de trace que permitem que a execução do programa seja acompanhada, tão bem como obter um grafo orientado que representa o aspecto estático do programa. O projeto apresentado por este trabalho visa implementar um instrumentador de código para a linguagem Delphi com o auxílio da linguagem de descrição de instrumentação IDeL, que generaliza e torna mais eficiente o processo de desenvolvimento de instrumentadores de código.

Palavras-chave: Instrumentação. Linguagem Delphi. Teste de Software. IDeL.

RONDINA, Gustavo Garcia. **Instrumentador de código para a linguagem Delphi.** 2005. 122 f. Trabalho de Conclusão de Curso (Bacharelado em Ciência da Computação) – Centro Universitário Eurípides de Marília, Fundação de Ensino Eurípides Soares da Rocha, Marília, 2005.

ABSTRACT

Several activities in Software Engineering demand a detailed analysis of the programs while they are being executed. The Structural technique of Software Testing, for example, is based on criterions that evaluate the code coverage of the programs under test to assure that every statement is executed at least once. To make this possible it is necessary to submit the program to a process named code instrumentation, which is responsible for the decoration of the program's source code. This decoration records relevant information in a trace file that allows one to keep track of the program execution as well to derive an oriented graph that represents the program static aspect. The project presented in this work intends to implement a code instrumentator to the Delphi programming language with the aid of the IDeL instrumentation description language, which makes the code instrumentator development process more efficient and general.

Keywords: Instrumentation. Delphi Programming Language. Software Testing. IDeL.

LISTA DE FIGURAS

SUMÁRIO

CAPÍTULO 1 – INTRODUÇÃO

O Projeto de Conclusão de Curso apresentado por este trabalho espera contribuir com a área de engenharia de software através do fornecimento de um instrumentador[1] de código para a linguagem Delphi. Um instrumentador de código é responsável por alterar o código fonte de um programa por meio da inserção de instruções especiais que não alteram a semântica do programa. Tais instruções monitoram a execução do programa enquanto o mesmo está sendo executado, permitindo assim que seja possível identificar quais partes do código foram exercitadas. Tal ferramenta é indispensável quando se trata de critérios de teste caixa-branca, que necessitam de informações referentes à execução dos programas testados. As duas seções seguintes apresentam os objetivos e a motivação do trabalho.

1.1. Objetivos

Este trabalho tem como principal objetivo o desenvolvimento de um instrumentador de código para a linguagem Delphi. O desenvolvimento do instrumentador é exaustivo e complexo, pois a aplicação final deve estar de acordo com a sintaxe da gramática da linguagem Delphi. A fim de simplificar o processo de implementação, uma linguagem de descrição de instrumentação chamada IDeL foi utilizada. Com essa linguagem é possível generalizar a implementação, diminuindo a complexidade do projeto e por conseqüência o tempo gasto.

Uma vez concluído, o projeto serve também como guia de referência da linguagem IDeL. Futuras empreitadas encontrarão um grau de dificuldade reduzido no que se refere aos

[1] A tarefa de instrumentação e a ferramenta instrumentadora são exiplicadas com mais detalhes na Seção 3.3.1.

aspectos operacionais e detalhes técnicos da linguagem e do seu compilador IDeLGen, pois este trabalho trata tais aspectos. Uma outra conseqüência advinda da conclusão do projeto é a abertura de portas para novas ferramentas de teste para a linguagem Delphi, já que a maioria dos critérios da técnica de Teste Estrutural exige algum tipo de análise de cobertura.

1.2. Motivação

A única forma de assegurar que um software possui qualidade é através de exaustivos testes suportados por técnicas cuja eficácia está empiricamente comprovada na literatura. Por ser uma tarefa extensa, existem ferramentas que servem de apoio à atividade de teste. Linguagem Delphi, apesar de muito popular e comercialmente aceita como confiável, apresenta uma carência de ferramentas de suporte ao teste de software. Além de (DUNIT, 2005), nenhuma abordagem automatizada foi encontrada para testar programas Delphi. Quando se fala especificamente na técnica de Teste Estrutural, a carência é ainda maior. Com o desenvolvimento do instrumentador espera-se contribuir com a construção de uma ou mais ferramentas de teste para a linguagem Delphi.

1.3 Organização do Trabalho

Este trabalho está dividido em 7 capítulos. O presente Capítulo apresenta a introdução, objetivos e motivação do projeto. O Capítulo 2 tem o intuito de introduzir o leitor às principais construções e conceitos da linguagem de programação Delphi. O Capítulo 3 explica em detalhes alguns conceitos relacionados à atividade de teste. O Capítulo 4 contém a teoria básica referente à gramáticas e às ferramentas Lex e YACC. O Capítulo 5 introduz a

linguagem de descrição de instrumentadores IDeL, que é explicada em mais detalhes por meio da própria implementação do instrumentador no Capítulo 6. Por fim, o Capítulo 7 apresenta as conclusões e considerações finais, tão bem como os trabalhos futuros possíveis a partir deste mesmo.

CAPÍTULO 2 - DELPHI

2.1. Considerações Iniciais

Neste Capítulo procura-se apresentar os conceitos gerais da linguagem de programação Delphi (BORLAND, 2002), como a forma em que os módulos das aplicações são dispostos, o relacionamento entre estes módulos e a constituição básica de um programa Delphi. A compreensão de tais tópicos é de grande valia para o completo entendimento do restante deste trabalho, já que o desenvolvimento de um instrumentador de código está intimamente relacionado com a linguagem em que os programas instrumentados estão escritos.

Todavia, foge do escopo deste Capítulo e deste trabalho apresentar os aspectos operacionais da ferramenta Borland Delphi (BORLAND 2002) ou de qualquer outra ferramenta relacionada ao desenvolvimento de programas e componentes na linguagem Delphi. O foco reside no código propriamente dito e nas construções gramaticais da linguagem, e não em uma implementação específica de um compilador. Foge também do escopo a apresentação de conceitos básicos de programação estruturada e de programação na linguagem Pascal.

2.2. Linguagem Delphi

Delphi é uma linguagem de programação compilada de alto nível e fortemente tipada que oferece diversos benefícios aos programadores, incluindo a produção de código compacto e de fácil leitura, tempo de compilação reduzido se comparado com outras linguagens, e a utilização de múltiplos arquivos que permitem uma programação modular e eficiente. Devido

ao fato de ser derivada da linguagem Object Pascal (BORLAND, 1993), a linguagem Delphi foi projetada para suportar o modelo de desenvolvimento orientado a objetos. Esse paradigma tem se destacado tanto no meio acadêmico quanto no setor comercial por oferecer diversos aperfeiçoamentos no processo de produção de software (SEBESTA, 1999), como a possibilidade de reutilização de código e uma forma mais natural de modelar problemas do mundo real.

2.3. Organização do Programa

Os programas Delphi normalmente são divididos em vários módulos chamados de `Units` (Unidades), cada qual localizado em um arquivo diferente, dessa forma é possível modularizar o desenvolvimento de aplicações. Cada programa é iniciado com um cabeçalho cuja função é especificar seu nome. Logo após o cabeçalho pode ocorrer uma cláusula **`uses`**, que é opcional, seguida de um bloco de declarações e de um bloco de instruções.

A cláusula **`uses`** é empregada para listar `Units` que estão ligadas ao programa em questão, ou seja, cujos procedimentos, funções e métodos estão sendo utilizados, da mesma forma que se utiliza bibliotecas em outras linguagens de programação para o reaproveitamento de código. Dessa forma as `Units` podem ser compartilhados por diversos programas e evita-se a replicação do mesmo código em arquivos diferentes. Além disso, essa cláusula fornece ao compilador informações referentes a dependências existentes entre os módulos. Como essas informações são armazenadas nos próprios módulos, a maioria dos programas escritos na linguagem Delphi não requerem *makefiles*, arquivos de cabeçalho, ou diretivas do tipo *include* para o pré-processador, como ocorre com a linguagem C, por exemplo.

2.4. Código Fonte Delphi

O compilador Delphi espera encontrar três tipos de arquivos fontes: **i)** arquivos que contém `Units` (cuja extensão é `.pas`); **ii)** arquivos de projeto (cuja extensão é `.dpr`); e **iii)** arquivos que representam pacotes (cuja extensão é `.dpk`). Um programa Delphi deve conter um único arquivo de projeto, opcionalmente um arquivo de pacotes e diversos arquivos `Units`, nos quais se localiza quase que todo o código da aplicação. Dessa forma, esse Capítulo irá abordar apenas os arquivos que contém `Units` e o arquivo de projetos, que possui uma quantidade mínima código fonte.

Existem outros tipos de arquivos que são produzidos ou necessários quando se utiliza uma implementação específica da linguagem Delphi ou um ambiente de desenvolvimento integrado em particular, porém não cabe a este trabalho descrever tais arquivos e ambientes. Informações detalhadas podem ser encontradas em (BORLAND, 1993), (BORLAND, 2002) e (BORLAND, 2004).

2.5. Programas e Units

Como apresentado anteriormente, uma aplicação Delphi é constituída de vários módulos chamados de `Units`. As `Units` são ligadas através de um módulo de código especial que contém o cabeçalho do programa. Este arquivo pode ser visto como o módulo central da aplicação, responsável pela conexão e comunicação de todos os outros módulos, como a função `main()` de um programa C. Cada `Unit` se localiza em seu próprio arquivo fonte que é compilado separadamente. Após a compilação, as `Units` são ligadas às bibliotecas padrão e a aplicação é criada (BORLAND, 2004).

2.5.1. Estrutura e Sintaxe de um Programa

O módulo principal de uma aplicação Delphi é armazenado num arquivo fonte cuja extensão é `.dpr` (acrônimo para *Delphi Project Resource* – Arquivo de Projeto Delphi), enquanto o restante do código fonte da aplicação é armazenado em arquivos `Units` do tipo `.pas`, remetendo à extensão de arquivos escritos na linguagem Pascal tradicional. Para construir um projeto o compilador precisa tanto do arquivo fonte principal quanto das `Units`, estejam já pré-compilados ou não.

O arquivo fonte principal (isto é, o arquivo DPR) apresenta o cabeçalho do programa, contendo o nome do programa a ser construído, opcionalmente uma cláusula **`uses`**, além de um bloco de declarações e um bloco de instruções executáveis. Na Figura 1 é apresentado o arquivo fonte principal de uma aplicação chamada `Calculadora`.

```
01    program Calculadora;
02
03    uses
04          REAbout in 'REABOUT.PAS' {AboutBox},
05          REMain in 'REMain.pas' {MainForm};
06
07    {$R *.RES}
08
09    var
10          t : string;
11
12    begin
13          t := 'Calculadora';
14          Application.Title := t;
15          Application.CreateForm(TMainForm, MainForm);
16          Application.Run;
17    end.
```

Figura 1 – Exemplo de uma calculadora – Arquivo principal

A linha 1 contém o cabeçalho do programa, que consiste na palavra chave **`program`** seguida de um identificador válido e de um ponto-e-vírgula. O identificador é o próprio nome

da aplicação, que nesse caso é `Calculadora`, além disso o compilador Delphi exige que este identificador seja também o nome do arquivo. Nesse exemplo o arquivo de projeto seria nomeado como `CALCULADORA.DPR` (a linguagem Delphi não é sensível à caixa das letras). A cláusula **`uses`** se estende da linha 3 até a linha 5, e especifica que existe a dependência de três `Units` adicionais: `REAbout` e `REMain`, localizadas nos arquivos `READOUT.PAS` e `REMain.pas`, respectivamente. Como visto, é possível especificar mais de uma `Unit` com a mesma cláusula **`uses`**. A linha 7 contém a diretiva **`$R`**, utilizada para relacionar os arquivos de ambiente do projeto com o programa. As linhas 9 e 10 contém o bloco de declarações, e as linhas 12 a 17 contém o bloco de instruções executáveis, delimitado pelas palavras chaves **`begin`** e **`end`**.

O código fonte apresentado na Figura 1 constitui um arquivo de projeto típico de uma aplicação Delphi. Normalmente estes arquivos são curtos, pois a maior parte da lógica da aplicação está inserida nos arquivos que contém as `Units`, na forma de procedimentos e funções (BORLAND, 2002).

2.5.2. Estrutura e Sintaxe de uma Unit

Um arquivo contendo uma `Unit` consiste de tipos (incluindo classes), constantes, variáveis e rotinas (procedimentos e funções) agrupadas a fim de realizar uma tarefa específica e propriamente delimitada. Cada `Unit` é definida em um arquivo separado cuja extensão é `.pas`. A Figura 2 ilustra a estrutura básica de uma `Unit`.

Como pode-se observar, a sintaxe da `Unit` é semelhante àquela utilizada no arquivo de projeto. O cabeçalho da `Unit` contido na linha 1 especifica o seu nome, através de um

identificador válido da linguagem[2]. Os nomes das `Units` devem ser únicos em um projeto Delphi, ou seja, não pode haver duas `Units` homônimas.

```
1     unit UnitPadrao;
2
3     interface
4     uses // Lista de dependências
5
6     // Código da Interface
7
8     implementation
9     uses // Lista de dependências
10
11          // Implementção de métodos, procedimentos e funções.
12
13    initialization
14
15    // Código de inicialização da Unit.
16
17    finalization
18
19    // Código de finalizacão da Unit
20
21    end.
```

Figura 2 – Estrutura de uma Unit

As linhas de 3 a 6 delimitam uma seção de Interface, que pode inicia-se com a palavra chave **`interface`** e se estende até a ocorrência da palavra chave **`implementation`**. Nessa seção são declaradas constantes, tipos, variáveis, procedimentos e funções que estarão disponíveis para outros módulos do programa. Essas entidades são denominadas públicas pois podem estar presentes no código de qualquer outra `Unit`. A declaração de um procedimento ou de uma função dentro da seção de interface consiste apenas na assinatura da rotina, isto é, em seu nome, seus parâmetros e seu tipo de retorno (caso seja uma função). O bloco de instruções contido no corpo da rotina deve estar presente na seção de implementação (BORLAND, 1993).

[2] Para aplicações desenvolvidas com ferramentas Borland, o nome da Unit deve coincidir com o nome do arquivo em que a mesma foi salva.

A seção de implementação estende-se da linha 8 até a linha 11 e é iniciada pela palavra reservada **`implementation`**, prosseguindo até o começo da seção de inicialização (caso não haja uma seção de inicialização, prossegue até o final da `Unit`). O bloco de implementação define os procedimentos e funções que foram declarados na seção de interface. As definições e as chamadas das rotinas podem ocorrer em qualquer ordem, sendo possível omitir a lista de parâmetros das funções e dos procedimentos públicos quando a mesma estiver presente na seção de interface. Se uma lista de parâmetros for incluída, esta deve ser idêntica àquela especificada na declaração da rotina presente na seção de interface. Além de conter a definição de rotinas públicas, a seção de implementação pode possuir declaração de constantes, tipos, variáveis e rotinas privadas, isto é, que só existem no escopo do arquivo onde a `Unit` está definida (BORLAND, 1993).

A seção de inicialização começa na linha 13 com a palavra reservada **`initialization`** e prossegue até o início da seção de finalização, se a mesma existir. Caso não exista uma seção de finalização, a inicialização estende-se até o fim do arquivo. Esta é uma seção opcional e contém instruções que são executadas no inicio da execução do programa ou de um módulo que utiliza essa `Unit`. É muito comum utilizar este bloco para inicializar estrutura de dados, arquivos e variáveis que são utilizadas pelas rotinas da `Unit`. As seções de inicialização de cada `Unit` são executadas na mesma ordem em que ocorrem na cláusula **`uses`** de cada módulo que as utiliza.

