UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA ELÉTRICA
DEPARTAMENTO DE ENGENHARIA DA COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

FEVEREIRO 1991

GFC - UMA FERRAMENTA MULTILINGUAGEM PARA GERAÇÃO DE GRAFO DE PROGRAMA

Por: Mauro Carnassale

Orientador: Prof. Dr. Mario Jino

DISSERTAÇÃO APRESENTADA À FACULDADE DE ENGENHARIA ELÉTRICA, DA UNIVERSIDADE ESTADUAL DE CAMPINAS, COMO REQUISITO PARCIAL PARA OBTENÇÃO DO TÍTULO DE MESTRE EM ENGENHARIA ELÉTRICA.

Este exemplar corresponde à redação final da tese defendida por Mauro Carnassale e aprovada pela Comissão Julgadora em 25 de fevereiro de 1991.

Mario Jino

À minha filha Karina

A G R A D E C I M E N T O S

Quero deixar registrado meus mais sinceros agradecimentos, em primeiro lugar, ao nosso Bom Deus, Criador e Mantenedor da vida e de tudo que é bom;

Ao Prof. Dr. Mario Jino pela receptividade, dedicada e eficiente orientação, e oportunidades oferecidas;

Ao Dr. Fuad Gattaz Sobrinho pelo dinamismo e profundo conhecimento transmitidos;

Ao Prof. Dr. Arthur João Catto, por ter me ensinado pacientemente as primeiras e importantes lições da ciência da computação que até hoje se constituem em um sólido alicerce;

Ao Prof. Dr. Léo Pini Magalhães pelas primeiras orientações na FEE, pela amizade, e incentivo sempre oportuno;

À Profa. Dra. Beatriz Mascia Daltrini pelo seu apoio, atenção e simpatia;

Ao Prof. Dr. Clésio Luís Tozzi pelas aulas ministradas, seu interesse, e confiança depositada em mim;

Ao Prof. Dr. Maurício Ferreira Magalhães pelas aulas ministradas e constante disposição em ajudar;

Aos meus amigos José Carlos Maldonado e Marcos Lordello Chaim, pela colaboração e espírito de equipe demonstrados durante todo o tempo, especialmente pela inestimável contribuição na formalização da linguagem LI e por um número incontável de pequenas e grandes ajudas; igualmente ao Rubens pelo sua ajuda principalmente na confecção do Manual do Usuário da GFC;

Aos meus amigos do Projeto Fábrica de Software por partilharem seus conhecimentos e amizade, particularmente ao Clênio, à Fernanda, ao Fernando, ao Miguel, ao Oscar, ao Silvio, à Sônia, ao Tobar e ao Wagner;

Aos meus amigos do Banco do Brasil que me proporcionaram mais uma vitória na carreira profissional;

Aos meus pais, pela cuidado e carinho sempre presentes;

À minha noiva pela sua incansável ajuda e por ter suportado pacientemente minha ausência e, meus monólogos sobre este trabalho, quando presente;

E a todos quantos de alguma forma utilizarão este trabalho para o avanço da ciência em seu emprego pacífico.

ÍNDICE

L I S T A D E F I G U R A S

L I S T A D E T A B E L A S

R E S U M O

Grafos de programa têm sido utilizados há muito tempo e, apesar de serem um conceito tradicional, sua aplicabilidade atual é grande e crescente. Muitas ferramentas e alguns ambientes utilizam-se dessa representação da estrutura lógica do programa para atingirem seus objetivos.

O propósito deste trabalho é apresentar a arquitetura, aspectos fundamentais de implementação e principais aplicações de uma ferramenta que produz o grafo de programa a partir de programas escritos em diversas linguagens procedimentais.

Os programas são traduzidos para uma linguagem intermediária, denominada LI, a partir da qual obtêm-se o grafo de programa e outros produtos relevantes. A ferramenta também aceita programas escritos diretamente em LI. Sua finalidade principal é apoiar outras ferramentas ou aplicações que se utilizam do grafo de programa.

C A P í T U L O I

INTRODUÇÃO

Grafos de programa são, há muito tempo, utilizados na análise de programas em geral [HEC77] e, em particular, em aplicações na otimização de programas e na análise de fluxo de dados [GRI71], [WAI84].

Mesmo informalmente, na busca de compreensão do código ou na detecção de erros, programadores desenhavam (e até hoje alguns ainda o fazem) o fluxograma do programa fonte que, na verdade, pode ser considerado o grafo do programa se fizermos uma abstração das figuras que indicam os processamentos. Outra atividade relacionada com o grafo de programa, não menos frequente, é a instrumentação de programas, que consiste em inserir instruções no código fonte para coletar informação durante a execução do programa.

Alguns compiladores modernos, como TURBO C, da Borland International [TUR88] e utilitários como o DBX da Sun Microsystems [DEB90], possuem poderosas rotinas de depuração que, entre outras tarefas, mostram, em tempo de execução, a sequência em que instruções do programa são executadas, permitindo um ganho razoável na produtividade e na qualidade da programação.

Sem dúvida, ao tratarmos sobre grafo de programa e suas aplicações, estamos tocando num ponto sensível da informática, que é o desenvolvimento de software. A capacidade de processamento (instalado) cresceu 40% (quarenta por cento) ao ano entre 1963 e 1983, enquanto a produtividade de desenvolvimento do software registrou uma elevação de apenas 3% a 8% (três a oito por cento) ao ano, nesse mesmo período [HOR84].

Em contraste com o decrescente custo do hardware, o custo do software vem crescendo a cada ano. Segundo Boehm [BOE87] se o software em todo o mundo continuar crescendo à taxa de 12% (doze por cento) ao ano, teremos em 1995 um gasto de 450 (quatrocentos e cinquenta) bilhões de dólares somente com software.

Uma característica marcante da Engenharia de Software tem sido a automatização do processo de desenvolvimento de software, através de ambientes automatizados que se propõem a suportar todas as atividades desse processo [AKH89]. As necessidades criadas pelo processo de automatização forçaram o surgimento de soluções alternativas quanto aos métodos, técnicas e ferramentas até então usados. Nesse contexto, tem sido reconhecida, entre outras, a necessidade de construir ferramentas automáticas e tanto quanto possível genéricas, e que sejam também flexíveis e expansíveis.

Objetivando, principalmente, contribuir para a solução desse problema, apresentaremos uma ferramenta multilinguagem,

denominada GFC, que traduz programas fontes escritos em diversas linguagens procedimentais para programas em uma linguagem intermediária, definida neste trabalho e que denominaremos LI. A partir de programas traduzidos ou escritos diretamente em LI, a GFC gera o grafo de programa, além de prover facilidades para outras ferramentas que se apóiam na estrutura lógica da codificação.

Na organização deste trabalho procuramos, em primeiro lugar, definir grafo de programa e apresentar sua importância e principais usos. Em seguida, no Capítulo III, discutimos a necessidade de automatização de grafo de programa, os passos requeridos para essa automatização e a abordagem adotada na solução proposta. Uma linguagem intermediária, definida no Capítulo IV, resulta como produto principal dessa abordagem. No Capítulo V, são descritos a GFC, sua arquitetura, aspectos importantes da implementação e estruturas de dados utilizadas. Mostramos também como configurar a GFC para uma linguagem específica. No Capítulo VI apresentamos, através de exemplos de diversas linguagens procedimentais, a geração do grafo de programa para cada comando da LI. Finalmente, são relatados no Capítulo VII as conclusões e os principais resultados obtidos. Apresentamos também algumas extensões e sugestões para trabalhos futuros. No apêndice são apresentadas as tabelas configuradoras das linguagens C e COBOL para a GFC, atualmente implementadas.

C A P Í T U L O I I

IMPORTÂNCIA E PRINCIPAIS USOS DO GRAFO DE PROGRAMA

2.1 GRAFO DE PROGRAMA

Iniciaremos este capítulo definindo grafo de programa e formulando alguns conceitos correlacionados como bloco, desvio e sequência. Em seguida, mostraremos a utilidade e a importância do grafo de programa, destacando seu emprego em teste de programas e na evolução do software.

Primeiramente, vejamos o que é um fluxo de controle em um programa. Na verdade, um programa em computador é uma sequência finita de instruções válidas [OVI80]. O controle da sequência dessas instruções é uma abstração da execução sequencial implementada por contadores de instrução ("program counters") existentes em computadores. Seleções, repetições e desvios incondicionais são abstrações de um mecanismo de controle muito simples, mas extremamente poderoso, que circuitos de máquina oferecem para a modificação explícita do indicador de instruções. A descrição da ordem de execução dos comandos de um programa é chamada de fluxo de controle [GHE87].

Aho e Ullman [AHO73], apoiados em conceitos da teoria dos grafos [DEO74], [FUR73], definem grafo de fluxo como sendo: um grafo dirigido rotulado "G" contendo um nó distinto "n", tal que cada nó de "G" é acessível a partir de "n". O nó "n" é chamado de nó inicial. Mais especificamente, definem grafo de fluxo de controle de programa, também chamado de grafo de programa, como sendo um grafo de fluxo no qual cada nó do grafo corresponde a um bloco básico do programa.

Antes porém, de definirmos o que vem a ser bloco básico, devemos identificar os mecanismos que governam o fluxo de controle sobre instruções individuais - sequência e desvio.

- SEQUÊNCIA é o mecanismo mais simples, usado para indicar que a execução dos comandos que aparecem escritos no programa na ordem "A"; "B"; são executados nessa mesma ordem, ou seja, a execução de "B" deve seguir-se imediatamente à execução do comando "A".

- DESVIO é usado para alterar a sequência de execução dos comandos. O desvio pode ser especificado de três formas: seleção, repetição e desvio incondicional.

Por sua vez, bloco básico, bloco sequencial ou simplesmente bloco é um conjunto de comandos de um programa "P", que aparecem escritos imediatamente um após o outro, tal que, ao executarmos o primeiro comando do conjunto, a execução prossegue (sequencialmente) até o último comando desse conjunto, sem nenhum desvio.

Para atingirmos o objetivo de obter o fluxo de controle de um programa "P" de uma linguagem procedimental "L", devemos fazer as associações entre:

- blocos de "P" (onde não há desvio no fluxo de execução de P) e nós (vértices de um grafo);

- desvios e arcos (que ligam os nós entre si).

Sejam os nós "i" e "j" de um grafo de programa correspondentes aos blocos "i" e "j", respectivamente, do programa; um arco é traçado do nó "i" para o nó "j" se a execução do bloco "j" seguir-se à execução do bloco "i", na sequência de execução do programa.

Considere o seguinte programa para o algoritmo Euclideano, extraído da página 910 da referência [AHO73], cuja saída principal deve ser o máximo divisor comum entre os inteiros "p" e "q":

```
            read p
            read q
loop:       r <- remainder (p,q)
            if r = 0 goto done
            p <- q
            q <- r
            goto loop
done:       write q
            halt
```

Os quatro blocos básicos do programa são:

```
Bloco 1   read p
          read q
Bloco 2   loop:    r <- remainder (p,q)
                   if r = 0 goto done
Bloco 3   p <- q
          q <- r
          goto loop
Bloco 4   done: write q
                halt
```

O grafo de programa, representado por um dígrafo correspondente é mostrado na Figura 2.1.

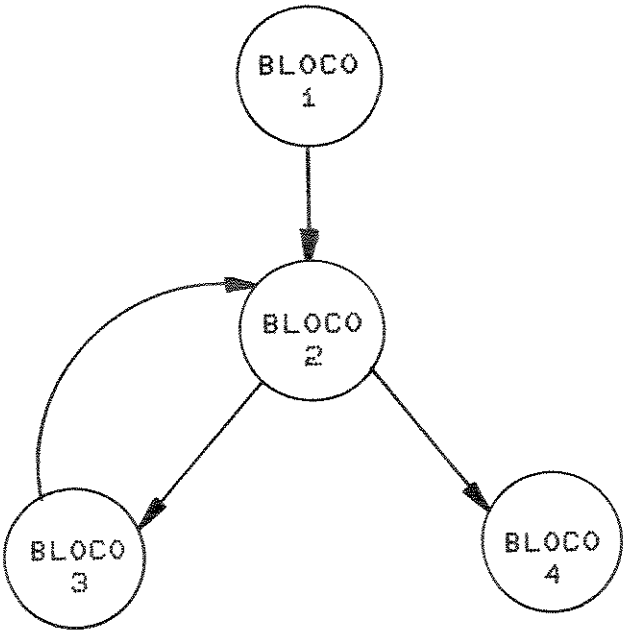


Figura 2.1: Grafo de Programa para o Algoritmo Euclideano

Observemos que o grafo de programa, que consiste basicamente em um grafo dirigido, é simplesmente uma notação para representar, com precisão e clareza, algébrica ou geometricamente, o fluxo de controle lógico de uma unidade de programa.

Não abordaremos linguagens não procedimentais pois, segundo Martim [MAR85], tal classe de linguagens descreve quais resultados devem ser obtidos, mas não especifica a sequência de passos (procedimento) pela qual estes resultados serão obtidos. Como exemplo de linguagens não procedimentais podemos citar: SMALLTALK, LISP e PROLOG. Por outro lado, ADA, ALGOL, C, COBOL, FORTRAN, PASCAL são exemplos de linguagens procedimentais.

Concluindo, queremos ressaltar que, obter o grafo de programa significa conhecer a estrutura de controle do projeto procedimental, isto é, conhecer a estrutura lógica da codificação, que é de grande importância na aquisição de conhecimento sobre o software, não apenas de sua funcionalidade, como principalmente, do conjunto de suas características internas [HET87].

2.2 GRAFO DE PROGRAMA E TESTE DE PROGRAMA

Nesta seção veremos quão importante é o grafo de programa na obtenção e avaliação da qualidade do software, onde qualidade é definida como "adequação a: requisitos funcionais e de desempenho estabelecidos explicitamente; padrões documentados e explícitos de desenvolvimento e; características que são esperadas de todo software produzido profissionalmente" [PRE87]. Mostraremos também que a geração do grafo de programa é pré-requisito para a implementação de uma estratégia de teste de programa.

O glossário do IEEE contém a seguinte definição: "teste é o processo de se experimentar ou avaliar um sistema por meios manuais ou automáticos, de modo a verificar se ele atende às necessidades especificadas, ou a identificar as diferenças entre os resultados esperados e reais" [IEE83].

Uma extensão dessa definição feita por Hetzel [HET87] é que "toda e qualquer ação realizada com o intuito de mensurar ou aprimorar uma característica do software pode ser considerada uma atividade de teste".

São diversas as técnicas encontradas na literatura para teste de programa [BEI82], [BOR87], [DUN84], [HET87], [PRE87], [VEL86]. Uma solução completa seria a de executar o programa esgotando todos os possíveis dados de entrada, porém, o teste exaustivo torna-se quase sempre impraticável, devido a restrições

materiais (custo, tempo, recursos, etc.). Assim sendo, passou-se a tentar definir quais as características que testes de programas deveriam apresentar, de modo a garantir certo grau de qualidade.

Primeiramente, convém lembrar que as técnicas de teste estão classificadas em duas grandes categorias: Testes Estruturais (Caixa Branca) e Testes Funcionais (Caixa Preta) [MYE79], [PRE87]. Essa classificação baseia-se na origem dos dados de teste: os que geram dados de teste a partir da especificação ou do entendimento do que o programa deva realizar; e os que o fazem em função da estrutura lógica do programa.

Veremos a seguir, em que se concentram essas duas categorias de técnicas de teste:

- Teste Funcional - as técnicas baseadas na especificação propõem a definição de casos de teste, de tal maneira que abranjam todas as situações válidas e inválidas (sobrecarga, etc.) a que um programa deva ser submetido a fim de satisfazer à especificação. A obtenção de um bom conjunto de casos de testes funcionais para um programa exige que os casos normais e anormais (fora do domínio de entrada) de operação de programa sejam cobertos de modo planejado e sistemático, pondo à prova todas as entradas e saídas do programa bem como suas funções.

- Teste Estrutural - como é materialmente impossível realizarmos testes que cubram a totalidade dos casos possíveis, devemos nos limitar a um subconjunto de casos. O problema é,

então, relacionarmos um subconjunto do domínio de entrada, de tamanho razoável, que tenha alta (ou maior) probabilidade de detectar um número grande (ou o maior número) de erros. Com base no código fonte, poderíamos desejar exercitar todos os caminhos possíveis dentro do mesmo. Como isto não é possível, na maioria dos casos práticos, as técnicas de Caixa Branca baseiam-se em critérios de cobertura que limitam os caminhos requeridos a subconjuntos do conjunto de todos os caminhos. Os métodos são chamados "de cobertura", pois se dedicam a executar "pelo menos uma vez" cada comando do programa, ou cada decisão, ou cada condição dentro de uma decisão, ou uma combinação destes [BOR87].

Ultimamente, têm sido introduzidos critérios de Teste Estrutural baseados em análise de fluxo de dados [MAL88a], [RAP85], que associam os tipos de ocorrências de variáveis (definição e uso) à estrutura lógica do programa.

Grafos de programa são indispensáveis ao teste estrutural pois, obter o grafo de programa, é o primeiro passo a ser dado quando da aplicação de métodos básicos de teste estrutural [HET87]. Existem muitos métodos propostos na literatura do tipo Caixa Branca para testar o software [CHU87], [MAL88a], [MYE79], [NTA88], [PRE87], [RAP85]. Tais métodos vêm cada vez mais sendo aprimorados objetivando, com um mínimo de casos de teste, submeter o programa a situações diversas a fim de descobrir o maior número de tipos de erros.

2.3 GRAFO DE PROGRAMA E EVOLUÇÃO DO SOFTWARE

Um aspecto da Engenharia de Software que ultimamente vem ganhando destaque é a Evolução do Software [NG90]. Segundo Lehman [LEH80], o termo "Evolução do Software" pode ser utilizado em lugar de "Manutenção do Software", por ser este último mais adequado às mudanças ocasionadas em elementos com propriedades físicas (alteram-se com o passar do tempo), ao contrário de elementos lógicos (como software) que evoluem de modo a melhorar seu desempenho e atributos.

A Evolução busca, entre outras metas, preparar o software existente para mudanças [SOB84], isto é, dotar o software de propriedades que lhe permitam ser facilmente adaptado a novas situações. A Evolução está baseada na análise da estrutura do código fonte, que por sua vez depende do grafo de programa.

Os métodos que buscam conferir maior qualidade ao software enfrentam, em geral, um problema comum - a dependência direta ou indireta da especificação. Geralmente, partem do pressuposto de que a especificação está correta e completa. Como podemos garantir isso? Seja qual for a abordagem escolhida para medirmos a qualidade de um programa, não podemos olvidar que o problema fundamental, e em muitos casos subjetivo, é a qualidade da especificação.

Conforme podemos observar na Tabela 2.1, há três situações em que podemos encontrar a especificação em relação ao código fonte ativo. Na situação "A", existe somente o código fonte. Na situação "B", a especificação existe, porém incompleta e desatualizada, portanto não há correspondência entre o código fonte e a especificação. Na situação "C" encontramos a situação ideal, onde a especificação existe, está completa e atualizada, e condiz com o código fonte.

