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

SUPORTE AO TESTE DE PROGRAMAS FORTRAN-77 NO AMBIENTE POKE-TOOL

Autor: *Rubens Pontes da Fonseca*

Orientador: *Prof. Dr. Jose' Carlos Maldonado*

Co-Orientador: *Prof. Dr. Ma'rio Jino*

Dissertação apresentada à Faculdade de Engenharia Elétrica da Universidade Estadual de Campinas como parte dos requisitos para obtenção do grau de Mestre em Engenharia Elétrica.

Janeiro de 1993

A

Noélia,
Pedro Henrique,
Claudia e
Otávio

AGRADECIMENTOS

Inicialmente, gostaria de deixar registrado minha gratidão ao nosso Bom Deus, Criador e Mantenedor da vida e de tudo que é bom;

Ao Prof. Dr. Mário Jino pela oportunidade, receptividade e eficiente orientação;

Ao Prof. Dr. José Carlos Maldonado pela dedicação, otimismo, idéias, motivação e principalmente pela amizade e paciência;

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

Aos meus amigos do Grupo de Teste de Software do Departamento de Engenharia da Computação e Automação Industrial da Universidade Estadual de Campinas: Marcos Lordello Chaim, Mauro Carnassale, Sílvia Regina Vergilio, Plínio de Sá Leitão Jr. e Plínio Vilela, sempre presente;

À Cia. Siderúrgica Nacional pela oportunidade e suporte financeiro e aos amigos que nela trabalham pelo incentivo;

À Fundação Gen. Edmundo de Macedo Soares e Silva (FUGEMSS) pelo apoio administrativo, e em especial à Pedagoga Ignês Mello de Oliveira.

RESUMO

As atividades de teste conduzidas manualmente são tão sujeitas a erros quanto as outras atividades do desenvolvimento de software. Critérios de teste estrutural baseados em análise de fluxo de dados têm sido propostos para o teste de unidade; não existe uma ferramenta de teste que apóie a aplicação desses critérios para programas implementados na linguagem FORTRAN-77. No DCA/FEE/UNICAMP foi desenvolvida uma ferramenta de teste que apóia a aplicação dos Critérios Potenciais Usos PU (uma família de critérios de fluxo de dados), denominada POKE-TOOL, configurável para diversas linguagens e que está operacional para as linguagens C e COBOL. Este trabalho apresenta a configuração da POKE-TOOL para a linguagem FORTRAN-77. A POKE-TOOL/versão FORTRAN-77 foi validada parcialmente com sua aplicação no teste de um conjunto de programas selecionados da literatura; os resultados dessa atividade são também apresentados e brevemente analisados neste trabalho.

ABSTRACT

Testing activities applied manually are as error-prone as other software development activities. Data-flow based structured testing criteria have been proposed for unit testing; there is no testing tool supporting these criteria for programs implemented in the FORTRAN-77. At DCA/FEE/UNICAMP a configurable testing tool, named POKE-TOOL, which supports the application of the Potential Uses Criteria (a data-flow based testing criteria family) has been developed and is operational for languages C and COBOL. This work presents the configuration of POKE-TOOL for FORTRAN-77. POKE-TOOL/version FORTRAN-77 has been partially validated using a program set selected from the literature; the results of this activity are also presented and briefly analysed in this work.

ÍNDICE

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO

1

INTRODUÇÃO

Neste capítulo são discutidos o contexto, a motivação, e a importância deste trabalho dentro da área de Engenharia de Software e os objetivos principais a serem atingidos. A organização da dissertação é apresentada na última seção deste capítulo.

1.1 CONTEXTO EM QUE O TRABALHO SE INSERE

A sociedade moderna se depara cada vez mais com o uso intenso de computadores em quase todas as suas atividades.

O software se torna o elemento principal na evolução de sistemas e produtos baseados em computadores. Estima-se que, já no começo desta década, uma organização média estará gastando 80 por cento de seu orçamento de computação no software, e apenas 20 por cento no hardware [YOU89].

Uma nova disciplina surgiu em resposta às necessidades de desenvolvimento de softwares que atendam a requisitos de custo, tempo e qualidade: é a Engenharia de Software.

A Engenharia de Software engloba três elementos —*métodos, ferramentas e procedimentos*—, que possibilitam ao gerente controlar o processo de desenvolvimento de software e fornecem uma base para a construção de software de alta qualidade, de uma maneira produtiva [PRE87].

Pressman apresenta o processo de desenvolvimento de software como tendo três fases genéricas. São elas: a *definição*, o *desenvolvimento* e a *manutenção* [PRE87].

Na *fase de definição* são identificadas as informações que deverão ser processadas, as funções e o desempenho desejados, as interfaces que deverão ser estabelecidas, as restrições do projeto e os critérios de validação requeridos. É a fase da identificação e definição do que deverá ser feito e envolve três passos: Análise do Sistema, Planejamento do Projeto de Software e Análise de Requisitos.

Na *fase de desenvolvimento* dá-se enfase ao estabelecimento de *como* os objetivos serão alcançados. Descreve-se a estrutura dos dados, a arquitetura do software, detalhes de procedimentos, a tradução para uma linguagem de programação e a especificação dos testes. Esta fase é executada em três passos: Projeto do Software, Codificação e Teste do Software.

A *fase de manutenção* enfoca as mudanças associadas com a correção de erros identificados pelo usuário, adaptações requeridas em função da evolução do ambiente de software e modificações decorrentes de mudanças nos requisitos dos usuários. Portanto, a manutenção pode ser corretiva, adaptativa ou perfectiva [PRE87].

Essas fases e os passos relacionados são complementados por um conjunto de outras atividades denominado *Garantia de Qualidade de Software*.

A Garantia de Qualidade de Software é uma atividade técnica

cujo objetivo é garantir que tanto o processo de desenvolvimento quanto o produto atinjam níveis de qualidade especificados. Neste contexto destacam-se algumas definições [BUC84] :

1) Qualidade: É a totalidade dos atributos e características de um produto ou serviço que residem em sua habilidade de satisfazer a determinadas necessidades.
2) Garantia de Qualidade: É um plano padronizado e sistemático de todas as ações necessárias para fornecer confiança suficiente de que um ítem ou projeto atende a requisitos técnicos estabelecidos.
3) Controle de Qualidade : São aquelas ações de Garantia de Qualidade que fornecem um meio para controlar e medir as características de um ítem, processo ou recurso em relação a requisitos estabelecidos.

Um dos aspectos motivadores para a implantação de um *Controle da Qualidade de Software* está diretamente ligado com a progressão apresentada pelos custos de correção e modificação de um software, através das fases do seu *ciclo de vida*.

Um levantamento feito [BOE81] a respeito do incremento nos custos para a correção (reparação ou modificação) de um software, considerando a fase na qual o erro foi detetado e corrigido, indicou que, se um erro nos requisitos de um software é identificado e corrigido durante o planejamento e a fase de definição de requisitos, sua correção é relativamente simples pois refere-se somente à atualização da especificação de requisitos. Se o mesmo erro não é corrigido até a fase de manutenção, a correção envolve vários documentos, entre eles, o conjunto de especificações, o código, os manuais do usuário e de manutenção, e o material de treinamento.

Além disso, correções em fases finais do desenvolvimento envolvem uma maior formalização quanto à aprovação e controle de modificações; torna também mais extensa a atividade de revalidação da correção. Esses fatores combinados tornam, tipicamente, em grandes projetos, a correção de um erro até 100 vezes mais cara na

fase de manutenção do que quando feitos na fase de requisitos [BOEB1].

Considerando-se apenas o efeito nos custos causados por erros no software, apresenta-se a experiência da IBM [PRE87]. Ela é mostrada na Tabela 1.1 e reflete o impacto da correção de defeitos nas diversas fases do ciclo de vida.

Um outro fator motivador para a busca de um maior controle da qualidade do software produzido por uma organização é constituido pelos resultados de pesquisas feitas a respeito do custo da manutenção de software em relação ao custo total (relativo a toda sua vida útil). São mostrados na Tabela 1.2 os resultados obtidos por vários autores [ART88] .

Segundo Page-Jones [PAG88], a manutenção corresponde a 67% do custo de toda a vida útil de um sistema; isso representa o dobro da quantia gasta durante o desenvolvimento (33%). Nesses 33% incluem-se: levantamento de necessidades(3%), especificações(3%), projeto(5%), codificação(7%), teste de unidades(8%) e teste de integração(7%). Testes respondem por 15% do custo total. Assim, 82% do investimento feito no desenvolvimento de um software pode ser consumido em testes, correção, modificação e melhoria.

Esses resultados mostram que, em geral, 80% ou mais da mão de obra disponível de um centro de desenvolvimento de sistemas pode estar voltada para a manutenção. Para buscar a melhoria desses resultados, deve-se adotar técnicas de Garantia de Qualidade desde as fases iniciais do processo de desenvolvimento de software.

Validação e *Verificação* são duas atividades centrais de Garantia de Qualidade de Software [MAL91].

A atividade de verificação fornece uma avaliação das funções de processamento. Verifica se o processo de desenvolvimento de software está correto e se existe adequação aos padrões pré-estabelecidos da organização. Já o processo de validação analisa características do sistema usando condições nominais e anômalas de performance. O produto é comparado com os requisitos

Tabela 1.1 - Custo de um Erro de Software por Fase.

FASE	CUSTO
Projeto	1,00
Pouco antes dos testes	6,50
Durante os testes	15,00
Após a entrega	67,00

Tabela 1.2 - Custo da Manutenção Dentro do "Ciclo de Vida" de Software.

FONTE	ANO	MANUTENÇÃO (%)
Canning	1972	60
Boehm	1973	40-80
DeRose & Nyman	1973	60-70
Mills	1976	75
Zelkowitz	1979	67
Cashman & Holt	1979	60-80

originais do usuário. A maior parte do esforço de validação tem sido realizada no final do período de desenvolvimento, quando o produto é testado.

A *Verificação* responde à pergunta, "Estou construindo corretamente o produto?"; a *Validação* por outro lado responde à pergunta "Estou construindo o produto correto?" [EVA87].

No processo de desenvolvimento de software todos os erros são erros humanos e, apesar da introdução de melhores métodos de desenvolvimento, melhores ferramentas de suporte e treinamento de pessoal, erros permanecem presentes nos diversos produtos de software produzidos e liberados [HOW87]; nesse contexto, a atividade de *teste* continuará a ter um papel importante no desenvolvimento de software [MAL91].