Por fim ocorre a seção de finalização, iniciada na palavra chave **`finalization`** (linha 17) e terminada no final do arquivo da `Unit`. Esta seção é opcional e só pode existir se houver também uma seção de inicialização. No bloco de finalização são inseridos as instruções que serão executadas quando o programa estiver finalizando sua execução. É muito comum colocar nessa seção códigos referentes a liberação de memória ou de outros recursos do sistema que foram alocados previamente. As seções de finalização de cada `Unit` são

executadas na ordem oposta em que foram executadas durante a inicialização. Uma vez que a seção de inicilização de uma `Unit` é executada, é garantido que a seção de finalização da mesma `Unit` será executada quando a aplicação for terminada. Porém, se houver um erro de execução, o código dessa seção pode não ser executado completamente.

Após a compilação do arquivo `MinhaUnidade.pas`, contendo uma `Unit` chamada `MinhaUnidade`, será gerado um arquivo chamado `MinhaUnidade.dpu` ou `MinhaUnidade.dcu`, dependendo da implementação do compilador Delphi empregado. Após a compilação de todas as `Units` e do módulo principal, todos os arquivos devem ser ligados através de um *linker* que irá gerar um único arquivo executável. Detalhes referentes à compilação e ligação de programas Delphi podem ser encontrados em (BORLAND, 2002) e (BORLAND, 2004).

2.5.3. Exemplo de uma Unit

Para fins ilustrativos, a Figura 3 apresenta uma `Unit` chamada `UPower`, que contém apenas uma função chamada `power(x, y)` (SIMÃO et al. A, 2002) que retorna o valor de x elevado ao expoente y. Essa `Unit` apresentada é mínima, porém funciona corretamente quando compilada em qualquer implementação da linguagem Delphi.

Para uma outra `Unit` aproveitar o código contido na Figura 3, basta utilizar a cláusula `uses` seguida do identificador `UPower`, assim será possível utilizar a função `power(x, y)` como se a ela estivesse definida no mesmo arquivo, sem a necessidade de replicação de código.

```
 1     unit UPower;
 2
 3     interface
 4
 5     function power(x : integer; y : integer) : integer;
 6
 7     implementation
 8
 9     function power(x : integer; y : integer) : integer;
10     var
11           p, z : integer;
12     begin
13           if (y < 0) then
14                p := y
15           else
16                p := -y;
17
18           z := 1;
19
20           while (p <> 0) do
21           begin
22               z := z * x;
23               p := p + 1
24           end;
25
26           if y < 0 then
27               z := 1 div z;
28
29          power := z
30     end;
31     end.
```

Figura 3 – Unit UPower - A função power(x, y) calcula xy.

2.5.4. Considerações Finais

As estruturas e construções da linguagem Delphi são similares em muitos aspectos àquelas presentes na linguagem Pascal e ObjectPascal. Dessa forma, não cabe a este trabalho descrever cada uma dessas estruturas. Quando necessário, são apresentados mais detalhes referentes a certos pontos da linguagem necessários para a compreensão de determinadas partes desta monografia.

Informações aprofundadas a respeito da sintaxe e da semântica da linguagem Pascal e ObjectPascal podem ser encontradas em (WIRTH, 1985) e (BORLAND, 1993),

respectivamente. O restante deste trabalho presume que o leitor possua conhecimentos básicos de programação na linguagem Pascal, como estruturas de controle, estruturas de repetição, procedimentos e funções, além de conceitos de programação estruturada e programação orientada a objetos. No Apêndice A pode ser encontrada a gramática da linguagem Delphi no formato da ferramenta `YACC`, utilizada para descrever e construir compiladores. Como as ações semânticas foram eliminadas dessa descrição, a forma em que a gramática é apresentada é muito similar a um diagrama BNF convencional.

Apesar de comercialmetne aceita, a linguagem Delphi é carente de ferramentas de apoio à atividade de teste de software. Além de (DUNIT, 2005), que apresenta uma ferramenta de teste baseada na metodologia *Extreme Programming*, nenhuma outra ferramenta foi encontrada. Esse é um problema sério, uma vez que a atividade de teste de software é imprescindível para a garantia da qualidade de uma aplicação.

CAPÍTULO 3 – TESTE DE SOFTWARE

Um software possui qualidade quando ele atende a todos os requisitos funcionais, implícitos ou explícitos, presentes em sua especificação (PRESSMAN, 1997). A melhor maneira de demonstrar que um programa cumpre sua especificação é através do teste de software (RAPPS e WEYUKER, 1985). Segundo (MYERS, 1979) a atividade de teste tem o objetivo de executar um programa com o intuito de encontrar um erro. Se o teste for executado com sucesso, ele irá descobrir erros. Além disso, os dados obtidos com o teste fornecem uma boa indicação da confiabilidade do software, aumentando assim sua qualidade.

A atividade de teste pode ser dividida em quatro etapas: planejamento do teste, projeto dos casos de teste, execução e avaliação dos resultados do teste (MALDONADO, 2001). Para a correta realização da atividade de teste se faz necessária a aplicação de técnicas, critérios e estratégias disciplinadas e teoricamente fundamentadas que auxiliam na elaboração de casos de teste que possuem elevadas chances de encontrar um erro ainda não descoberto. É importante ter em mente, porém, que a atividade aponta a presença de erros e defeitos, e não a ausência dos mesmos.

Os critérios de teste de software para a elaboração de casos de teste são estabelecidos de acordo com diferentes abordagens. Uma delas é conhecida como teste de caixa-preta, ou Teste Funcional, que se preocupa em assegurar que o software satisfaça os requisitos funcionais e não-funcionais determinados por sua especificação, verificando se determinados dados de entrada e produzem as saídas esperadas, testando assim se as funções do mesmo são operacionais. O teste funcional também é conhecido como teste comportamental e geralmente é conduzido na interface do software, não se preocupando com os detalhes da implementação. Como exemplos de critérios pertinentes à técnica funcional podem ser citados o Particionamento em Classes de Equivalência, Análise do Valor Limite e o Grafo de Cause-

Efeito (PRESSMAN, 1997).

Outra abordagem existente é a técnica baseada em erros, onde são levantados os erros mais comuns cometidos pelos programadores durante o desenvolvimento do software para derivar requerimentos do teste (MALDONADO, 2001). A Análise de Mutantes (DELAMARO, 1993) é um exemplo de critério que utiliza esta técnica.

A técnica de teste estrutural, também conhecida como teste de caixa-branca, visa testar a estrutura interna lógica do software. Os casos de teste, derivados utilizando detalhes do projeto estrutural do software, garantem que todos os caminhos independentes em um módulo sejam executados, que todas as decisões lógicas assumam valores verdadeiros e falsos, que todos os comandos de repetição sejam executados e que as estruturas de dados do programa sejam válidas.

Cada uma das técnicas compreende um conjunto de critérios que definem os requisitos necessários que os casos de teste precisam cobrir. Vale ressaltar ainda que uma técnica não é alternativa à outra, elas são abordagens complementares, sendo que cada uma cobre diferentes classes de erros (PRESSMAN, 1997; MALDONADO 2001) A abordagem estrutural, por exemplo, identifica erros na estrutura lógica e procedural do programa, enquanto a funcional identifica funções do software que não estão de acordo com a especificação, erros na interface, erros nas estruturas de dados e no acesso aos dados externos.

3.1. Teste Estrutural

Este trabalho de conclusão de curso visa desenvolver um instrumentador de código, necessário para realizar atividades de análise de cobertura essenciais para a técnica de teste estrutural. Dessa forma, esta técnica é descrita com mais detalhes durante as próximas seções.

3.1.1. Grafo de Fluxo de Controle

A maioria dos critérios envolvidos na técnica de teste estrutural utiliza uma representação gráfica do programa a ser testado, conhecida como Grafo de Fluxo de Controle (GFC). A partir da representação gráfica do programa é possível determinar quais componentes serão executados e definir os fundamentos para os critérios estruturais e os requisitos de teste. Um GFC é um grafo orientado cuja função descrever o fluxo de controle lógico de um programa. A Figura 4 ilustra a notação utilizada para construir um GFC a partir das estruturas lógicas presentes no código fonte de um programa.

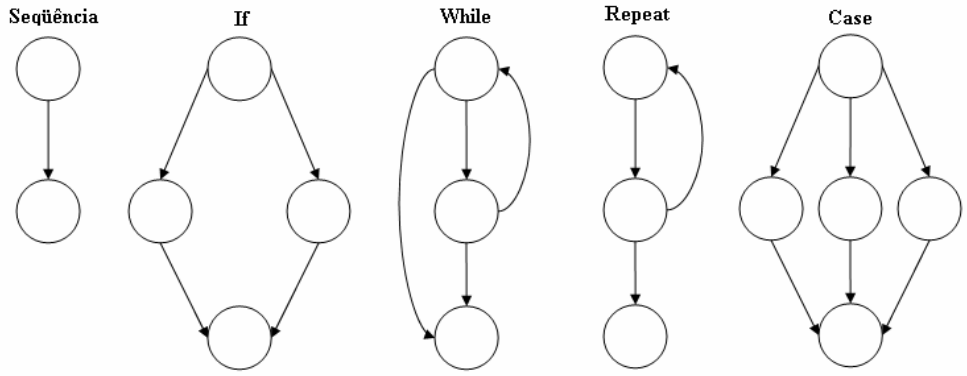

Figura 4 – Notação utilizada para construir um CFG – Principais estruturas

Cada nó do GFC representa um bloco de comandos do programa, que são executados de maneira atômica, ou seja, a execução de um comando pertencente ao bloco faz com que todo o bloco seja executado, sem desvios de fluxo dentro de um bloco (RAPPS e WEYUKER, 1985). Os comandos do bloco são seqüenciais e possuem apenas um predecessor e um sucessor, com exceção do primeiro comando e do último, respectivamente. Cada GFC possui apenas um nó de entrada e um nó de saída. Os nós do grafo são conectados

através de arestas ou arcos orientados, que denotam a transferência de controle de fluxo do programa entre diferentes blocos. Um nó que possui um comando condicional é denominado nó predicativo e dele partem pelo menos duas arestas.

No grafo, um caminho é qualquer seqüência finita de nós. Um caminho simples é uma seqüência finita de nós onde todos os nós pertencentes a este caminho são distintos, exceto possivelmente o primeiro e o último. Se todos os nós são distintos o caminho é dito livre de laço. Um caminho completo é aquele onde o primeiro nó é o nó de entrada, e o último nó é o nó de saída do grafo (RAPPS e WEYUKER, 1985).

A Figura 5(b) apresenta o GFC da função `power(x, y)` (SIMÃO et al. A), contida na `Unit` UPower da Figura 3. Nesse grafo (1, 2, 4, 5, 6, 5, 7, 8, 9) é um caminho e (1, 2, 4, 5, 6, 7, 8, 9) é um caminho livre de laço. A Figura 5(a) mostra novamente o código da função `power(x, y)`, só que agora com comentários que ilustram quais linhas de código correspondem a quais nós.

```
 1    function power(x : integer; y : integer) : integer;
 2    var
 3          p, z : integer;
 4    begin
 5          if (y < 0) then          // nó 1
 6                p := y             // nó 2
 7          else
 8                p := -y;           // nó 3
 9
10          z := 1;                  // nó 4
11
12          while (p <> 0) do        // nó 5
13          begin
14                z := z * x;        // nó 6
15                p := p + 1         // nó 6
16          end;
17
18          if y < 0 then            // nó 7
19               z := 1 div z;       // nó 8
20
21          power := z               // nó 9
22    end;
```

Figura 5(a) – Função power(x, y). Os comentários indicam os nós.

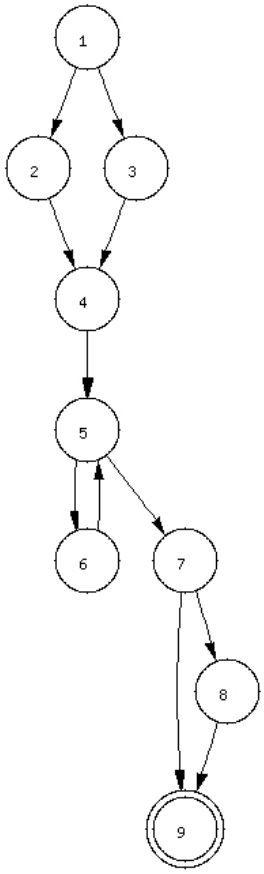

Figura 5(b) – GFC da função power(x, y) da Figura 5(a).

3.1.2. Critérios de Teste Estrutural

O teste estrutural engloba diversos critérios que são empregados durante a derivação dos requisitos de teste. Os principais são: critérios baseados na complexidade, critérios baseados em fluxo de controle e critérios baseados em fluxo de dados.

3.1.2.1. Critérios Baseados na Complexidade

Um destes critérios de teste estrutural foi proposto por (MCCABE, 1989) e consiste em determinar a complexidade de um programa procedural. A complexidade ciclomática é uma métrica que proporciona a medida quantitativa da complexidade lógica de um programa. Esta medida define o número de caminhos independentes existentes no CFG, ou seja, um caminho que introduz pelo menos um novo nó ou um novo arco. Os casos de teste são derivados, portanto, com o intuito de cobrir os caminhos existentes, executando cada instrução pelo menos uma vez (PRESSMAN, 1997).

3.1.2.2. Critérios Baseados em Fluxo de Controle

Estes critérios visam projetar casos de teste que executam as condições lógicas e desvios de fluxo, além de outras características de controle presentes em um programa, determinando assim os requisitos para o teste. Os critérios mais conhecidos são:

- **todos-nós**: exige que cada nó do CFG, ou seja, cada bloco de comandos seqüenciais, seja executado pelo menos uma vez.
- **todos-arcos**: requer que todos os comandos de transferência de controle e desvio de fluxo sejam exercitados pelo menos uma vez.
- **todos-caminhos**: requer que todos os caminhos de um programa sejam executados.

O critério todos-caminhos pode transmitir a idéia de que, quando aplicado, é possível garantir programas cem por cento livres de erros, uma vez que todas as estruturas lógicas do

programa foram executadas e verificadas. Porém, até mesmo nos programas mais simples o número de caminhos lógicos possíveis é muito alto, sendo necessário o desenvolvimento de uma quantidade excessiva de casos de teste, tornando a atividade excessivamente exaustiva e muitas vezes impraticável (PRESSMAN, 1997). Além disso, não é possível determinar antecipadamente se um caminho é ou não executável, e existem erros que o critério baseado em fluxo de controle não consegue descobrir (RAPPS e WEYUKER, 1985; PRESSMAN, 1997).

3.1.2.3. Critérios Baseados em Fluxo de Dados

Estes critérios determinam os requisitos de teste através da análise do fluxo de dados do programa, fazendo a seleção de caminhos baseando-se na localização das definições e dos usos das variáveis existentes, testando assim o resultado de toda e qualquer computação que ocorre no programa (RAPPS e WEYUKER, 1985; PRESSMAN, 1997). Os casos de teste são projetados com o intuito de executar os caminhos identificados. Os critérios baseados em fluxo de dados são complementares em relação ao critério baseado em fluxo de controle, uma vez que eles podem ser utilizados para descobrir erros relacionados à dependência de dados ou erros computacionais, através do teste das definições das variáveis e das referências a estas definições (MALDONADO et al, 2001; PRESSMAN, 1997).

Para auxiliar o emprego deste critério de teste foi proposto o Grafo Def-Uso (RAPPS e WEYUKER, 1985), que é uma extensão do GFC, onde são adicionadas informações referentes às ocorrências das variáveis nos vértices e arcos do grafo, podendo ser uma definição ou um uso. A partir do grafo Def-Uso é possível estabelecer os requerimentos do teste.