Tabela 2.1: Situações do Software Atual

SITUAÇÃO	ESPECIFICAÇÃO	RELAÇÃO:ESPECIFICAÇÃO/ CÓDIGO FONTE
A	Inexiste	Inexiste
B	Existe, porém incompleta ou desatualizada	O código não confere com a especificação
C	Existe, completa e atualizada	O código condiz com a especificação

Segundo Ng e Yeh [NG90], há, somente nos EUA, US$ 300 bilhões em software nas situações "A" e "B" (dados de 1990). Mesmo que todos passassem a utilizar novas técnicas e métodos de desenvolvimento que garantissem a qualidade do software a ser produzido, o que fazer com todo esse software atualmente em uso? O pouco software que se encontra na situação "C", sem diligente cuidado pode, naturalmente pela mudança de requerimentos, passar para a situação "B".

Pelo menos em nosso país, a urgência nas modificações, muitas vezes por força de alterações legais (por exemplo, as muitas e repentinas medidas provisórias que começam a vigorar instantaneamente) aliadas a uma política de recursos humanos deficiente, principalmente no que se refere a treinamento de pessoal, não permitem a formação de uma consciência profissional que dê a devida importância à produção e atualização sistemática da documentação. A incompreensão de que o grande problema é a manutenção do software impede que se busque com a devida atenção uma solução eficaz e eficiente...

A Figura 2.2 mostra a vida útil de sistemas de software em geral e descreve a proporção do gasto em relação ao custo total após a liberação do sistema.

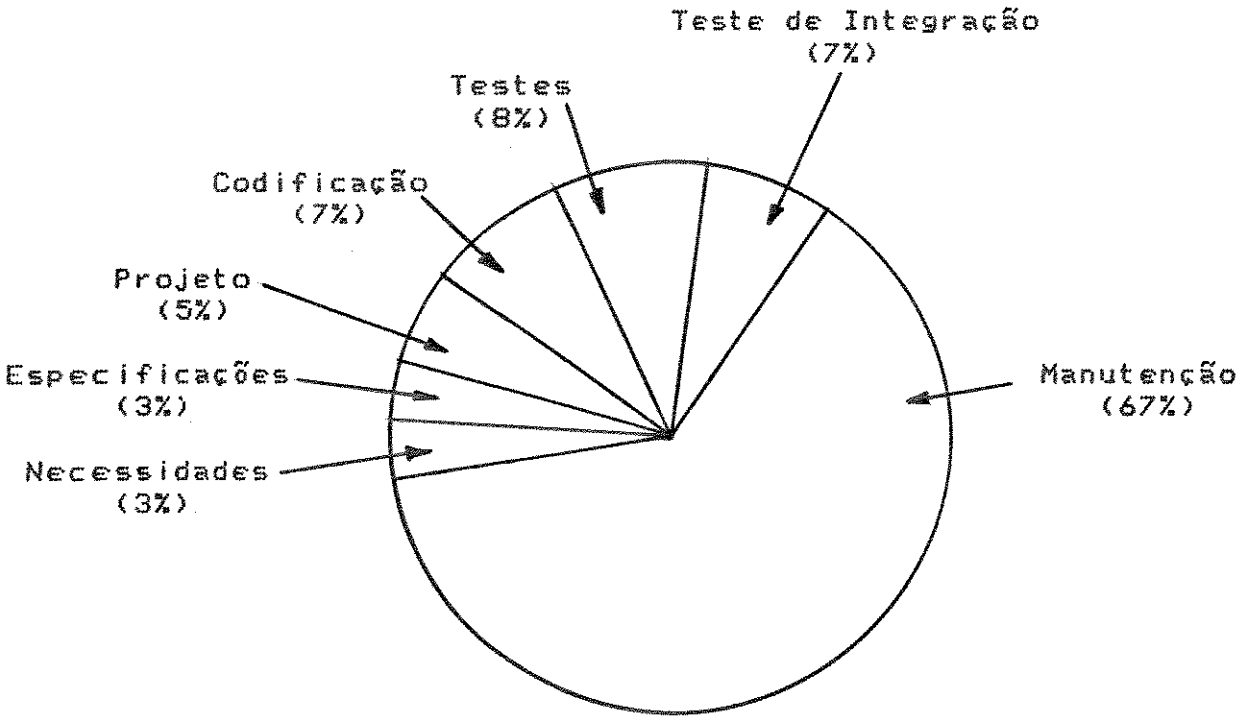


Figura 2.2: Distribuição dos Custos de Sistemas de Software em Geral

Como podemos observar na Figura 2.2, sessenta e sete por cento do custo de toda a vida útil do sistema - o dobro da quantia gasta durante o desenvolvimento - é dispendido na manutenção. Testes e depuração respondem por 15% (quinze por cento) do custo total; Enquanto 15% (quinze por cento) é gasto na construção, 82% (oitenta e dois por cento) do investimento é consumido em acertos e conservação [PAG88].

Esse quadro caótico faz com que, em geral, 80% (oitenta por cento) ou mais, da mão de obra disponível de um centro de desenvolvimento de sistemas precise estar voltada para a manutenção de sistemas. E mesmo assim, os resultados deixam a desejar, pois os profissionais da área se vêm às voltas com sistemas nas situações "A" e "B" de cujo desenvolvimento muitas vezes não participaram. Mesmo que o tivessem feito, como guardar na memória detalhes de implementação ou soluções lógicas adotadas há muito tempo?

Diante dessa realidade assustadora concluímos que a única fonte segura de que dispomos para obter informações sobre sistemas nessas situações, é o código fonte, produzido quase sempre de forma artesanal.

Uma solução para reverter esse quadro, apresentada por Sobrinho [SOB84], é a remodularização e a obtenção da especificação do software a partir do código fonte, onde a primeira fase dessa metodologia consiste em preparar o software existente para mudança. O passo inicial dessa fase é avaliar a estrutura do software original. Tal análise, em primeira instância, é obtida através da geração do grafo do programa.

C A P Í T U L O I I I

GERAÇÃO AUTOMÁTICA DO GRAFO DE PROGRAMA

3.1 INTRODUÇÃO

No capítulo anterior examinamos grafo de programa e algumas aplicações. Neste capítulo, apresentaremos resumidamente algumas ferramentas automatizadas disponíveis no mercado, e outras no âmbito da pesquisa. Encerraremos este capítulo apresentando a abordagem adotada na solução proposta, onde destacamos a característica multilinguagem da ferramenta.

3.2 FERRAMENTAS QUE SE UTILIZAM DO GRAFO DE PROGRAMA

Posto que mal difundidas, os profissionais da engenharia de software podem contar com algumas ferramentas úteis que se apóiam em grafo de programa. Dentre estas podemos citar as ferramentas para medição da cobertura dos testes; para análise lógica de programas; para otimização de programas; e para análise de fluxo de dados associado à estrutura da codificação.

Primeiramente vamos considerar uma ferramenta que mede e avalia a cobertura de testes. Como exemplo temos o software da Programming Aids, Inc. denominado CHECKOUT, que instrumentaliza o código fonte fazendo um acompanhamento de cada instrução, computando o número de vezes que cada instrução foi executada durante o processamento. O CHECKOUT tem capacidade de acumular informações de até oito execuções consecutivas do mesmo programa.

Na mesma categoria encontramos o SOFTOOL, produzido pela SOFTOOL Corporation. O SOFTOOL é um conjunto integrado de programas dirigido a ambientes de programação COBOL ou FORTRAN. A técnica básica empregada é a instrumentação, associada ao desenvolvimento estruturado top-down. Os resultados da cobertura dos testes são apresentados em nível de programa, parágrafo ou instrução.

Uma segunda categoria de ferramentas é a dos analisadores da estrutura de programas, usados principalmente para propiciar facilidades quanto ao entendimento do programa e como auxiliar na tarefa de manutenção do software. Como subfunção, podem ser usados para documentação e teste de programas. Como exemplo encontramos o SCAN/370, dirigido a ambientes de programação COBOL. O SCAN/370 comenta cada parágrafo ou seção e sugere maneiras alternativas de codificação; assinala o domínio de cada comando PERFORM; lista os trechos de código inacessíveis; e produz um diagrama hierárquico do programa.

Uma terceira categoria é a dos otimizadores. A otimização de programas é um recurso oferecido como opção em diversos compiladores. Um exemplo conhecido é o CAPEX, desenvolvido originalmente para otimização automática de programas escritos em COBOL e para fornecer informações que auxiliem no ajuste fino do desempenho de programas. Atualmente o CAPEX está adaptado para detectar parágrafos e instruções não exercitados durante a execução [HET87].

Queremos dar destaque à última categoria aqui abordada: ferramentas que, baseadas na análise de fluxo de dados, associam os tipos de ocorrências de variáveis (definição e uso) à estrutura lógica do programa. Encontram-se no âmbito acadêmico. Dentre elas queremos citar algumas que realizam análise de cobertura: a ferramenta criada por Herman [HER76], que suporta o primeiro critério de teste baseado na análise de fluxo de dados; a ASSET que apóia a família de critérios baseada na análise de fluxo de dados proposta por Rapps e Weyuker [FRA85], [FRA88]; e a ferramenta que suporta os critérios de Laski e Korel, [KOR87], [LAS83].

Especificamente, queremos ressaltar a proposta de Maldonado, Chaim e Jino [MAL88a], [MAL88b], que estabelece os critérios Potenciais Usos. Estes critérios, semelhantemente aos de Herman [HER76], Laski [LAS83], Ntafos [NTA84] e Rapps [RAP85], baseiam-se na análise de fluxo de dados para a derivação de casos de teste. No entanto, as associações requeridas por estes

critérios são o exercício de caminhos entre a definição e um possível (potencial) uso de uma variável.

A ferramenta POKE-TOOL (Potential Uses Criteria Tool for Program Testing) [CHA91], [MAL89] está apoiada na ferramenta GFC [CAR90c], o que contribui para que seja multilinguagem. Resumidamente, a POKE-TOOL, na versão atual, suporta aplicação dos Critérios Potenciais Usos para o teste de unidades de software implementadas na linguagem C; dado um conjunto de casos de teste "T", a POKE -TOOL realiza a análise de cobertura do conjunto "T" em relação a um critério previamente selecionado. Ainda, os resultados da análise estática provida pela ferrramenta POKE-TOOL podem ser utilizados para apoiar a atividade de geração de casos de testes propriamente dita.

Queremos ainda considerar uma abordagem para o gerenciamento do software dentro do ambiente de manutenção, citada no Capítulo II. Como dissemos, a manutenção é o problema crítico em todo ambiente de produção de software, devido à inconsciência dessa realidade aliada à carência de técnicas, métodos e ferramentas que minimizem o esforço requerido para esta atividade. Proposta para a primeira etapa de solução desse problema é apresentada por Carnassale M., Evangelista,S.R.M., Moura,M.F. e Ternes,S. [CAR90a], através de um conjunto de ferramentas necessárias para a criação de um ambiente, baseado em Sobrinho [SOB84], onde um software poderá ser reestruturado, remodularizado e documentado a partir do código fonte.

A ferramenta GFC integra esse ambiente, que atualmente está em desenvolvimento como parte de um projeto maior chamado Projeto Fábrica de Software [CAR90b].

As ferramentas aqui apresentadas não geram exclusivamente o grafo de programa como produto, mesmo porque, o grafo de programa além de ser um resultado final útil e suficiente para várias aplicações, é também alicerce para muitas outras. Portanto, para viabilizar a automatização da análise da estrutura do código fonte (Seção 2.3), bem como a automatização das Técnicas de Teste Estrutural e de Análise de Fluxo de Dados (Seção 2.2), é necessária a obtenção automática do grafo de programas.

3.3 ABORDAGEM ADOTADA

A abordagem adotada foi, sem dúvida, o aspecto mais estudado deste trabalho. Decidimos, primeiramente, fazer da GFC uma ferramenta portável e multilinguagem. Esta decisão gerou a necessidade de definirmos uma linguagem intermediária, comum às representações de fluxo de controle das linguagens procedimentais, sem perder, todavia, as informações do programa fonte em sua linguagem original.

Para atender um número maior de situações, optamos por contemplar, também, programas não estruturados. A seguir damos

com mais detalhes as razões para essas abordagens, precedidas de uma síntese dos passos necessários para a automatização.

3.3.1 PASSOS NA AUTOMATIZAÇÃO

De forma análoga à compilação de um programa [SET83], o primeiro passo a ser dado para a obtenção automática do grafo de programa é reconhecer os itens léxicos do programa fonte a ser analisado, processo chamado de análise léxica.

Por sua vez, os itens léxicos, reconhecidos e classificados (por exemplo, em palavras reservadas, identificadores, delimitadores, operadores, números), são convenientemente tratados por um analisador sintático.

O analisador sintático agrupa os itens léxicos utilizando-se de uma série de regras de sintaxe, que constituem a gramática da linguagem fonte, e que definem, em última instância, a estrutura sintática do programa fonte.

O reconhecimento dessa estrutura é essencial para o processo de geração do grafo de programa, pois terão que ser detectados, não só os comandos que alteram o fluxo de execução, como também, pontos de entrada e saída, e construções do programa fonte que porventura estejam em desacordo com a gramática.

Uma vez feito este reconhecimento, resta-nos representar as informações obtidas sobre o fluxo de execução do

programa analisado. Nessa representação devem constar informações suficientes para modelarmos o grafo de programa.

3.3.2 MULTILINGUAGEM E PORTABILIDADE

Não é pequena a diversidade de linguagens de programação existentes e em uso, mesmo se considerarmos somente a categoria de linguagens procedimentais. Portanto, generalidade é uma das características desejáveis que ferramentas devem possuir para atender à crescente demanda de ambientes automatizados de desenvolvimento de software.

Procurando atingir, também, esse objetivo, dotamos a GFC de aspectos multilinguagem que lhe conferem a característica de generalidade, uma vez que o objetivo principal da GFC é suportar ferramentas ou aplicações que utilizem seus produtos. Portanto, a decisão de desenvolvê-la para contemplar várias linguagens e ambientes, foi, sem dúvida, um fator multiplicador de sua utilidade.

Outrossim, o requerimento de portabilidade para a GFC levou-nos a torná-la independente de qualquer outro software, sem que, com isso fossem afetadas sua eficiência e eficácia. Essa independência aliada ao fato de ter sido projetada para três ambientes (MS-DOS, UNIX e VMS) permite à GFC atender a um número maior de usuários e aplicações.

3.3.3 POR QUE UMA LINGUAGEM INTERMEDIÁRIA?

A decisão de incluir nos requerimentos da GFC a característica de ferramenta multilinguagem, levou-nos a escolher uma forma de representação comum para os programas codificados nas diversas linguagens abrangidas. Após meticuloso estudo das linguagens ALGOL-60, C, COBOL, FORTRAN, MODULA-2, e PASCAL definimos uma linguagem intermediária, denominada LI.

Esse princípio de representação intermediária foi usado com sucesso quando Wirth criou o P-code para imbutir portabilidade à linguagem PASCAL [WIR71]. Um outro exemplo é a linguagem DIANA, usada como linguagem intermediária para ADA [GOO83].

A LI possui sintaxe e semântica muito simples e é composta por elementos chamados átomos da LI que representam genericamente vários tipos de comandos que alteram, ou não, o fluxo de execução nas linguagens procedimentais.

A programação direta em LI pode ser feita a partir do desenho do software, mas, não em substituição à programação em uma linguagem convencional, pois nenhum código executável será gerado a partir de um programa em LI. No entanto, devido à extrema simplicidade da linguagem, os módulos poderão ser facilmente compostos em LI, como também testados, avaliados e alterados quanto à sua estrutura, mesmo antes da fase de codificação.

O programa escrito diretamente em LI, ou traduzido de outra linguagem, será utilizado para a produção do grafo de programa. A geração da LI como passo intermediário anterior à produção do grafo de programa é uma alternativa importante para eliminar esforço redundante na construção de ferramentas específicas para cada linguagem de programação.

A existência de uma linguagem intermediária propicia também, um alicerce cômodo para o desenvolvimento de outras ferramentas que necessitem utilizar o grafo de programa, ou mesmo, fazer uso apenas de programas em LI.

A gramática da LI, sua descrição detalhada bem como a representação de trechos de programas de diversas linguagens para código em LI são encontradas no Capítulo IV.

3.3.4 ABRANGÊNCIA DE PROGRAMAS NÃO ESTRUTURADOS

Posto que programação estruturada é assunto consolidado há bastante tempo, existe ainda em uso um número incontável de programas, que foram concebidos fora dessa mentalidade e que não podem ser ignorados. Aliás, provavelmente são os que mais necessitam de uma ferramenta que auxilie na análise da estrutura ou na evolução do código.

Diante dessa realidade, programas não estruturados não poderiam ser excluídos das facilidades produzidas pela GFC.

3.3.5 PRESERVAÇÃO DA RELAÇÃO COM O CÓDIGO FONTE

Os comandos da LI, por si só, não explicitam detalhes do código fonte. Se nossa preocupação fosse somente gerar o grafo de programa muitas informações sobre o programa fonte que não afetam o fluxo de controle poderiam ser desprezadas; por exemplo, declarações ou atribuições de variáveis. No entanto, diante do objetivo de apoiar outras ferramentas, tornou-se imperiosa a preservação de todas as informações sobre o programa fonte, pois obviamente algumas ferramentas necessitarão desses dados para aplicações específicas.

Cada comando de um programa em LI, resultado de uma tradução, tem em seus comandos uma correspondência, direta ou indireta, com o trecho do programa fonte de onde foi traduzido. Decidimos, então, para solucionar o problema acima citado, associar a cada elemento da LI o endereço da porção do código fonte correspondente. Dessa maneira, qualquer aplicação que necessite informações adicionais sobre o código fonte, poderá obtê-las a partir do código em LI.

Como é preservado o relacionamento entre programa fonte e programa em LI, bem como exemplos de correspondência direta e indireta de seus endereços, serão discutidos no Capítulo IV.

C A P í T U L O I V

A LINGUAGEM INTERMEDIÁRIA LI

4.1 INTRODUÇÃO

As razões que nos levaram a optar por uma linguagem intermediária, bem como sua importância, foram discutidas na Seção 3.3. Neste capítulo, apresentaremos uma extensão da descrição da LI [CHA91], sua sintaxe, semântica e alguns exemplos de tradução de programas de várias linguagens procedimentais para LI. Por último, teceremos algumas considerações sobre a definição da LI.

4.2 TIPOS DE COMANDOS E ESTRUTURA DOS ÁTOMOS DA LI

A LI, basicamente, tem o propósito de representar o fluxo de execução de um programa; em virtude do que, possui dois tipos de comandos :

- comandos sequenciais, e
- comandos de controle de fluxo.

Os comandos sequenciais da LI representam os comandos das linguagens procedimentais que não alteram o fluxo de execução, como por exemplo, uma declaração de variável ou estrutura de dados, ou uma computação (comandos de atribuição ou chamadas de funções).

Os comandos de controle de fluxo da LI são equivalentes aos comandos das linguagens procedimentais que causam seleção, seleção múltipla, repetição e desvio incondicional.

Os comandos da LI, tanto sequenciais quanto os de controle de fluxo serão descritos detalhadamente nas próximas seções.