O *teste de software* é uma das atividades de Garantia de Qualidade de Software e envolve as atividades de *planejamento*, *projeto de casos de teste*, *execução de casos de teste* e *análise dos resultados obtidos*. Segundo Pressman [PRE87], a ausência de planejamento das atividades de desenvolvimento é uma das origens da *crise do software*.

Em todos os ramos da Engenharia, teste representa uma parte fundamental e é essencial no desenvolvimento de software [HOW87]. Teste pode ser considerado como a atividade final de verificação e validação de um projeto de software.

A Engenharia de Software busca projetar casos de teste que tenham uma alta probabilidade de encontrar a maioria dos defeitos com um mínimo de tempo e esforço.

A qualidade não é tangível. A finalidade do teste é tornar a qualidade visível ou melhor, dar subsídio para a avaliação da qualidade de um software.

Myers [MYE79] lista os três princípios de teste mais importantes. São eles:

- Teste é o processo de executar um programa com o intuito de encontrar erros.
- Um bom caso de teste é aquele que possui uma alta

probabilidade de encontrar um erro ainda não descoberto.

- Um teste bem sucedido é aquele que revela um erro ainda não descoberto.

Myers [MYE79] também ressalta que "Um teste não pode mostrar a ausência de defeitos; ele pode somente mostrar que defeitos estão presentes", e ainda " Um teste que não encontra erros não é um teste bem sucedido".

Em [ART88] são definidos quatro níveis de teste de software numa abordagem "bottom-up". São eles:

Teste de unidade - processa o teste individual de módulos de um projeto de software. Compara a função de um módulo com a sua especificação e visa identificar erros de lógica e de implementação.

Teste de Integração - integra todos os módulos para compor a estrutura do software e testa suas interfaces. Visa identificar erros de interface entre os módulos;

Teste de Sistema - verifica se todos os requisitos funcionais e de desempenho do sistema estão satisfeitos; e

Teste de Aceitação - processa testes com dados e ambiente reais do usuário.

O planejamento da atividade de teste deve fazer parte do planejamento global do sistema, e culmina em um *Plano de Teste*, que constitui um documento crucial no ciclo de vida de desenvolvimento de software. Nesse plano, são estimados recursos e são definidas estratégias, métodos e técnicas do teste caracterizando-se um critério de aceitação do software em desenvolvimento; a falta de tempo e de recursos, e a indisponibilidade de ferramentas adequadas são os principais problemas enfrentados pelas equipes de teste [MAL91].

Na elaboração de um plano de teste devem ser considerados os critérios de parada e de medição da qualidade dos testes.

Devido ao processo interativo dos testes de software, torna-se difícil determinar exatamente quando parar de testar. Arthur [ART88], Myers [MYE79], Musa e Ackerman [MUS89] e,

Schneidwind e Keller [SCH92] discutem algumas diretrizes para auxiliar a decisão de parada de testes.

Um dos resultados teóricos mais importantes na área de teste de programas é que não existe um procedimento de teste de propósito geral que possa ser usado para provar a correção de um programa (conforme citado em [MAL91]).

Existem muitas técnicas de teste bastante diferentes [COW88]. As técnicas *funcional* e *estrutural* são as predominantes.

A técnica funcional ou técnica *caixa-preta* usa como base para teste os requisitos definidos na especificação. O teste funcional envolve dois passos principais: a identificação das funções que espera-se que o software execute e a criação de dados de teste que verifiquem se as funções são realmente executadas pelo software. Nenhuma consideração é feita quanto a *como* o programa executa essas funções.

A técnica estrutural de teste ou de *caixa branca* é baseada na implementação do programa. Utiliza a estrutura de controle do mesmo para derivar os requisitos de teste.

As técnicas de teste funcional e estrutural são complementares pois cobrem classes distintas de erros [MYE79, PRE87]. A técnica estrutural de teste é mais adequada para o teste de unidade.

Vários critérios para a seleção de casos de teste estrutural foram estabelecidos. Primeiramente surgiram os critérios baseados unicamente no fluxo de controle de programas. O teste estrutural envolvendo a execução de um programa pode requerer:

- Todos os comandos em um programa devem ser executados pelo menos uma vez (*todos-nós*);
- Todas as transferências de controle em um programa devem ser executados pelo menos uma vez (*todos-ramos*), e
- Todos os caminhos em um programa devem ser executados pelo menos uma vez (*todos-caminhos*).

No contexto de teste estrutural, o teste exaustivo em que todos os caminhos possíveis em um programa fossem exercitados

seria o ideal. Entretanto, isso encontra pelo menos dois obstáculos.

O primeiro obstáculo é o grande número de caminhos possíveis (no caso de programas com "loops"). Isto pode levar a um número infinito de casos de teste inviabilizando sua aplicação por questões de custo e tempo. O segundo obstáculo é o número de *caminhos não executáveis* encontrado na maioria dos programas reais [MAL91]. Um caminho é não executável se não existe um conjunto de valores que possam ser atribuidos às variáveis de entrada do programa, que cause a execução desse caminho [FRA87]. Portanto, esses dois problemas constituem uma forte restrição à aplicação dos critérios baseados unicamente no fluxo de controle de programas.

Ntafos [NTA88] refere-se a vários critérios de testes estruturais baseados na análise de fluxo de dados. A análise de fluxo de dados estabelece que a ocorrência de uma variável pode ser de dois tipos: *definição e uso*. No contexto de teste de software, métodos e critérios de projeto de casos de teste baseados em análise de fluxo de dados requerem que as interações que envolvem definições de variáveis de programa e subsequentes referências a essas definições sejam testadas. Portanto, para a derivação de casos de teste, esses critérios baseiam-se nas associações entre a definição de uma variável e os seus possíveis subsequentes usos.

Especificamente, Maldonado, Chaim e Jino [MAL88a,MAL91], introduziram os Critérios Potenciais Usos, baseados no conceito *potencial uso*; os Critérios Potenciais Usos são critérios de teste estrutural, baseados na análise de fluxo de dados e consistem, fundamentalmente, em variações da família de critérios apresentada por Rapps e Weyuker [RAP82, RAP85]; são denominados: critérios *todos-potenciais-du-caminhos*, *todos-potenciais-usos* e *todos-potenciais-usos/du*. Associações são requeridas independentemente da ocorrência explicita de uma referência a uma determinada definição: se um uso pode existir —*um potencial uso*— a

potencial associação é requerida.

A aplicação dos critérios baseados em análise de fluxo de dados sem o apoio de uma ferramenta automatizada é limitada a programas muito simples [KOR85]. Um sistema automatizado possibilita ganhos de eficiência e a eliminação de erros na atividade de teste.

Um levantamento de ferramentas de teste estrutural de programas é apresentado no Capítulo 2. Dessas ferramentas, algumas foram desenvolvidas para tratarem programas escritos na linguagem FORTRAN.

1.2 OBJETIVO DO TRABALHO

A linguagem FORTRAN é considerada por Rice [RIC83] como sendo uma linguagem "standard" para computações numéricas e científicas. Em sua opinião FORTRAN é padrão por três razões:

a) a maioria dos cientistas e engenheiros que fazem cálculos com computador a conhece e pode usá-la no seu local de trabalho;
b) existe uma quantidade enorme de software disponível em FORTRAN; e
c) a linguagem FORTRAN permite que se faça a maioria das tarefas de cálculo numérico de uma maneira direta e confortável.

Grande é a quantidade de compiladores desenvolvidos para a linguagem FORTRAN e uma vasta gama de software para aplicações científicas e industriais codificado nela está disponível. Muitas são as aplicações em que essa linguagem é especificada.

O apoio automatizado de teste para programas escritos na linguagem FORTRAN deverá ser uma contribuição relevante para o desenvolvimento de sistemas com qualidade.

Este trabalho objetiva a especificação e a implementação da configuração para a linguagem de programação FORTRAN-77 [ANS78] de uma ferramenta de teste de software denominada POKE-TOOL (**PO**tential Uses **CRI**teria **TOOL** for Program Testing) [CHA91a, CHA91b]. Esta ferramenta apóia a aplicação dos critérios Potenciais Usos [MAL91]. A POKE-TOOL permite a análise de cobertura de um conjunto de casos de teste. Ela fornece ao usuário os caminhos necessários para satisfazer os critérios Potenciais Usos e é capaz de verificar se um conjunto de casos de teste fornecido pelo usuário executa todos os caminhos requeridos. A POKE-TOOL é uma ferramenta flexível e permite ser configurada para uma linguagem procedural de programação de interesse do usuário.

Para atingir o objetivo deste trabalho, as seguintes etapas são fundamentais:

- Estudo cuidadoso das características da linguagem FORTRAN-77;
- Instanciação dos modelos de implementação dos critérios Potenciais Usos;
- Seleção de um conjunto mínimo de programas que represente um núcleo de um "benchmark" que possibilitará a validação final da ferramenta;
- Implementação propriamente dita dos modelos;
- Teste dos programas de acordo de acordo com os critérios P.U.;
- Análise dos resultados obtidos.

Dada a importância da linguagem de programação FORTRAN-77 em ambientes científicos e industriais, e o desconhecimento da existência de uma ferramenta de teste baseado em fluxo de dados para FORTRAN-77 [HOR92], espera-se que este trabalho dê uma grande contribuição no sentido da produção de software de qualidade.

1.3 ORGANIZAÇÃO DO TRABALHO

Neste capítulo tratou-se da apresentação do contexto em que esta Dissertação se insere chamando a atenção para a importância de uma ferramenta de teste estrutural baseado em fluxo de dados que apoie testes em programas escritos na linguagem FORTRAN. Buscou-se também mostrar a tarefa de testar como sendo uma das etapas da Garantia de Qualidade de Software, e a importância da identificação de defeitos nas fases iniciais do desenvolvimento de software. Também foi descrito o objetivo que se espera alcançar com o desenvolvimento deste trabalho.

No Capítulo 2 são introduzidos conceitos básicos referentes aos critérios Potenciais Usos, a arquitetura e aspectos de configuração da POKE-TOOL. São descritas sucintamente, algumas ferramentas de teste estrutural de programas para uma avaliação comparativa com a POKE-TOOL.

No Capítulo 3 é mostrado o resultado de um estudo sobre a linguagem FORTRAN. Os aspectos históricos, características gerais e erros potenciais relacionados com a linguagem são descritos neste Capítulo.

No Capítulo 4 é apresentada a descrição dos modelos de implementação necessários para a aplicação dos critérios Potenciais Usos a unidades de programas codificadas na linguagem FORTRAN.