A definição de uma variável ocorre quando um valor é armazenado na porção de

memória que ela referencia, ou seja, quando a variável ocorre no lado esquerdo de uma expressão, em comandos de entrada ou quando são utilizadas para armazenar valores retornados por procedimentos ou funções.

Um uso significa que a variável foi referenciada e seu valor foi utilizado. Existem dois tipos distintos de usos, o uso computacional e o uso predicativo, chamados respectivamente de **c-uso** e **p-uso**. Um uso computacional afeta diretamente o resultado da computação onde ele aparece, como em uma operação aritmética, por exemplo. O uso predicativo determina uma alteração no fluxo de controle do programa, como quando uma variável está presente em uma expressão booleana ou relacional no predicado de um comando de desvio. O c-uso de uma variável *x* é considerado um c-uso global quando não há nenhuma definição da mesma dentro do nó onde ocorre o c-uso. Quando uma variável *x* possui um c-uso apenas no nó onde é definida, então este uso é considerado um c-uso local (RAPPS e WEYUKER, 1985).

Sendo a variável *x* definida no nó *i* de um grafo Def-Uso, o caminho cujo nó inicial é *i* e o nó final é *j* é classificado como livre de definição em relação a variável *x* se a mesma não é redefinida neste caminho. A notação *(i, j, x)* indica que a variável *x* é definida no nó *i* e existe um c-uso da mesma no nó *j*, sendo que o caminho do nó *i* ao nó *j* é livre de definição em relação à variável *x*. A notação *(i, (j, k), x)* indica que a variável *x* é definida no nó *i* e existe um p-uso da mesma no arco *(j, k)*, sendo que o caminho do nó *i* ao arco *(j, k)* é livre de definição em relação à variável *x*.

A definição de uma variável *x* no nó *i* é uma definição global se ela é a última definição de *x* que ocorre em *i* e existe um caminho livre de definição em relação a variável *x* que vai de *i* até um nó contendo um c-uso global de *x* ou até um arco contendo um p-uso de *x*. Dessa maneira, uma definição global define uma variável que pode ser utilizada fora do nó em que a definição ocorre. Se a definição de uma variável *x* em um nó *i* não for global, então ela é chamada de definição local.

Em (RAPPS e WEYUKER, 1985) são estabelecidos alguns conjuntos e definições que visam auxiliar a derivação dos requerimentos de teste. O conjunto *def(i)* é composto de todas as variáveis que são definidas globalmente no nó *i*. Todas as variáveis que possuem um c-uso global no nó *i* fazem parte do conjunto *c-uso(i)*, assim como todas as variáveis que possuem um p-uso no arco *(i, j)* fazem parte do conjunto *p-uso(i, j)*. Considerando uma variável *x* definida em um nó *i*, o conjunto *dcu(x, i)* é composto de todos os nós *j* que possuem um c-uso de *x* e que possuem um caminho livre de definição em relação a *x* do nó *i* até o nó *j*. O conjunto *dpu(x, i)* é o conjunto de todos os arcos *(j, k)* que possuem um p-uso e que possuem um caminho livre de definição em relação a *x* que vai do nó *i* até *j*.

Um caminho $(n_1, \ldots, n_j, n_k)$ é chamado de du-caminho em relação a variável *x* se n_1 possui uma definição global de *x* e se: 1) n_k possui um c-uso de *x* e $(n_1, \ldots, n_j, n_k)$ é um caminho livre de definição em relação a *x*; ou 2) o arco (n_j, n_k) possui um p-uso de *x,* e $(n_1, \ldots, n_j)$ é um caminho livre de laços em relação a variável *x*. Na Figura 5(b), o caminho (1, 2, 4, 5, 6) é um du-caminho em relação a variável x, pois a mesma é definida no nó 1 e possui um c-uso no nó 6, sendo que o caminho que se inicia no nó 1 e termina no nó 6 é livre de definição em relação a *x*.

Diversos critérios são utilizados para selecionar os caminhos do grafo Def-Uso que serão exercitados durante a atividade de teste, conforme (RAPPS e WEYUKER, 1985). Considerando um grafo Def-Uso *G* e um conjunto *P* de caminhos completos de *G*, então os seguintes critérios são identificados:

- **todas-definições**: para cada nó *i* de *G* e para cada variável *x* definida em *i*, *P* inclui um caminho livre de definição em relação a *x* que vai de *i* até algum elemento de *dcu(x, i)* ou *dpu(x, i).*
- **todos-c-usos/alguns-p-usos**: este critério requer que cada c-uso de uma variável *x*

definida em um nó *i* de *G* deve estar incluso em algum caminho de *P*.

- **todos-p-usos/alguns-c-usos**: este critério requer que cada p-uso de uma variável *x* definida em um nó *i* de *G* deve estar incluso em algum caminho de *P*.
- **todos-usos**: para cada nó *i* de *G* e para cada variável *x* definida em *i*, *P* inclui um caminho livre de definição em relação a *x* que vai de *i* até todos os elementos de *dcu(x, i)* e todos os elementos de *dpu(x, i)*.
- **todos-du-caminhos**: para cada nó *i* de *G* e para cada variável *x* definida em *i*, *P* inclui todos os du-caminhos em relação a *x*. Assim, se existem múltiplos du-caminhos partindo de uma definição global para um determinado uso, todos precisam estar incluídos em *P*.
- **todos-potenciais-usos**: requer que para todo nó *i* e para a variável *x*, existindo uma definição de *x* em *i*, 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 com relação a variável *x*, seja excercitado (MALDONADO, 1989).

3.2. Ferramentas de Teste

A atividade de teste pode consumir até 40% dos esforços despendidos no desenvolvimento de um software (PRESSMAN, 1997). Empregar ferramentas que auxiliam o teste pode diminuir o custo e o tempo gasto durante esta fase do projeto, sem reduzir a eficácia. Além disso, o uso de ferramentas automatizadas aumenta a confiabilidade do produto final. Com o avanço da pesquisa na área de teste de software diversas ferramentas foram desenvolvidas a fim de automatizar a atividade, entre elas destacam-se as ferramentas `xSuds` (ARGRAWAL et al, 1998), `Proteum` (DELAMARO, 1993), `PROTEUM/IM`

(DELAMARO, 1997), `PokeTool` (MALDONADO, CHAIM e JINO, 1989) e `JaBUTi` (VINCENZI, 2003).

3.3. Instrumentação de Código

A instrumentação é uma técnica de análise de programas freqüentemente empregada no processo de engenharia de software com diversos propósitos, como monitoramento da execução de um programa, análise de cobertura para determinados critérios de teste de software (como o teste estrutural, por exemplo), geração de requisitos de teste, depuração e identificação de erros, análise de performance, além de reengenharia ou engenharia reversa. De um ponto de vista abstrato, o processo de instrumentação de um software pode ser dividido em duas tarefas principais e distintas. Uma delas consiste em derivar a estrutura do software a partir do código fonte, e a outra consiste na inserção de instruções especiais cuja função é obter informações sobre a execução do software em tempo real.

Para a primeira tarefa desta análise, ocorre uma varredura no código fonte do programa a ser instrumentado e as informações relevantes são obtidas. Na maior parte dos casos essas informações se referem a estruturas de fluxo de dados e estruturas de fluxo de controle. Dessa forma, do fonte do programa (ou da função) abstrai-se um grafo cujos nós e arestas são, respectivamente, as instruções e os desvios existentes no fluxo de controle do programa. Em seguida os dados referentes a definições e usos de variáveis, obtidos na varredura do programa, são associados aos nós e às arestas do grafo gerado, obtendo-se assim um grafo Def-Uso.

O grafo do programa representa o aspecto estático do código fonte, porém muitas vezes é necessário obter também informações adicionais detalhadas a respeito do seu aspecto dinâmico, ou seja, informações referentes a execução do programa. Pode-se querer tomar

conhecimento, por exemplo, quais instruções foram executadas e em que ordem, ou ainda quais caminhos do grafo foram exercitados. Para tal, são inseridas instruções especiais no código fonte que não alteram seu significado semântico, ou seja, o funcionamento do programa será o mesmo. Essas instruções especiais tem o propósito de monitorar a execução do programa, coletar e armazenar as informações importantes em um arquivo de trace. A tarefa de inserir essas instruções de monitoramento no código chama-se decoração do código.

Essas duas tarefas tomadas em conjunto são denominadas de instrumentação de um programa. A técnica de instrumentação está presente, de uma forma ou de outra, em todos os critérios de teste de software que exigem análise de cobertura. O critério todos-nós, por exemplo, exige que cada nó do grafo gerado no processo de instrumentação seja exercitado pelo menos uma vez. Para garantir que esse critério seja satisfeito, as informações referentes à execução do programa que foram armazenadas no arquivo de trace são avaliadas.

Por ser um processo demorado e propenso ao erro, a instrumentação de programas é normalmente realizada com o auxílio de um software, chamado de instrumentador, que automatiza o processo. O instrumentador faz uma análise do programa de acordo com a gramática da linguagem em que o programa foi escrito e reconhece as estruturas relevantes. Com base na sintaxe da linguagem, o instrumentador deriva tanto o grafo do programa quanto o código decorado (isto é, o código alterado com as instruções especiais de monitoramento). Dessa forma, o processo de instrumentação se torna mais confiável, rápido e de fácil integração com outras aplicações, como uma ferramenta de análise de cobertura ou de derivação de requisitos de testes, por exemplo.

CAPÍTULO 4 – GRAMÁTICAS

Neste Capítulo são apresentados os fundamentos de teoria das gramáticas e linguagens, necessários para a compreensão da discussão apresentada nos Capítulos posteriores. O presente texto é de caráter introdutório a respeito das gramáticas livres de contexto e árvores sintáticas. Além disso, ao final desse Capítulo pode ser encontrada uma breve descrição das ferramentas Lex e YACC, utilizadas na construção do instrumentador para a linguagem Delphi a que se refere este trabalho.

4.1. Definição

Uma gramática pode ser vista como um conjunto finito de dispositivos capazes de representarem linguagens geralmente infinitas (SIMÃO et al A, 2002). Dada uma gramática *G*, a linguagem *L(G)* é o conjunto (não necessariamente finito) de todas as sentenças que podem ser obtidas por meio das produções de *G*. As produções são regras de substituições, ou seja, partindo de um símbolo inicial é possível realizar substituições até que uma sentença pertencente à gramática seja encontrada. A maioria das linguagens de programação é caracterizada por gramáticas, o que simplifica o processo de desenvolvimento de compiladores e interpretadores (HOPCROFT e ULLMAN, 1979).

4.2. Hierarquia de Gramáticas

De acordo com Noam Chomsky (CHOMSKY, 1956), as gramáticas são classificadas em quatro categorias, baseadas nos tipos de produções que elas possuem. As gramáticas mais abrangentes são chamadas de Gramáticas Irrestritas ou do **tipo 0**, que não impõe restrições quanto aos tipos de produções possíveis. Em um nível inferior estão as gramáticas do **tipo 1**, ou Gramáticas Sensíveis ao Contexto, que possuem restrições quanto ao tamanho da cadeia ou sentença gerada por suas regras de produção. O terceiro nível compreende as gramáticas do **tipo 2**, ou Gramáticas Livres de Contexto, em que um símbolo não terminal pode ser sempre substituído. Assim, o lado esquerdo de qualquer regra de produção deve ser formado sempre por um único símbolo não terminal. O quarto e menos abrangente nível é representado pelas gramáticas do **tipo 3**, ou Gramáticas Regulares (DELAMARO, 1998).

É importante notar que uma sentença que pode ser gerada por uma gramática de nível inferior (tipo 2, por exemplo), também pode ser produzida por meio das regras de uma gramática de nível superior (tipo 0, por exemplo), porém o contrário não é verdadeiro. Informações mais detalhadas a respeito da classificação e definição de cada um dos tipos de gramáticas podem ser encontradas em (CHOMSKY, 1956), (DELAMARO, 1998), (HOPCROFT e ULLMAN, 1979), (SIMÃO et al A, 2002) e (AHO, SETHI e ULLMAN, 1985).

4.3. Gramáticas Livres de Contexto

As Gramáticas Livres de Contexto (GLC) representam um conjunto expressivo e de grande significância prática na ciência da computação. Através de GLCs é possível definir a

maioria das linguagens de programação, formalizando assim a análise sintática e simplificando o processo de tradução de programas, além de serem úteis em qualquer outra aplicação que se beneficie do processamento de cadeias de caracteres (HOPCROFT e ULLMAN, 1979).

Uma GLC é usualmente descrita por meio das regras de produção que ela possui, dispostas através de uma notação chamada de BNF (Acrônimo para *Backus-Naur Form* – Forma de Backus-Naur). Dessa maneira, a definição de linguagens de programação se torna simples, facilitando a construção de compiladores e interpretadores (HOPCROFT e ULLMAN, 1979).

4.3.1. Convenções

Para facilitar a compreensão do restante deste texto, as seguintes convenções são adotas nessa Seção:

- letras minúsculas isoladas, dígitos e caracteres especiais são símbolos terminais (ex: `a`, `b`, `c`, `d`);
- qualquer texto entre aspas simples é um símbolo terminal (ex: `'else'`, `'while'`);
- letras maiúsculas isoladas são símbolos não terminais (ex: `A`, `B`, `C`);
- qualquer texto contido entre os sinais "<" e ">" é um símbolo não terminal (ex: `<OP>`, `<ID>`);
- letras gregas minúsculas são cadeias de símbolos terminais e não terminais (ex: α, β, γ, δ).

4.3.2. Definição Formal

Formalmente uma GLC é definida como uma quádrupla *G = (N, T, P, S)*, onde *N* e *T* são conjuntos finitos disjuntos de símbolos não terminais e símbolos terminais, respectivamente. O conjunto finito *P* contém as regras de produção da gramática, cada produção na forma *(A,α)*, em que *A* é um símbolo não terminal e α é uma cadeia de símbolos contida no conjunto *(V U T)**. A variável *S* é chamada de símbolo inicial, sendo *S* pertencente ao conjunto *V* (HOPCROFT e ULLMAN, 1979).

Uma produção *(A, α)* é uma regra de substituição representada na forma *A ::= α*. Essas regras são aplicadas partindo do símbolo inicial *S*, substituindo-se o símbolo não terminal que aparece do lado esquerdo de uma produção pela palavra que aparece do lado direito (SIMÃO et al A, 2002). Quando a palavra obtida possuir apenas símbolos terminais, diz-se que a palavra pertence à linguagem *L(G)* definida pela gramática *G*. Caso a cadeia obtida ainda contenha símbolos não terminais, as substituições devem ser aplicadas até que só haja apenas símbolos terminais (DELAMARO, 2004).

A gramática[3] G_1, cujas produções são exibidas no BNF da Figura 6(a), é um exemplo simples que serve para definir expressões aritméticas (HOPCROFT e ULLMAN, 1979). Considera-se $G_1 = (N, T, P, E)$, o conjunto de símbolos não terminais $N = \{E\}$, o conjunto de símbolos terminais $T = \{+, -, (,), \texttt{id}\}$; e o símbolo inicial sendo *E*. As produções *P* são mostradas na Figura 6(a). A Figura 6(b) ilustra uma forma simplificada de esquematizar o BNF. A barra vertical | tem o significado de "*ou*".

[3] A gramática G_1 apresenta ambigüidade, que é um problema no desenvolvimento de compiladores. Para fins explicativos, porém, ela é adequada por sua simplicidade.

```
1) E  ::=  E + E
2) E  ::=  E * E
3) E  ::=  ( E )
4) E  ::=  id
```

Figura 6 (a) – BNF da gramática G1.

```
E  ::=  E + E | E * E | ( E ) | id
```

Figura 6 (b) – Outra forma de representar G1.