Uma das características da LI é que todos os seus átomos são seguidos de números que respectivamente identificam:

- o início do átomo no programa fonte (a quantos bytes da posição inicial do arquivo fonte começa o átomo);

- o comprimento do átomo (de quantos bytes é composto o átomo); e

- o número da linha onde se encontra o átomo.

Antes de apresentarmos a estrutura da LI convém estabelecermos a notação. Utilizaremos a notação de Backus-Naur [SEB89], onde: os terminais serão representados em letra maiúscula; os não-terminais entre parêntesis angulares; e as chaves aparecerão sublinhadas quando forem parte integrante da LI. Os meta-símbolos usados são sete ao todo, a saber: ::= , / , (,) , { , } , $^+$.

Vejamos então, como é a estrutura de um átomo da LI:

⟨atm_li⟩ ::= ⟨atomo⟩⟨inicio⟩⟨comprimento⟩⟨linha⟩.

onde,

⟨atomo⟩::= $DCL / $S / { / } /
$IF / $ELSE / $CASE / $C / $NC / $CC /
$WHILE / $FOR / $REPEAT / $UNTIL /
$GOTO / $BREAK / $CONTINUE / $RETURN /
$ROTC / $ROTD / LABEL

⟨inicio⟩ ::= NUM

⟨comprimento⟩ ::= NUM

⟨linha⟩ ::= NUM.

Os terminais que aparecem nessa descrição representam exatamente a sequência de caracteres indicada pelos próprios terminais; com exceção de $S, $C, $NC, NUM e LABEL. Estes terminais indicam:

- $S a sequência de caracteres $Sn;
- $C a sequência $C(n)n;
- $NC a sequência $NC(n)n;

onde "n" pertence ao conjunto N* (naturais exceto o elemento nulo),

- NUM um elemento do conjunto N*; e
- LABEL uma sequência qualquer de caracteres, que é literalmente um rótulo no programa fonte.

A utilidade dos ponteiros dos átomos é possibilitar o acesso ao código fonte associado ao átomo da LI, o que é necessário em algumas aplicações, como ocorre na ferramenta POKE-TOOL [CHA91], citada na Seção 3.2. Consideremos o seguinte comando em linguagem ALGOL:

boolean a,b,c;

seria traduzido para a LI como:

$DCL 150 14 10.

O comando LI significa uma declaração que começa a 150 bytes do início do arquivo fonte, tem 14 bytes de comprimento e está localizado na décima linha. Os espaços em branco são contados para efeito de endereçamento, se ocorrerem entre itens léxicos que serão traduzidos para apenas um átomo da LI (no exemplo acima o branco entre "n" e "a" é contado). O caracter de controle "return", sempre que ocorrer, também é contado. Notemos que toda uma linha de comando em ALGOL originou um átomo da LI.

A seguir, serão apresentados os diversos comandos da LI e para descrevê-los utilizaremos a mesma notação estabelecida nesta seção.

4.2.1 COMANDOS SEQUENCIAIS

Os comandos de programas fontes que não afetam o fluxo de controle são chamados de comandos sequenciais ou comandos simples. Ao serem traduzidos para comandos em LI, são sintetizados produzindo uma significativa compactação do código gerado.Os comandos sequenciais são os seguintes:

⟨dcl⟩ ::= $DCL ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩ e

⟨s⟩ ::= $S ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩.

Onde, ⟨dcl⟩ denota uma declaração de variável e ⟨s⟩ uma computação; sendo que uma computação pode ser uma atribuição de valor a uma variável (através de uma expressão ou chamada de função), ou somente uma chamada de procedimento.

Por exemplo, o comando em linguagem C:

x = fopen("arquivo.c","r");

seria traduzido para:

$S1 1024 27 23.

Observemos que $S1 acima representa o primeiro comando sequencial do programa em LI, ou seja, o número que segue aos caracteres "$S" indica a ordem em que o comando sequencial aparece no programa.

4.2.2 COMANDOS DE SELEÇÃO

São os seguintes os comandos de seleção:

⟨if⟩ ::= ⟨if_atm⟩ ⟨cond_atm⟩ ⟨statement_1⟩ /

⟨if_atm⟩⟨cond_atm⟩⟨statement_1⟩⟨else_atm⟩ ⟨statement_2⟩

onde,

⟨if_atm⟩ ::= $IF ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩;

⟨cond_atm⟩ ::= $C ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩;

⟨else_atm⟩ ::= $ELSE ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩ e

⟨statement_i⟩ é o não-terminal que denota todos os possíveis comandos da LI, agrupados ou não.

⟨statement_i⟩ será definido formalmente mais adiante. Este comando significa o "if" tradicional das linguagens do estilo ALGOL, ou seja, os "comandos" em ⟨statement_i⟩ serão executados se ⟨cond_atm⟩ for verdadeiro e os "comandos" em ⟨statement_2⟩ serão executados caso contrário. Considere o seguinte trecho de programa em MODULA-2:

```
...
if  j<>0 then k:=k/j; else writeln(`error: division  by
zero´);
...
```

Este comando traduzido para a LI, seria:

```
$IF        1031      2     10
$C(1)1     1034      4     10
$S1        1039     12     10
$ELSE      1052      4     10
$S2        1057     35     10
```

Os números que aparecem em $C(i)i indicam, respectivamente, o número de predicados que possui a condição e a ordem de aparição no programa. Sobre a quantificação dos predicados discutiremos na Seção 4.3.

A solução para a ambiguidade que poderia ser gerada pelo encadeamento de $IF e $ELSE sem delimitadores de bloco "{" e "}" foi tomada inspirada na maioria da linguagens estudadas, onde o ELSE é associado com o mais recente IF sem ELSE, salvo o uso explícito de chaves que podem forçar a associação apropriada.

4.2.3 COMANDO DE SELEÇÃO MÚLTIPLA

Definimos um comando de seleção múltipla, assim representado:

⟨case⟩ ::=

⟨case_atm⟩ ⟨case_cond_atm⟩ [((⟨rotc_atm⟩ / ⟨rotd_atm⟩) (⟨statement⟩))$^{+}$]

onde,

⟨case_atm⟩ ::= $CASE ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩;

⟨case_cond_atm⟩ ::= $CC ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩;

⟨rotc_atm⟩ ::= $ROTC ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩; e

⟨rotd_atm⟩ ::= $ROTD ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩.

O não-terminal ⟨case_atm⟩ representa o átomo que inicia o comando de seleção múltipla; ⟨case_cond_atm⟩ representa a condição do comando; e ⟨rotc_atm⟩ representa os possíveis rótulos para as sequências de comandos indicadas por ⟨statement⟩. O não-terminal ⟨rotd_atm⟩ representa o rótulo para a sequência de comandos a ser executada quando a condição não combina com nenhum rótulo ⟨rotc_atm⟩.

A semântica do comando acima é equivalente ao comando "switch" da linguagem C, isto é, a execução começa no rótulo que ocorreu a combinação e executa os comandos desse rótulo mais os comandos dos rótulos que o seguem, a menos que seja encontrado um comando válido de desvio incondicional. O comando "break", por exemplo, causa a imediata saída do comando de seleção múltipla

para o comando que o segue imediatamente. Os comandos de desvio incondicional serão discutidos na Subseção 4.2.5.

Para exemplificar, considere o seguinte trecho de programa em Ada [GHE87]:

```
case OPERATOR of
  when "." => RESULT := OPERAND1 and OPERAND2;
  when "+" => RESULT := OPERAND1 or OPERAND2;
  when "=" => RESULT := OPERAND1 = OPERAND2;
  when others => ... produce error message ...
end case;
```

A respectiva tradução para a LI é:

```
$CASE      1001     4     5
$CC        1006     8     5
(          1015     2     5
$ROTC      1020    11     6
$S1        1032    32     6
$BREAK        0     0     0
$ROTC      1067    11     7
$S2        1079    31     7
$BREAK        0     0     0
$ROTC      1113    11     8
$S3        1125    30     8
$BREAK        0     0     0
$ROTD      1158    14     9
$S4        1173    29     9
)          1203     9    10
```

Observemos que foi acrescentado o comando "break" no trecho em LI para manter a semântica do "case" da linguagem Ada. Notemos também, que os ponteiros do comando "break" são iguais a 0 para indicar que não há correspondência direta no código fonte para o referido comando LI.

4.2.4 COMANDOS DE REPETIÇÃO

A LI fornece comandos para repetição tanto para um número fixo de iterações quanto para repetições que dependam de uma condição.

No caso de um número fixo de repetições, temos um comando semelhante ao "for" das linguagens do estilo ALGOL. O "for" da LI é definido como:

⟨for⟩ ::= ⟨for_atm⟩ ⟨s1⟩ ⟨cond_for_atm⟩ ⟨s2⟩ ⟨statement⟩

onde,

⟨for_atm⟩ ::= $FOR ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩; e

⟨cond_for_atm⟩ ::= $C ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩.

O não-terminal ⟨for_atm⟩ indica o comando "for" da LI; o não-terminal ⟨s1⟩ representa a iniciação das variáveis de controle do "for", através de um comando sequencial; ⟨cond_for_atm⟩ representa a condição; ⟨s2⟩ representa o comando sequencial que altera as variáveis de controle a cada iteração do "for"; e ⟨statement⟩ representa o corpo do comando. O comando "for" da LI é inspirado no comando equivalente da linguagem C, possuindo a mesma semântica. No trecho de programa em C, o comando:

```
for (i=0; i < nfiles; i++)
        fstat[i] = 0;
```

é traduzido para a LI da seguinte maneira:

```
$FOR      110     3     12
$Si       115     4     12
$C(1)i    120    11     12
$S2       132     3     12
$S3       144    13     13
```

A associação dos comandos do código fonte para a LI não é sempre direta como no exemplo anterior. Considere o trecho de programa em Pascal [GHE87]:

```
type  day =  (sunday,monday,tuesday,wednesday,thursday,
friday,saturday);
var week_day: day;
...
for week_day := monday to friday do
...;
```

traduzido para LI, temos:

```
$DCL       51     70     2
$DCL      123     18     3
...
$FOR      303      3     7
$Si       307     18     7
$C(i)i      0      0     0
$S2       326      9     7
...
```

Observemos que na condição do comando "for" os ponteiros são iguais a 0, novamente indicando que o átomo da LI não possui correspondência direta no arquivo fonte. Deixamos a decisão de como mapear a linguagem fonte para a LI com o usúario configurador da GFC.

Os comandos de iteração cujo número de repetições é dirigido por uma condição, são também equivalentes aos tradicionais "while" e "repeat-until" das linguagens do estilo ALGOL. O "while" da LI é definido como:

⟨while⟩ ::= ⟨while_atm⟩ ⟨cond_while_atm⟩ ⟨statement⟩

onde,

⟨while_atm⟩ ::= $WHILE ⟨inicio⟩⟨comprimento⟩⟨linha⟩;e

⟨cond_while_atm⟩ ::= $C ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩.

⟨while_atm⟩ indica o comando "while"; ⟨cond_while_atm⟩ a condição; e ⟨statement⟩ o corpo do "while". O corpo será executado enquanto a condição em ⟨cond_while_atm⟩ permanecer verdadeira.

O "repeat-until" da LI é definido como:

⟨repeat_until⟩ ::= ⟨repeat_atm⟩ ⟨statement⟩ ⟨until_atm⟩ ⟨cond_until_atm⟩

onde,

⟨repeat_atm⟩ ::= $REPEAT ⟨inicio⟩⟨comprimento⟩ ⟨linha⟩;

⟨until_atm⟩ ::= $UNTIL ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩;e

⟨cond_until_atm⟩ ::= ($C / $NC) ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩.

⟨repeat_atm⟩ indica o início do comando "repeat-until"; ⟨statement⟩ o corpo do comando; ⟨until_atm⟩ indica o fim do comando; e ⟨cond_until_atm⟩ representa a condição de término. A semântica desse comando da LI é igual ao comando equivalente das linguagens "ALGOL-like", ou seja, o corpo da iteração é executado pelo menos uma vez e o teste da condição é realizado depois da execução do corpo. Notemos que a condição de término é composta por $C ou $NC. Isto ocorre porque, na maioria das linguagens "ALGOL-like", o corpo do "repeat-until" é executado até que uma

dada condição seja satisfeita; porém, em algumas linguagens como C, o comando "repeat-until" equivalente funciona de maneira que o corpo é executado enquanto a condição permanecer verdadeira. Devido a esse fato, a condição desse comando pode ser $C se for o primeiro caso (e aí estaria indicando um "repeat-until" tradicional), ou $NC se for o segundo caso (e nesta situação teríamos a repetição do laço até ser negada a condição).

Considere o seguinte trecho de programa em C, extraído da página 65 da referência [KER87]:

```
...
do { /* gera os digitos em ordem inversa */
      s[i++] = n % 10 +'0'; /* obtem proximo digito */
   } while ((n /= 10)>0); /* remove o digito */
...
```

Este trecho seria assim traduzido para LI:

```
...
$REPEAT     1000     2    12
{           1003     1    12
$S1         1050    21    13
}           1102     1    14
$UNTIL      1104     5    15
$NC(1)1     1110    13    15
...
```

4.2.5 COMANDOS DE DESVIO INCONDICIONAL

Os comandos de desvios incondicional provocam a mundança do fluxo de execução em um programa. A LI possui um comando de transferência incondicional irrestrito do tipo "goto" e comandos de transferência incondicional controlada; estes últimos têm sua utilização limitada a algumas situações e seu efeito é bem previsível.

O comando "goto" da LI é definido como:

⟨goto⟩ ::= ⟨goto_atm⟩ ⟨label_atm⟩

onde,

⟨goto_atm⟩ ::= $GOTO ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩; e

⟨label_atm⟩ ::= LABEL ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩.

⟨goto_atm⟩ representa o comando "goto" da LI; e ⟨label_atm⟩ representa o rótulo para onde deve ser dirigido o fluxo de execução quando encontrado o comando "goto".

Os comandos de desvio incondicional controlados da LI são:

⟨break⟩ ::= $BREAK ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩;

⟨continue⟩ ::= $CONTINUE ⟨inicio⟩⟨comprimento⟩⟨linha⟩;e

⟨return⟩ ::= $RETURN ⟨inicio⟩ ⟨comprimento⟩ ⟨linha⟩.

Esses comandos de transferência incondicional foram inspirados nos seus homônimos da linguagem C, e por isso, possuem efeitos idênticos. O "break" causa o fim do comando de iteração ("for", "while" ou "repeat-until") mais próximo que o engloba.

Ainda, dentro de um comando "case" da LI, o "break" causa o desvio para o primeiro comando fora do "case". O comando "continue" provoca o desvio para a próxima iteração do laço que o engloba. No caso dos comandos "repeat-until" e "while", ao ser encontrado o "continue", o fluxo de execução é desviado para o teste da condição da iteração; no caso do comando "for", o fluxo de execução é desviado para o comando que altera as variáveis de controle. O comando "return" causa o fim do procedimento que está sendo executado e o retorno para a unidade que o chamou.

4.2.6 COMANDOS COMPOSTOS E PROGRAMAS

Até aqui foram descritos os comandos individuais da LI, entretanto, para concluir sua definição, falta ainda descrever como se agrupam comandos na LI e como esses comandos são organizados em um programa.

O não-terminal ⟨statement⟩ representa um único comando da LI ou um agrupamento deles entre chaves, definido como:

```
⟨statement⟩ ::= { ( ⟨stament⟩) } /
                ⟨dcl⟩ /
                ⟨s⟩ /
                ⟨if⟩/
                ⟨case⟩/
                ⟨for⟩ /
                ⟨while⟩ /
                ⟨repeat_until⟩ /
                LABEL ⟨statement⟩ /
                ⟨goto⟩ /
                ⟨break⟩ /
                ⟨continue⟩ /
                ⟨return⟩.
```

Os programas em LI são definidos da seguinte maneira:

⟨program⟩ ::= { (⟨dcl⟩ / ⟨s⟩) } ⟨statement⟩.

Para exemplificar, mostraremos a seguir o resultado da tradução para LI de dois programas, com diferentes opções de saída. Primeiramente um programa em linguagem C que se encontra na página 61 da referência [KER87]:

```
main() /* contar digitos, espaco branco, outros */
{
int c,i,nbranco,noutro,ndigito[10];
nbranco = noutro = 0;
for (i=0; i<10; i++)
      ndigito[i] = 0;
while ((c = getchar()) != EOF)
      switch (c) {
      case '0':
      case '1':
      case '2':
      case '3':
      case '4':
      case '5':
      case '6':
      case '7':
      case '8':
      case '9':
                  ndigito[c-'0']++;
                  break;
      case ' ':
      case '\n':
      case '\t':
                  nbranco++;
                  break;
      default:
                  noutro++;
                  break;
      }
  printf("digitos =");
  for (i=0; i<10; i++)
      printf("%d",ndigito[i]);
  printf("\nespaco branco = %d, outros = %d\n",
  nbranco, noutro);
}
```

Arquivo obtido da tradução do exemplo acima:

```
$DCL                    1       6       1
(                      52       1       2
$DCL                   54      35       3
$S1                    90      21       4
$FOR                  112       3       5
$S2                   117       4       5
$C(01)1               122       5       5
$S3                   128       3       5
$S4                   139      15       6
$WHILE                155       5       7
$C(01)2               161      24       7
$CASE                 192       6       8
$CC                   199       3       8
(                     203       1       8
$ROTC                 211       9       9
$ROTC                 227       9      10
$ROTC                 243       9      11
$ROTC                 259       9      12
$ROTC                 275       9      13
$ROTC                 291       9      14
$ROTC                 307       9      15
$ROTC                 323       9      16
$ROTC                 339       9      17
$ROTC                 355       9      18
$S5                   381      17      19
$BREAK                415       6      20
$ROTC                 428       9      21
$ROTC                 444      10      22
$ROTC                 461      10      23
$S6                   488      10      24
$BREAK                515       6      25
$ROTD                 528       8      26
$S7                   553       9      27
$BREAK                579       6      28
)                     592       1      29
$S8                   596      20      30
$FOR                  619       3      31
$S9                   624       4      31
$C(01)3               629       5      31
$S10                  635       3      31
$S11                  646      24      32
$S12                  673      65      33
)                     740       1      35
```

A seguir apresentamos a tradução para a LI de um programa em COBOL (somente a Procedure Division), que se encontra na página 397 da referência [STE78]:

```
PROCEDURE DIVISION.
   OPEN INPUT MEDICAL-FILE, OUTPUT DIAGNOSIS-REPORT.
   MOVE SPACES TO HEADER.
   MOVE "DIAGNOSIS REPORT" TO LITERAL1.
   WRITE HEADER.
   MOVE SPACES TO DETAIL-LINE.
BEGIN.
   READ MEDICAL-FILE AT END GO TO EOJ.
   IF TEMPERATURE = 1 AND LUNG-INFECTION = 1
      MOVE "PNEUMONIA" TO DIAGNOSIS
      GO TO WRITE-RTN.
   IF (LUNG-INFECTION = 1 AND SNIFFLES = 1) OR
      (LUNG-INFECTION = 1 AND SORE-THROAT = 1) OR
      (TEMPERATURE =1 ) OR
      (SNIFFLES = 1 AND  SORE-THROAT =1)
      MOVE "COLD" TO DIAGNOSIS
      GO TO WRITE-RTN.
   MOVE "PHONY" TO DIAGNOSIS.
WRITE-RTN.
   MOVE PATIENT-NAME TO NAME.
   WRITE DETAIL-LINE.
   GO TO BEGIN.
EOJ.
   CLOSE MEDICAL-FILE, DIAGNOSIS-REPORT.
   STOP RUN.
```

Abaixo mostramos o resultado obtido com a execução do TRASIN utilizando a opção de saída (eleita pelo usuário) que suprime as colunas de endereçamento ao código fonte "inicio" e "comprimento" :

```
      $S1          2          )          11
      $S2          3        $IF          12
      $S3          4    $C(07)3          12
      $S4          5          (          15
      $S5          6        $S8          15
   BEGIN           7      $GOTO          16
      $S6          8  WRITE-RTN          16
      $IF          8          )          16
  $C(01)1          8        $S9          17
    $GOTO          8  WRITE-RTN          18
      EOJ          8       $S10          19
      $IF          9       $S11          20
  $C(02)2          9      $GOTO          21
        (         10     BEGIN           21
     $S7          10       EOJ           22
    $GOTO         11      $S12           23
WRITE-RTN         11    RETURN           24
```

4.3 CONSIDERAÇÕES SOBRE A DEFINIÇÃO DA LI

Reservamos esta seção para algumas considerações sobre a definição da LI. Discutiremos sobre a escolha do conjunto de comandos e palavras reservadas que compõem a LI, sobre a quantificação de predicados nas condições e algumas soluções adotadas.