O Capítulo 5 é dedicado à descrição das principais etapas da implementação da configuração da ferramenta POKE-TOOL para a linguagem FORTRAN-77. Apresenta ainda o resultado da aplicação de um conjunto de casos de teste que possibilita a comprovação da operacionalidade do protótipo desenvolvido. Aliado ao Exemplo apresentado no Apêndice E, constitui uma constatação de alcance dos objetivos propostos.

O Capítulo 6 traz uma síntese dos problemas encontrados e dos resultados obtidos; tambem sugere alguns trabalhos futuros

decorrentes desta dissertação.

Esta dissertação apresenta ainda, 6 Apêndices. O Apêndice A contém uma síntese da descrição da Linguagem Intermediária (LI). A LI permite a abstração do fluxo de controle de programas [CAR91].

O Apêndice B é o que descreve os Modelos de Implementação dos Critérios Potenciais Usos que são instanciados para a linguagem alvo da configuração da POKE-TOOL.

A Tabela de Transições Léxicas e a Tabela de Palavras-Chave utilizadas para a configuração do Analisador Léxico para o FORTRAN-77 são mostradas no Apêndice C.

O Apêndice D apresenta a descrição sintática da linguagem FORTRAN-77 da forma como é utilizada para a configuração do Analisador Sintático da POKE-TOOL.

No Apêndice E tem-se um exemplo completo da aplicação da presente versão da POKE-TOOL que permite a visualização de sua operacionalidade. O exemplo escolhido trata do cálculo de imposto de renda devido para um conjunto de salários, segundo os critérios usuais na Inglaterra. Este exemplo é representativo, pois possui algumas das estruturas de controle mais comumente encontradas em programas FORTRAN-77.

O último Apêndice, o F traz alguns exemplos, colhidos ao acaso, de rotinas desenvolvidas para o Módulo LI (Analisadores Léxico e Sintático) e para o Módulo Pokernel que é o responsável pela identificação das definições de variáveis e instrumentação do código fonte.

CAPÍTULO

2

UMA FERRAMENTA AUTOMATIZADA PARA TESTE ESTRUTURAL DE PROGRAMAS BASEADO EM ANÁLISE DE FLUXO DE DADOS

Neste capítulo é apresentada a terminologia e alguns conceitos básicos utilizados no desenvolvimento deste trabalho. São introduzidos os critérios Potenciais Usos [MAL88a,MAL88b, MAL89,MAL91] que são critérios de teste estrutural de software baseados em análise de fluxo de dados. A arquitetura e os aspectos de configuração de uma ferramenta de testes, a POKETOOL [CHA91a,CHA91b], também são discutidos. Antes, para dar uma visão global e histórica da automatização do teste de programa, são apresentadas algumas ferramentas de teste estrutural de programas encontradas na literatura.

2.1 FERRAMENTAS AUTOMATIZADAS

Segundo Chaim [CHA91b], o uso de uma ferramenta de software para auxílio ao teste de programas pode ser vinculado a um critério de teste de duas maneiras. Na primeira a ferramenta de software pode utilizar o critério de teste como um guia para a *geração de casos de teste* que satisfaçam o critério; na outra, há a possibilidade de utilização do critério para a *análise de cobertura* de um conjunto de casos de teste, isto é, a ferramenta verifica se os casos de teste aplicados preencheram os requisitos de teste do critério.

Métricas de Cobertura de teste são uma alternativa de medição de qualidade dos testes [HAL91]. Métricas de cobertura de teste são estabelecidas a partir do conhecimento de detalhes de implementação e o objetivo é garantir que os testes "cobriram" todos os elementos requeridos; usualmente, essas métricas estão fortemente relacionadas com critérios de teste estrutural.

As ferramentas de teste estrutural de programas fazem, em quase sua totalidade, análise de cobertura, segundo algum critério de teste selecionado, de um conjunto de casos de teste. Apesar de não auxiliarem diretamente na determinação dos dados de entrada de um programa, necessários para a execução de caminhos específicos, a maioria das ferramentas de teste apresenta ao usuário quais os requisitos de teste exigidos para que os critérios sejam satisfeitos. Assim, de certa forma, orientam e auxiliam os usuários na elaboração dos casos de teste. A seguir são descritos as principais ferramentas de teste disponíveis na literatura.

2.1.1 DAVE

É uma das mais antigas ferramentas conhecidas de teste de programas FORTRAN [OST76]. É uma ferramenta que não necessita da execução do programa, pois faz apenas uma análise do código fonte.

A DAVE utiliza técnicas de análise de fluxo de dados e grafos de programa. Identifica muitas classes de erros de programa e anomalias de fluxo de dados.

2.1.2 Ferramenta de Herman

A ferramenta de Herman [HER76] suporta a aplicação do primeiro critério baseado em análise de fluxo de dados; tal critério é semelhante ao critério *todos-c-usos* [RAP82,RAP85] proposto posteriormente. A ferramenta realiza a análise de cobertura desse critério para programas escritos em linguagem FORTRAN.

2.1.3 RXVP80

RXVP80 [DEU82] é um sistema que faz basicamente análise de cobertura do teste de ramos em programas escritos em FORTRAN. Além de apoiar o teste dinâmico, via instrumentação, essa ferramenta fornece ainda: análise estática do código fonte com a geração do grafo de chamada dos módulos (unidades) constituintes do sistema em teste; geração do grafo de fluxo de controle dos módulos; verificação de tipos nas chamadas de procedimentos e a geração de relatórios que fornecem referências cruzadas entre código fonte, variáveis e módulos.

É uma ferramenta comercial, distribuída pela General Research Corporation, Santa Barbara, California, EUA.

2.1.4 TCAT

A ferramenta TCAT (*Test-Coverage Analysis Tool*) [REI90] realiza o teste de unidades segundo o critério todos os ramos; produz estatísticas para o último caso de teste ou para todos os casos de teste. Existem versões disponíveis para Ada, C, COBOL, FORTRAN-77 e Pascal. É uma ferramenta comercial fornecida por Software Research Corporation, San Francisco, Califórnia, EUA.

2.1.5 ASSET

ASSET (*A System to Select and Evaluate Tests*) [FRA87] apóia basicamente um conjunto de critérios baseados em análise de fluxo de dados, introduzidos por Rapps e Weyuker [RAP82,RAP85], em programas escritos em linguagem Pascal; foi desenvolvida na New York University.

2.1.6 PROTESTE

PROTESTE [PRI90] tem como objetivo um ambiente completo para suporte ao teste estrutural de programas, incluindo tanto critérios baseados unicamente no fluxo de controle (por exemplo, critérios todos os nós e todos os ramos) como critérios baseados em análise de fluxo de dados (por exemplo, critérios de Rapps e Weyuker e Potenciais Usos); apóia o teste de programas escritos em linguagem Pascal. PROTESTE é um protótipo desenvolvido na Universidade Federal do Rio Grande do Sul.

2.1.7 ATAC

ATAC (*Automatic Test Analysis for C*) é uma ferramenta desenvolvida no Bell Communications Research, e suporta o teste de unidades escritas em linguagem C; realiza análise de cobertura através dos critérios todos os nós, todos os ramos e todos os usos. Ela informa os elementos requeridos pelo critério mas não executados na aplicação dos casos de teste. É uma ferramenta de domínio público, mas de uso restrito a universidades [HOR92].

2.1.8 LOGISCOPE

LOGISCOPE [FOR92] é uma ferramenta que analisa complexidade e cobertura de teste. Ela gera grafos de estrutura e fornece métricas em várias linguagens. Ela produz grafos que mostram o fluxo de controle de programas e a sua estrutura de uma forma textual. A LOGISCOPE, segundo os autores, identifica as condições necessárias para testar os caminhos dentro do programa e faz

análise de cobertura; os autores não deixam explícito o critério utilizado para este fim. A ferramenta trata programas escritos nas linguagens: ADA, C, COBOL, FORTRAN, MODULA-2, PASCAL, PL1. É uma ferramenta comercial fornecida pela Verilog, Toulouse, França.

2.1.9 POKE-TOOL

A POKE-TOOL [CHA91b] é uma ferramenta que apóia a aplicação dos critérios Potenciais Usos [MAL91]. Essa ferramenta já está configurada para testar programas escritos nas linguagens "C" e COBOL. Ela é uma ferramenta interativa orientada para sessão de trabalho. Executa a análise estática do código fonte, preparação para o teste, submissão de casos de teste e gerenciamento dos resultados dos testes. A POKE-TOOL é um protótipo disponível para ambientes do tipo PC sob o sistema operacional DOS. Foi desenvolvida na Faculdade de Engenharia Elétrica da Universidade Estadual de Campinas. Esta ferramenta por ser alvo do presente trabalho é descrita em maiores detalhes na Seção 2.4.

A Tabela 2.1 apresenta uma comparação entre a POKE-TOOL e algumas ferramentas de teste encontradas na literatura. As informações apresentadas são as disponíveis nos artigos consultados e as características comparadas foram as utilizadas em [HOR92].

2.2 CONCEITOS BÁSICOS

A terminologia e os conceitos apresentados nesta seção são essencialmente uma síntese dos trabalhos de Weyuker [WEY88] e Maldonado [MAL91].