Através da aplicação das regras da Figura 6(a) é possível obter, por exemplo, a expressão `(a + b) * c`. Na Figura 7 é apresentada a ordem em que as regras devem ser aplicadas para obter essa expressão (considerando `id` como sendo qualquer identificador, dígito ou número válido nessa gramática). A expressão obtida é, portanto, uma palavra ou sentença da linguagem $L(G_1)$ definida pela gramática G_1.

```
E  ::=  E * E                 produção 2
E  ::=  ( E ) * E             produção 3
E  ::=  ( E ) * id            produção 4
E  ::=  ( E + E ) * id        produção 1
E  ::=  ( E + id ) * id       produção 4
E  ::=  ( id + id ) * id      produção 1
```

Figura 7 – Ordem em que as produções devem ser aplicadas.

Como se pode ver, para qualquer seqüência $\gamma A \delta$, pode-se derivar outra seqüência $\gamma \alpha \delta$, se existe uma regra (A, α) na gramática.

4.3.3. Exemplo: Pascal Simplificado

A Figura 8 apresenta o exemplo de uma gramática G_2 na forma BNF que define uma versão truncada[4] da linguagem Pascal, contendo apenas expressões aritméticas simples, um comando de repetição (**`while`**) e um comando de controle (**`if`**). Apesar de trivial, o exemplo serve de ilustração para compreender melhor uma gramática livre de contexto.

```
<S>  ::=  <W>
<S>  ::=  <IF>
<S>  ::=  'begin' <S> 'end'
<S>  ::=  'exit' ';'
<S>  ::=  <ID> ':=' <E> ';'
<W>  ::=  'while' <E> 'do' <S>
<IF> ::=  'if' <E> 'then' <S> 'else' <S>
<E>  ::=  <ID>
<E>  ::=  <E> <OP> <ID>
<E>  ::=  '(' <E> ')'
<OP> ::=  '-' | '+' | '<' | '>' | '='
<ID> ::=  identificador | inteiro
```

Figura 8 – BNF da gramática G2 – versão truncada da linguagem Pascal.

Seguindo as convenções previamente estabelecidas, a gramática $G_2 = (N, T, P, S)$, cujas regras de produção P estão ilustradas na Figura 8, é constituída do conjunto de não terminais $N = \{S, W, IF, ID, E, OP\}$, da conjunto de símbolos terminais $T = \{$`'begin'`, `'end'`, `'exit'`, `';'`, `':='`, `'while'`, `'do'`, `'if'`, `'then'`, `'else'`, `'('`, `')'`, `'+'`, `'-'`, `'<'`, `'>'`, `'='`$\}$, e do símbolo inicial S. A Figura 9 apresenta um programa válido na linguagem $L(G_2)$.

[4] A gramática é um subconjunto da linguagem Pascal. <S> é uma regra de produção para comando (*statement)*, <E> para expressão, <IF> para o comando if, <W> para o comando while, <OP> representa um operador e <ID> é um identificador válido ou um número inteiro.

```
1     while (a < 10) do
2     begin
3           if (a = 0) then
4                 exit;
5           else
6                 a := a + b;
7     end
```

Figura 9 – Programa válido na linguagem definida pela gramática G2.

4.3.4. Árvores Sintáticas

As derivações de uma gramática podem ser sumarizadas em uma árvore sintática (ou árvore de derivações (HOPCROFT e ULLMAN, 1979)) que facilita a compreensão do todo e a construção de analisadores sintáticos. Uma árvore sintática é uma árvore cujos nós internos são os símbolos não terminais da gramática, as folhas são os símbolos terminais e o nó raiz é o símbolo inicial. Se um nó <*A*> possui os nós filhos $\beta_1, \beta_2, \beta_3, \ldots, \beta_k$, então deve existir uma regra de substituição na forma apresentada na Figura 10.

$$\langle A \rangle ::= \beta_1 \beta_2 \beta_3 \ldots \beta_4$$

Figura 10 – Filhos de um nó.

A Figura 11 contém a árvore sintática do programa da Figura 9 relativa à gramática G_2. Tomando os símbolos terminais (folhas da árvore) em seqüência, da esquerda para a direita, é possível obter a sentença pertencente à linguagem $L(G_2)$, que no caso é o próprio programa da Figura 9. Representar a estrutura sintática do programa através de uma árvore possibilita uma melhor visualização das aplicações das produções da gramática e permite ainda verificar se há ambigüidade nas regras de substituição.

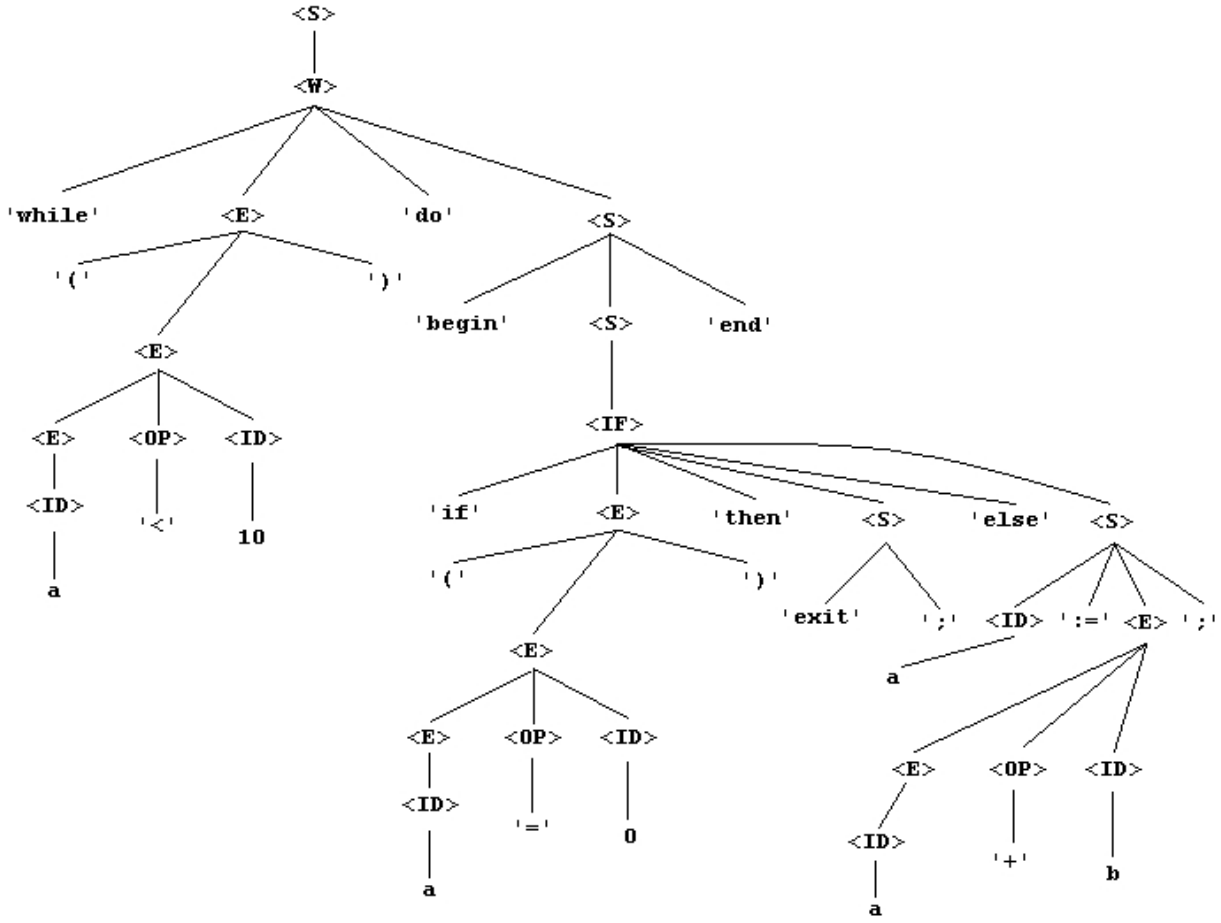

Figura 11 – Árvore sintática do programa da Figura 9.

4.3.5. Geradores de Compiladores

As ferramentas Lex e YACC (LEVINE, MASON e BROWN, 1992; LESK, SCHMIDT e JOHNSON, 2005), auxiliam no desenvolvimento de programas que lidam com entrada estruturada, que pode variar desde um editor de texto simples que suporta busca de padrões no texto até um avançado compilador que traduz código alto nível para código objeto otimizado. Para interpretar uma entrada estruturada duas tarefas são necessárias: identificar as unidades mínimas da entrada e as relações existentes entre tais unidades.

Por convenção chama-se tais unidades de *tokens*, que são símbolos indivisíveis, tais como a palavra reservada '**`else`**' na gramática G_2 da Figura 8. A tarefa de identificar os *tokens* da entrada é chamada de análise léxica, e é realizada por um programa chamado analisador léxico, que tem a única função de obter os dados da entrada e retornar os *tokens*, um a um, quando requisitado.

Após a identificação dos *tokens* é necessário definir os relacionamentos existentes entre eles. Essa tarefa cabe ao analisador sintático que, tendo as especificações de uma gramática *G*, é capaz de verificar se uma seqüência de *tokens* (símbolos terminais da gramática) pertencem à linguagem *L(G)*. A análise léxica e a análise sintática são duas importantes tarefas realizadas por um compilador, entre outras que não interessam a este trabalho, como a análise semântica e a tradução do código.

Como é possível perceber, construir um analisador léxico e sintático partindo do nada, apenas usando uma linguagem estruturada como C ou Pascal, pode ser uma tarefa extensa, muito complexa e propensa a erros. Para isso existem os chamados geradores de compiladores, como o Lex e o YACC.

4.3.5.1. Lex

A ferramenta Lex[5] é empregada na construção de analisadores léxicos. Para tal, basta escrever a especificação léxica da gramática, ou seja, definir quais são os *tokens* (símbolos terminais) da gramática por meio de uma sintaxe que o Lex entende, e então utilizar essa especificação como entrada para o Lex. Os padrões serão identificados utilizando expressões regulares e o resultado será o código fonte de um analisador léxico na linguagem C, mais

[5] A ferramenta Lex original é proprietária, portanto neste projeto utilizou-se a ferramenta Flex (*Fast Lex)*, que é uma implementação livre (FREE SOFTWARE FUNDATION B, 2005) do padrão lex.

confiável e quase tão rápido quanto um escrito “à mão” (LEVINE, MASON e BROWN, 1992).

O código fonte gerado não é um programa executável, mas uma rotina chamada `yylex()` que quando invocada retorna o próximo *token* identificado na entrada a ser processada. Dessa forma é possível integrar o código gerado pelo Lex com outras aplicações, como analisadores sintáticos gerados pelo YACC.

O exemplo da Figura 12 apresenta o código de uma especificação léxica simples, utilizada para reconhecer e separar os *tokens* da gramática da Figura 8.

```
1    %{
2    /* especificação léxica simples */
3
4    #include <y.tab.h>
5    %}
6
7    %option noyywrap
8
9    %%
10
11   [\t ]+      /* ignora  espacos em branco */;
12   begin       { count(); return(BEGIN);     }
13   end         { count(); return(END);       }
14   exit        { count(); return(EXIT);      }
15   while       { count(); return(WHILE);     }
16   do          { count(); return(DO);        }
17   if          { count(); return(IF);        }
18   then        { count(); return(THEN);      }
19   else        { count(); return(ELSE);      }
20   "+"         { count(); return(PLUS);      }
21   "-"         { count(); return(MINUS);     }
22   "<"         { count(); return(LT);        }
23   ">"         { count(); return(GT);        }
24   "="         { count(); return(EQUAL);     }
25   "("         { count(); return(LPAR);      }
26   ")"         { count(); return(RPAR);      }
27   ";"         { count(); return(SEMICOLON); }
28   ":="        { count(); return(ATTRIB);    }
29   [0-9]+      { count(); return(INTEGER);   }
30   [a-zA-Z_]   { count(); return(IDENTIFIER);}
31   \n          { linecount();                }
32   %%
```

Figura 12 – Especificação léxica simples. Sempre que um token é encontrado, ele é contado e retornado.

No exemplo, as linhas 11 a 31 definem os padrões que devem ser identificados e separados quando a rotina de análise léxica for invocada. A rotina `count()`, escrita na linguagem C, presente nesse exemplo, simplesmente conta quantos *tokens* já foram lidos. A rotina `linecount()` conta quantas linhas foram interpretadas. Quando um padrão é identificado, a rotina `count()` incrementa o contador de *tokens* e logo em seguida a rotina `yylex()`, que é o analisador léxico propriamente dito, retorna o *token* identificado para o programa que o invocou. A possibilidade de utilizar código na linguagem C junto às especificações léxica permite a criação de softwares avançados. É possível, por exemplo, inserir cada *token* reconhecido em uma tabela de símbolos que será utilizar por outra aplicação posteriormente.

4.3.5.2. YACC

A ferramenta YACC[6] (*Yet Another Compiler Compiler* – Mais um Compilador de Compilador) é utilizada na construção de analisadores sintáticos (ou *parsers*). É preciso passar para o YACC a especificação de uma gramática livre de contexto e os *tokens* retornados pelo analisador léxico. Como resultado, será produzida uma rotina escrita em C capaz de reconhecer as sentenças da gramática especificada. A rotina detecta quando um conjunto de *tokens* combina com uma das produções da gramática e também é capaz de detectar erros sintáticos.

Um *parser* gerado pelo YACC não é tão rápido e eficiente quanto um *parser* escrito por um programador experiente, porém a facilidade e a rapidez existentes na geração de um *parser* com o auxílio do YACC são suficientes para encorajar o programador a não escrever

[6] A ferramenta YACC original foi desenvolvida pela AT&T e é proprietária. Neste projeto usou-se a ferramenta GNU Bison (FREE SOFTWARE FUNDATION A, 2005), que é uma implementação livre do padrão YACC.

seu próprio. Caso o *parser* precise de futuras alterações, é muito mais fácil alterar a especificação da gramática e gerar um novo analisador sintático do que altear um *parser* estático, escrito manualmente (LEVINE, MASON e BROWN, 1992). Além disso, o código gerado pelo YACC possui menos erros, já que diversos programadores depuram o código e corrigem os erros encontrados, uma vez que existem implementações livres do YACC, como o GNU Bison (FREE SOFTWARE FUNDATION A, 2005).

A Figura 13 contém a especificação de um parser para a gramática da Figura 8. Essa especificação deve ser utilizada junto ao analisador léxico da Figura 12 que terá a função de retornar os *tokens* para o analisador sintático. O *parser* gerado pelo YACC deve ser invocado por outros softwares através da rotina `yyparse()`. Através dessa rotina é possível integrar o código gerado pelo YACC com outras aplicações.

É importante destacar que este exemplo é meramente ilustrativo. Sua única função é imprimir a mensagem “entrada válida” na saída padrão quando uma sentença válida da linguagem for identificada e uma mensagem de erro quando encontrar uma entrada inválida. O YACC suporta a definições de ações semânticas, utilizadas na avaliação de expressões e regras da gramática, necessárias na análise semântica. Tais ações exigem uma discussão mais aprofundada que não é pertinente ao conteúdo deste trabalho, já que a construção do instrumentador para linguagem Delphi não exige tais ações na especificação YACC da gramática. A forma como foram apresentadas as Figuras 12 e 13 são suficientes para a construção do instrumentador. Mais informações sobre Lex e YACC podem ser encontradas em (LEVINE, MASON e BROWN, 1992) e (AHO, SETHI e ULLMAN, 2005)

Para gerar um instrumentador de código para uma linguagem qualquer, o IDeLGen necessita das especificações léxicas e sintáticas de tal linguagem descritos na forma Lex e YACC, respectivamente.

```
1     %{
2     #include <stdio.h>
3     %}
4
5     %token BEGIN END EXIT WHILE DO
6     %token IF THEN ELSE PLUS MINUS LT GT
7     %token EQUAL LPAR RPAR SEMICOLON
8     %token ATTRIB INTEGER INDENTIFIER
9
10    %%
11
12    p   :   s
13        ;   { printf("entrada válida\n");
14
15    s   :   w
16        |   if
17        |   BEGIN s END
18        |   EXIT SEMICOLON
19        |   IDENTIFIER ATTRIB e SEMICOLON
20        ;
21
22    w   :   WHILE e DO s
23        ;
24
25    if  :   IF e THEN s ELSE s
26        ;
27
28    e   :   IDENTIFIER
29        |   e op IDENTIFIER
30        |   LPAR e RPAR
31        ;
32
33    op  :   MINUS
34        |   PLUS
35        |   LT
36        |   GT
37        |   EQUAL
38        ;
39    %%
40
41
42    yyerror(char *s)
43    {
44        fprintf(stderr, "%s\n", s);
45    }
```

Figura 13 – Especificação sintática simples. As ações semânticas foram excluídas.