Primeiramente, precisávamos escolher, para compor a LI, um conjunto de comandos que representasse todas as possibilidades de fluxo de controle das linguagens procedimentais, o que poderia ter sido feito usando bem poucos tipos de comandos, através dos quais representaríamos os demais. No entanto, economizar comandos só aumentaria a complexidade do trabalho de tradução e não apresentaria ganho significativo algum. Com este raciocínio, escolhemos para a LI um conjunto de comandos, definido nas seções anteriores, que minimiza a tarefa do tradutor TRA*LI*, onde, na maioria dos casos, a tradução é quase que direta, como podemos observar ao considerarmos os comandos e estruturas das linguagens ALGOL 60 [RUT67], [SEG81]; C [KER87]; COBOL [BAS81], [STE78]; FORTRAN 77 [DAV83], [HEH87]; MODULA-2 [THA85]; e PASCAL [JEN84], sintetizadas no quadro a seguir:

Tabela 4.1: Comparação entre Comandos de Algumas Linguagens e a LI

TIPOS DE COMANDO	LI	ALGOL-60	C	COBOL-ANSI	FORTRAN	MODULA-2	PASCAL
SELEÇÃO	IF	IF-THEN	IF	IF,END, INVALID,SIZE ERROR,WHEN	IF, IF-THEN	IF-THEN	IF-THEN
	IF-ELSE	IF-THEN-ELSE	IF-ELSE	IF-ELSE, IF-OTHERWISE	IF-THEN-ELSE, ELSEIF,	IF-THEN-ELSE	IF-THEN-ELSE
SELEÇÃO MULTIPLA	CASE	CASE	SWITCH	GO TO DEPENDING	GO TO (computado) IF (aritmético)	CASE	CASE
REPETIÇÃO	FOR	FOR	FOR	PERFORM-VARYING, EXAMINE, SEARCH	DO	FOR	FOR
	WHILE	WHILE-DO	WHILE	PERFORM-UNTIL PERFORM-TIMES	--	WHILE-DO	WHILE-DO
	REPEAT-UNTIL	DO-UNTIL	DO-WHILE	--	--	REPEAT-UNTIL,LOOP	REPEAT-UNTIL
DESVIOS INCONDICIONAIS	GOTO	GOTO	GOTO	GO TO	GO TO	--	GOTO
	BREAK	--	BREAK	--	--	EXIT	--
	CONTINUE	--	CONTINUE	--	CONTINUE	--	--
	RETURN	--	RETURN	STOP RUN	RETURN	--	--

No caso particular da linguagem FORTRAN, cuja gramática é dependente de contexto, restrições adicionais deverão ser impostas a fim de utilizarmos a GFC tal como foi concebida.

Como já dissemos, a sintaxe e a semântica da LI são muito simples. Assim sendo, a quantidade de comandos escolhidos não interfere na complexidade da análise da LI e consequente geração do grafo de programa, pelo módulo GERMA.

Uma dificuldade com a qual nos deparamos foi a seleção do conjunto de palavras reservadas da LI. Por se tratar de uma ferramenta multilinguagem, palavras reservadas em uma determinada linguagem são simples identificadores em outras, e vice-versa. De fato, identificadores (ou até mesmo números em algumas linguagens) podem ser usados como rótulos. Poderia ocorrer então, que um rótulo traduzido para a LI de uma determinada linguagem coincidisse com uma palavra reservada da LI. Por essa razão e para simplificarmos a análise da LI, precedemos cada comando da LI de um símbolo "$" e mantivemos inalterado na tradução o rótulo da linguagem fonte.

Quanto à definição da gramática da LI, posto que simples, tivemos que tomar certo cuidado para manter a representatividade das estruturas das linguagens procedimentais, como por exemplo, neste trecho de programa em ALGOL, onde vemos um rótulo aparecendo no meio de uma estrutura:

```
boolean a;
if a then rot1: a:= false;
```

Um outro ponto a considerar, é a quantificação de predicados em uma condição. Conforme afirma Myers [MYE79], a quantidade de predicados de uma condição altera a complexidade do comando que a contém. Como uma das funções da GFC é suportar aplicações sobre métricas de software, fornecemos essa informação diretamente do programa em LI.

C A P í T U L O V

A FERRAMENTA GFC

5.1 INTRODUÇÃO

Depois de termos apresentado a abordagem adotada na solução proposta e a descrição detalhada da LI, resta-nos descrever com mais detalhes a arquitetura da GFC e os aspectos mais importantes de sua implementação. Apresentaremos cada módulo separadamente, suas principais características, importância de suas funções, e as razões para a modularização. Também veremos como configurar a GFC para uma nova linguagem, bem como as estruturas de dados utilizadas.

5.2 DESCRIÇÃO FUNCIONAL DA GFC

Nesta seção, para obtermos uma visão geral de todos os procedimentos da GFC, apresentamos, o DFD-Gane [GAN79] geral e o primeiro nível de detalhamento, Figuras 5.1 e 5.2 respectivamente.

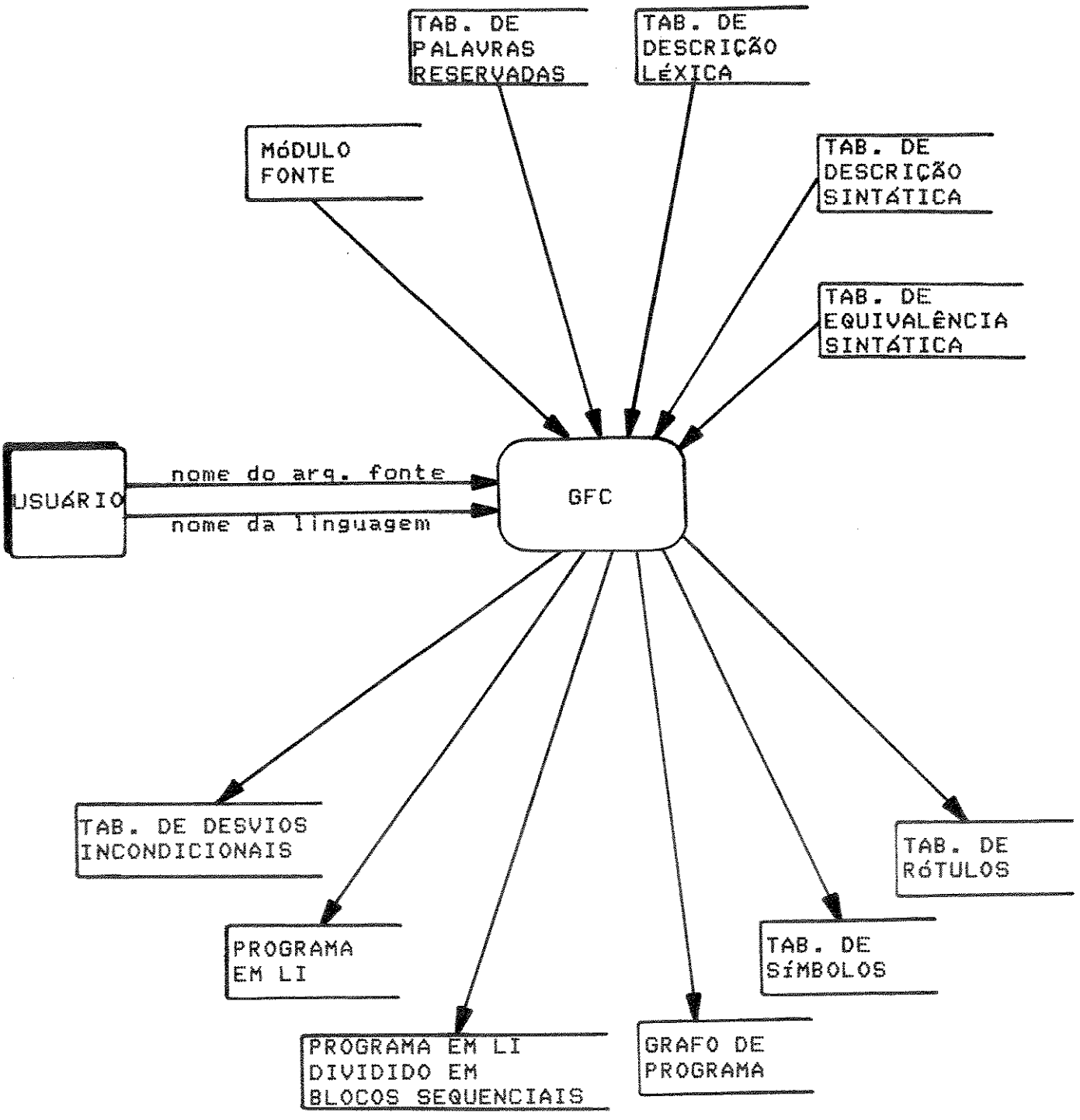


Figura 5.1: DFD Geral da GFC

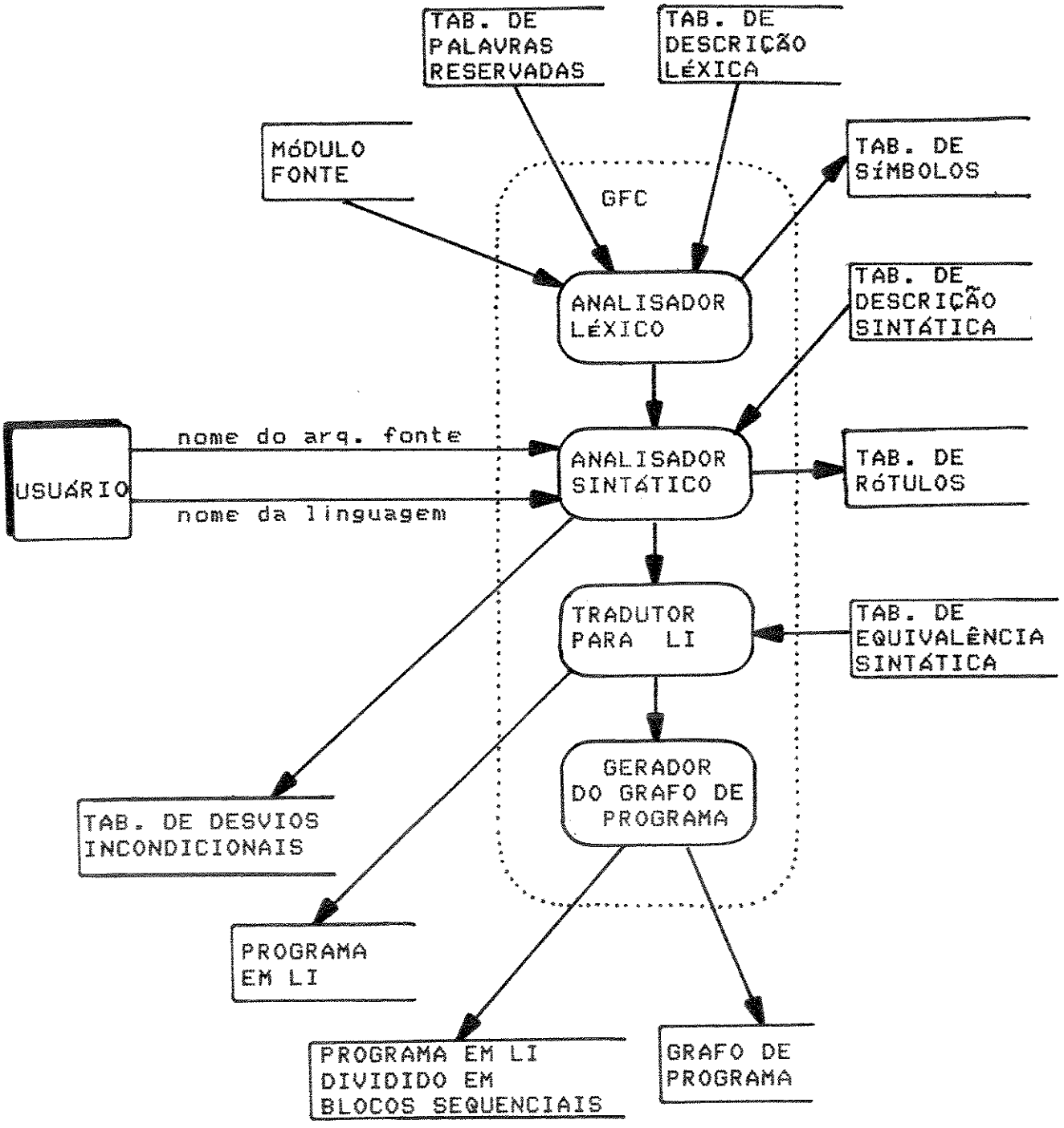


Figura 5.2: DFD Primeiro Nível de Detalhamento da GFC

Através da Figura 5.1 podemos observar quais são as entradas, tabelas de consulta e produtos da GFC. Nas próximas seções trataremos como essas entradas se transformarão em produtos, bem como o significado e importância de cada uma das tabelas. Entretanto, apresentamos aqui, as principais restrições sobre a entrada, que na verdade, pouco limitam o uso da ferramenta, porém, fizeram-se necessárias para racionalizar a implementação:

- o módulo ou programa fonte a ser analisado deve estar isento de erros de sintaxe;

- programas que se utilizam de certas extensões de linguagem (tipo definições em C) que serão substituidas por um simples macroprocessador, devem estar pré-processados;

- programas que contêm comandos que alteram sua estrutura lógica em tempo de execução podem ser analisados, porém não apresentarão todos os resultados reais, já que tais comandos não possuem equivalência semântica na LI, podendo ser tratados como comandos sequenciais;

Como pudemos perceber através da Figura 5.2, temos claramente definidos quatro procedimentos: Um Analisador Léxico, um Analisador Sintático, um Tradutor para LI e um Modelador do Grafo de Programa, cujas funções principais descrevemos a seguir:

- Analisador Léxico - Em conformidade com o citado na Seção 3.3, o primeiro passo a ser dado para a obtenção automática do grafo de programa é fazer a análise léxica do programa fonte

codificado em uma determinada linguagem procedimental.

A função deste Analisador, em particular, é simplificada, pois os requerimentos de nossa abordagem não exigem uma análise léxica tradicional. Estamos especificamente interessados em obter palavras reservadas, identificadores, delimitadores e, dependendo da linguagem, alguns símbolos especiais. São eliminados nesta fase os comentários do programa, os espaços em branco e os caracteres de controle. Todas essas particularidades são ditadas pela tabela da descrição léxica da linguagem do programa fonte, consultadas pelo Analisador.

Uma tabela de palavras reservadas da linguagem do programa fonte é também fornecida para o Analisador verificar se uma cadeia de caracteres será classificada como identificador ou como palavra reservada. Obtemos como resultado deste procedimento uma tabela de símbolos e os itens léxicos do programa fonte que nos interessam, devidamente classificados.

- Analisador Sintático - O Analisador Sintático agrupa os itens léxicos, fornecidos pelo Analisador Léxico utilizando-se de tabelas descritivas das regras de sintaxe da linguagem fonte, reconhecendo, através dessas tabelas, a estrutura do programa fonte. Por vezes, são necessárias algumas ações semânticas para complementar a atuação das tabelas. Essas ações podem ser entendidas como procedimentos que complementam as informações contidas nas tabelas de descrição léxica e sintática. Obviamente as tabelas descritivas da sintaxe são específicas para cada

linguagem, e em nosso caso, bem mais simplificadas, pois não estamos interessados em gerar código executável e sim, selecionar os comandos que afetam o fluxo de execução do programa em análise. Como resultado deste procedimento obtemos uma tabela de rótulos, uma de desvios incondicionais e a estrutura do programa fonte reconhecida.

- Tradutor para a LI - Simultaneamente ao reconhecimento da estrutura do programa fonte sua tradução é feita para a linguagem intermediária LI (descrita no Capítulo IV).

Esta tradução é baseada em informações que se encontram na tabela de equivalência sintática. Após a tradução, como resultado deste procedimento, temos um programa em LI cujo grafo de programa é equivalente ao do programa fonte.

- Modelador do Grafo de Programa - Uma vez obtido um programa em LI que representa o fluxo do programa fonte original, resta-nos modelar as informações obtidas para gerar o grafo de programa. Como dissemos, o grafo de programa é simplesmente uma notação para representar o fluxo de controle lógico de uma unidade de programa. Dessa forma, o que mais nos interessa nesta fase é representar os blocos de execução sequencial e as respectivas ligações entre eles. O resultado deste procedimento é o programa LI dividido em blocos sequenciais e uma matriz de adjacência que representa o grafo de programa.

5.3 ARQUITETURA DA GFC E ASPECTOS IMPORTANTES DA IMPLEMENTAÇÃO

Os quatro procedimentos descritos na Seção 5.2 - Analisador Léxico, Analisador Sintático, Tradutor para a LI e Modelador do Grafo de Programa - foram implementados através de três programas - TRASIN, IDENBLOC e GERMA - conforme podemos observar na Figura 5.3.

O programa TRASIN é o tradutor genérico que executa as funções de analisador léxico e sintático, e tradutor para a LI; o programa IDENBLOC é o identificador e numerador de blocos sequenciais que prepara o programa em LI para a fase seguinte; e o programa GERMA é o gerador de matriz de adjacência que representa o grafo de programa. Nas próximas seções consideraremos, cada um desses programas.