Seja um grafo de fluxo de controle $G = (N, A, s)$ onde N representa o conjunto de nós, A representa o conjunto de arcos, e

Tabela 2.1 Comparação entre Ferramentas de Teste

	COMERCIAIS			PROTÓTIPO DOMÍNIO PÚBLICO
CARACTERÍSTICAS	RXUP80	TCAT	LOGISCOPE	ATAC
LINGUAGENS SUPORTADAS	FORTRAN	FORTRAN-77, ADA, C, COBOL, PASCAL	ADA, C, COBOL, PASCAL, ADABAS2, FORTRAN	C
GERAÇÃO DE CASOS DE TESTE	NÃO	NÃO	NÃO	NÃO
EDIÇÃO DE CASOS DE TESTE	NÃO	--	NÃO	NÃO
AVALIAÇÃO AUTOMÁTICA DA CORREÇÃO DE PROGRAMAS	NÃO	NÃO	NÃO	NÃO
GRAVAÇÃO DE CAMINHOS NÃO EXECUTÁVEIS	NÃO	--	NÃO	NÃO
ELIMINAÇÃO DE CASOS DE TESTE REDUNDANTES	NÃO	--	--	SIM
FASE DA ATIVIDADE DE TESTE SUPORTADA	UNIDADE SISTEMA	UNIDADE SISTEMA	UNIDADE	UNIDADE SISTEMA
INTERFACE	LINHA DE COMANDO	ORIENT. JANELAS	GRAFICA / MENU	LINHA DE COMANDO
EXECUÇÃO AUTOMÁTICA DE PROGRAMAS	NÃO	--	--	NÃO

	PROTÓTIPOS			
CARACTERÍSTICAS	HERMAN	ASSET	PROTESTE	POKE-TOOL
LINGUAGENS SUPORTADAS	FORTRAN	PASCAL	PASCAL, C	C, COBOL
GERAÇÃO DE CASOS DE TESTE	NÃO	NÃO	SIM	NÃO
EDIÇÃO DE CASOS DE TESTE	NÃO	NÃO	NÃO	NÃO
AVALIAÇÃO AUTOMÁTICA DA CORREÇÃO DE PROGRAMAS	NÃO	NÃO	NÃO	NÃO
GRAVAÇÃO DE CAMINHOS NÃO EXECUTÁVEIS	NÃO	NÃO	NÃO	SIM
ELIMINAÇÃO DE CASOS DE TESTE REDUNDANTES	NÃO	NÃO	NÃO	NÃO
FASE DA ATIVIDADE DE TESTE SUPORTADA	UNIDADE	UNIDADE	UNIDADE	UNIDADE
INTERFACE	LINHA DE COMANDO	MENU	GRAFICA / MENU	MENU
EXECUÇÃO AUTOMÁTICA DE PROGRAMAS	NÃO	NÃO	SIM	SIM

s o nó inicial. Um *caminho* é uma sequência finita de nós $(n_1, n_2, \ldots, n_k)$, $k \geq 2$, tal que existe um arco do nó n_i para n_{i+1} para $i = 1,2,\ldots,k-1$. Um caminho é um *caminho simples* se todos os nós que compõem esse caminho, exceto possivelmente o primeiro e o último, são distintos. Um caminho é dito *caminho livre de laço* se todos os nós que o compõem são distintos. Um *caminho completo* é um caminho onde o primeiro nó é o nó inicial e o último nó é o nó de saída do grafo G.

Os critérios Potenciais Usos são critérios de teste baseados em análise de fluxo de dados [HER76,RAP82,RAP85,FRA88,MAL88]. Um *modelo de fluxo de dados* é usado para a definição das ocorrências de variáveis em um programa. As ocorrências de uma variável em um programa podem ser uma *definição*, um *uso* ou uma *indefinição*.

Em geral, uma ocorrência de variável é uma definição se seu valor pode ser alterado em um comando de atribuição, de entrada ou de chamada de procedimento. A ocorrência de uma variável é um uso quando a referência a essa variável não a estiver definindo: um uso computacional (c-*uso*) afeta diretamente a computação sendo realizada ou permite que o resultado de uma definição anterior possa ser observado; um uso predicativo (p-*uso*) afeta diretamente o fluxo de controle de um programa. Uma variável está indefinida quando, ou não se tem acesso ao seu valor, ou sua localização deixa de estar amarrada na memória.

O grafo de fluxo de controle estendido pela associação a cada nó i do conjunto de variáveis definidas em i ($def_g(i)$) é denominado *grafo def*.

Os elementos do fluxo de execução de um programa são caracterizados pelos tipos de ocorrência das variáveis.

Um caminho $(i, n_1, \ldots, n_m, j)$, $m \geq 0$, que não contenha definição de variável nos nós $n_1, \ldots, n_m$ é chamado *caminho livre de definição* com respeito a $(c.r.a)$ x do nó i ao nó j e do nó i ao arco (n_m, j).

Um caminho livre de definição $(n_1, n_2, \ldots, n_j, n_k)$ onde o caminho $(n_1, n_2, \ldots, n_j)$ é um caminho livre de laço e n_1 tem uma

definição de x, é denominado *potencial-du-caminho (c.r.a) x*.

Para a definição dos critérios Potenciais Usos é necessária a introdução dos seguintes conceitos: $pdcu(x,i)$ = { nós j | existe um caminho livre de definição c.r.a x de i a j }; $pdpu(x,i)$ = { arcos (j,k) | existe um caminho livre de definição c.r.a x de i a (j,k)}; uma *potencial associação definição-c-uso* é a tripla $[i,j,x]$ onde $x \in defg(i)$ e $j \in pdcu\ (x,i)$; e uma *potencial associação definição-p-uso* é a tripla $[i,(j,k),x]$ onde $x \in defg(i)$ e $(j,k) \in pdpu\ (x,i)$. Uma *potencial associação* é definida como uma potencial associação definição-c-uso, uma potencial associação definição-p-uso ou um potencial-du-caminho.

Um caminho $\pi_1 = (i_1, i_2, \ldots, i_k)$ é dito estar incluído em um conjunto Π de caminhos se Π contém um caminho $\pi_2 = (n_1, n_2 \ldots, n_m)$ tal que $i_1 = n_j$, $i_2 = n_{j+1}, \ldots, i_k = n_{j+k-1}$, para algum j, $1 \leq j \leq m - k + 1$. Então, π_1 está incluído em π_2 ou que π_1 é um sub-caminho de π_2.

Um caminho completo π cobre uma potencial associação definição-c-uso $[i,j,x]$ (respectivamente, uma potencial associação definição-p-uso $[i,(j,k),x]$) se ele inclui um caminho livre de definição c.r.a x de i para j (respectivamente, de i para (j,k)). π cobre um potencial-du-caminho π_1 se π_1 está incluído em π. Um conjunto Π de caminhos cobre uma potencial associação se algum elemento do conjunto cobri-la.

2.3 CRITÉRIOS POTENCIAIS USOS

Os critérios Potenciais Usos — *todos-potenciais-usos*, *todos-potenciais-usos/du* e *todos-potenciais-du-caminhos* — são critérios baseados na análise de fluxo de dados e requerem, basicamente que caminhos livres de definição, em relação a qualquer nó i que

possua definição de variável e qualquer variável x definida em i, sejam executados independentemente de ocorrer uso dessa variável nesses caminhos [MAL88a,MAL88b,MAL89,MAL91]. Neste sentido, pode-se verificar, por exemplo, que o valor de x não foi alterado nesses caminhos (possivelmente devido a efeitos colaterais) ganhando-se, desta forma, maior confiança de que a computação correta é realizada; isto está de acordo com uma filosofia discutida por Myers [MYE79]: um erro está claramente presente se um programa não faz o que supõe-se que ele faça, mas erros também estão presentes se um programa faz o que supõe-se que não faça. Ainda, a aplicação dos critérios Potenciais Usos torna mais fácil a detecção de erros causados por dependências de fluxo de dados ausentes [MAL91] originadas, por exemplo, por omissão de uso de variáveis.

- **Critério Todos-potenciais-usos**: Um conjunto de caminhos Π satisfaz o critério todos-potenciais-usos se, para todo nó i e para toda variável x para a qual existe uma definição em i, Π inclui pelo menos um caminho livre de definição c.r.a x do nó i para todo nó e para todo arco possível de ser alcançado a partir de i.
- **Critério Todos-potenciais-usos/du**: Um conjunto de caminhos Π satisfaz o critério todos-potenciais-usos/du se, para todo nó i e para toda variável x para a qual existe uma definição em i, Π inclui pelo menos um potencial-du-caminho c.r.a x do nó i para todo nó e para todo arco possível de ser alcançado a partir de i.
- **Critério Todos-potenciais-du-caminhos**: Um conjunto de caminhos Π satisfaz o critério todos-potenciais-du-caminhos se, para todo nó i e para toda variável x para a qual existe uma definição em i, Π inclui todos os potenciais-du-caminhos c.r.a x em relação ao nó i.

2.4 ARQUITETURA DA FERRAMENTA POKE-TOOL

Nesta seção é discutida a arquitetura proposta por Chaim [MAL89] para a ferramenta de teste POKE-TOOL e são abordadas as funções suportadas pela ferramenta.

Segundo Deutsch [DEU82], existem cinco funções básicas que devem ser realizadas por uma ferramenta deste tipo:

a) análise do código fonte e criação de uma base de dados;
b) geração de relatórios baseada em análise estática do código fonte, que indique problemas potenciais ou existentes e identifique a estrutura de dados e de controle do software;
c) instrumentação do código fonte permitindo a coleta de dados relativos à execução de casos de teste;
d) análise dos resultados dos testes e geração de relatórios;
e) geração de relatórios de apoio ao teste para auxiliar na organização das atividades de teste e para derivar conjuntos de entrada (dados de teste) para testes específicos.

Com a ferramenta POKE-TOOL, tenciona-se apoiar todas essas funções da atividade de teste.

A Figura 2.1 mostra a arquitetura geral da POKE-TOOL que terá como entrada o programa a ser testado, a seleção do critério a ser utilizado e um conjunto de casos de teste. Na hipótese do conjunto de casos de teste fornecido ser vazio, a POKE-TOOL fornecerá um conjunto de caminhos necessários para satisfazer o critério selecionado, orientando desta forma a própria seleção de casos de teste. Se o conjunto de casos de teste fornecido não for vazio, será produzida uma relação de caminhos requeridos pelo critério mas ainda não executados. Pretende-se que esta ferramenta apóie o teste de programas implementados na linguagem FORTRAN.