CAPÍTULO 5 – IDEL & IDELGEN[7]

A programação de um instrumentador sem ferramentas de apoio é complexa e toma tempo, já que o instrumentador depende da gramática da linguagem para a qual o mesmo está sendo confeccionado. Além disso, depois de pronto o instrumentador não funcionará para outra linguagem. Para evitar tais problemas, este trabalho utilizou o sistema IDeL & IDeLgen que auxilia na construção de instrumentadores. Este Capítulo trata desse sistema.

5.1. Gerador de Instrumentadores

O sistema utilizado para gerar o instrumentador Delphi é composto de uma linguagem de especificação de instrumentadores chamada IDeL (*Instrumentation Description Language* – Linguagem para Descrição de Instrumentadores), e de um "compilador" para essa linguagem chamado IDeLgen (*IDeL Generator* – Gerador IDeL). A abordagem utilizada por esse sistema pode ser vista como geração de instrumentadores, pois através da linguagem IDeL especifica-se as características desejadas no instrumentador que se quer gerar.

Dessa forma, usando uma linguagem de especificação intermediária, o processo de construção de instrumentadores é simplificado e generalizado. É possível construir um instrumentador para a linguagem C, por exemplo, e futuramente alterá-lo para que se torne um instrumentador para a linguagem C++ ou Java sem muito esforço, diferente do que aconteceria se fosse necessário alterar o código fonte de um instrumentador escrito puramente em uma linguagem de programação convencional.

[7] Este Capítulo foi baseado em (SIMÃO et al A, 2002) e (SIMÃO et al B, 2002).

5.2. Aspectos Operacionais

Para facilitar a compreensão do restante deste Capítulo, esta Seção mostra como trabalhar com sistema IDeL & IDeLGen. Dada uma gramática *GRM*, um programa escrito em uma linguagem *L(GRM)* a e uma descrição de instrumentação, o sistema é capaz de gerar o programa instrumentado e o grafo do programa.

A gramática *GRM* deve ser fornecida através de dois arquivos, um correspondente às suas especificações léxicas (`GRM.l`) e o outro correspondente às especificações sintáticas (`GRM.y`). O arquivo léxico deve ser compatível com a sintaxe da ferramenta Lex, e o arquivo sintático deve ser compatível com a sintaxe da ferramenta YACC, discutidas no Capítulo 4. As ações semânticas do arquivo sintático não devem estar presentes, apenas a descrição das produções da gramática.

Os dois arquivos da gramática devem ser passados para a aplicação IDeLGen, que irá interpretá-los e produzir um programa executável chamado `idel.GRM`. Este programa é o instrumentador para a linguagem definida pela gramática *GRM* e reconhece programas escritos nessa linguagem. Ele deve ser executado juntamente com o arquivo `GRM.idel`, que contém as especificações de instrumentação na linguagem IDeL, e com programa `P.GRM` que será instrumentado. Como resultado serão produzidos o grafo do programa e o programa decorado. O funcionamento do sistema é ilustrado na Figura 14.

A Figura 15(a) mostra na prática como executar o IDeLGen e gerar o instrumentador, e a Figura 15(b) mostra como executar o instrumentador e obter o programa instrumentado e o grafo do programa.

Na Figura 15(a) a ferramenta IDeLGen é chamada com o parâmetro `pascal`, significando que a descrição da gramática está localizada nos arquivos `pascal.l` e `pascal.y`. Após a execução desse comando, o executável `idel.pascal` é produzido. Na

Figura 15(b) o instrumentador `idel.pascal` é executado e recebe como parâmetros o arquivo `pascal.idel`, que contém a especificação IDeL para um instrumentador da linguagem Pascal, e o arquivo `exemplo.pas`, que é o programa que será instrumentado. O Capítulo 6 descreve uma forma mais flexível de trabalhar com o sistema.

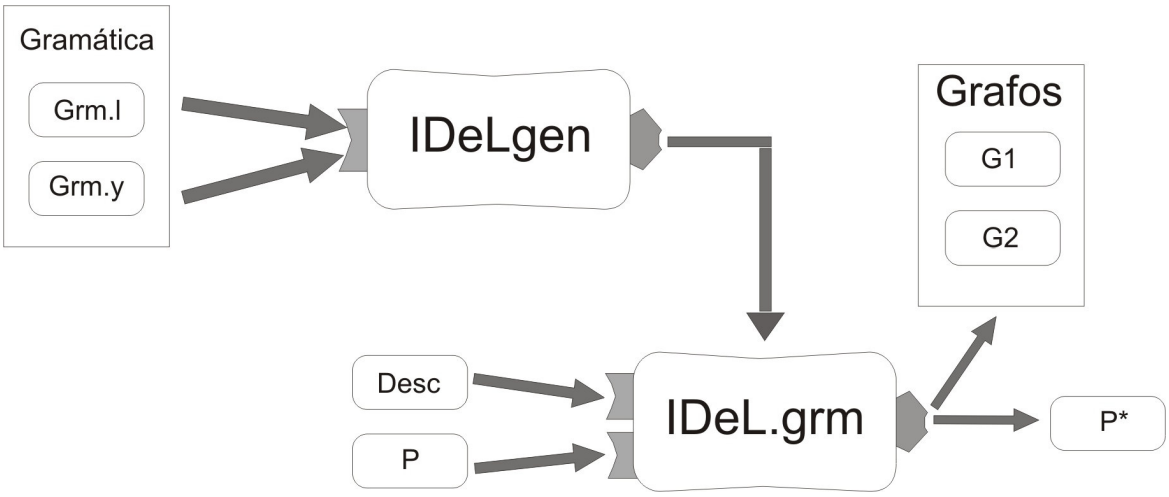

Figura 14 – Esquema geral do funcionamento do sistema IDeL & IDeLGen.

```
gustgr@lost% idelgen pascal
```

Figura 15 (a) – Geração do instrumentador idel.pascal.

```
gustgr@lost% ./idel.pasl pascal.idel exemplo.pas
```

Figura 15 (b) – Instrumentação do programa exemplo.pas.

5.3. Conceitos Iniciais

Como foi visto no Capítulo 4, uma árvore sintática representa a estrutura sintática de um programa, isto é, seu aspecto estático. O instrumentador é capaz de interpretar essa estrutura sintática e a partir dela derivar o grafo do programa e decidir onde inserir as instruções de monitoração. Contudo, é necessária uma maneira genérica de especificar, selecionar e alterar porções da árvore sintática nas quais as construções da IDeL devem ser aplicadas, isso significa que é preciso dividir a árvore sintática em sub-árvores e interpretá-las de acordo com as especificações na linguagem IDeL. Essas sub-árvores são chamadas de árvores sintáticas de padrões. Após a identificação de uma estrutura sintática do programa, é preciso realizar uma comparação a fim de se certificar de que a estrutura identificada combina com a árvore de padrões especificada.

Arvores Sintáticas de Padrões

Para formar uma árvore sintática de padrões (ASP) necessita-se de um conjunto *M* de meta-variáveis e estender a árvore sintática para permitir que as folhas sejam, além de símbolos terminais, meta-variáveis. Além disso, o nó raiz de uma ASP pode ser qualquer símbolo não terminal, não apenas o símbolo inicial, como ocorria com as árvores sintáticas tradicionais. Nessa nova abordagem, cada meta-variável está associada a um símbolo não terminal, denominado o tipo da meta-variável. Uma meta-variável pode estar livre ou conectada. Toda variável conectada está associada a uma sub-árvore que pode ser gerada a partir do seu tipo. Dessa forma, observa-se que uma árvore sintática é apenas um tipo especial de árvore de padrões em que todas as meta-variáveis (se existentes) estão conectadas.

A Figura 16 exibe uma árvore de padrões relacionada com a gramática apresentada na Figura 8. Para distinguir as meta-variáveis dos identificadores tradicionais, elas são prefixadas com um sinal de dois-pontos (`:`). É importante destacar que mesmo utilizando meta-variáveis, os filhos de um nó ainda devem estar de acordo com as produções permitidas pela gramática, isto é, uma meta-variável pode aparecer somente onde poderia também um símbolo não terminal do seu tipo.

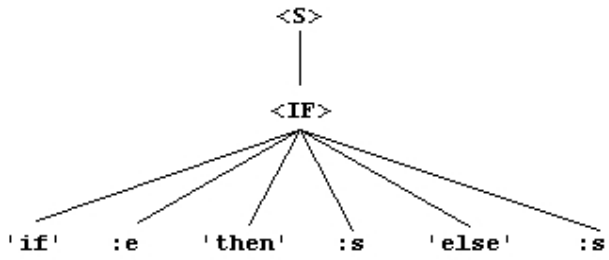

Figura 16 – Árvore sintática de padrões do comando condicional if.

A linguagem IDeL exige a especificação de padrões para que seja possível fazer a comparação de árvores. O padrão mais simples é formado por uma meta-variável que representa o nó inicial. Esse padrão é expressado pelo símbolo não terminal do nó raiz (no caso, `S`) posicionado entre colchetes. Por exemplo, `[S]` é um padrão cujo nó inicial é uma meta-variável do tipo `<S>`. Padrões mais avançados são representados com o auxílio dos sinais < e >. Entre esses sinais é inserida a seqüência de símbolos e meta-variáveis que deve ser interpretada para gerar a árvore de padrões equivalente. O padrão da Figura 16, por exemplo, é definido por `[S< if :e then :s else :s >]`. Nessa representação também deve-se respeitar as regras da gramática, e as meta-variáveis (no caso `:e` e `:s`) devem ser declaradas de acordo com os tipos apropriados (no caso `<E>` e `<S>`, respectivamente).

5.4. IDeL: Principais Características

O sistema IDeL & IDeLGen traduz os programas a serem instrumentados para um formato intermediário genérico, dessa forma é possível trabalhar com um mecanismo abstrato para realizar a instrumentação de programas em diferentes linguagens. O formato intermediário abordado pela linguagem IDeL é a árvore sintática, pois os programas escritos na maior parte das linguagens podem ser convertidos para árvores sintáticas por meio de técnicas devidamente conhecidas e estabelecidas na teoria das gramáticas (AHO, SETHI e ULLMAN, 1985). Dessa forma, todo programa a ser instrumentado tem seu código transformado em uma árvore sintática.

5.4.1. Árvore Sintática Aprimorada

Conforme explicado anteriormente, o processo de instrumentação envolve a derivação do grafo do programa, rastreamento de informações relevantes (como uso e definições de variáveis) a inserção de instruções de monitoramento no código (decoração do código). Para realizar essas tarefas, o sistema IDeL & IDeLGen utiliza uma extensão da árvore sintática capaz de armazenar dados especiais em determinados nós, chamada Árvore Sintática Aprimorada.

Um nó interno nessa extensão da árvore sintática tem dois objetos associados a si: um mapa de nós e a tabela de implementação. O mapa de nós é usado para representar o relacionamento entre o nome real de um nó e um nome simbólico. Por exemplo, com o mapa de nós o nó `45` (nome real) pode ser referenciado pelo nome simbólico `$begin`. A tabela de implementação tem a finalidade de armazenar informações referentes ao tipo de

instrumentação e decoração que deverá ocorrer naquele nó em particular. A tabela de implementação pode definir, por exemplo, que uma ponta de prova (*checkpoint*) será inserida antes ou depois do nó. Dessa maneira é possível instrumentar a árvore sintática de maneira flexível, tratando cada nó em particular.

5.4.1.1. Mapeamento dos Nós

O processo de instrumentação exige que algumas informações referentes ao contexto sejam coletadas. É interessante saber, por exemplo, se uma instrução **`break`** ocorreu dentro de uma estrutura de repetição. Todavia, para fins de simplicidade e praticidade, o instrumentador gerado analisa as construções de um programa individualmente. Isso significa que informações sobre o contexto devem estar disponíveis quando se estiver examinando uma construção qualquer em particular.

Para tal, associa-se um mapa de nós com cada nó da árvore sintática. Nesse mapa os nomes reais dos nós podem ser atribuídos ou obtidos por meio de nomes simbólicos. O mapa é organizado de forma hierárquica, de modo que se um determinado nó não corresponde a nenhum dos nomes simbólicos existentes no mapa, os mapas do nós predecessores são consultados recursivamente até que o nome simbólico correspondente seja encontrado.

Por exemplo, o nome simbólico `$break` pode ser atribuído ao nó que sucede o comando de repetição que estiver sendo examinado. Se uma instrução **`break`** for encontrada, basta referir-se diretamente ao nó `$break`. Então, o algoritmo recursivo irá obter o nó mais interno da árvore que possui o nome simbólico `$break`.

5.4.1.2. Tabela de Implementação

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 maior parte das vezes, as alterações são realizadas antes ou depois de uma determinada estrutura, e estão relacionadas com um nó 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 atravessada e cada alteração é realizada baseada nas informações presentes na tabela de implementação de cada um dos nós.

5.4.2. Estrutura Geral da Linguagem

O arquivo de especificação da instrumentação escrito na linguagem IDeL é composto de três partes: identificação das unidades, processamento das unidades e implementação. Cada uma dessas partes corresponde à uma fase do processo de instrumentação.

5.4.2.1. Identificação das Unidades

A primeira fase 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. As unidades são identificadas através de uma lista de padrões, sempre que um dos padrões presentes na lista é

encontrado na árvore sintática, a unidade é processada. Na próxima etapa é gerado um grafo de programa para cada unidade identificada nesta fase.

5.4.2.2. Processamento das Unidades

Após a identificação das unidades ocorre o processamento das mesmas. Esta é a principal fase na geração do instrumentador e é quando o grafo do programa é derivado e são inseridas as alterações referentes ao tipo de instrumentação na árvore sintática. Esta etapa é dividida em uma série da passos de processamento (declarados com o comando **`step`**) que são aplicados em seqüência nas unidades identificadas.

Um passo de processamento consiste na aplicação de um ou mais padrões de instrumentação, que são compostos por diversas seções: seção de nome, de declaração das meta-variáveis, de comparação, de declaração dos nós, de topologia dos grafos, de atribuições e seção de instrumentação. Apenas as seções de nome e de comparação de padrões são obrigatórias. O processamento das unidades é tratado em mais detalhes no Capítulo 6, em que é discutida a construção do instrumentador para linguagem Delphi.

5.4.2.3. Implementação

A terceira etapa consiste na parte de implementação, que tem a finalidade de prover implementações concretas para as declarações abstratas **`instrument`** localizadas na seção de processamento das unidades. Essa seção é composta por uma lista de declarações **`implement`** que contém as ações que devem ser tomadas para alterar o programa e o grafo do programa.