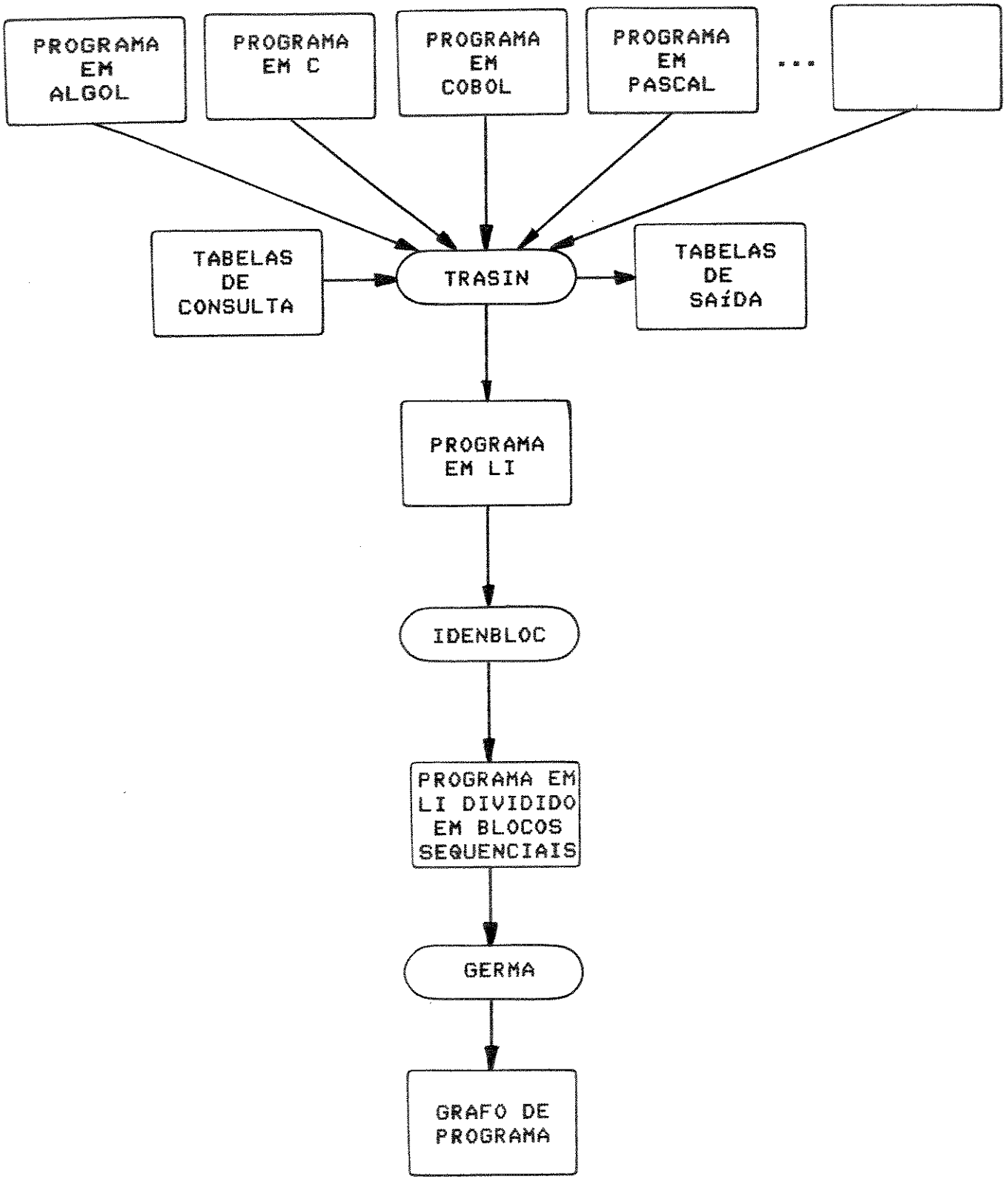


Figura 5.3: Arquitetura da Ferramenta GFC

5.3.1 O PROGRAMA TRASIN

O TRASIN consiste de três módulos: dois analisadores, um léxico (ANALEX) e outro sintático (ANASIN), e um tradutor para a LI (TRALI). Respectivamente, fazem a análise léxica, a análise sintática do programa fonte e a sua tradução para um programa em LI. Agrupando estes três módulos em um programa, obtemos a tradução para a LI em apenas um passo, conforme observamos na Figura 5.4.

O TRASIN é um programa gerenciador universal, orientado por tabelas descritivas da linguagem especificada pelo usuário. Tal abordagem confere a ferramenta a generalidade desejada.

As tabelas orientadoras do TRASIN são específicas, carregadas dinâmica e individualmente por uma função do sistema para cada linguagem. Por exemplo, se o sistema tiver como entrada um programa escrito em COBOL, serão carregadas as tabelas de transições léxicas, palavras reservadas, transições sintáticas e equivalência sintática referentes a linguagem COBOL.

Em alguns casos, para analisar devidamente uma linguagem, necessitamos recorrer a ações "semânticas". Uma ação "semântica" é um comando codificado em linguagem C, que pode fazer parte de uma biblioteca de ações do sistema, ou ser carregada juntamente com as tabelas descritivas da linguagem fonte. Um conjunto de ações "semânticas" para a linguagem C e COBOL é fornecido no apêndice, a título de exemplo.

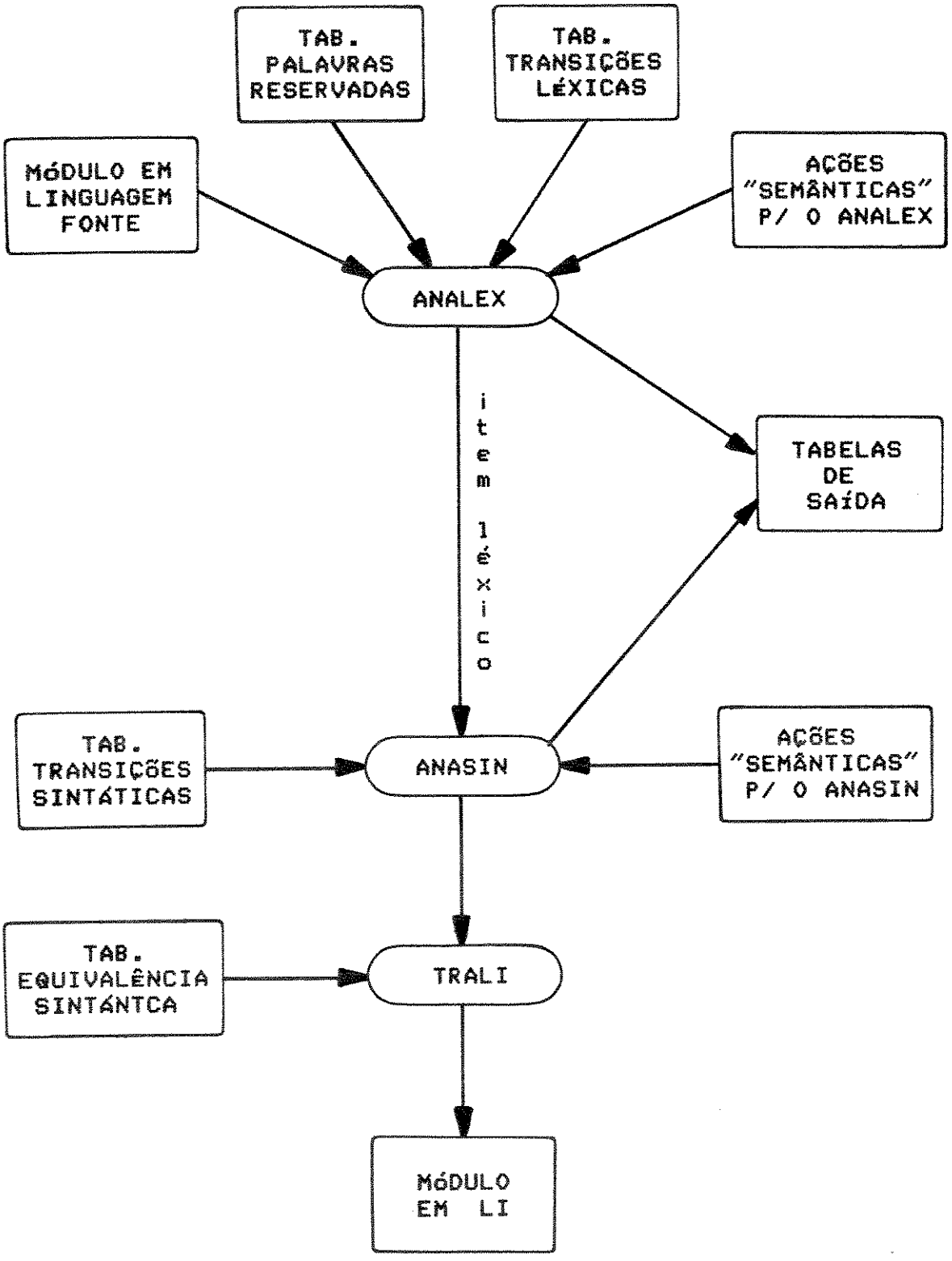


Figura 5.4: Arquitetura do Programa TRASIN

Apesar desta solução não ser a que proporciona maior velocidade de processamento, o programa gerenciador acrescido das tabelas requerem relativamente pouco espaço de memória. O maior ganho, no entanto, reside no fato de que, para a incorporação de uma nova linguagem é necessária unicamente a inserção das tabelas descritivas correspondentes - o programa gerenciador é universal.

Como o sistema é modular, podemos obter suas saídas intermediárias (programa traduzido para LI, tabela de símbolos, tabela de rótulos e tabela de desvios incondicionais) com a execução apenas do TRASIN. O produto principal é um programa em LI; usando a notação estabelecida no Capítulo IV, descrevemos a seguir as saídas secundárias:

- Tabela de Símbolos é assim representada:

⟨ident_fte⟩ ::= ⟨ident⟩⟨inicio⟩⟨comprimento⟩⟨linha⟩

onde,

⟨ident⟩ são os identificadores da liguagem fonte e opcionalmente nomes de função de biblioteca.

- Tabela de Rótulos é assim descrita:

⟨rot_atm⟩ ::= LABEL⟨inicio⟩⟨comprimento⟩⟨linha⟩

onde,

LABEL é o ponto de entrada (rótulo) para onde deve ser dirigido o fluxo de execução quando encontrado um comando "goto".

- Tabela de Desvios Incondicionais:

<got_atm> ::= LABEL<inicio><comprimento><linha>

onde,

LABEL é o rótulo que aparece imediatamente após um comando "goto".

Notemos que símbolos, desvios e rótulos contêm os respectivos endereços no programa fonte, de maneira análoga aos átomos da LI.

No programa TRASIN não há restrição de tamanho do módulo fonte.

5.3.2 O PROGRAMA IDENBLOC

O programa IDENBLOC é composto de um módulo muito simples, cuja característica principal é reduzir a complexidade da geração da matriz de adjacência, bem como conferir mais uma possibilidade de utilização da GFC, uma vez que um dos propósitos básicos deste sistema é servir de suporte para outras aplicações.

Como dissemos, o IDENBLOC divide em blocos sequenciais um programa em LI, e associando a cada bloco um nó, podemos utilizar os conceitos da teoria dos grafos [AND80], [SWA81], [SZW86], [VEL79].

Poderá ocorrer que um bloco contenha muitos átomos da linguagem ou apenas um. De fato, a cada átomo LI do programa

traduzido é associado um número (sequencial a partir de 1) que indica o número do bloco sequencial ao qual o átomo pertence.

A saída do IDENBLOC é, portanto, o mesmo programa em LI acrescido do número do nó em cada átomo. Utilizando a mesma notação estabelecida no Capítulo IV, temos:

⟨atm_nli⟩ ::= ⟨atomo⟩⟨no⟩⟨inicio⟩⟨comprimento⟩⟨linha⟩

onde,

⟨no⟩ ::= NUM.

Podemos obter um programa em LI acrescido do número de nós em cada átomo, com a execução dos programas TRASIN seguido do IDENBLOC. Se o programa foi codificado diretamente em LI, basta executar o IDENBLOC; em ambos os casos, não há restrição de tamanho do programa de entrada.

5.3.3 O PROGRAMA GERMA

A saída do IDENBLOC, bem como as duas tabelas geradas pelo TRASIN (tabela de rótulos e tabela de desvios incondicionais) serão as entradas do terceiro e último programa deste sistema, e é composto por dois módulos, o M-1 e o M-2.

Utilizamos nesta fase o YACC - Yet Another Compiler-Compiler, que é essencialmente um analisador gramatical do tipo LALR(1), ou seja, que faz uma análise da esquerda para a direita, resultando a derivação direita e consultando um próximo item

léxico da entrada (LookAhead, Left-to-right parsing producing Rightmost derivation). Resumidamente, podemos dizer que o YACC é um compilador de compiladores [JOH75].

O M-1 contém uma rotina de baixo nível para reconhecer as entradas básicas (itens léxicos da LI).

O M-2 contém a especificação do processamento de entrada o qual inclui regras descrevendo a estrutura dessa entrada, ou seja, a gramática LI (ver Capítulo IV), e um código (em linguagem C) a ser executado quando essas regras forem reconhecidas.

A cada regra gramatical da LI, associamos ações que serão executadas toda vez que determinada regra for reconhecida pelo processo de entrada. Uma ação é um comando qualquer em C e, como tal, pode executar entradas e saídas, chamar subprogramas, etc. Dessa forma, à medida que o YACC faz a análise do programa em LI, isto é, analisa se as sentenças da LI satisfazem o conjunto de regras gramaticais definidas, o grafo de programa é obtido, pois as ações estão construídas exatamente para sua obtenção, cuja representação lógica é dada pela matriz de adjacência.

A representação do grafo de programa através da matriz de adjacência ocorre da seguinte forma: se há caminho possível entre um bloco sequencial "i" e um bloco sequencial "j", então a posição (i,j) da matriz de adjacência conterá 1, caso contrário conterá 0.

O espaço em memória para essa matriz é alocado de acordo com o número de nós gerado pelo IDENBLOC, e a restrição é o tamanho de memória disponível na máquina utilizada.

Devido à simplicidade da gramática da LI e o problema da generalidade já ter sido resolvido pelo programa TRASIN, escolhemos utilizar o YACC pelas facilidades que ele proporciona a nível de codificação [JOH75].

Nosso cuidado maior nesta fase foi manter a estrutura lógica original do programa fonte de entrada, e manter a associação de cada nó com a porção do código fonte correspondente.

Para este programa, o módulo do programa fonte a ser analisado deve corresponder a um subprograma de uma linguagem, por exemplo, uma função em C ou um parágrafo em COBOL.

5.4. ESTRUTURAS DE DADOS UTILIZADAS

Estruturas são especialmente apropriadas para gerenciar arranjos de variáveis inter-relacionadas, porque em muitas ocasiões essas variáveis podem ser tratadas como um todo ao invés de entidades separadas [KER87].

Uma estrutura idêntica à estrutura dos átomos da LI (Capítulo IV), foi utilizada na representação dos dados para implementar o programa TRASIN. A estrutura é etiquetada como

"atm_li" e composta por uma cadeia de caracteres chamada "atomo" e por três outras variáveis ordinárias do tipo inteiro, respectivamente chamadas de "incio", "comprimento" e "linha". As informações carregadas por essas variáveis podem ser vistas na Seção 4.2.

Para atender a uma particularidade desta implementação, onde usamos alocação dinâmica de memória, um campo nomeado "ap_atm" é acrescentado na estrutura com a função de apontar para o próximo elemento da estrutura; e uma pilha sintática associada ao módulo TRALI, utilizada quando a tradução não é direta.

No programa TRASIN são também referenciadas três tabelas que descrevem uma linguagem fonte e uma tabela de equivalência sintática entre a linguagem intermediária LI e a linguagem fonte, as quais orientam o gerenciador universal na sua tarefa. A seguir, apresentamos cada uma delas e seus respectivos conteúdos:

- Tabela de Palavras Reservadas - contém todos os identificadores reservados da linguagem fonte e sua classificação. Cada registro está dividido em dois campos, o primeiro contém uma palavra reservada ou, a critério do usuário configurador, um nome de função da biblioteca da linguagem; o segundo contém um número inteiro que significa a classe da palavra reservada (ver seção 5.5).

- Tabela de Descrição Léxica - contém as regras de formação léxica da linguagem fonte. Cada registro divide-se em

cinco campos, onde o primeiro contém uma lista de cartacteres da linguagem convenientemente selecionados; o segundo e o terceiro campos indicam, respectivamente, o estado atual e o próximo estado na análise léxica; o quarto campo, se diferente de zero, informa a ação "semântica" que deverá ser acionada; o último campo é usado para dinamizar a pesquisa na tabela, direcionando o analisador para a próxima linha na sequência da análise.

- Tabela de Descrição Sintática - contém as regras gramaticais da linguagem fonte. A divisão de cada registro é idêntica a da Tabela de Descrição Léxica. No entanto, o primeiro campo contém itens léxicos ao invés de caracteres; e o segundo e terceiro campos indicam, respectivamente, o estado atual e o próximo estado na análise sintática.

- Tabela de Equivalência Sintática - relaciona a estrutura da LI com a estrutura da linguagem fonte. O número de campos de cada registro é variável. O primeiro campo indica o número de equivalências que aquele registro contém, ou seja, por quantos conjuntos de átomos da LI o comando da linguagem fonte poderá ser traduzido; o segundo campo contém o item léxico da linguagem fonte ou apenas a sua classe; o terceiro campo campo contém um indicador de equivalência, usado para impedir ambiguidade quando dois ou mais comandos têm identidade léxica porém diferem sintática ou semanticamente; o quarto campo contém a quantidade de átomos da LI que serão utilizados na tradução do comando que aparece no campo dois para a equivalência específica

identificada no campo três; os próximos campos conterão átomos da LI, cada campo um átomo, tantos quantos a quantificação do campo quatro.

No programa IDENBLOC essa estrutura é redefinida com a etiqueta de "atm_nli", onde toda a estrutura anterior é mantida, e acrescentada uma nova variável ordinária do tipo inteiro, chamada "no", que obviamente, conterá o número do nó atribuído ao átomo da LI.

No programa GERMA, uma nova necessidade de estrutura surge para representar o grafo de programa. Optamos pela forma mais usual - uma matriz quadrada, booleana e esparsa chamada matriz de adjacência [AUT76], onde as linhas representam os nós orígem e as colunas representam os nós destino.

Sem dúvida, a interface com outras ferramentas, especialmente com a POKE-TOOL [CHA91], influenciou nas soluções adotadas sobre as estruturas de dados da GFC.

5.5. CONFIGURANDO A GFC PARA UMA NOVA LINGUAGEM

Para o usuário poder utilizar-se do potencial da GFC, sem dúvida, ele precisa saber como configurar a GFC para uma determinada linguagem procedimental de sua preferência.

Basicamente, o usuário configurador terá que construir as quatro tabelas de acordo com a descrição apresentada na seção

anterior e, eventualmente, adicionar alguma ação "semântica" às já existentes na GFC. A sequência é a seguinte:

- Passo 1 - identificar todas as palavras reservadas da nova linguagem e classificá-las em ordem alfabética crescente. Essa será a ordem da tabela de palavras reservadas. Nomes de função de biblioteca poderão ser incluídos a critério do usuário; se não incluídos, serão tratados como identificadores;

- Passo 2 - identificar as palavras reservadas que representam comandos que alteram o fluxo de controle (ver Seção 4.2), e a cada uma atribuir um distinto número inteiro igual ou superior a 9 e diferente de 99.