São nove as funções básicas que constituem a POKE-TOOL como

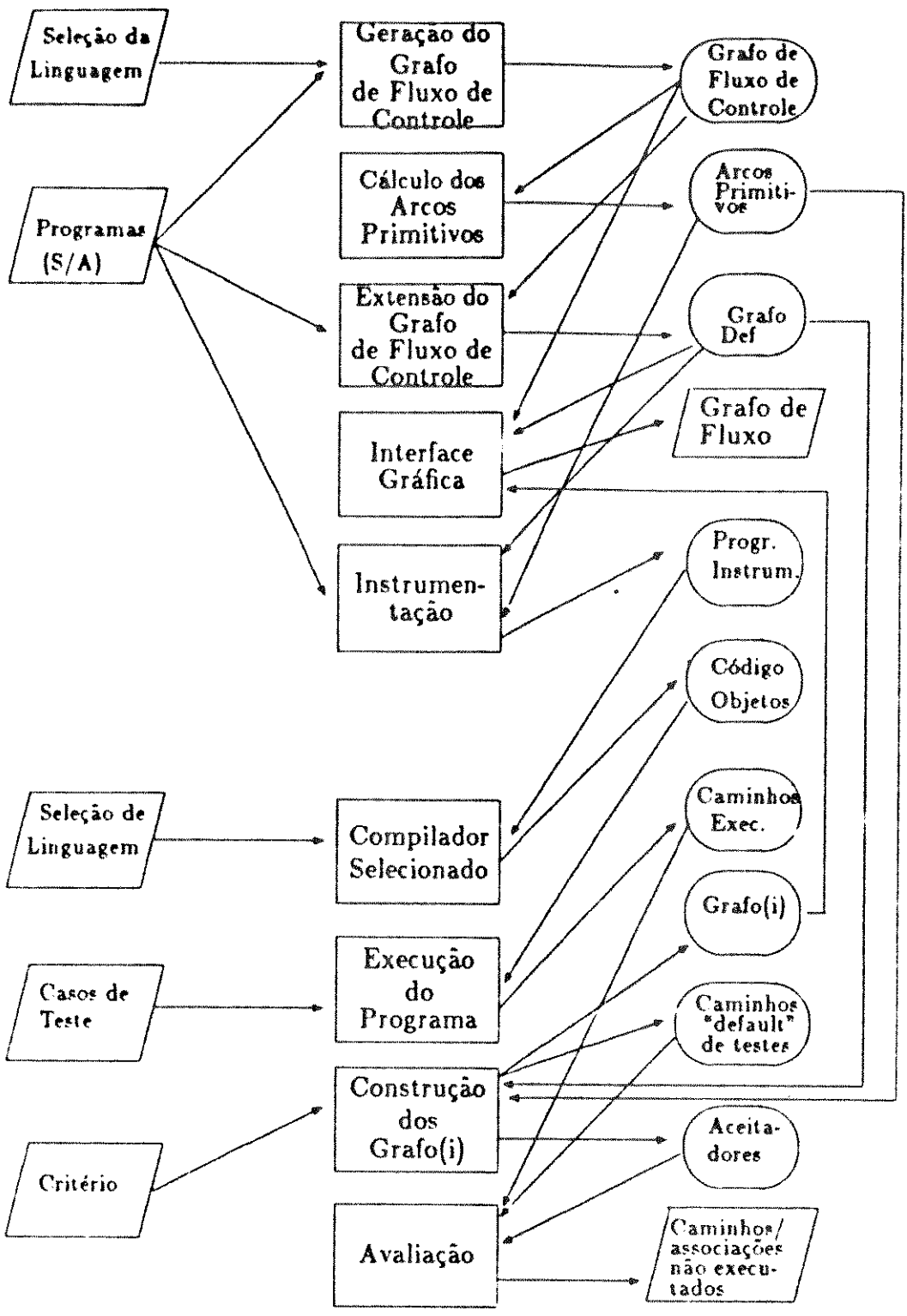


Figura 2.1: Arquitetura da Ferramenta POKE-TOOL

mostradas na Figura 2.1. e descritas abaixo:

GRAFO DE FLUXO DE CONTROLE: Esta função produz o grafo de fluxo de controle do programa fonte a ser testado. Para a produção do grafo de fluxo de controle para várias linguagens utiliza-se uma tabela descritora da sintaxe da linguagem de implementação que é utilizada na análise sintática do programa fonte; é gerada uma versão em *Linguagem Intermediária* (LI) [CAR91] do programa fonte, de onde se extrai o fluxo de controle. A definição da correspondência entre os comandos da linguagem fonte e os comandos da linguagem intermediária faz parte do processo de configuração.

CÁLCULO DOS ARCOS PRIMITIVOS: Calcula os arcos primitivos [CHU87] do grafo de fluxo de controle. O conjunto de arcos primitivos servirá de base para a instrumentação do programa fonte e para a construção dos autômatos utilizados na avaliação de adequação de conjuntos de casos de teste.

EXTENSÃO DO GRAFO DE FLUXO DE CONTROLE: Associa a cada nó i do grafo de fluxo de controle o conjunto de variáveis definidas no nó i, produzindo um grafo denominado *grafo def*, de acordo com o modelo de fluxo de dados pré-estabelecido.

INTERFACE GRÁFICA: Apresenta os grafos de fluxo de controle para o usuário.

INSTRUMENTAÇÃO: Insere comandos de escrita (pontas de prova) no programa fonte para cada um dos nós da unidade em teste, gerando uma nova versão do programa — versão instrumentada — que produz um "trace" da execução dos casos de teste.

COMPILADOR SELECIONADO: Consiste de um compilador da linguagem fonte na qual o programa em teste foi implementado.

EXECUÇÃO DO PROGRAMA: Controla a execução do programa em teste — unidade em teste — produzindo um conjunto dos caminhos executados pelos casos de teste fornecidos. Adicionalmente, produz um registro de pares <entrada,saída> associados aos

respectivos caminhos executados. Os caminhos executados são descritos através de seqüências de números dos nós do grafo de fluxo de controle do programa fonte.

CONSTRUÇÃO DOS GRAFO(i): Com base no grafo def e no conjunto de arcos primitivos, esta função constrói os grafo(i) [MAL88]. Adicionalmente, fornece um conjunto de caminhos e associações requeridos para satisfazer os critérios Potenciais Usos. Ainda são fornecidos os descritores de caminhos e associações, em termos dos arcos primitivos, a serem utilizados na avaliação de um conjunto de casos de teste qualquer.

AVALIAÇÃO: Esta função verifica se o conjunto de caminhos ou associações executados satisfaz o critério selecionado. Em caso negativo, produz uma relação de caminhos (associações) requeridos pelo critério e não executados e uma medida percentual da cobertura atingida pelo conjunto de casos de teste, ou seja, uma relação entre os caminhos (associações) executados e o número de caminhos (associações) requeridos.

De forma resumida, pode-se dizer que a partir do programa fonte, a POKE-TOOL determina o grafo de fluxo de controle. A seguir, este grafo é estendido incorporando-se informações de fluxo de dados, obtendo-se o grafo def; o conjunto de arcos primitivos é calculado, obtendo-se o grafo com redução de herdeiros para fluxo de dados [MAL91].

Os arcos primitivos são utilizados para construir descritores (expressões regulares) dos caminhos (associações) requeridos. A instrumentação auxilia na determinação dos caminhos efetivamente executados pelo conjunto de casos de teste fornecido. Os descritores são utilizados para verificar se o critério selecionado foi satisfeito; esta verificação dá-se pela implementação de aceitadores de expressões regulares. A partir do grafo def e do conjunto de arcos primitivos constroem-se os grafo(i), caracterizando-se os arcos primitivos em cada um desses grafo(i). Em seguida, os descritores dos caminhos (associações)

requeridos são elaborados.

Na hipótese da POKE-TOOL ser utilizada na avaliação da adequação de um conjunto de casos de teste, são determinados os caminhos (associações) efetivamente executados e verifica-se se os aceitadores correspondentes aos descritores dos caminhos (associações) requeridos estão no estado final (o critério foi satisfeito); caso contrário, é fornecida ao usuário uma lista de caminhos requeridos pelo critério e não executados pelo conjunto de casos de teste. No caso da POKE-TOOL ser utilizada para auxiliar na geração de casos de teste, o conjunto "default" de caminhos requeridos pelo critério (obtido durante a construção dos grafo(i)) é fornecido.

Por suas características, a ferramenta POKE-TOOL pode constituir-se também num suporte extremamente útil tanto às atividades de depuração como às de manutenção de programas.

2.5 ASPECTOS DE CONFIGURAÇÃO DA FERRAMENTA POKE-TOOL

Para a configuração da POKE-TOOL visando a aplicação dos critérios Potenciais Usos a unidades de programas, são necessárias, a caracterização do fluxo de execução da unidade e a instrumentação do código fonte para a determinação dos caminhos efetivamente executados durante a aplicação dos casos de teste. Adicionalmente, informações de fluxo de dados são agregadas às informações de fluxo de controle da unidade de programa para a caracterização dos componentes requeridos pelo critério de teste.

A arquitetura da POKE-TOOL estabelece uma distinção entre as tarefas dependentes do código fonte e as demais, permitindo que tais tarefas sejam isoladas em módulos distintos dos módulos que realizam tarefas independentes da linguagem [CHA91b].

As funções *Extensão do Grafo de Fluxo de Controle* e *Instrumentação* são implementadas pelo módulo **POKERNEL** da POKE-TOOL. Assim, esse módulo determina quais variáveis são definidas em um dado nó e insere os comandos para a definição de pontas de prova, declaração de variáveis auxiliares e comandos para a manipulação de arquivos. É, portanto, dependente da linguagem alvo da configuração.

A função *Grafo de Fluxo de Controle* é subdividida em dois módulos: o primeiro — o módulo **LI** — realiza o mapeamento da unidade em teste para a unidade em teste em Lingugem Intermediária; o segundo — o módulo **CHANOMAT** — realiza a geração do grafo de fluxo de controle a partir da unidade em LI. Aparentemente essa função seria extremamente dependente da linguagem de programação da unidade em teste; entretanto, essa dependência se restringe ao mapeamento para a LI, pois a geração do grafo de fluxo de controle é genérica.

O módulo **LI** da POKE-TOOL é dependente da linguagem fonte, pois é ele quem faz a tradução da unidade em teste para a unidade em teste em LI.

A seguir, apresentamos conforme [LEI92], uma relação das tarefas a serem realizadas por um usuário configurador da POKE-TOOL para obter uma configuração da ferramenta para uma nova linguagem. Os passos **A**, **B** e **D** são aplicados tanto para a configuração do módulo LI, quanto para a configuração do módulo POKERNEL; o passo **C** está associado somente ao módulo POKERNEL.

A. Configurar o analisador léxico da POKE-TOOL. Essa tarefa demanda que o usuário conheça bem os aspectos léxicos da linguagem alvo e está dividida em algumas subtarefas:

A.1. Desenvolver um autômato finito [SET81] que realize a análise léxica da linguagem e especificar a tabela de palavras reservadas da linguagem.

A.2. Identificar as ações semânticas associadas às transições do autômato finito.

A.3. Transcrever esse autômato para a notação aceita pelo analisador genérico da POKE-TOOL.

A.4. Complementar o analisador léxico através de rotinas ("ações semânticas") que realizem a separação dos átomos da linguagem fonte, colocando-os nas estruturas de dados da POKE-TOOL utilizadas para armazená-los.

A.5. Depurar o analisador léxico conjuntamente com as ações semânticas desenvolvidas.