Na segunda fase da instrumentação as declarações **`instrument`** apenas indicam que

tipo de alteração devem ser realizadas em determinado nó, porém não indicam como isso deve ocorrer. Isso cabe à etapa de implementação. Fazendo uma analogia com linguagens de programação convencionais, as declarações instrument podem ser vistas como chamadas rotinas, e a fase de implementação é responsável por fornecer o corpo dessas rotinas. Para mais detalhes veja o Capítulo 6.

5.4.3. Algoritmo

Após a geração do executável pelo IDeLGen, a descrição do instrumentador deve ser interpretada. A interpretação das especificações do instrumentador (escrito na linguagem IDeL) a fim de instrumentar um programa *P* (escrito em uma linguagem *L* qualquer) é composta de cinco etapas:

1ª) Interpretação: o programa *P* é interpretado pelo instrumentador gerado pelo IDeLGen. Se o programa estiver correto em relação à linguagem *L*, uma árvore sintática é construída.

2ª) Identificação das Unidades: partindo do nó raiz, a árvore sintática é atravessada e tenta-se encontrar padrões que combinem com aqueles definidos na primeira parte do arquivo de especificação de instrumentação. Sempre que um nó na árvore sintática combinar com um desses padrões, um novo grafo é criado e o nó encontrado é processado (3ª etapa). Se nenhum nó combinar com o padrão das unidades, a fase da implementação (4ª etapa) é iniciada.

3ª) Processamento das unidades: cada passo descrito na segunda parte do arquivo de instrumentação é executado em seqüência. Os passos são processados a partir do nó inicial da

árvore até as folhas, ou das folhas até o nó inicial. A direção do processamento depende da forma com que os passos foram declarados. Para cada nó da árvore, cada um dos padrões declarados nos passos são testados até que uma combinação ocorra, então o padrão que combinou é aplicado. Após a execução do último passo, um algoritmo de minimização é aplicado no grafo. Em seguida o grafo é reorganizado e gravado em um arquivo de saída formatado como XML ou DOT (GANSNER e NORTH, 2001), de fácil interpretação por muitos softwares disponínveis.

4ª) Implementação: partindo do nó raiz, a árvore sintática é atravessada e verifica-se a tabela de implementação presente em cada nó. Caso uma implementação precise ser realizada, o sistema procura por uma declaração **`implement`** que seja compatível com aquela que está especificada na tabela de implementação do nó. Se a declaração compatível for encontrada, a árvore sintática é devidamente alterada.

5ª) Varredura da Árvore: a árvore sintática é atravessada e cada nó terminal é coletado e armazenado em um arquivo de saída. A seqüência de terminais contida nesse arquivo é justamente o programa P instrumentado.

CAPÍTULO 6 – INSTRUMENTADOR PARA LINGUAGEM DELPHI

Este Capítulo apresenta a implementação do instrumentador para a linguagem Delphi, escrita na linguagem IDeL. Apenas as estruturas mais importantes são abordadas. A especificação completa do instrumentador pode ser encontrada no Apêndice B, devidamente comentada. Além disso, através da descrição do instrumentador para Delphi este Capítulo se aprofunda nos detalhes operacionais e de implementação da linguagem IDeL, omitidos no Capítulo 5.

6.1. Gramática da Linguagem Delphi

Para gerar o instrumentador com o IDeLGen foi necessário escrever as especificações léxicas e sintáticas da gramática da linguagem Delphi no formato reconhecido pelas ferramentas Lex e YACC. Apesar das extensas pesquisas bibliográficas realizadas, nenhuma gramática livre de contexto completa foi encontrada para que fosse possível escrever suas especificações. Em (SEMANTIC DESIGNS, 2005) foi encontrado um software que possui um BNF da gramática desejada, porém o software é proprietário e precisa ser licenciado, o que tornou seu uso inviável. Em (BORLAND, 1993) foi encontrada uma gramática parcial da linguagem Object Pascal disposta no formato BNF, e foi esse o ponto de partida deste trabalho. Após consultas em vários manuais de referências (BORLAND, 1993; BORLAND, 2002; BORLAND, 2004) foi possível aprimorar a gramática e atingir um resultado aparentemente completo, superior a qualquer outro encontrado nas pesquisas bibliográficas.

A gramática da linguagem Delphi está disposta em dois arquivos. O arquivo `delphi.l` contém as especificações léxicas no formato reconhecido pela ferramenta Lex. O arquivo `delphi.y` contém as especificações sintáticas no formato reconhecido pela

ferramenta YACC. A gramática pode ser consultada em sua totalidade no Apêndice A. É importante ressaltar que o arquivo sintático usado para gerar o instrumentador deve conter apenas as regras de produções no formato YACC, qualquer outra construção como ações semânticas, funções na linguagem C ou mesmo comentários acarretam erros na hora de gerar o instrumentador.

Além disso, no arquivo léxico é necessário fazer uma chamada à função `count()` a cada reconhecimento de *tokens*, caso contrário a IDeL não será capaz de ler o código fonte do programa que será instrumentado. Essa função pode ser definida no próprio arquivo léxico e tem a finalidade de incrementar um contador sempre que um *token* for identificado. Para exemplo de um arquivo léxico funcional consultar o Apêndice A.

6.2. Geração do Instrumentador

Dispondo da gramática da linguagem Delphi, o executável do instrumentador foi gerado através da ferramenta IDeLGen, que interpretou os arquivos `delphi.y` e `delphi.l` para realizar essa tarefa. A geração do instrumentador pode ser vista na Figura 17.

```
gustgr@lost% idelgen delphi
Building tree constructor
#
#
#
#
#
Building operator parser
#
#
#
#
#
Building idel.delphi
Done.
```

Figura 17 – Geração do instrumentador para linguagem Delphi.

O arquivo executável `idel.delphi` gerado é o instrumentador que deve ser utilizado junto ao arquivo `delphi.idel` e ao programa a ser instrumentado. Esse executável verifica se o código fonte do programa está de acordo com a linguagem Delphi, e caso esteja, gera uma árvore sintática na qual será realizada a instrumentação de acordo com a descrição o arquivo `delphi.idel`.

6.3. Descrição do Instrumentador

A descrição do instrumentador para a linguagem Delphi está no arquivo `delphi.idel`. Esse arquivo é dividido em três partes principais: identificação das unidades, processamentos das unidades e implementação. A Figura 18 ilustra o esqueleto do arquivo de descrição do instrumentador.

```
instrumenter <nome_do_instrumentador>

## Parte 1
unit
      # identificação da unidade
end unit

## Parte 2
step <nome_do_passo>

pattern <nome_do_padrão>
      # definição do padrão
end pattern

## Parte 3
implementation

implement
      # implementação
end implement

end instrument
```

Figura 18 – Esqueleto do arquivo IDeL.

6.3.1. Identificação das Unidades

Uma 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. A seção de identificação de Unidades aparece logo no início do arquivo delphi.idel e visa identificar essas construções da linguagem Delphi para posteriormente ocorrer a geração das árvores sintáticas e a instrumentação. Para cada unidade será gerada um grafo diferente. Por exemplo, se um programa possuir as funções `soma()` e `fatorial()` e for instrumentado, as duas funções serão reconhecidas como unidades e dois grafos de programa diferentes serão gerados ao fim da instrumentação.

Uma limitação da linguagem IDeL é que a linguagem permite reconhecimento de apenas um tipo de unidade. Essa abordagem funciona bem para linguagens como C ou Java, onde toda a lógica do programa está localizada dentro de funções e métodos, respectivamente. Porém, a linguagem Delphi possui mais do que um tipo de unidades. A solução encontrada foi desenvolver três versões do arquivo `delphi.idel`. Uma delas tratando a identificação de funções, a segunda tratando a identificação de procedimentos e a terceira tratando a identificação do corpo do programa principal. As outras duas seções dos arquivos são as mesmas, a única seção que é diferenciada é a seção de identificação de unidades. Este texto aborda apenas a identificação de funções, pois o processo para identificar procedimentos e o corpo do programa principal é análogo.

A Figura 19 mostra a seção de identificação de unidades do arquivo `delphi.idel` que reconhece funções. A palavra reservada **`unit`**, na linha 1, indica que a seção de identificação está começando. Na linha 2 é iniciada a declaração das meta-variáveis, utilizadas no padrão da linha 11. Os tipos das meta-variáveis estão localizados entre os colchetes, e correspondem às regras da gramática. As linhas 8 e 9 definem o nome pelo qual a

unidade vai ser identificada, que nesse caso é o próprio nome da função. A linha 12 marca o fim da seção de identificação de unidades.

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

Figura 19 – Seção de identificação das unidades do arquivo delphi.idel.

As linhas 10 e 11 constituem a seção de comparação e contém 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 da linguagem Delphi, conforme pode ser visto na árvore sintática de padrões na Figura 20, que corresponde à produção `func_impl` da gramática. Na descrição dos padrões é possível utilizar os *tokens* da gramática Delphi, como ocorre com os *tokens* '`function`', '`begin`', '`end`' e '`;`' no padrão da Figura 19. Quando esse padrão for identificado na árvore sintática do programa, a fase do processamento das unidades se inicia.

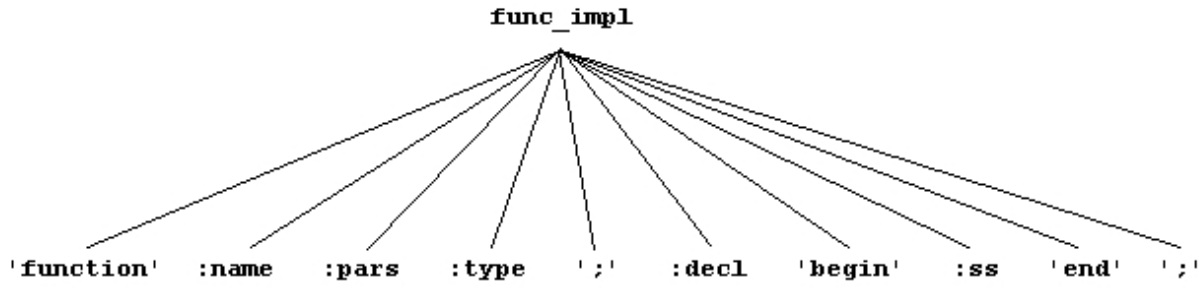

Figura 20 – Árvore de padrões da produção func_impl.

6.3.2. Processamento das Unidades

A seção de processamento das unidades é composta de zero ou mais passos iniciados com a palavra reservada **`step`**. Cada passo contém uma série de padrões que são testados e aplicados na árvore sintática caso seja possível (ver Seção 5.4.3). Os passos e os padrões mais importantes são discutidos nas subseções seguintes. A Seção 6.3.2.6 (passo `MakeGraph`) é a mais importante pois trata as construções das linguagens Delphi e ObjectPascal.

6.3.2.1. Passo FindFunction

Assim que uma unidade for identificada, o primeiro passo que é aplicado é o `FindFunction` (Figura 21) que tem a finalidade de encontrar cada função na árvore sintática e criar dois nós, chamados `init` e `exit`, que correspondem ao nó inicial e final do grafo, respectivamente. Esse passo é composto apenas pelo padrão `Function`, cuja declaração é iniciada na linha 3. Esse padrão é similar ao padrão de identificação de unidades, como se pode ver pela linha 11, que define o padrão.

As linhas 12 e 13 utilizam o comando IDeL **`declare`** para declarar os nós iniciais e finais. Na linguagem IDeL um nó é referenciado através de um sinal "`$`" seguido por um identificador válido, que é constituído por uma seqüência de letras e dígitos iniciada por uma letra. As linhas 14 e 15 correspondem à uma seção de atribuição de nomes simbólicos, e está relacionada com o mapeamento de nós. A atribuição da linha 15 serve para indicar que ocorre uma definição dos parâmetros formais da função no nó `init`. A declaração de um padrão é explicada com mais detalhes na Seção 6.3.2.6, que descreve o passo `MakeGraph`.

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

Figura 21 – Passo FindFunction.

6.3.2.2. Passo MarkStatements

O próximo passo a ser aplicado na árvore é o `MarkStatements` (Figura 22). Ele é composto pelo padrão `FooStatement`, declarado na linha 3. Esse padrão tem a finalidade de encontrar todos os comandos presentes na árvore sintática (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.

6.3.2.3. Passo LinkStatement

Após o passo `MarkStatements`, o passo `LinkStatement` atravessa 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 23, esse passo é

composto pelos padrões `Statement` (linha 3) e `List` (linha 13).

```
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 22 – Passo MarkStatements.

Uma referência na forma `$begin:s` significa referenciar o nó `$begin` da árvore de padrões na qual a meta-variável `:s` está unificada, ou seja, o nó `$begin` da árvore de padrões da meta-variável `:s`. Se uma meta-variável não for indicada, no caso de `$begin`, o nome simbólico é procurado na árvore de padrões encontrada. Caso um nome simbólico não esteja associado a nenhum nó da árvore, um nó nulo é retornado.

As linhas 9 e 10 da Figura 23 atribuem os nomes simbólicos `$begin` e `$end` aos nós `$begin:s` e `$end:s`, respectivamente. Isso significa que quando for solicitado o nó `$begin` e `$end`, os nós `$begin:s` e `$end:s` serão obtidos, respectivamente. Dessa forma a lista de comandos é conectada ao grafo. Analogamente, nas linhas 22 e 23, quando o nó `$begin` for solicitado, o nó `$begin:ss`, que é o nó inicial da lista de comando, será obtido. Quando o nó `$end` for solicitado, o nó `$end:s`, que é o nó final do último comando da lista, será obtido. As linhas 19 e 20, responsáveis pela topologia do grafo, criam uma aresta que parte do último nó da lista de comandos e vai até o primeiro nó do próximo comando.

```
 1     step LinkStatementList
 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 23 – Passo LinkStatementList.

6.3.2.4. Passo JoinStatement

O passo `JoinStatement` (Figura 24) 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. 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 24 – Pass JoinStatement.

6.3.2.5. Passo JoinToFunction

O passo JoinToFunction (Figura 25) utiliza o padrão `Function1` para conectar o nó inicial (`$init`) de uma unidade com o nó inicial da lista de comandos que constitui o corpo da função, `$begin:ss`. O nó final da lista de comandos (`$end:ss`) é conectado ao nó final da unidade (`$exit`). A seção de instrumentação (linhas 15 a 18) marca na tabela de implementação dos nós `$init` e `$exit` que serão realizadas duas alterações no código associado a esses nós.

6.3.2.6. 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, uma vez que a compreensão de tais padrões possibilita um entendimento do todo. O passo pode ser encontrado na íntegra no Apêndice B, juntamente com o restante da especificação IDeL do

instrumentador.

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

Figura 25 – Passo JoinToFunction.

Com a aplicação dos passos anteriores, qualquer referência aos nós `$begin` e `$end` de uma instrução devem ser entendida como as instruções que aparecem antes e após a instrução, respectivamente.