- Passo 3 - atribuir o número 99 às demais palavras reservadas (que não representam comandos que alteram o fluxo de controle). Os números de 0 a 9 são reservados para o sistema. Neste ponto temos construída a tabela de palavras reservadas;

- Passo 4 - descrever o léxico da nova linguagem através da construção de uma tabela de transições léxicas, de acordo com as estruturas de dados apresentada na Seção 5.4, lembrando que comentários, espaços em branco e caracteres de controle do programa fonte são descartados; se existirem caracteres na linguagem fonte considerados úteis à análise sintática, por exemplo "(" , ")" e ";" em C, e "." em COBOL, deverão ser convenientemente tratados nesta tabela;

- Passo 5 - descrever a sintaxe da nova linguagem através da construção de uma tabela de transições sintáticas,

consoante às estruturas de dados apresentada na Seção 5.4, lembrando que nos interessam somente comandos relacionados com o fluxo de controle de programas;

Observação: a técnica usada para a construção das tabelas citadas no Passo 4 e no Passo 5 é a representação por meio de uma máquina abstrata chamada Autômato de Estados Finitos [SET83].

- Passo 6 - Finalmente, o usuário deve construir a tabela de equivalência sintática entre a nova linguagem e a LI, segundo a estrutura de dados apresentada na seção anterior. Na construção dessa tabela convém lembrar que um comando qualquer da linguagem fonte pode ser representado por um ou mais comandos em LI; o contrário não é verdade.

Para criar uma nova tabela de equivalência aconselhamos ao usuário configurador estudar com atenção o Capítulo IV, bem como a estrutura da nova linguagem. O grafo de programa de cada comando da nova linguagem deve ser conhecido e o usuário deve preocupar-se em manter a equivalência entre o grafo de programa do comando em LI e o grafo de programa do(s) comando(s) correspondente(s) da nova linguagem.

No apêndice, apresentamos todas as tabelas aqui tratadas, para a configuração das linguagens C e COBOL na GFC.

C A P í T U L O V I

GERAÇÃO DO GRAFO DE PROGRAMA

6.1 INTRODUÇÃO

Exemplificamos a definição da LI, no Capítulo IV, utilizando código fonte das linguagens ADA, ALGOL, C, COBOL e PASCAL e suas respectivas traduções para LI. Basicamente, aproveitaremos esses exemplos para apresentar, neste Capítulo, a matriz de adjacência e o desenho do grafo de programa obtidos a partir do código em LI.

Por ser desnecessário para o propósito deste capítulo, omitiremos do código em LI os endereços referentes ao programa fonte e o caractere "$" que precedem os átomos; acrescentaremos, porém, o número do nó em cada átomo.

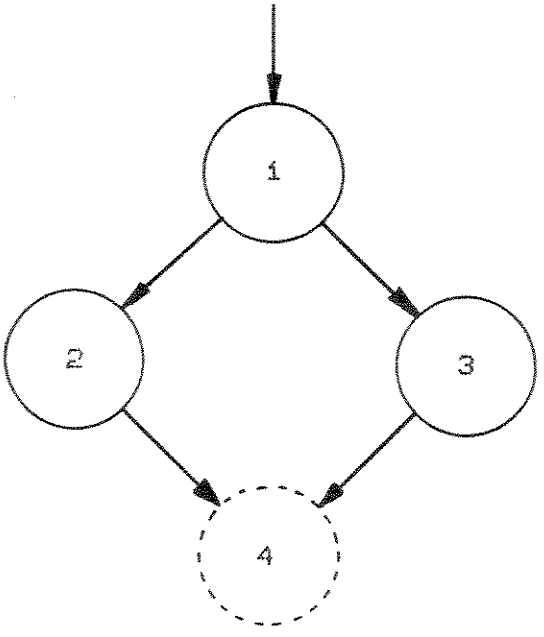
6.2 COMANDOS DE SELEÇÃO

Repetimos a seguir, o trecho de programa em LI da Subseção 4.2.2, acrescido do número do nó:

```
IF          1
C(1)1       1
S1          2
ELSE        3
S2          3
```

a matriz de adjacência e o desenho do grafo, respectivamente são:

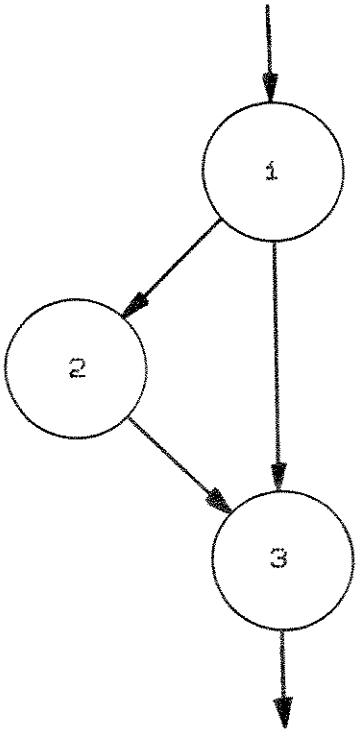
	1	2	3	4
1	0	1	1	0
2	0	0	0	1
3	0	0	0	1
4	0	0	0	0



no caso de não existir o "ELSE" teríamos:

IF	1
C(1)1	1
S1	2
S2	3

	1	2	3
1	0	1	1
2	0	0	1
3	0	0	0



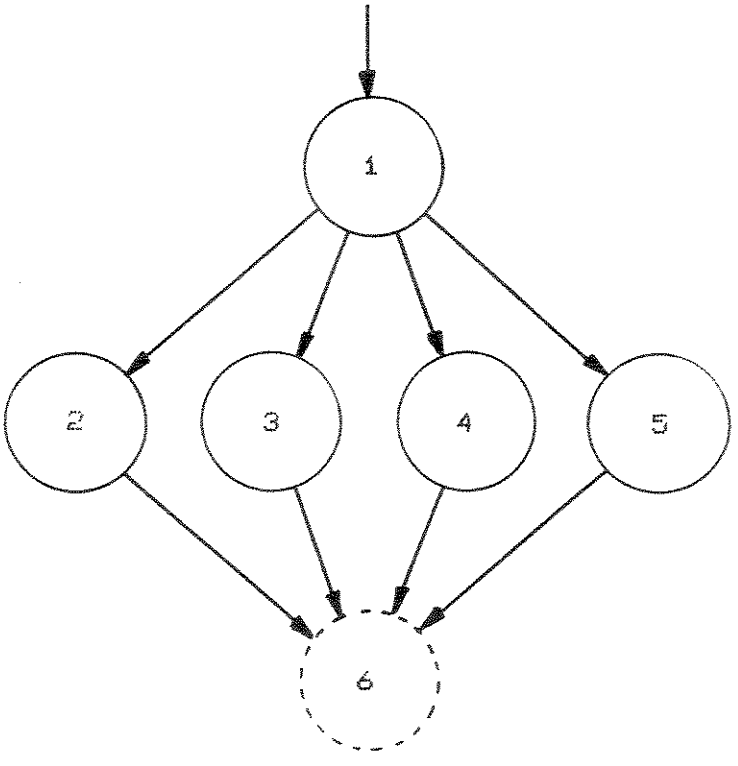
6.3 COMANDO DE SELEÇÃO MÚLTIPLA

Aproveitando, também, o trecho de programa em LI da Subseção 4.2.3, temos:

```
CASE        1
CC          1
{           1
ROTC        2
S1          2
BREAK       2
ROTC        3
S2          3
BREAK       3
ROTC        4
S3          4
BREAK       4
ROTD        5
S4          5
}           5
```

reproduzindo a matriz de adjacência e o desenho do grafo, temos:

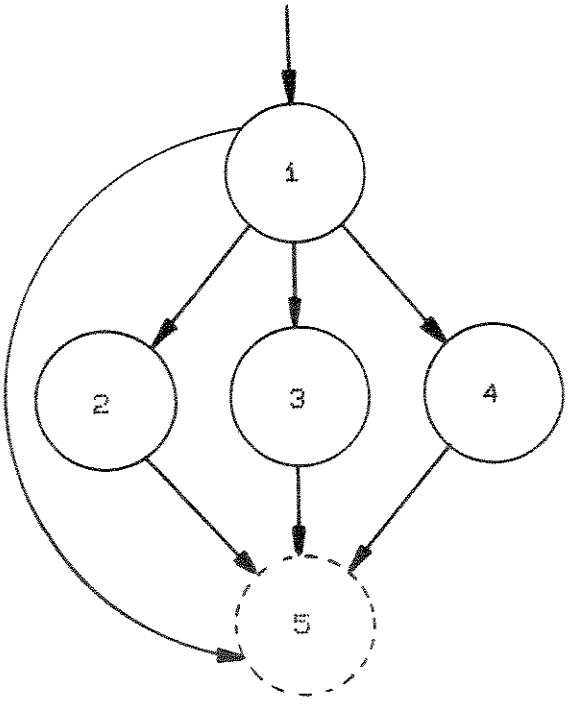
	1	2	3	4	5	6
1	0	1	1	1	1	0
2	0	0	0	0	0	1
3	0	0	0	0	0	1
4	0	0	0	0	0	1
5	0	0	0	0	0	1
6	0	0	0	0	0	0



Na hipótese de não existir o "ROTD" (nó 5), teríamos:

CASE	1
CC	1
(	1
ROTC	2
S1	2
BREAK	2
ROTC	3
S2	3
BREAK	3
ROTC	4
S3	4
BREAK	4
)	4

	1	2	3	4	5
1	0	1	1	1	1
2	0	0	0	0	1
3	0	0	0	0	1
4	0	0	0	0	1
5	0	0	0	0	0



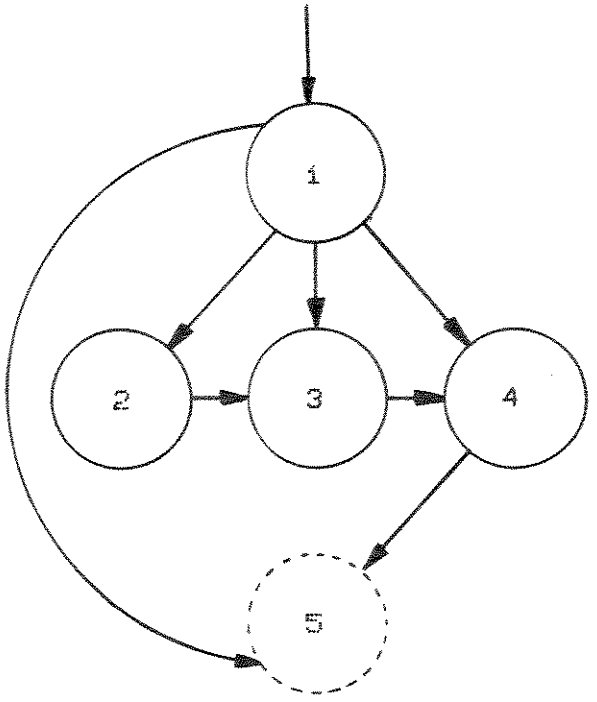
O arco (1,5) é traçado para cobrir o caso em que não ocorre coincidência entre a expressão "CC" e algum "ROTC". A execução, então, passa para o comando seguinte ao CASE, sem executar nenhum ROTC.

Se a semântica da linguagem fonte determinar que após a execução de um caso "ROTC" o fluxo prosseguirá para o próximo caso, e não existir desvio incondicional explícito, então teríamos:

```
CASE      1
CC        1
(         1
ROTC      2
S1        2
ROTC      3
S2        3
ROTC      4
S3        4
)         4
```

e a matriz de adjacência e o grafo seriam:

	1	2	3	4	5
1	0	1	1	1	1
2	0	0	1	0	0
3	0	0	0	1	0
4	0	0	0	0	1
5	0	0	0	0	0



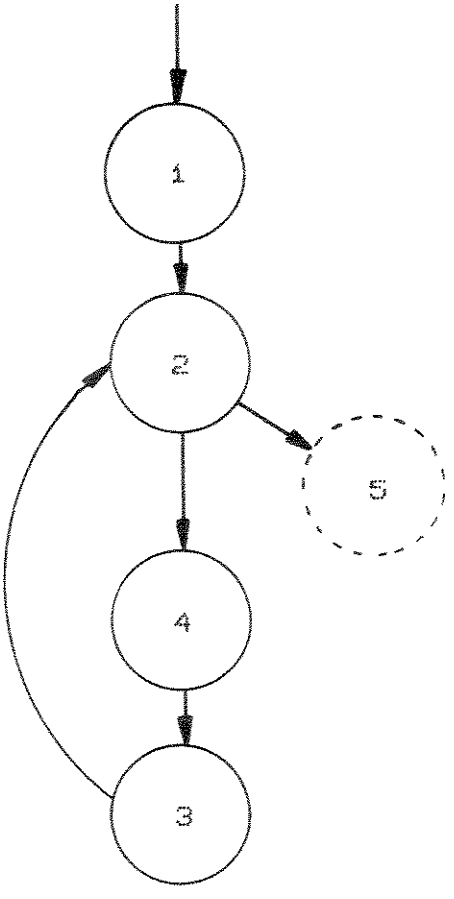
6.4 COMANDOS DE REPETIÇÃO

O primeiro trecho de programa em LI que aparece na Subseção 4.2.4 traduz um comando com um número fixo de repetições, ei-lo:

```
FOR          1
S1           1
C(1)1        2
S2           3
S3           4
```

gerando a matriz de adjacência e o desenho do grafo, obtemos:

	1	2	3	4	5
1	0	1	0	0	0
2	0	0	0	1	1
3	0	1	0	0	0
4	0	0	1	0	0
5	0	0	0	0	0



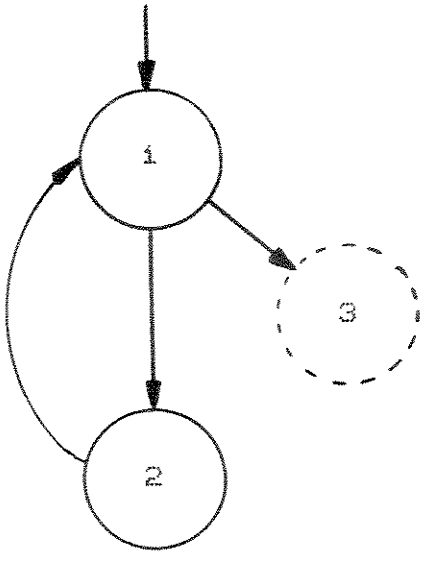
Notemos que a iniciação da variável de controle do "FOR" representada por "S1", aparece no primeiro nó, enquanto que o incremento dessa variável, representado por "S3", aparece no nó 3, ambos separados da condição do comando "C(1)1" e do corpo de execução "S3". Em alguns casos, e neste em particular, poderíamos ter fundido em um, os nós 3 e 4, sem ocasionar nenhuma incorreção, porém, a título de padronização e clareza preferimos manter esta separação em todos os casos.

Por sua vez o comando:

```
WHILE      1
C(1)1      1
S1         2
```

é assim representado:

	1	2	3
1	0	1	1
2	1	0	0
3	0	0	0



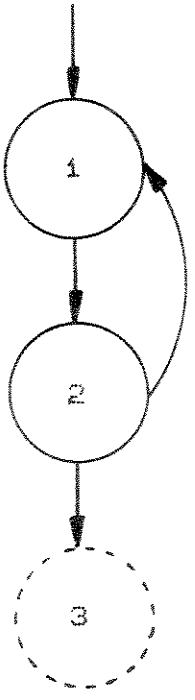
Notemos que o comando "WHILE" deve aparecer sozinho em um nó, uma vez que existe um laço chegando ao nó 1 e esse nó contém uma condição.

O último comando de repetição definido e exemplificado na Subseção 4.2.4 é o "REPEAT-UNTIL", vejamos então o trecho em LI:

```
REPEAT      1
(           1
S1          1
)           1
UNTIL       2
NC(1)1      2
```

com a execução do programa GERMA, obtemos:

	1	2	3
1	0	1	0
2	1	0	1
3	0	0	0

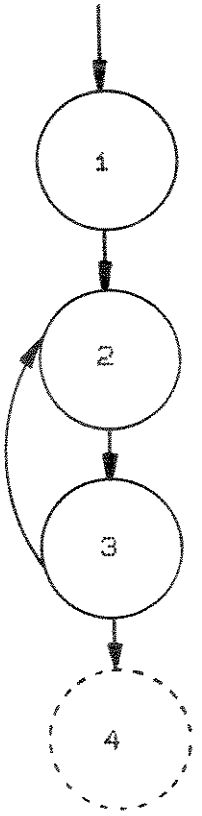


O nó que contém o átomo "REPEAT", pode conter comandos sequenciais que imediatamente o sucedem, pois nesse nó não existe qualquer condição, como é o caso do segundo nó no exemplo abaixo:

```
S1          1
S2          1
REPEAT      2
S3          2
S4          2
UNTIL       3
C(1)1       3
```

a matriz de adjacência e o desenho do grafo correspondente são:

	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	0	1	0	1
4	0	0	0	0



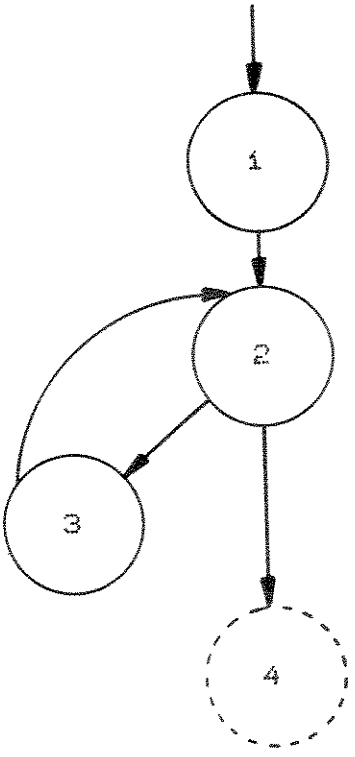
6.5 COMANDOS DE DESVIO INCONDICIONAL

Primeiramente vamos tratar o comando de desvio incondicional irrestrito. Para exemplificar utilizamos um comando GOTO que aparece após um comando de seleção:

```
S1              1
LABEL1          2
S2              2
IF C(1)1        2
GOTO            3
LABEL1          3
```

gerando a matriz de adjacência e desenhando o grafo de programa, obtemos:

	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	0	1	0	0
4	0	0	0	0



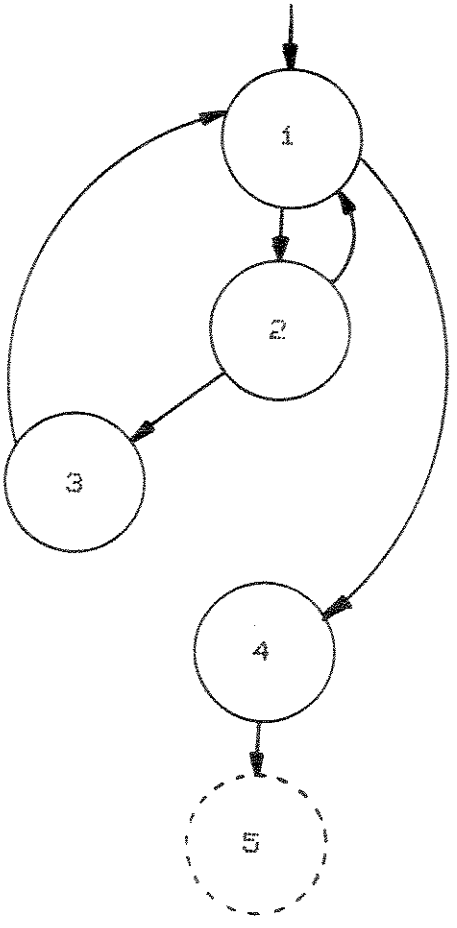
Os comandos de desvio controlados, conforme vimos na Subseção 4.2.5, são "BREAK", "CONTINUE" e "RETURN". O comando "BREAK" já foi exemplificado na Seção 6.3. Apresentamos, então, dois trechos de programa onde ocorrem os outros dois desvios. Obviamente, estes comandos somente têm sentido se fizerem parte de uma estrutura que os justifique.