B. Configurar o analisador sintático da POKE-TOOL. Essa tarefa demanda que o usuário conheça bem a sintaxe e a semântica da linguagem alvo e está dividida em algumas subtarefas:

B.1. Descrever a gramática da linguagem fonte na forma de *grafos sintáticos* [WIR76].

B.2. Identificar os pontos onde devem ser ativadas as rotinas semânticas nos grafos sintáticos.

B.3. Transcrever os grafos sintáticos para a notação aceita pela POKE-TOOL.

B.4. Complementar o analisador sintático com rotinas semânticas; no caso do módulo **LI**, essas rotinas identificam as equivalências entre a linguagem alvo e a LI e geram os átomos da LI que irão compor a versão em linguagem intermediária da unidade em teste; no caso do módulo POKERNEL, essas rotinas realizam a identificação das variáveis definidas gerando, portanto, o grafo def.

B.5. Depurar o analisador sintático conjuntamente com as rotinas desenvolvidas.

C. Ajustar os procedimentos que inserem código fonte na nova versão da unidade em teste com as instruções da nova linguagem de programação aceita pela POKE-TOOL.

D. Compilar e ligar os arquivos que constituem os analisadores léxico e sintático genéricos e as ações e rotinas semânticas com os demais arquivos que constituem a POKE-TOOL; no caso do módulo POKERNEL, também são agregados os procedimentos de

inserção de código fonte.

Para se alcançar uma ampliação das aplicações da ferramenta POKE-TOOL, buscou-se dotá-la da característica de multilinguagem isto é, tornar factível a sua configuração para várias linguagens procedimentais de programação. Isto levou à escolha de uma forma de representação comum para os programas codificados nas diversas linguagens abrangidas. Após estudo das linguagens ALGOL-60, C, COBOL, FORTRAN, MODULA-2, e PASCAL foi definida uma Linguagem Intermediária denominada LI [CAR91]. No Apêndice A é apresentado um subconjunto da LI, que foi utilizado para o mapeamento da linguagem FORTRAN-77.

No Apêndice B são descritos os modelos de implementação dos critérios Potenciais Usos. A instanciação para a Linguagem FORTRAN-77 desses modelos é descrita no Capítulo 4.

2.6. CONSIDERAÇÕES FINAIS

Foram apresentados a terminologia e os conceitos que nortearam o desenvolvimento deste trabalho.

A arquitetura e os aspectos de configuração da ferramenta POKE-TOOL também foram descritos. A POKE-TOOL apóia o teste estrutural de programas baseado em análise de fluxo de dados, segundo os critérios Potenciais Usos, dos quais foram caracterizados os conceitos básicos.

Para salientar a importância da POKE-TOOL no contexto de ferramentas de teste estrutural de programas, foram introduzidas, também, características de outras ferramentas encontradas na literatura que contribuem para uma análise comparativa.

Quando da especificação dos modelos de implementação da POKE-TOOL, foi identificada a necessidade de se dotar a linguagem

intermediária LI de novos átomos que foram introduzidos neste capítulo.

Antes de se apresentar os aspectos de implementação da configuração da POKE-TOOL para FORTRAN-77, é mostrado no Capítulo 3 um levantamento feito sobre algumas características da linguagem e alguns erros passíveis de serem praticados por aqueles que programam em FORTRAN.

CAPÍTULO

3

A LINGUAGEM FORTRAN

3.1 HISTÓRICO

FORTRAN foi a primeira linguagem de programação de alto nível desenvolvida para uso não acadêmico [MAR85] . Sua geração se deu quando um grupo na IBM em Nova York, chefiado por John Backus, estava investigando a possibilidade de um tradutor de linguagem simbólica [HOR84] . A grande questão naquela época não era se tal linguagem poderia ser projetada e compilada, mas sim se seria suficientemente eficiente. Essa preocupação teve um grande impacto no projeto e, hoje, encontram-se como resultado muitos compiladores FORTRAN que geram código bastante eficiente. O FORTRAN 0 (FORmula TRANslation) foi primeiro projetado a partir de 1954, acompanhado de dois anos e meio de construção do compilador (total aproximado de 25 homens/ano [COL69]) . Ele não era o FORTRAN que conhecemos hoje. Não existia o comando FORMAT e nenhuma função definida pelo usuário. Variáveis de quando muito dois caracteres eram permitidas, o que era uma inovação para a época.

Quando a primeira versão do compilador FORTRAN foi para os testes de campo, o sistema mostrou muitos erros. Houve um longo período durante o qual muitos duvidaram que um dia esse sistema

pudesse trabalhar de uma maneira prática [COL69]. Consistia de mais de 25.000 instruções em linguagem de máquina, e representava uma ordem de complexidade nunca tentada anteriormente em programas de computador.

Em novembro de 1954, John Backus e seu grupo produziram o relatório intitulado " The IBM Mathematical FORmula TRANslating System : FORTRAN ". Esse documento dizia, entre outras coisas, que a nova linguagem poderia eliminar erros de programação e o processo de depuração. Baseado nesta premissa, o primeiro compilador FORTRAN incluia um pouco de verificação de erros de sintaxe [SEB89] .

As potencialmente grandes vantagens oferecidas pelo FORTRAN foram alcançadas devido à persistência de diversos grupos que usaram a linguagem na solução de problemas do cotidiano [COL69] .

Um compilador FORTRAN trabalhando de uma forma prática foi um grande marco. Mas antes de representar um fim, ele era apenas o início do desenvolvimento de sistemas de programação e de linguagens procedimentais. Desde a sua criação, o FORTRAN tem alcançado um grande número de versões representando sucessivas melhorias derivadas da extensiva experiência em seu uso. Essas versões permitiram a ampliação da sua aplicação. Avanços tecnológicos em linguagens de programação e investimentos dos fabricantes, entre outros, são fatores que contribuiram para o aparecimento dessas novas versões.

A linguagem FORTRAN foi originalmente desenvolvida para ser aplicada na solucão de problemas científicos e de engenharia e tem sido utilizada com êxito para tal fim por vários anos.

FORTRAN foi a primeira linguagem de programação a ser adotada como padrão nacional nos EUA, quando uma versão foi aprovada pela ASA (que se tornou mais tarde ANSI). O FORTRAN foi padronizado primeiramente em 1966 (FORTRAN IV) e, em 1978, uma nova versão revisada do padrão era liberada. Essa nova versão, a *ANSI X3.9 1978 FORTRAN-77* [ANS78], mantém muito das características do

FORTRAN IV e adiciona facilidades para o tratamento de cadeias de caracteres, matrizes, comandos para tratamento de laços lógicos, e um comando IF com uma cláusula ELSE opcional que não compunham a versão anterior. O objetivo do comitê que elaborou o padrão foi preservar as características positivas da linguagem, tais como compilação e execução de código eficientes [HOR87] .

FORTRAN 8X é o nome da próxima versão do FORTRAN [MET82] ,que ainda não é um padrão aprovado, embora se espera que venha a tornar-se o padrão FORTRAN (ANSI) dos anos 90. As novas características do FORTRAN 8X incluem "arrays" alocados dinâmicamente, muitas operações sobre "arrays", tipos de dados definidos pelo usuário, módulos, e comando CASE [SEB89] .

Já existem disponíveis versões de compiladores segundo o padrão FORTRAN 90 ISO (DIS 1539:1991). A linguagem FORTRAN 90 apresenta características que a torna comparável às linguagens como ADA e C++; ela oferece: novas construções de controle, tipos derivados, apontadores, alocação dinâmica de memória, blocos de interface, argumentos opcionais para procedimentos; procedimentos recursivos e operações sobre matrizes entre outras.

O único antigo rival do FORTRAN como linguagem de alto nível era o Algol-60. As formas como as duas linguagens foram projetadas mostram contraste, assim como seus níveis de aceitação e uso. Em vários aspectos o Algol mostrava-se superior tanto em seu projeto quanto em seu objetivo. Entretanto, o FORTRAN foi a linguagem escolhida pela grande maioria dos programadores ao escreverem os seus programas aplicativos, apesar de o Algol 60 ser o preferido nos círculos acadêmicos [MAR85].

Muitas outras linguagens foram criadas após o FORTRAN e, obviamente, muitas delas dizem-se superiores. Apesar disso, o uso do FORTRAN persiste. Há diversas razões para isto. Uma delas é que, sendo a primeira linguagem de alto nível, o FORTRAN foi aprendido por muitos programadores e uma vasta quantidade de software foi escrita com sua utilização. Os programadores que

dominam completamente a linguagem relutam naturalmente em mudar para outra. Além disso, é mais razoável utilizar os diversos programas úteis escritos e intensamente testados em FORTRAN do que reescrevê-los em outra linguagem [MAR85] .

3.2 CARACTERÍSTICAS GERAIS DA LINGUAGEM

Em uma linguagem de programação sequências de palavras podem ser combinadas em sentenças que, por sua vez, formam programas. A sintaxe de uma linguagem é um conjunto de regras que determinam se uma sentença é bem formada ou não. A essas regras dá-se o nome de produções.

A tarefa de um compilador é reconhecer sentenças bem formadas de uma linguagem e gerar código que seja semanticamente equivalente às sentenças.

Além das vinte e seis letras maiúsculas que compõem o alfabeto inglês (A.....Z) a linguagem FORTRAN utiliza os dez algarismos arábicos (0.....9). O padrão ANSI FORTRAN 77 acrescenta a estes mais treze caracteres especiais (= + * / ( ) , . $ ' :) e o caracter branco.

Quanto à formação de identificadores, a linguagem FORTRAN 77 ainda restringe identificadores a seis caracteres. Isso frequentemente impossibilita a utilização de nomes significativos para as variáveis.

Em FORTRAN não são usados símbolos especiais para definição de blocos de comandos. No comando IF BLOCO o início se dá com a palavra IF e após um bloco de comandos aparece ENDIF que sinaliza o final do bloco.

O FORTRAN apesar de ser uma linguagem que deve ser compilada, possui um comando, o FORMAT, que é sempre interpretado [SEB89] .

O FORTRAN traz vantagens sobre linguagens do tipo Algol 60. O FORTRAN tem o recurso de compilação separada de diversas partes de um único programa. Uma pessoa pode desenvolver independentemente parte de um sistema de software e produzir código objeto para ele em uma biblioteca. Uma segunda pessoa pode usar este código objeto enquanto constrói outra parte do sistema.