6.3.2.6.1. Comando de Repetição While-Do

A Figura 26 apresenta o padrão utilizado para instrumentar a instrução **`while`** da linguagem Delphi. Quando uma árvore de padrões que combine com ao padrão da linha 6 for encontrada na árvore sintática, esse padrão de instrumentação será aplicado. Considerando o comportamento semântico padrão de uma estrutura **`while`** genérica, as quatro tarefas que o instrumentador deve realizar são as seguintes:

```
 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 26 – Padrão que instrumenta o comando de repetição while.

1ª) criar um nó para a expressão de controle do **`while`**, utilizada para testar se o corpo da estrutura será executado ou não. O nó que representa essa expressão é chamado de `$control` e foi declarado na linha 7.

2ª) conectar os nós através de arcos que indicam os potenciais desvios de fluxo do programa. Os desvios de fluxo possíveis são: **i)** da instrução antes do **`while`** até a expressão de controle (linha 9), **ii)** da expressão de controle para o início do corpo do **`while`** (linha 10), **iii)** do fim do corpo do **`while`** para o início da expressão de controle (linha 11) e **iv)** da expressão de controle para a instrução que ocorre após o **`while`** (linha 12).

3ª) obter informações de contexto que permitam a correta instrumentação: **i)** quando uma exceção for lançada com o comando **`raise`**, o **`while`** deve ser interrompido (linha 14), **ii)**

quando um comando **`exit`** for encontrado, o nó que ocorre após o **`while`** deve ser utilizado (linha 15), **iii)** ocorre uma definição de variável na expressão de controle (linha 16) e **iv)** ocorre um p-uso de variável na expressão de controle (linha 17).

4ª) incluir pontas de prova (*checkpoint*) para monitorar e coletar informações quando o programa instrumentado for executado: **i)** um *checkpoint* deve ser inserido antes do **`while`** (linha 19), **ii)** um checkpoint deve ser inserido antes do corpo do **`while`** (linha 20), **iii)** um *checkpoint* deve ser inserido após o **`while`** (linha 21) e **iv)** um *checkpoint* deve ser inserido antes da expressão de controle (linha 22), para registrar quando ela é avaliada.

A linha 1 do padrão de instrumentação define o nome do padrão ("`While`", nesse caso). Esse nome não tem influência nenhuma na semântica da linguagem IDeL, serve apenas para documentação e para esclarecer um pouco o código. As linhas 2 a 4 declaram duas meta-variáveis que serão utilizadas por esse padrão. A meta-variável `:e` representa a produção `<expr>` da gramática, enquanto a meta-variável `:s` representa a produção `<stmt>`.

As linhas 5 e 6 contém o padrão que será utilizado para fazer a comparação com a árvore sintática da unidade. Quando uma árvore de padrões que combine com esse padrão for encontrada, o padrão de instrumentação `While` é aplicado. A linha 7 contém uma seção de declaração responsável pela criação de nós, relacionada com a 1ª tarefa. Um novo nó é criado e adicionado ao grafo, então à esse nó é atribuído o nome simbólico `$control`.

As linhas 8 a 12 são utilizadas para definir a topologia do grafo, e estão relacionadas com a 1ª tarefa. Nessas linhas são indicadas as arestas que definem os desvios de controle do **`while`**. A linha 9, por exemplo, cria um arco que conecta o nó `$begin` ao nó `$control`. Já a linha 10 cria um arco que conecta o nó `$control` ao nó `$begin` da árvore da meta-variável `:s`.

As linhas 13 a 17 constituem a seção de atribuição. Nessa seção são feitas atribuições de nomes simbólicos aos nós, de acordo com a 3ª tarefa. Na linha 17 o nome simbólico `$pusage` é atribuído ao nó. Este nome identifica um p-uso de uma variável naquele nó.

As linhas 18 a 22 definem que tipo de instrumentação que será realizada, de acordo com a 4ª tarefa. A linha 21 especifica que um *checkpoint* deve ser inserido logo após o comando `while`. Na seção de implementação (Seção 5.4.2.3), terceira parte do arquivo de especificação, essas declarações abstratas são convertidas em códigos concretos. A palavra reservada **`self`** referencia ao nó atual da árvore sintática. O grafo da instrução **`while`**, após a instrumentação, pode ser visto na Figura 27. Os nós `$begin:s` e `$end:s` são conectados quando o padrão responsável por instruções for processado, o padrão `While` tem a finalidade de instrumentar apenas o comando **`while`**, nada mais do que isso. Essa estratégia torna a descrição da instrumentação modular pois as construções da linguagem são tratadas individualmente.

6.3.2.6.2. Comando Condicional If-Then-Else

O mesmo esquema é utilizado para escrever a instrumentação do comando condicional `If-Then-Else` (Figura 28). As linhas 8 a 12 conectam os nós do grafo. A linha 18 conecta o nó da instrução que aparece antes do comando **`if`** com o primeiro nó do corpo do **`if`**. A linha 19 conecta o nó da instrução que aparece antes do comando **`if`** ao primeiro nó do corpo do comando **`else`**. A linha 20 conecta o último nó do corpo do comando **`if`** ao primeiro nó da instrução que aparece após o **`if`**. A linha 21 conecta o último nó do corpo do comando **`else`** ao primeiro nó da instrução que aparece após o **`if`**.

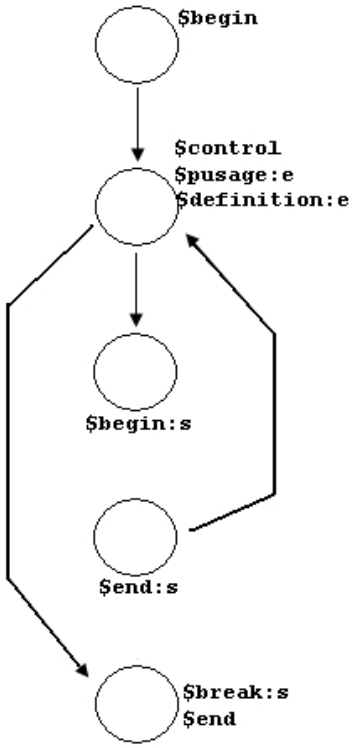

Figura 27 – Grafo do comando de repetição while.

Assim como no comando **`while`**, os nós `$begin:s1` e `$begin:s2` são conectados com os nós `$end:s1` e `$end:s2`, respectivamente, quando o padrão de instrumentação para instruções for aplicado na árvore. As linhas 13 a 17 fazem as atribuições necessárias para indicar ocorrência de lançamento de exceções e definições e usos de variáveis. As linhas 18 a 23 definem o tipo de instrumentação que será realizada com o comando **`if`**.

```
 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 28 – Padrão que instrumenta o comando condicional If-Then-Else.

6.3.3. Implementação

Na seção de processamento das unidades as declarações **`instrument`** são abstratas. A implementação da instrumentação depende da linguagem, pois as alterações que serão feitas no código devem ocorrer na linguagem em que o programa foi originalmente escrito. A seção de implementações é composta de diversos blocos implement, cada um definindo um tipo de instrumentação.

Por exemplo, a implementação da declaração **`instrument`** presente na linha 16 da Figura 25 é apresentada na Figura 29. Essa implementação é utilizada para alterar o início de uma unidade (função). O seu nome está definido na linha 10, assim como sua sintaxe, que indica que essa implementação deve ser utilizada para inserir um *checkpoint* antes (indicado pela palavra chave *before*) de um nó `$init`.

```
 1     implement
 2     var
 3           :name   as  [func_or_proc_or_method_name]
 4           :type   as  [proc_or_func_fptype]
 5           :pars   as  [fp_list]
 6           :decl   as  [impl_decl_sect_list]
 7           :ss     as  [stmt_list]
 8           :file   as  [string]
 9           :n  as  [constant]
10     init $init before
11           [func_impl< function :name :pars :type ; :decl begin :ss end ; >]
12     binding :n  to node $init
13     binding :file   to literal [string< '[:name].trace.tc' >]
14     as
15           [func_impl< function :name :pars :type ; :decl
16                 type TextFile = Text;
17                 var TraceFile : TextFile;
18
19                 function check(n : integer) : boolean;
20                 begin
21                       writeln(TraceFile, n);
22                       check := true;
23                 end;
24
25                 begin
26                       assign(TraceFile, :file);
27                       append(TraceFile);
28                       begin
29                             :ss
30                       end
31                 end ; >]
32     end implement
```

Figura 29 – Implementação do checkpoint init.

Após a linha 14, que contém a palavra reservada **`as`**, deve ser indicado o padrão pelo qual o padrão da linha 11 será substituído. Esse novo padrão pode apresentar qualquer construção da linguagem suportada pela gramática. O padrão 15 a 31 é utilizado para inserir instruções que irão abrir um arquivo onde serão armazenadas as informações coletadas em tempo de execução (linha 26 e 27) e declarar a função **`check()`** (linhas 19 a 23), que será utilizada para gravar no arquivo. Quando o programa instrumentado for executado, um arquivo de trace será gerado com o nome da função seguido da extensão `.trace.tc`

(`soma.trace.tc` para uma função chamada `soma()`, por exemplo). Assim, cada função instrumentada irá gerar um arquivo de *trace* separado.

A implementação exit utilizada na linha 17 da Figura 25 é similar. Ela faz com que o código associado ao último nó da unidade (função) seja alterado. Essa alteração simplesmente irá fechar o arquivo de *trace* aberto na implementação da Figura 29. Essa é uma operação esperada, já que a função terminou sua execução e o arquivo de *trace* não deve permanecer aberto para leitura ou escrita.

A implementação *checkpoint* da Figura 30 faz com que uma instrução `writeln()` seja inserida antes de uma outra instrução. Essa implementação é utilizada na linha 20 da Figura 26.

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

Figura 30 – Implementação para inserir um checkpoint antes de uma instrução.

6.4. Execução do Instrumentador

Como exemplo de uma instrumentação típica, a função `power()` da Figura 5(a), localizada no arquivo `teste.pas`, foi instrumentada. Os arquivos da gramática `delphi.l` e `delphi.y` foram utilizados para gerar o instrumentador `idel.delphi`, conforme mostra a Figura 17. O arquivo `delphi.idel` contém a descrição IDeL do instrumentador. A Figura

31 ilustra a função sendo instrumentada.

```
gustgr@lost% ./idel.delphi -p teste.pas -i delphi.idel -g teste -o teste.inst.pas
processing source...
processing instrumenter...
running ...
generating teste.power.dot
done.
```

Figura 31 – Instrumentação do programa teste.pas.

No comando utilizado para instrumentar o arquivo teste.pas, a opção `-p` serve para indicar 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 (ie. `nome_base.funcao.dot`) e a opção `-o` informa o arquivo final instrumentado.

Como resultado, obtém-se a função instrumentada no arquivo `teste.inst.pas` (Figura 32) e o grafo do programa no formato DOT (Figura 33(a)), no arquivo `teste.power.dot`. Um arquivo DOT é gerado para cada função. Este arquivo pode ser convertido para o formato GIF (Figura 33(b)) utilizando o pacote de software GraphViz (GASNER e NORTH, 2001). O Apêndice C apresenta exemplos das principais estruturas da linguagem Delphi e os grafos correspondentes, todos gerados com o instrumentador para Delphi desenvolvido neste trabalho.

```
 1       function power(x : integer ; y : integer) : integer;
 2
 3       var p ,z : integer;
 4       var TraceFile : TextFile;
 5
 6           function check (n : integer) : boolean;
 7           begin
 8               writeln (TraceFile , n);
 9               check := true;
10           end;
11
12       begin
13           assign(TraceFile , 'power.trace.tc');
14           append(TraceFile);
15
16           begin
17               begin
18                   writeln(TraceFile , 1);
19                   begin
20                       if check(1) and (y<0) then
21                       begin
22                           writeln(TraceFile , 2);
23                           p := y
24                       end
25                       else
26                       begin
27                           writeln(TraceFile , 4);
28                           p := - y
29                       end;
30                       writeln (TraceFile , 6)
31                   end
32               end;
33
34               z := 1;
35
36               begin
37                   writeln(TraceFile , 6);
38                   begin
39                       while check(7) and (p <> 0) do
40                       begin
41                           writeln(TraceFile , 8) ;
42                           begin
43                               z := z * x;
44                               p := p + 1
45                           end
46                       end;
47                       writeln(TraceFile , 11)
48                   end
49               end;
50
51               begin
52                   writeln (TraceFile , 11);
53                   begin
54                       if check(11) and (y<0) then
55                       begin
56                           writeln(TraceFile , 12);
57                           z := 1 div z
58                       end;
59                       writeln(TraceFile , 14)
60                   end
61               end;
62
63               power := z;
64               close(TraceFile)
65           end
66       end;
```

Figura 32 – Arquivo teste.inst.pas - Função power(x, y) instrumentada.

```
digraph gfc {
node [shape = doublecircle] 15;
node [shape = circle] 1;
/* pusage of y at 1 */
node [shape = circle] 2;
/* cusage of y at 2 */
node [shape = circle] 3;
node [shape = circle] 4;
/* cusage of y at 4 */
node [shape = circle] 5;
node [shape = circle] 6;
node [shape = circle] 10;
node [shape = circle] 8;
/* cusage of z at 8 */
/* cusage of x at 8 */
node [shape = circle] 9;
/* cusage of p at 9 */
node [shape = circle] 11;
/* pusage of y at 11 */
node [shape = circle] 12;
/* cusage of z at 12 */
node [shape = circle] 13;
node [shape = circle] 14;
/* cusage of z at 14 */
node [shape = circle] 7;
/* pusage of p at 7 */
1 -> 2;
1 -> 4;
3 -> 6;
5 -> 6;
6 -> 7;
7 -> 8;
10 -> 7;
7 -> 11;
11 -> 14;
11 -> 12;
13 -> 14;
2 -> 3;
4 -> 5;
8 -> 9;
9 -> 10;
12 -> 13;
14 -> 15;
 }
```

Figura 33 (a) – Árquivo teste.power.dot gerado pelo instrumentador.

O grafo da Figura 33(b) não é idêntico ao grafo da Figura 5(b), como se pode notar pelo tamanho do primeiro. A diferença é que na Figura 5(b) o grafo foi totalmente minimizado, porém na Figura 33(b), os nós que apresentam informações relevantes individuais (como definições ou usos) não foram mesclados com os nós subseqüentes para preservar as características individuais de cada nó. Isso se deve a uma decisão tomada quando o instrumentador foi implementado, que pode ser facilmente revertida se necessário.

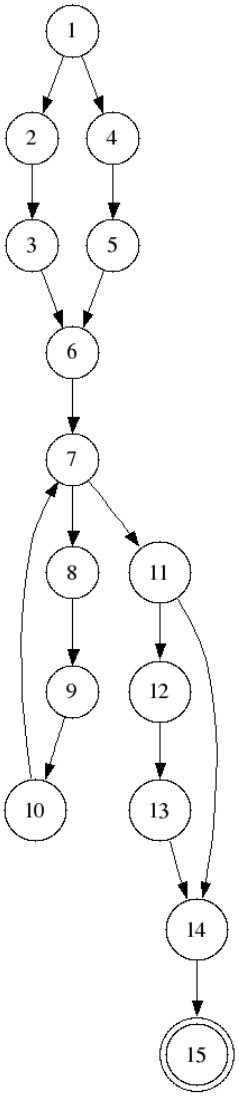

Figura 33 (b) – Grafo teste.power.gif gerado a partir do arquivo teste.power.dot.

CAPÍTULO 7 – CONCLUSÕES

O processo de instrumentação é de fundamental importância em diversas áreas da Engenharia de Software, sendo mais confiável e preciso quando realizado com o auxílio de um software instrumentador. Este trabalho atingiu seu principal objetivo, pois concluiu com sucesso a construção de um instrumentador para linguagem Delphi. O instrumentador foi desenvolvido com o apoio da linguagem de descrição de instrumentação IDeL, e o seu respectivo compilador IDeLGen. Com essa linguagem foi possível projetar um instrumentador flexível que pode ser atualizado sem grande complexidade, caso seja necessário.

O instrumentador funciona tanto para a linguagem de programação Delphi quanto para a linguagem Object Pascal, na qual a primeira se baseia. A instrumentação de um programa resulta em um programa instrumentado, isto é, decorado com instruções de monitoramento que não alteram a semântica do programa para que seja possível acompanhar sua execução em tempo real. Além disso, é gerado um grafo que representa o aspecto estático do programa, no formato XML ou DOT, que pode ser interpretado por diversas aplicações existentes.