Primeiramente, vejamos o "CONTINUE":

```
WHILE      1
C(2)1      1
IF         2
C(1)2      2
CONTINUE   3
S1         4
```

a matriz de adjacência e o grafo de programa respectivo são:

	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	0
3	1	0	0	0	0
4	0	0	0	0	1
5	0	0	0	0	0

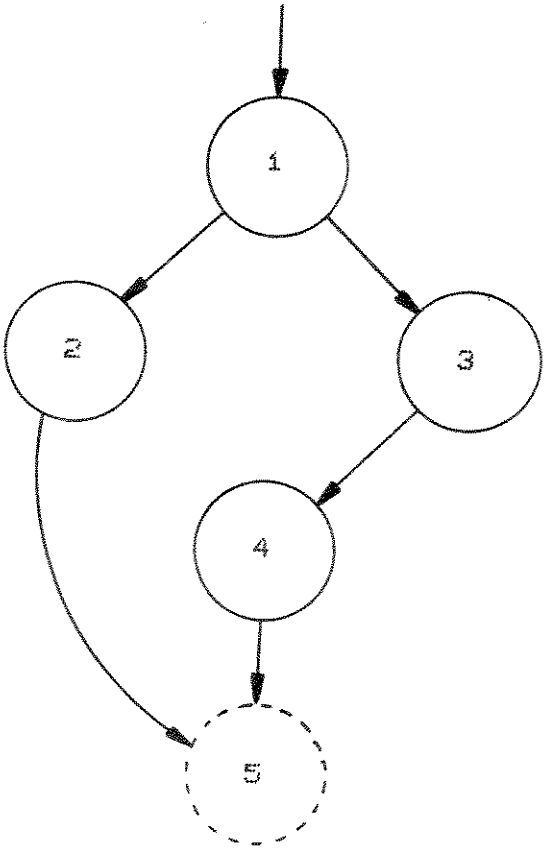


Finalizando, temos o comando "RETURN":

IF	1
C(3)1	1
RETURN	2
ELSE	3
S1	3
S2	4

a matriz de adjacência e o grafo de programa, respectivamente, são:

	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	0	1
3	0	0	0	1	0
4	0	0	0	0	1
5	0	0	0	0	0



6.6 UM EXEMPLO COM DIVERSOS COMANDOS

Embora já tenhamos visto nas seções anteriores, exemplos de todos os comandos, nesta seção apresentamos um exemplo que contém uma combinação maior de comandos. Particularmente, queremos mostrar a utilização de todos os comandos de desvio incondicional num único módulo. Consideremos, portanto, a seguinte sequência de comandos em LI: (a endentação foi incluída para facilitar o leitor; na prática ela não ocorre)

```
DCL                    1
{                      1
   S1                  1
   S2                  1
   REPEAT              2
      CASE             2
      CC               2
      {                2
         ROTC          3
            S3         3
         ROTC          4
            DCL        4
            S4         4
         ROTD          5
            S5         5
            S6         5
      }                6
   IF                  7
   C(02)1              7
      CONTINUE         8
   IF                  9
   C(01)2              9
      GOTO            10
      LABEL01         10
   ELSE               11
      BREAK           11
   UNTIL              12
   NC(01)3            12
   IF                 13
   C(01)4             13
      S7              14
```

```
    ELSE                15
       RETURN           15
LABEL01                 16
    S8                  16
    WHILE               17
    C(03)5              17
    FOR                 18
    S9                  18
    C(01)6              19
    S10                 20
    {                   21
       S11              21
       S12              21
    }                   21
}                       22
```

a matriz de adjacência correspondente é:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
12	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
17	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1
18	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0
20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
21	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

6.7 CONSIDERAÇÕES SOBRE A GERAÇÃO DO GRAFO DE PROGRAMA

Não obstante uma das restrições de entrada ser a de que os programas submetidos à execução da GFC estejam isentos de erro de sintaxe (Capítulo V), a GFC detecta se ocorrer violação dessa restrição. Naturalmente, essa robustez restringe-se a desrespeitos da sintaxe da LI, uma vez que o usuário configurador, ao criar as tabelas descritivas de uma determinada linguagem, ditará as regras de reconhecimento da estrutura dessa linguagem. Portanto, o grafo de programa, obtido a partir de um programa em LI traduzido do código fonte representa, em última análise, o fluxo de controle determinado pelo usuário configurador que construiu a tabela de equivalência sintática entre a LI e a linguagem fonte em questão.

C A P í T U L O V I I

CONCLUSÕES

7.1 A GFC E PRINCIPAIS USOS

Apresentamos a arquitetura, a abordagem adotada, os aspectos fundamentais de implementação e principais aplicações de uma ferramenta multilinguagem, que traduz programas fontes escritos em diversas linguagens procedimentais, para programas em linguagem intermediária LI. A partir dos programas traduzidos ou escritos diretamente em linguagem intermediária, é gerado o grafo de programa, representado pela matriz de adjacência. Outros produtos como uma tabela de símbolos, uma tabela de desvios incondicionais e uma de rótulos são também gerados. Para ampliar sua utilidade foram inseridas facilidades como a possibilidade de acessar um determinado trecho do programa fonte original através do código traduzido para a LI.

A definição da linguagem intermediária LI conferiu flexibilidade à ferramenta e a conveniente composição de seus elementos pode representar qualquer comando das linguagens procedimentais. Para criá-la foram estudadas as linguagens ALGOL, C, COBOL, FORTRAN, MODULA-2 e PASCAL.

Foi considerado, também, como novas linguagens podem ser incorporadas à ferramenta sem modificação de sua estrutura,

através da inclusão de descritores léxicos e sintáticos da nova linguagem.

Apresentamos o largo uso de grafo de programa em teste estrutural de programas, especialmente na seleção de casos de teste baseada em análise de fluxo de dados, em cujo contexto insere-se a GFC através de seu uso pela POKE-TOOL, cujo resultado citamos na Seção 7.3.

Mencionamos a integração da GFC no Projeto Fábrica de Software, onde está sendo desenvolvido um ambiente que objetiva evoluir o software existente através da reestruturação, remodularização e documentação total do código fonte. Esse ambiente é composto por diversas ferramentas como o "Slicer", o "Tracer" e o "Partitioner" [CAR90b], [SOB84]. Tais ferramentas baseiam-se numa representação do código fonte denominada "and-or-tree" que pode ser obtida através da linguagem intermediária LI.

Outros usos estão voltados para a obtenção de métricas sobre o software [MCC76], [PER81], [PRE87]; para a manutenção do software [MAR83]; para a análise da estrutura lógica da codificação e documentação do software [CHO90]; instrumentação do software [BEI84]; avaliação e aprimoramento da qualidade do software [HET87], [PRE87]; engenharia reversa [CHI90]; e evolução do software [SOB84], [NG90].

7.2 OBJETIVO PRINCIPAL

A GFC foi concebida levando em conta uma importante função no âmbito científico, que é suportar outras ferramentas que se apóiam em grafo de programa, e facilitar pesquisas que, de algum modo possam servir-se de seus produtos. A fim de ser mais abrangente, a GFC foi projetada para ambientes MS-DOS, UNIX e VMS. Foi codificada em C, e aplica-se a linguagens procedimentais de alto-nível, estruturadas ou não, descritas de acordo com a interface requerida pela GFC.

7.3 RESULTADOS OBTIDOS

Na atual versão da GFC, foram testados programas codificados em linguagens C e COBOL. Além da bateria de testes a qual foi submetida durante a fase de desenvolvimento, permaneceu em exposição no painel de ferramentas do IV Simpósio Brasileiro de Engenharia de Software [CAR90d], sendo atendidos na ocasião, pedidos de execução de usuários com distintos interesses.

No entanto, apresentamos como teste mais significativo os resultados positivos obtidos, através da aplicação efetiva dos Critérios Potenciais Usos descritos em [MAL88a], no uso da ferramenta POKE-TOOL (Potential Use Criteria Tool for Program Testing) [MAL89], [CHA91], citada na Seção 3.2, que utiliza a GFC para obter a versão em LI de programas codificados em C.

7.4 PRINCIPAIS DIFICULDADES NA IMPLEMENTAÇÃO

A implementação teve duplo objetivo, a comprovação da viabilidade do modelo proposto e a criação de uma ferramenta de utilidade ampla. Segundo Page-Jones [PAG88] nenhuma especificação é tão real e tangível quanto um sistema em funcionamento, onde o usário pode ver, tocar, sentir e lidar com o software.

Todavia, algumas dificuldades foram encontradas ao longo do desenvolvimento do projeto que tiveram que ser contornadas, resultando em alteração da definição inicial, sem contudo, perder-se o objetivo principal.

A linguagem FORTRAN, por exemplo, que tem um comportamento atípico relativamente ao conjunto de linguagens procedimentais, não pôde ser tratada pela GFC a menos que lhe fossem impostas restrições específicas que normalizem seu comportamento (por exemplo, palavras reservadas só poderiam ser usadas como tais).

Na linguagem COBOL, o comando "ALTER" que altera o fluxo de controle em tempo de execução não pôde ser semanticamente representado pela LI; optamos por representá-lo na LI através de um comando sequencial.

Uma dificuldade na implementação foi identificar o melhor conjunto de comandos para a LI de forma a representar adequadamente todos os comandos das linguagens procedimentais, bem como definir uma gramática para a LI que representasse as estruturas dessas linguagens.

Sem dúvida, outra dificuldade encontrada foi manter a fidelidade no endereçamento do programa fonte através do código em LI, nos casos em que a associação entre ambos não é direta, por exemplo, nos casos em que a condição está implícita no comando fonte e explícita no respectivo comando em LI.

Uma preocupação com a configuração da GFC para novas linguagens é estar nas mãos do usuário configurador a garantia de equivalência entre o grafo de programa dos comandos em LI e os da nova linguagem. O que podemos dizer a esse respeito é que o modelo oferece maior liberdade ao usuário e consequentemente lhe delega maiores responsabilidades.

7.5 TRABALHOS FUTUROS

Seria cansativo e dispersivo tentar esgotar neste trabalho todas as possíveis aplicações do grafo de programa. Nosso propósito foi mostrar que apesar do uso de grafo de programa ser tradicional, sua aplicabilidade atual é grande e crescente.

A experiência obtida na definição e implementação do modelo foi muito importante para estabelecermos as necessidades envolvidas na geração do grafo de programa para linguagens procedimentais.

Como trabalhos futuros propomos prover a GFC de maior facilidade para incorporação de novas linguagens, através de geradores automáticos de tabelas descritivas léxicas, sintáticas

e de equivalência com a LI. Sugerimos também obter-se grafos de programa para descrição da interação entre módulos, o que julgamos uma extensão natural deste trabalho. Quanto à linguagem FORTRAN, acreditamos que a definição de um pré-processador poderia adequá-la aos requerimentos da GFC.

Uma outra extensão interessante para o modelo é obter-se uma representação formal do código fonte que auxiliasse na análise da funcionalidade do software e, para tanto, propomos a obtenção de tabelas de decisão funcional associadas ao grafo de programa.

REFERÊNCIAS BIBLIOGRÁFICAS

[AHO73] Aho,A.V., Ullman,J.D., The Theory of Parsing,Translation, and Compiling - Vol. II: Compiling, Englewood Cliffs, Printice-Hall, 1973.

[AKH89] Akhras,F.N., Costa,M.C.C., "Meta-modelos para ferramentas genéricas integráveis ao ambiente SIPS", III Simpósio Brasileiro de Engenharia de Software, Recife, Out 1989, pp.113-122.

[AND80] Andrade,M.C.Q., A Criação no Processo Decisório - O Grafo como Opção Metodológica, Rio de Janeiro, Livros Técnicos e Científicos, 1980.

[AUT76] Authié,G., Otimizacao em Grafos, Campinas, Editora da UNICAMP, 1976.

[BAS81] Bastos, A.C., Programação COBOL, 3a. ed., Rio de Janeiro, Livros Técnicos e Científicos, 1981.

[BEI82] Beizer,B., Software Testing Techniques, New York, Van Nostrand Reinhold, 1982.

[BEI84] _____, Software System Testing and Quality Assurance, New York, Van Nostrand Reinhold, 1984.

[BOE87] Boehm,B.W., "Improving software productivity", IEEE Computer, Set 1987, pp.43-57.

[BOR87] Boria,J., Ingenieria de Software, Buenos Aires, Editorial Kapelusz, 1987.

[CAR90a] Carnassale,M., Evangelista,S.R.M., Moura,M.F.,Ternes,S., "Uma proposta de ferramenta para preparação de software para mudança", Relatório Técnico - Projeto Fábrica de Software (Banco do Brasil, EMBRAPA e CTI), Campinas , Jan 1990.

[CAR90b] _____, "A model for preparing software for changes", apresentado no II Workshop on the Brazilian Software Plant Project, Campinas, Mar 1990 (a ser publicado nos anais do Congresso).

[CAR90c] Carnassale,M., Jino,M., "Uma ferramenta multilinguagem para geração de fluxo de controle", XXIII Congresso Nacional de Informática, SUCESU, Rio de Janeiro, Ago 1990 (anais em meio magnético).

[CAR90d] _____, "GFC - uma ferramenta para geração de grafo de programa", IV Simpósio Brasileiro de Eng. de Software, Águas de S.Pedro - SP, Out 1990, pp.263-264.

[CHA91] Chaim,M.L., Maldonado,J.C., Jino,M., "Projeto de uma ferramenta de teste de programas", Relatório Técnico - DCA/FEE/UNICAMP, Campinas, Jan 1991.

[CHI90] Chikofsky,E.J., CrossII,J.H., "Reverse engineering and design recovery: a taxonomy", IEEE Software, Vol.7, No.1, Jan 1990, pp.13-17.

[CHO90] Choi,S.C., Scacchi,W., "Extracting and restructuring the design of large systems", IEEE Software, Vol.7, No.1, Jan 1990, pp.66-71.

[CHU87] Chusho,T., "Test data selection and quality estimation based on the concept of essencial branches for path testing", IEEE Trans. Software Eng., vol. SE-13, No.5, Mai 1987, pp.509-517.

[DAV83] Davis,G.B., Hofmann, T.R., FORTRAN 77: A Structured, Disciplined Style, New York, McGraw Hill, 1983.

[DEB90] Debugging Tool Manual 800-3849-10, Mountain View, Sun Microsystems, 1990.

[DEO74] Deo,N., Graph Theory with Applications to Engineering and Computer Science, Englewood Cliffs, Prentice-Hall, 1974.

[DUN84] Dunn, R.H., Software Defect Removal, New York, MacGraw Hill, 1984.

[FRA85] Frankl,F.G., Weyuker,E.J., "Data flow testing tool", in Proc. Softfair II, San Francisco - CA, Dez 1985, pp.46-53.

[FRA88] _____, "An aplicable family of data flow testing criteria", IEEE Trans. Software Eng., Vol.14, No.10, Out 1988, pp.1483-1498.

[FUR73] Furtado,A.L., Teoria dos Grafos - Algoritmos, Rio de Janeiro, Livros Técnicos e Científicos, 1973.

[GAN79] Gane,C.P., Sarson,T., Structured Systems Analysis: Tools and Techniques, Englewood Cliffs, Prentice-Hall, 1979.

[GHE87] Ghezzi,C., Mehdi,J., Programming Language Concepts, 2a. ed., New York, John Wiley, 1987.

[GOO83] Goos,G., Wulf,W.A., Evans Jr.,A., Butler.K.J. (edit.), DIANA An Intermediate Language for Ada, Berlin Heidelberg, Springer-Verlag, 1983.

[GRI71] Gries,D., Compiler Construction for Digital Computers, New York, John Wiley, 1971.

[HEC77] Hecht,M.S., Flow Analysis of Computer Programs, New York, North-Holland, 1977.

[HEH87] Hehl,M.E., Linguagem de Programação Estruturada FORTRAM 77, 2a. ed., S.Paulo, McGraw Hill, 1987.

[HER76] Herman,P.M., "Data flow approach to program testing", Australian Computer Journal, Vol.8, No.3, Nov 1976, pp.92-96.

[HET87] Hetzel,W., Guia Completo ao Teste de Software, Rio de Janeiro, Editora Campos, 1987.

[HOR84] Horowitz,E., Munson,B., "An expansive view of reusable software", IEEE Trans. Software Eng., Vol. SE-10, Set 1984, pp.477-487.

[IEE83] IEEE Standard Glossary of Software Engineering Terminology - Std 729, New York, Institute of Eletrical and Eletronics Engineers, 1983.

[JEN84] Jensen,K., Wirth,N., PASCAL - Manual del Usuario e Informe, 2a. ed., Buenos Aires, El Atneo, 1984.

[JOH75] Johnson,S.C., "YACC: Yet Another Compiler-Compiler", Computing Science Technical Report, No.32, Bell Laboratories, New Jersey, Murray Hill, 1975.

[KER87] Kernigham,B.W., Ritchie,D.M., C a Linguagem de Programação, 3a. ed., Porto Alegre, Editora Campos, 1987.

[KOR87] Korel,B. "The program dependence graph in static program testing", Information Processing Letters, Vol.24, No.2, Jan 1987, pp.103-108,

[LAS83] Laski,J.W., Korel,B., "A data flow oriented program testing strategy", IEEE Trans. Software Eng., Vol. SE-9, No.3, Mai 1983, pp.347-354.

[LEH80] Lehman,M.M., "Programs, life cycles and laws of software evolution", Proc IEEE, Vol.68,No.9, Set 1980, pp.199-215.

[MAL88a] Maldonado,J.C., Chaim,M.L., Jino,M., "Seleção de casos de testes baseada em fluxo de dados através dos critérios

potenciais usos", II Simpósio Brasileiro de Eng. de Software, Canela-RS, Out 1988, pp.24-35.

[MAL88b] _____, "Resultados do estudo de uma família de critérios de teste de programas baseado em fluxo de dados", Relatório Técnico DCA/RT/001/88 - DCA/FEE/UNICAMP, Campinas, Dez 1988.

[MAL89] _____, "Arquitetura de uma ferramenta de teste de software de apoio aos critérios potenciais usos", XXII Congresso Nacional de Informática, SUCESU, São Paulo, Set 1989, pp.92-101.

[MAR83] Martin,J., McClure,C., Software Maintenance: The Problem and Its Solutions, Englewood Cliffs, Prentice-Hall, 1983.

[MAR85] Martin, J., Fourth Generation Languages - Vol.1: Principles, New Jersey, Prentice-Hall, 1985.

[MCC76] McCabe,T., "A complexity measure", IEEE Trans.Software Eng., Vol. SE-2, No.4, Dez 1976, pp.308-320.