3.3 POTENCIAIS ERROS ASSOCIADOS A ALGUNS COMANDOS FORTRAN

Apresenta-se aqui algumas características da linguagem FORTRAN e possíveis erros não detectados por alguns compiladores.

. Compilação Independente

Como já dito, o FORTRAN permite compilação independente. Isto significa que unidades de programa podem ser compiladas sem informação sobre qualquer outra unidade do mesmo programa.

Tal compilação permite mais facilmente a construção de grandes sistemas, mas isto não é uma forma muito segura. Um aspecto negativo desta facilidade reside no fato de que, uma vez produzido o código objeto, torna-se impossível verificar se o mesmo está sendo acessado pelo legítimo chamador. Por exemplo, para um procedimento FORTRAN com dois parâmetros, um do tipo inteiro e outro do tipo real, poderia não haver uma forma de verificar se uma chamada está fornecendo valores com tipos corretos, ou até mesmo se está fornecendo o número correto de valores. Outras possibilidades de erros devidos à compilação independente são ainda mostrados neste capítulo.

. Palavras Especiais

As palavras-chave em FORTRAN têm um sentido mais restrito do que em linguagen "C" [KER90] e Pascal [BOR87], entre outras.

Palavras-chave em FORTRAN não são nomes reservados como em outras linguagens [MIC87]. O compilador reconhece as palavras-chave por seus contextos. Por exemplo, um programa pode ter um vetor com nome READ ou GOTO. Usando esses nomes, entretanto, constroi-se programas de difícil entendimento e leitura. Por isso os programadores devem evitar nomes que possam ser confundidos como comandos FORTRAN.

. **Sintaxe**

A sintaxe de uma linguagem de programação pode ter um efeito dramático na integridade , confiabilidade e segurança de programas [HOR84] . Consideremos o comando FORTRAN

```
DO10I=1.5
  A(I)=X+B(I)
10 CONTINUE
```

que, à primeira vista, poderia ser identificado como um erro de construção do comando DO: um ponto foi colocado onde deveria aparecer uma vírgula. Entretanto, este comando seria interpretado, indevidamente, como correto: atribuição do valor 1.5 à variável DO10I.

. **Forma e significado**

No projeto de uma linguagem de programação, para auxiliar na legibilidade, a aparência dos comandos deve, pelo menos parcialmente, indicar sua ação. Em FORTRAN, a semântica nem sempre segue a sintaxe [SEB89] . O princípio é violado por duas construções da linguagem que possuem aparência similar mas têm diferentes significados: é o caso do GO TO assinalado e do GO TO computado.

Por exemplo :

```
GO TO I, (10,20,30)
```

e

```
GO TO (10,20,30), I
```

no primeiro comando, a variável I identifica o número do rótulo do comando para o qual deve se dar o desvio. No segundo, a variável I indica uma ordenação. Isto é, se I é igual a 1, o desvio se dá para o comando relacionado com primeiro rótulo da lista. Se I é igual a três, o desvio se dá para o comando relacionado com o terceiro rótulo da lista. E assim sucessivamente.

. **Verificação de Tipos**

Alguns compiladores FORTRAN não executam verificação de tipos em tempo de compilação nem em tempo de execução. Isto pode acarretar incontáveis erros. Nesses compiladores, uma variável do tipo INTEGER pode ser usada como parâmetro na chamada de uma subrotina que espera parâmetros do tipo REAL. Esta inconsistência não é identificada.

. **Sinonímia ("Aliasing")**

A sinonímia permite ao programador definir nomes diferentes para a mesma localização de memória [HOR84] . Recentemente, esse recurso vem sendo apontado como tendo grande efeito negativo sobre a confiabilidade de programas. No FORTRAN, o comando EQUIVALENCE tem essa propriedade. Não existe mecanismo pelo qual o usuário ou o sistema possa determinar o tipo do valor corrente armazenado em tal localização; então nenhuma verificação é feita. Com o seu uso é possível que duas variáveis de tipos diferentes de um programa possam referenciar a mesma célula de memória.

. **Recursividade**

As especificações da linguagem FORTRAN, incluido o padrão ANSI 77, fazem a definição das áreas de memória para cada nome de variável quando do início do programa. Essas definicões permanecem ligadas às mesmas variáveis durante toda a execução do programa [HOR84]. Essa forma de alocação adotada pelo FORTRAN é estática. Todo dado é alocado em tempo de compilação. Em tempo de execução, quando uma subrotina é chamada, toda a sua área de armazenagem é colocada à parte em alguma região fixa, incluindo espaço para parâmetros e para endereço de retorno de subrotinas. Nas chamadas seguintes uma subrotina irá encontrar todas as suas áreas de armazenagem no mesmo lugar. Esse esquema simples de alocação de memória preserva o valor de dados através de sucessivas chamadas das subrotinas: entretanto, inviabiliza procedimentos recursivos.

. **Escopo de Variáveis**

Com relação ao escopo, cada variável em FORTRAN é declarada como local para os procedimentos (PROGRAM, SUBROUTINE ou FUNCTION) nos quais ela é usada. O efeito global é alcançado pela característica do comando COMMON. Variáveis que são nomeadas em um comando COMMON podem ser compartilhadas entre outros procedimentos que também possuem o comando COMMON. Entretanto, não é exigido que as variáveis em COMMON sejam referenciadas da mesma forma em diferentes subrotinas, e não existe verificação para identificar se as declarações COMMON são consistentes. Não obstante ter sua utilidade, o uso do COMMON pode trazer efeitos indesejáveis quanto à segurança. A opção de uso de bloco COMMON é possível e permite a um conjunto de procedimentos compartilhar um conjunto nomeado de variáveis exclusivamente.

Um problema com o uso do COMMON é que dois subprogramas podem, cada um, incluir um comando COMMON que especifica o mesmo nome de bloco e que, portanto, referem-se ao mesmo bloco. Cada um

dos subprogramas pode, por outro lado, especificar uma lista diferente de variáveis, com tamanhos e tipos que podem não corresponder em ambas as listas. Por exemplo, é perfeitamente legal ter as declarações como mostrado na Figura 3.1. em um subprograma, e as declarações como mostradas na Figura 3.2 em outro subprograma. O identificador BLOCK1 é o nome do bloco. A Figura 3.3 mostra esquematicamente com seriam vistas as variáveis em BLOCK1. Essa organização pode fazer sentido apenas se BLOCK1 é usado meramente para compartilhar área de armazenagem e não dados.

Na maioria dos casos, um dado em um bloco COMMON é para ser compartilhado usando o mesmo nome de variável e tipo. Um simples erro de ordenação na lista de variáveis em um comando COMMON pode causar armazenamento inadequado entre variáveis de tipos diferentes, o que pode causar um erro de difícil detecção.

Na construção de grandes programas, pode-se aproveitar as vantagens da compilação independente que o FORTRAN permite. Isso não é muito seguro [SEB89] . O único requisito da compilação independente na linguagem é que os nomes e tipos de variáveis alocadas pelo comando COMMON devem ser especificadas na unidade de programa sendo compilada. Uma característica da compilação independente é que as interfaces entre unidades compiladas separadamente não são verificadas quanto à consistência.

. Matrizes e Vetores

Para tratamento de matrizes e vetores a declaração de suas dimensões se faz por meio do comando DIMENSION. Em FORTRAN, deve-se escrever

```
DIMENSION A(100), B(10,10)
```

para declarar um vetor e uma matriz bi-dimensional de 100 elementos.

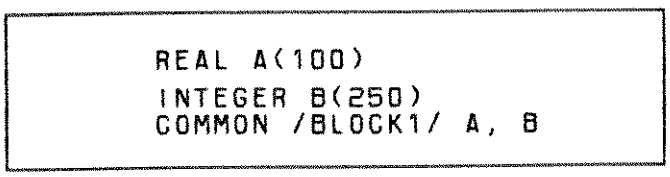


Figura 3.1 - Exemplo de Declaração COMMON em um Subprograma.

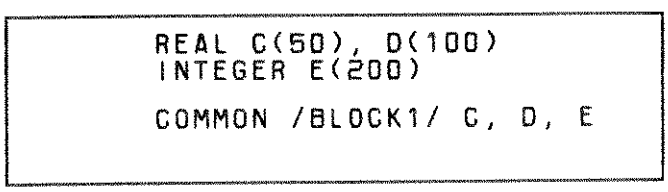


Figura 3.2 - Exemplo de Declaração COMMON em um Subprograma.

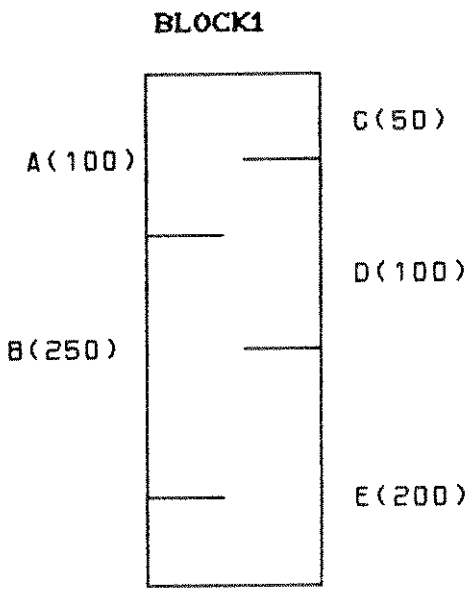


Figura 3.3 - Visão Esquemática de Partilhamento de Bloco COMMON.

A sintaxe de referência a vetores e matrizes é bastante universal: o nome do conjunto é seguido por uma lista de índices, a qual é envolvida por parenteses. Um problema que existe na linguagem é que eles também são usados para envolver os parâmetros em chamadas de subprogramas; isto faz com que as referências a "arrays" apareçam exatamente como as chamadas de função. Na linha de comando,

```
SUM = SUM + B(I)
```

só será possível a identificação de B como uma referência a um vetor ou a uma função através da consulta às declarações feitas de todos os vetores dentro da unidade sendo analisada. Se nenhuma declaração de B é encontrada, tal referência é considerada ser uma chamada de função.

Em FORTRAN 77 a inicialização de matrizes e vetores pode ser feita através do uso do comando DATA. O vetor LIST, por exemplo, pode ser inicializado da seguinte forma:

```
INTEGER LIST(3)
DATA LIST /0,5,5/
```

Algumas linguagens apresentam operações que tratam vetores e matrizes como unidades. Este não é o caso do FORTRAN 77, no qual somente elementos individuais dos vetores e matrizes podem ser manipulados. O FORTRAN 77 não fornece operações sobre matrizes.