Este trabalho apresentou um grau de desafio adequado a um projeto de conclusão de curso, já que algumas dificuldades foram encontradas enquanto o mesmo foi desenvolvido. A primeira barreira apareceu quando foi necessário escrever uma gramática livre de contexto para a linguagem Delphi. Mesmo com pesquisas e revisão bibliográfica não foi possível encontrar uma representação BNF completa da gramática, que teve que ser compilada ao longo do trabalho. Após a obtenção da gramática, foi necessário escrever a descrição do instrumentador na linguagem IDeL.

Apesar de muito bem projetada, a linguagem IDeL e o compilador IDeLGen não apresentam muita documentação e não são utilizados fora do meio acadêmico. Os únicos documentos disponíveis para consulta sobre os aspectos operacionais do sistema são dois

relatórios técnicos (SIMÃO et al A, 2002; SIMÃO et al B, 2002) que, apesar de grande utilidade para este trabalho, não abrangem toda a complexidade do gerador de instrumentador. Para a compreensão de parte dessa complexidade foi necessário analisar e estudar o código fonte do sistema, tarefa que tomou algum tempo devido à complexidade e extensão do código.

Este projeto cumpriu um segundo objetivo, não esperado quando este trabalho foi proposto, que é servir de documentação e manual de referência para a linguagem IDeL. Como o auxílio deste trabalho, se espera que o desenvolvimento de instrumentadores para outras linguagens futuramente se torne mais simples e rápido. Além disso, a construção do instrumentador para a linguagem Delphi serviu também como uma forma de validar o sistema IDeL & IDeLGen, mostrando sua viabilidade na construção de instrumentadores.

Para o aluno os benefícios acadêmicos foram valiosos. A revisão bibliográfica permitiu um estudo detalhado de ferramentas importantes como Lex e YACC, amplamente utilizadas no desenvolvimento de compiladores, além de um estudo mais aprofundado a respeito de teoria das gramáticas, o conhecimento dos fundamentos da linguagem de programação Delphi, do processo de instrumentação e de técnicas e critérios de teste de software. Todo o conhecimento adquirido com esta pesquisa não poderia ser obtido através de aulas convencionais, confirmando que o Trabalho de Conclusão de Curso tem valor inestimável na formação acadêmica dos alunos.

7.1. Trabalhos Futuros

Ao longo deste projeto e do constante estudo e manuseio do sistema IDeL & IDeLGen foi possível identificar melhorias que podem ser feitas ao sistema para que o mesmo seja mais eficiente e flexível. Trabalhos futuros podem incluir a alteração do sistema para que suporte a identificação mais de uma unidade em uma árvore sintática, evitando assim a generalização

das produções da gramática. No presente momento a linguagem IDeL suporta apenas a especificação e identificação de uma unidade em um arquivo de descrição IDeL. Apesar de funcionar bem para linguagens como C ou Java, onde toda a lógica do programa está contida dentro de funções e métodos, respectivamente, para linguagens como Delphi e Pascal, em que a lógica está contida em estruturas diferenciadas como funções e procedimentos, isso pode representar um problema. Além disso, seria interessante realizar alterações na interface da ferramenta IDeL para que a mesma exiba mensagens de erro mais intuitivas para o usuário.

Uma outra melhoria envolve alterações na gramática da linguagem IDeL para que seja possível especificar a criação de arcos diferenciados, úteis para indicar um desvio de fluxo secundário em um grafo, que se dá quando ocorre uma exceção ou um evento, por exemplo. Dessa forma permite-se que o entendimento do grafo seja facilitado, já que fica evidente quais desvios de fluxo não ocorrem normalmente no programa instrumentado. Outra possibilidade de continuação do projeto é desenvolver uma ferramenta de apoio ao teste estrutural para programas Delphi que utilizaria o instrumentador desenvolvido para realizar análise de cobertura, uma vez que o instrumentador mostrado neste trabalho pode ser empregado em diversos critérios de teste da técnica estrutural.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, A. V; SETHI, R.; ULLMAN, J. D. **Compilers: Priciples, Techniques and Tools.** 1 ed. Massachusets: Addison-Wesley, 1985. 344 p.

ARGRAWAL, H. et al. Mining system tests to aid software maintence. **IEEE Computer**. v. 31, n. 7, p. 64-73, Julho 1998.

BORLAND INPRISE CORPORATION, **Object Pascal Language Guide.** 1 ed. Borland, 1993, 234 p.

BORLAND SOFTWARE CORPORATION, **Delphi Developer's Guide.** 1 ed. Borland, 2002, 1106 p.

BORLAND EXCELLENCE ENDURES, **Delphi Language Guide**. 1 ed. Borland, 2004, 249 p.

CHOMSKY, N. Three models for the description of language. **IRE Trans. Of Information Theory.** v. 2, n. 2, p. 137-124, 1956.

DELAMARO, M. E. **Mutação de Interface: Um Critério de Adequação Interprocedimental para o Teste de Integração.** 1997. Tese (Doutorado em Física Computacional) – Instituto de Física de São Carlos, Universidade de São Paulo, São Carlos, Junho, 1997.

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

DELAMARO, M. E. **Linguagens Formais, Automatos e Teoria da Computação.** Notas de Aula, 1998.

DELAMARO, M. E. **Como Construir um Compilador Utilizando Ferramentas Java.** 1 ed. São Paulo: Novatec Editora Ltda, 2004, p. 1-7.

DUNIT. **An Xtreme testing framework for Borland Delphi programs**. Disponível em <http://dunit.sourceforge.net/>. Acesso em: 09 nov. 2005.

FREE SOFTWARE FUNDATION A. **GNU Bison**. Disponível em <http://www.gnu.org/software/bison/bison.html>. Acesso em: 09 nov. 2005.

FREE SOFTWARE FUNDATION B. **GNU Flex**. Disponível em <http://www.gnu.org/software/flex/ >. Acesso em: 09 nov. 2005.

GASNER, E. R; NORTH, S. C. An open graph visualization system and its applications to software engineering. **Software – Practice & Experience**, v. 30, n. 11, p. 1203-1233, 2001.

HOPCROFT, J. E.; ULLMAN, J. D. **Introduction to Automata Theory, Languages and Computation.** 1 ed. Massachusets: Addison-Wesley, 1979, 418 p.

LESK, M. E.; SCHMIDT, E.; JOHNSON, S. C. **The LEX & YACC Page.** Disponíve em <http://dinosaur.compilertools.net/>. Acesso em: 09 nov. 2005.

LEVINE, J.; MASON, T.; BROWN, D. **Lex & YACC**. 2 ed. atual. O'Reilly Media, 1992. 384 p.

MALDONADO, J. C. et al. **Introdução ao Teste de Software**. Minicurso. 2001.

MALDONADO, J. C.; CHAIM, M. L.; JINO, M. **Arquitetura de uma ferramenta de teste de apoio aos critérios potenciais usos**. In: XXII Congresso Nacional de Informática, 1989, São Paulo.

MCCABE, T. J.; BUTTER, C. W. Design complexity measurement and testing. **Communications of the ACM**, v. 32, n. 12, p. 1415-1425, Dezembro, 1989.

MYERS, G. J. **The Art of Software Testing**.1 ed. atual. Nova York: Wiley, 1979.

PRESSMAN, R. S. **Software Engineering**: A Practitioner's Approach. 4 ed. atual. McGraw-Hill, 1997.

RAPPS, S.; WEYUKER E. J. Selection software test data using data flow information. **IEEE Transactions on Software Engineering**, v. 11, n. 4, p. 367-375, Abril, 1985.

SEBESTA, R. W. **Concepts of Programing Languages**. 4 ed. Massachusets: Addison-Wesley, 1999, 669 p.

SEMANTIC DESIGNS. **Design Maintence System.** Disponível em <http://www.semdesigns.com/Products/DMS/>. Acesso em 09 nov. 2005.

SIMÃO, A. S. et al A. **Software Product Instrumentation Descriptuion**. 2002. Relatórios Técnicos do ICMC, N° 157 – Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo, São Carlos, 2002.

SIMÃO, A. S. et al B. **A Language for the Description of Program Instrumentation and Automatic Generation of Instrumentators** . Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo, São Carlos, 2002.

VINCENZI, A. M. R. et al. **JaBUTi –** Java Bytecode Understanding and Testing. 2003. Relatórios Técnicos do ICMC – Instituto de Ciências Matemática e Computação, Universidade de São Paulo, São Carlos, Março, 2003.

WIRTH, N. **Algorithms and Data Structures.** 1 ed.Prentice Hall, Novembro, 1985. 288 p.

APÊNDICE A – GRAMÁTICA DELPHI

A.1 Especificação Léxica – `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++;
                        }
                }
        }
}
```

A.2 Especificação Sintática – `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
        | proc_impl
        | func_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
        ;

proc_impl
        : PROCEDURE
func_or_proc_or_method_head proc_block
        ;

func_impl
        : FUNCTION
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 – DESCRIÇÃO DO INSTRUMENTADOR DELPHI

```
################################################
##  delphi.idel                                ##
##                                             ##
##  Instrumentador para Delphi/ObjectPascal    ##
##                                             ##
##    Gustavo Rondina                          ##
##                                             ##
##                                 Nov/2005    ##
################################################

## 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
       :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
       [func_impl< function :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
       :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
       [func_impl< function :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 $beginto  $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 $beginto  $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
       :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
       [func_impl< function :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

## O tratamento de excessao nao funciona muito bem,
## poderia estar melhor.
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
       $end:s       ->  $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

## O goto está aqui só para constar, na verdade
## nenhum desvio no grafo é gerado.
pattern Goto
var
       :g     as  [goto_stmt]
match
       [stmt< :g >]
graph
       $begin ->  $end
end pattern

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

## 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 close(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
       :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
       [func_impl< function :name :pars :type ; :decl begin :ss end ; >]
binding      :n     to node $init
binding :fileto literal [string< '[:name].trace.tc' >]
as
       [func_impl< function :name :pars :type ; :decl
              type TextFile = Text;
              var TraceFile : TextFile;

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

              begin
                     assign(TraceFile, :file);
                     append(TraceFile);
                     begin
                            :ss
                     end
              end ; >]
end implement

## Fecha o arquivo assim que termina a funcao
implement
var
```

```
      :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
      [func_impl< function :name :pars :type ; :decl
            type TextFile = Text;
            var TraceFile : TextFile;

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

            begin
                  assign(TraceFile, :file);
                  append(TraceFile);
                  begin
                        :ss
                  end
            end ; >]
binding     :n     to node      $exit
binding :fileto literal [string< '[:name].trace.tc' >]
as
      [func_impl< function :name :pars :type ; :decl
            type TextFile = Text;
            var TraceFile : TextFile;

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

            begin
                  assign(TraceFile, :file);
                  append(TraceFile);
                  begin
                        :ss ;
                        close(TraceFile)
                  end
            end ; >]
end implement

end instrumenter
```

APÊNDICE C – EXEMPLOS INSTRUMENTADOS

Os exemplos são meramente ilustrativos, não têm aplicação prática. Possuem apenas valor sintático.

C.1 Comando Repeat-Until

```
function repeatuntiltest(a : integer) : integer;
begin
      repeat
            a := a + 1;
            a := a + 1
      until (a < 0);

      repeatuntiltest := 0
end;
```

Figura 34(a) – Exemplo utilizando o comando Repeat-Until

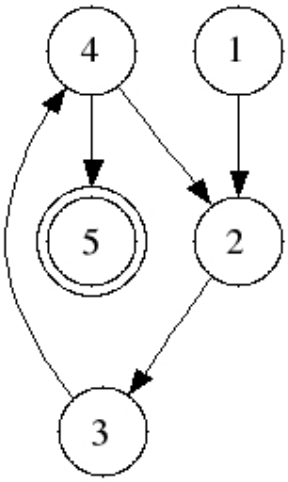

Figura 34(b) – Grafo correspondente.

C.2 Comando For-To-Do

```
function fortotest(a : integer) : integer;
var
      x : integer;
begin
      x := 10;

      for a := 0 to x + 6 do
      begin
            a := a + 1;
            a := a * a
      end;

      fortotest := 0
end;
```

Figura 35(a) – Exemplo utilizando o comando For-To-Do

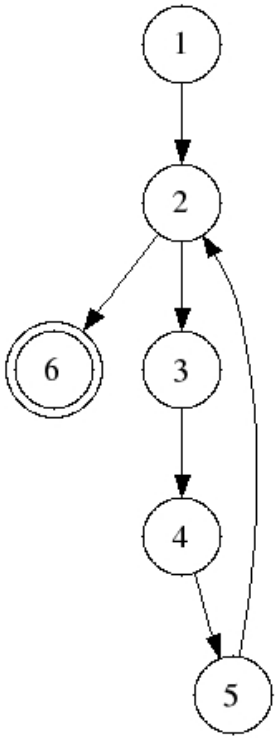

Figura 35(b) – Grafo correspondente.

C.3 Comando Case

```
function casetest(a : integer) : integer;
var
      c : integer;
begin
      c := 10;

      case a of
            1      : writeln(a);
            2      : writeln(a);
            3      : writeln(a);
      else
            begin
                  a := a + 1;
                  a := a * a;
                  a := a - 10;

                  if a > 0 then
                        a := a + 1
                  else
                        a := a - 1
            end
      end;

      casetest := 0
end;
```

Figura 36(a) – Exemplo utilizando o comando Case

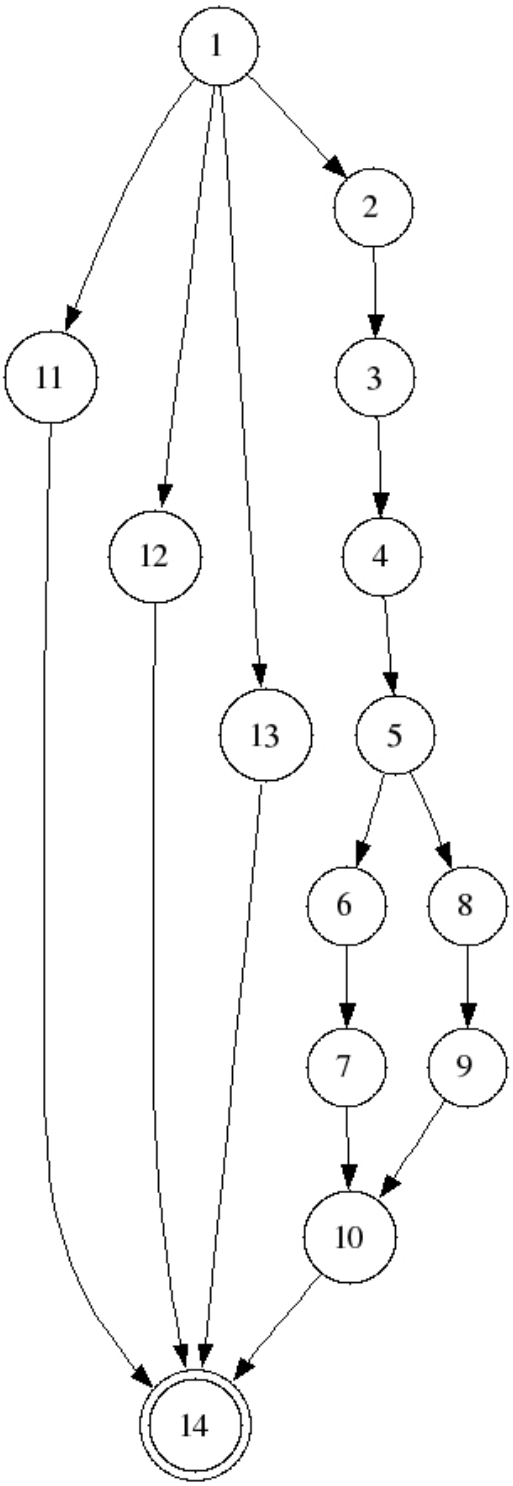

Figura 36(b) – Grafo correspondente.