[MYE79] Myers,G.J., The Art of Software Testing, New York, John Wiley, 1979.

[NG90] Ng,P.A., Yeh,R.T. (edit.), Modern Software Engineering - Fundations and Current Perspectives, New York, Van Nostrand Reinhold, 1990.

[NTA84] Ntafos,S.C., "On required element testing", IEEE Trans. Software Eng., Vol. SE-10, Nov 1984, pp. 795-803.

[NTA88] Ntafos,S.C., "A comparison of some structural testing strategies", IEEE Trans. Software Eng. Vol.14, No.6, Jun 1988, pp.868-873.

[OVI80] Ovideo,E.I., "Control flow, data flow and program complexity", Proc. IEEE Compsac 80, Out 1980, pp.146-152.

[PAG88] Page-Jones,M., Projeto Estruturado de Sistemas, São Paulo, McGraw Hill, 1988.

[PER81] Perlis,A., Sayward,F., Shaw,M., (edit.), Software Metrics: An Analysis and Evaluation, Cambridge, Massachusetts Institute of Technology Press, 1981.

[PRE87] Pressman,R.B., Software Engineering: A Practitioner's Approach, 2a. ed., New York, MacGraw-Hill, 1987.

[RAP85] Rapps,S., Weyuker,E.J., "Selecting software test data using data flow information", IEEE Trans. Software Eng., vol. SE-11, No.4, Abr 1985, pp.367-375.

[RUT67] Rutijhauser,H., Description of ALGOL 60, Berlim Heidelberg, Springer-Verlag, 1967.

[SEB89] Sebesta,R.W., Concepts of Programing Languages, Redwood City, Benjamin Cummings Publishing Company, 1989.

[SEG81] Segre,L.M., Linguagem de programação ALGOL, Rio Janeiro, Editora Campus, 1981.

[SET83] Setzer,V.W., Melo, I.S.H., A Construção de Um Compilador, Rio de Janeiro, Editora Campus, 1983.

[SOB84] Sobrinho,F.G., "Structural complexity: a basis for systematic software evolution", Tese de Doutorado, Department of Computer Science and College of Business and Management, University of Maryland, Jul 1984.

[STE78] Stern,N., Stern,R., Programação COBOL, Rio de Janeiro, Editora Guanabara Dois, 1978.

[SWA81] Swamy,M.N.S., Thulasiraman, K., Graphs, Networks, and Algorithms, New York, John Wiley, 1981.

[SZW86] Szwarcfiter,J.M., Grafos e Algoritmos Computacionais, Rio de Janeiro, Editora Campos, 1986.

[THA85] Thalmann, D., MODULA-2 (Computer program language), Berlim Heidelberg, Springer-Verlag, 1985.

[TUR88] Turbo Debugger User's Guide - versão 1.0, Scotts Valley, Borland International, 1988.

[VEL79] Veloso,P.A.S., Máquinas e Linguagens: uma Introdução à Teoria de Autômatos, Escola de Computação, USP/IME, São Paulo, 1979.

[VEL86] _____, Verificação e Construção de Programas, Campinas, Editora da UNICAMP, 1986.

[WAI84] Waite,W.M., Goos,G., Compiler Construction, New York, Springer-Verlag, 1984.

[WIR71] Wirth,N., "The design of PASCAL compiler", Software - Pratice and Experience 1, Vol.1, No.3, Jul-Set 1971, pp.309-333.

A P Ê N D I C E

TABELAS DESCRITIVAS DA LINGUAGEM C:

Tabela de Palavras Reservadas da Linguagem C:

auto 99
break 10
case 15
char 99
continue 25
close 99
default 30
define 99
do 35
double 99
else 45
entry 99
exit 99
extern 99
fclose 99
float 99
fopen 99
for 65
fprintf 99
fscanf 99
getchar 99
goto 70
if 75
include 99
int 99
long 99
main 99
open 99
printf 99
register 99
return 95
short 99
sizeof 99
sprintf 99
sscanf 99

Tabela de Transições Léxicas da Linguagem C:

```
"i \n\t\b\f\r" 0 0 0 0
"i/" 0 1 1 9
"i\"" 0 4 0 16
"i\'" 0 6 0 20
"cl( )()?;:/\"\' \n\t\b\r& " 0 8 1 24
"il" 0 9 1 26
"i( )()?;:" 0 0 4 0
"i&" 0 10 1 28
"i " 0 11 1 30
"c*" 1 0 5 0
"i*" 1 3 7 14
"i/" 2 0 0 0
"i*" 2 2 0 11
"c/*" 2 3 0 14
"i*" 3 2 0 11
"c*" 3 3 0 14
"i\"" 4 0 0 0
"i\\" 4 5 0 19
"c\"\\" 4 4 0 16
"c\0" 5 4 0 16
"i\'" 6 0 0 0
"c\'\\" 6 6 0 20
"i\\" 6 7 0 23
"c\0" 7 6 0 20
"il( )()?;:/\"\' \n\b\t\r\f& " 8 0 5 0
"cl( )()?;:/\"\' \n\b\t\r\f& " 8 8 1 24
"ild_." 9 9 2 26
"cld_." 9 0 3 0
"i&" 10 0 6 0
"c&" 10 0 5 0
"i " 11 0 6 0
"c " 11 0 5 0
```

Tabela de Transições Sintáticas da Linguagem C:

```
"c;  { } else return break continue case switch default  unsigned
char  int  void double float long short : if while do goto for  ?
struct union register extern auto static ( main" 0 0 0 0
"i;" 0 0 1 0
"i{" 0 0 13 0
"i}" 0 0 12 0
"ielse" 0 0 4 0
"i:" 0 0 14 0
"iif" 0 1 4 19
"iwhile" 0 1 10 19
"ido" 0 3 5 22
"igoto" 0 4 4 24
"ifor" 0 5 4 25
"i?" 0 9 0 32
"ichar  int  unsigned void double float long short register  auto
static" 0 19 0 52
"icase default" 0 12 19 39
"ireturn break continue" 0 13 19 41
"i(" 0 15 0 44
"iswitch" 0 17 4 48
"iextern typedef struct union" 0 10 0 34
"imain" 0 18 0 50
"i(" 1 2 0 20
"c)" 2 2 3 20
"i)" 2 0 15 0
"cwhile" 3 0 8 0
"iwhile" 3 1 4 19
"c\b" 4 14 2 43
"i(" 5 6 15 26
"c;" 6 6 0 26
"i;" 6 7 1 28
"c;" 7 7 6 28
"i;" 7 8 6 30
"c)" 8 8 7 30
"i)" 8 0 15 0
"c;" 9 9 0 32
"i;" 9 0 1 0
"c; {" 10 10 0 34
"i;" 10 0 17 0
"i{" 10 11 0 37
"c}" 11 11 11 37
"i}" 11 16 0  46
"c:" 12 12 0 39
"i:" 12 0 20 0
"i;" 13 0 20 0
"c;" 13 13 0 41
"i;" 14 0 0 0
```

```
"cint char double float short long unsigned" 15 0 8 0
"iint char double float short long unsigned" 15 0 0 0
"c;" 16 16 0 46
"i;" 16 0 17 0
"c{" 17 17 0 48
"i{" 17 0 16 0
"c{" 18 18 0 50
"i{" 18 0 18 0
"d0" 19 19 0 52
"s0" 19 20 0 54
"i(" 20 21 0 57
"i;" 20 0 17 0
"c( ;" 20 16 0 46
"c)" 21 21 22 57
"i)" 21 22 0 59
"i;" 22 0 17 0
"c;" 22 0 18 0
```

Tabela de Equivalência Sintática entre a Linguagem C e a LI:

```
i{ 13 1 {
i) 12 1 )
iif 4 1 $IF
ielse 4 1 $ELSE
ireturn 4 1 $RETURN
ibreak 4 1 $BREAK
icontinue 4 1 $CONTINUE
icase 4 1 $ROTC
iswitch 4 1 $CASE
idefault 4 1 $ROTD
2while 10 1 $UNTIL 4 1 $WHILE
ifor 4 1 $FOR
ido 5 1 $REPEAT
igoto 4 1 $GOTO
```

TABELAS DESCRITIVAS DA LINGUAGEM COBOL:

Tabela de Palavras Reservadas da Linguagem COBOL:

ACCEPT 50
ADD 50
ADDRESS 99
ADVANCING 99
AFTER 13
ALL 14
ALPHABETIC 11
ALTER 50
ALTERNATE 99
AND 12
APPLY 99
ARE 99
AREA 99
AREAS 99
ASCENDING 99
AT 99
BEFORE 99
BY 33
CALL 99
CLOSE 99
COMPUTE 50
CONSOLE 99
CORR 99
CURRENT-DATE 99
DEPENDING 28
DISP 99
DISPLAY 50
DISPLAY-ST 50
DIVIDE 50
EJECT 99
ELSE 21
END 22
END-OF-PAGE 99
EOP 99
EQUAL 11
EQUALS 11
ERASE 50
ERROR 26
EXAMINE 36
EXIT 50
FROM 32
GIVING 99
GO 27
GREATER 11

I-O 99
IF 20
INPUT 99
INPUT-OUTPUT 99
INTO 99
INVALID 24
IS 11
KEY 23
KEYS 23
LESS 11
LOW-VALUE 99
LOW-VALUES 99
MOVE 50
MULTIPLE 99
MULTIPLY 50
NEGATIVE 11
NEXT 99
NO 99
NOT 11
NOTE 99
NUMERIC 11
OF 99
OFF 99
ON 29
OPEN 50
OR 12
OTHERWISE 21
OUTPUT 99
PERFORM 30
POSITION 99
POSITIONING 99
POSITIVE 11
READ 50
READY 99
REMAINDER 99
RETURN 50
RETURN-CODE 99
REWIND 50
REWRITE 50
ROUNDED 99
RUN 35
SEARCH 16
SENTENCE 50
SIZE 25
SORT 50
SORT-CORE-SIZE 50
SORT-FILE-SIZE 50
SORT-MODE-SIZE 50
SPACE 11
SPACES 11

```
STOP 34
SUBTRACT 50
SUM 99
THEN 99
THROUGH 31
THRU 31
TIMES 19
TO 11
UNEQUAL 11
UNTIL 15
UP 99
UPDATE 50
UPON 99
USING 99
WHEN 99
WITH 99
WRITE 50
WRITE-ONLY 50
WRITE-VERIFY 50
ZERO 11
ZEROES 11
ZEROS 11
```

Tabela de Transições Léxicas da Linguagem COBOL:

```
"id" 0 1 0 1
"id" 1 2 0 2
"id" 2 3 0 3
"id" 3 4 0 4
"id" 4 5 0 5
"id" 5 6 0 6
"i " 6 7 0 9
"i*" 6 14 0 23
"i\n" 6 0 0 0
"i " 7 8 10 11
"ild-" 7 11 1 17
"i " 8 9 10 13
"ild-" 8 11 1 17
"i " 9 10 10 15
"ild-" 9 11 1 17
"i " 10 15 10 25
"ild-" 10 11 1 17
"ild-" 11 11 1 17
"i. " 11 12 2 19
"i." 12 13 8 21
"c." 12 12 0 19
"i " 13 13 0 21
"i\n" 13 0 0 0
"c\n" 14 14 0 23
"i\n" 14 0 0 0
"i ," 15 15 0 25
"i." 15 15 8 25
"i()" 15 15 13 25
"c.()ld ,\n\"" 15 15 6 25
"i\"" 15 16 0 33
"il" 15 18 1 38
"id" 15 19 1 44
"i\n" 15 21 14 55
"c\"\n" 16 16 0 33
"i\"" 16 15 0 25
"i\n" 16 17 0 36
"c\"" 17 17 0 36
"i\"" 17 16 0 25
"i ," 18 15 3 25
"i.()" 18 15 5 25
"il" 18 18 1 38
"id-" 18 20 1 50
"i\n" 18 21 14 55
"c.()ld- ,\n" 18 15 11 25
"id" 19 19 1 44
"il-" 19 20 1 50
"i.()" 19 15 4 25
```

```
"i\n" 19 21 7 55
"c.()ld- ,\n" 19 15 6 25
"i ," 19 15 7 25
"ild-" 20 20 1 50
"i ," 20 15 3 25
"i.()" 20 15 11 25
"i\n" 20 21 14 55
"c.()ld- ,\n" 20 15 7 25
"id" 21 22 10 56
"id" 22 23 10 57
"id" 23 24 10 58
"id" 24 25 10 59
"id" 25 26 10 60
"id" 26 27 10 61
"i-" 27 28 10 65
"i*" 27 14 12 23
"i " 27 7 12 9
"i\n" 27 0 0 0
"c " 28 18 9 38
"i " 28 7 9 9
```

Tabela de Transições Sintáticas da Linguagem COBOL:

```
"d3 8 9 16 18 20 21 22 24 25 27 30 34 37 50" 0 0 0 0
"s50" 0 26 0 64
"iIF" 0 1 2 15
"iSEARCH" 0 2 12 17
"s21 22" 0 0 4 0
"iINVALID" 0 4 2 21
"s9" 0 5 3 23
"iSIZE" 0 6 12 24
"iGO" 0 7 10 25
"iPERFORM" 0 11 12 33
"iWHEN" 0 0 4 0
"iSTOP" 0 20 12 53
"iEXAMINE" 0 21 12 55
"i." 0 0 15 0
"s8" 0 0 21 0
"s0 1 2 11 12" 1 1 7 15
"d0 1 2 11 12" 1 0 6 0
"iALL" 2 3 23 19
"cALL" 2 3 22 19
"cWHEN" 3 3 0 19
"iWHEN" 3 1 2 15
"iKEY" 4 4 0 21
"cKEY" 4 0 1 0
"i." 5 0 5 0
"iERROR" 6 0 20 0
"iTO" 7 7 0 25
"s0" 7 8 8 27
"d0" 8 0 9 0
"s0" 8 9 8 29
"s0" 9 9 8 29
"iDEPENDING" 9 10 11 31
"iON" 10 10 0 31
"s0" 10 0 18 0
"s0" 11 12 0 34
"s31" 12 12 0 34
"d0 31" 12 0 16 0
"s0" 12 13 0 37
"s0 1" 13 13 0 37
"iTIMES" 13 0 5 0
"iUNTIL" 13 14 0 42
"iVARYING" 13 15 0 44
"d0 1 15 17 19" 13 0 16 0
"s0 1 2 11 12" 14 14 7 42
"d0 1 2 11 12" 14 0 14 0
"s0" 15 16 0 45
"iFROM" 16 16 0 45
"s0 1" 16 17 0 47
```

```
"iBY" 17 17 0 47
"s0 1" 17 18 0 49
"iUNTIL" 18 19 0 50
"s0 1 2 11 12" 19 19 7 50
"iAFTER" 19 15 19 44
"d0 1 2 11 12 13" 19 0 17 0
"iRUN" 20 0 5 0
"cRUN" 20 0 13 0
"iTALLYING" 21 22 0 57
"iREPLACING" 21 24 0 61
"d0 1" 22 22 0 57
"s0 1" 22 23 0 59
"cREPLACING" 23 0 4 0
"iREPALCING" 23 24 0 61
"cBY" 24 24 0 61
"iBY" 24 25 0 63
"s0 1" 25 0 25 0
"d3 8 9 16 18 20 21 22 24 25 27 30 34 37 50" 26 26 0 64
"s3 8 9 16 18 20 21 22 24 25 27 30 34 37 50" 26 0 16 0
```

Tabela de Equivalência Sintática entre a Linguagem COBOL e a LI:

```
1IF 2 1 $IF
2WHEN 2 1 $IF 4 5 $BREAK ) $IF $C1 (
1INVALID 2 3 $IF $C1 (
1ERROR 5 3 $IF $C1 (
2GO 9 1 $GOTO 11 1 $CASE
1ELSE 4 3 ) $ELSE (
1OTHERWISE 4 3 ) $ELSE (
1END 4 3 $IF $C1 (
2. 2 1 ) 5 1 (
3PERFORM 5 3 $WHILE $C1 $S 14 1 $WHILE 17 2 $FOR $S
1RUN 5 1 $RETURN
2SEARCH 22 5 $FOR $S $C1 $S ( 23 4 $FOR $S $C1 $S
1EXAMINE 24 5 $FOR $S $C1 $S $S
```

AÇÕES SEMÂNTICAS UTILIZADAS NA ANÁLISE LÉXICA:

ASL-1 - Armazena caractere lido do módulo fonte numa cadeia e incrementa contador de caracteres;

ASL-2 - Despreza caractere lido do módulo fonte e incrementa contador de caracteres;

ASL-3 - Despreza caractere lido do módulo fonte e decrementa contador de caracteres;

ASL-4 - Despreza caractere lido do módulo fonte (não altera o contador de caracteres);

ASL-5 - Testa quantidade de caracteres numa determinada cadeia;

ASL-6 - "Esvazia" uma cadeia de caracteres armazenada, isto é, torna seu tamanho igual a zero;

ASL-7 - Chama a função "pesquisa-tab" cuja tarefa é pesquisar (busca binária) se uma determinada cadeia de caracteres encontra-se na tabela de palavras reservadas, se encontrada a função retorna a classe da palavra reservada ; se a cadeia não for encontrada então é chamada a função "grava-simbol" cuja tarefa é

gravar essa cadeia na tabela de símbolos; e a função "pesquisa-tab" retorna o valor 0 (que significa tratar-se de um identificador);

ASL-8 - Retorna o valor 1 quando é identificada uma determinada cadeia de caracteres que não interessa à análise sintática;

ASL-9 - Dada uma cadeia de caracteres qualquer, converte as letras maiúsculas em minúsculas;

ASL-10 - Dada uma cadeia de caracteres qualquer, converte as letras minúsculas em maiúsculas;
Observação: - a função "grava-simbol" além de gravar a cadeia de caracteres, no caso um identificador, grava também o endereço desse identificador referente ao código fonte, ou seja, comprimento, tamanho e número da linha;

ASL-11 - Chama a função "msg-erro" cuja função é emitir mensagem de erro devido a incompatibilidade na análise léxica.

AÇÕES SEMÂNTICAS UTILIZADAS NA ANÁLISE SINTÁTICA:

ASS-1 - Coloca um item léxico no topo da pilha sintática;

ASS-2 - Retira um item léxico do topo da pilha sintática; se pilha vazia emite mensagem de erro;

ASS-3 - Retira todos os itens léxicos da pilha sintática;

ASL-4 - Chama a função "msg-erro" cuja tarefa é emitir mensagem de erro devido a incompatibilidade na análise sintática;

ASS-5 - Grava item léxico na tabela de rótulos;

ASS-6 - Grava item léxico na tabela de desvios incondicionais;

ASS-7 - Grava átomo da LI em arquivo de saída com mesmo nome do arquivo de entrada que contém o módulo fonte analisado, porém, com a extensão "LI";

ASS-8 - Chama a função "conta-cond" que quantifica os predicados de uma expressão lógica retornando um valor inteiro maior que zero;

ASS-9 - Incrementa contador de comandos sequenciais;

ASS-10 - Incrementa contador de condições;

Observações: - Todas as ações que executam gravação, além de gravarem o elemento principal citado em cada uma delas, gravam também o endereço relativo ao programa fonte desses elementos, ou seja, comprimento, tamanho e número da linha.