. Comandos de Desvio

A linguagem FORTRAN em suas versões mais antigas, encorajava o uso do comando GO TO devido aos seus comandos condicionais. O uso do GO TO comprometia a segurança e a legibilidade do código fonte. Apesar do comando GO TO não ter sido abolido, esse problema foi em parte diminuido com a introdução no padrão ANSI FORTRAN 77 da forma IF-THEN-ELSE que requer um ENDIF. Para a implementação de algumas estruturas da programação estruturada como *while*, o comando GOTO ainda precisa ser utilizado.

. Passagem de Parâmetros

Na linguagem FORTRAN todos os parâmetros são passados por referência. Esta forma de passagem de parâmetros pode causar, em alguns ambientes, um erro sutil porém fatal. Pode ocorrer em um programa que contenha duas referências à constante 10, a primeira sendo um parâmetro em uma chamada à subprograma. Imagine-se que o

subprograma erroneamente modifique esse parâmetro, que corresponde a 10 para o valor 5. O compilador para este programa deve ter criado uma única posição para o valor 10 durante a compilação, como normalmente é feito, e deve usar aquela posição para todas as referências à constante 10 no programa. Mas após retornar do subprograma, todas as subsequentes ocorrências de 10 serão realmente referenciadas ao valor 5.

. **Chamadas de Funções**

A definição do FORTRAN 77 estabelece que expressões que têm chamadas de funções são legais somente se as funções não modificam os valores de outros operandos dentro da expressão. Não é fácil para o compilador determinar o efeito exato que uma função pode ter sobre variáveis fora da função, especialmente na presença de COMMON e EQUIVALENCE. Este é um caso em que a definição da linguagem especifica as condições sob as quais uma construção é legal, mas deixa para o programador a garantia de que tais construções são especificadas corretamente nos programas.

O FORTRAN permite expressões de modo misto. Isto não é considerado bom pois permite que o programador produza erros que não são detectados pelo compilador, diminuindo com isto a segurança. Por exemplo, no trecho de programa abaixo,

```
      INTEGER A, B, C
      REAL D
. . . . . .
      C = FUN (A+D)
. . . . . .
```

FUN é uma função que tem um único parametro do tipo INTEGER. No exemplo, a expressão contida em FUN referencia A e D. Erroneamente o programador colocou D ao invés de B. A é declarado INTEGER e D REAL. Como resultado do erro, A é convertido para o tipo REAL. O

resultado da expressão é então do tipo REAL. Para a função é enviado um valor REAL ao invés do INTEGER que ela esperava devido ao fato de que os tipos dos parâmetros não são verificados em subprogramas escritos por usuários do FORTRAN 77. O compilador não detecta o erro. Isto pode produzir resultados incorretos.

3.4 CONSIDERAÇÕES FINAIS

Após um breve histórico da linguagem FORTRAN, foram descritas neste capítulo características gerais da linguagem e potenciais erros associados a alguns de seus comandos.

As informações sobre os potenciais erros inerentes à linguagem FORTRAN auxiliam no estabelecimento de uma taxonomia de erros e é um subsídio importante para estudos de avaliação da adequação de critérios de teste em relação a esses erros.

O conhecimento desses possíveis erros possibilitará, também, uma programação preventiva que tornará o software gerado mais confiável.

A partir do descrito neste Capítulo, poderão ainda ser desenvolvidos "*checklists*" que auxiliarão na tarefa de inspeção do código escrito em FORTRAN. Em [BR075,MYE79,HOW87,ART88] são tratadas as vantagens e as técnicas de inspeção de código fonte. Ressalta-se que a análise estática e dinâmica são atividades complementares do ponto de vista da validação de um sistema de software.

CAPÍTULO

4

LINGUAGEM FORTRAN: INSTANCIAÇÃO DOS MODELOS DE IMPLEMENTAÇÃO DOS CRITÉRIOS POTENCIAIS USOS

Neste capítulo apresentam-se aspectos gerais dos requisitos necessários para a aplicação dos critérios Potenciais Usos (PU) [MAL88a,MAL88b,MAL89,MAL91] às unidades de programas codificadas na linguagem FORTRAN-77 e as soluções adotadas. São apresentados os modelos de fluxo de dados, fluxo de controle e instrumentação considerados na configuração da ferramenta POKE-TOOL [CHA91b] para a linguagem alvo.

4.1 MODELO DE FLUXO DE DADOS PARA O FORTRAN-77

No contexto de teste de software, métodos e critérios de projeto de casos de teste baseados em análise de fluxo de dados requerem que as interações que envolvem definições de variáveis de programa e subsequentes referências a essas variáveis sejam testadas; por sua vez, os Critérios Potenciais Usos requerem associações que implicam no exercício de caminhos entre uma definição de variável e um possível (potencial) uso dessa

variável. Analogamente aos outros critérios, os critérios Potenciais Usos necessitam de uma ferramenta para sua aplicação efetiva. A ferramenta POKE-TOOL [CHA91b] supre essa necessidade.

Para a aplicação dos critérios de testes estruturais Potenciais Usos em programas escritos na linguagem de programação FORTRAN-77 é necessário a configuração da ferramenta POKE-TOOL para esta linguagem.

Como descrito anteriormente, são três os tipos de ocorrências de variáveis: definição, uso e indefinição. Do ponto de vista dos critérios Potenciais Usos a ocorrência do uso não precisa ser identificada.

Na geração do *grafo def* para o FORTRAN-77, a cada nó identificado é associado o conjunto de variáveis nêle definidas. A definição de uma variável se dá quando um valor é armazenado em uma posição de memória.

Como em outras linguagens, no FORTRAN-77 uma variável é definida quando está:

- no lado esquerdo de um comando de atribuição;
- em um comando de entrada; e,
- em chamadas de procedimentos como parâmetros de saída;

No FORTRAN sempre ocorre definição de variável em comandos DO nos quais a variável controladora do laço é iniciada e, posteriormente, incrementada a cada possível iteração.

A linha de comando [RIC83]

```
DO  300 I = 1, NBIT
```

define inicialmente a variável I com valor igual a 1 e, enquanto I for menor ou igual a NBIT a define novamente incrementando de 1.

Uma definição em FORTRAN também pode se dar por meio do comando DATA. O comando DATA é um comando não-executável que é usado para definir valores iniciais para variáveis, conjuntos,

elementos de conjuntos e subcadeias.

A linha de comando [HEH86]

```
DATA Q,N / 5.6, -0.016, 17 /, CAR,M /'TITULO', 3*2 /
```

define as variáveis Q, N, CAR, M.

No FORTRAN-77, uma atribuição é feita a uma variável quando esta aparece à esquerda de um sinal de igualdade (=) seguido de uma constante ou de uma expressão. A definição de uma variável poderá se dar após o conjunto de declarações do programa, em qualquer ponto da unidade; independentemente de uma declaração explícita anterior da variável (declaração implícita).

No FORTRAN-77 o comando de entrada utilizado para definição de variáveis é o comando READ. O comando READ, em alguns casos, é uma combinação de comando de entrada com comando que altera o fluxo de controle.

Exemplo:

```
READ ( 1, END = 50, ERR = 80 ) A, B, C
```

Através do comando acima, definem-se as variáveis A, B e C que recebem valores de um dispositivo de entrada. Além disso, no caso de fim dos dados de entrada, o controle do programa é desviado para a linha de comando cujo rótulo é 50. No caso de algum erro no processamento da entrada de dados, o programa é desviado para a linha de comando que possui o rótulo 80. As opções END e ERR não aparecem necessariamente em todos os comandos READ. Portanto, o comando READ, além de promover a definição de variáveis, também pode causar desvio de fluxo de controle.

Na linguagem FORTRAN a passagem de valores entre procedimentos através de parâmetros se dá somente por referência. Portanto, as chamadas de subrotinas feitas na linguagem por meio

do comando "CALL" definem variáveis por referência.

Em um programa na linguagem FORTRAN a linha [CER87]

CALL COPIA (ORIG, COP, K)

define por referência as variáveis ORIG, COP e K.

A linha de comando [CER87]

SUBROUTINE BOLHA (VET,N)

define por referência as variáveis VET e N.

As variáveis compostas (estruturadas) no FORTRAN-77 são os vetores e matrizes. Convenciona-se que a definição de um elemento de um vetor ou matriz implica a definição do vetor ou da matriz.

No modelo de fluxo de dados adotado, considera-se que no nó de entrada ocorre a definição de todos os parâmetros e variáveis globais que ocorrem na unidade em teste.

Ocorre uma indefinição de variável se sua localização não estiver definida ("amarrada") na memória ou se não for possível o acesso ao seu valor [FRA87]. A indefinição de uma variável pode ocorrer devido ao encerramento da execução da unidade; neste sentido o nó de saida tem uma indefinição de todas as variáveis locais.

4.2 INSTRUMENTAÇÃO PARA UNIDADES EM FORTRAN-77

A instrumentação consiste em comandos adicionados ao programa, os quais têm por finalidade registrar a sua execução. A

instrumentação visa facilitar a análise posterior à execução dos casos de testes como, por exemplo, a análise de adequação de um dado conjunto de casos de teste. Com este objetivo, procede-se à modificação do código fonte escrito na linguagem FORTRAN, com a inserção de "pontas de prova" numeradas. Gera-se desta forma uma nova versão da unidade em teste. A esta nova versão, a "instrumentada", dá-se o nome de TESTEPROG.FOR. Um exemplo desta versão instrumentada é mostrado no arquivo TESTEPROG.FOR gerado para o programa INCTAX.FOR e reproduzido no Apêndice E.

Como dito anteriormente, as pontas de prova são numeradas. Esses números correspondem aos números dos nós do grafo de fluxo de controle da unidade em teste. Essa numeração permite a identificação do caminho executado pelo caso de teste fornecido.

Para a instrumentação de unidades de programas em FORTRAN, insere-se no código fonte chamadas à subrotina PROVA, com o número do nó passado como parâmetro .

A subrotina PROVA chamada por —CALL PROVA (NO)— tem a função de gravar no arquivo PATH.TES os caminhos efetivamente percorridos na execução da unidade com uma formatação própria.

Para facilitar a leitura do código instrumentado, são inseridas também linhas de comentário que informam o número do nó associado a cada linha de comando do programa fonte.

Segundo [CHA91] a escrita de todos os nós foi adotada para simplificar a implementação pois, para os modelos adotados de avaliação e de descrição dos caminhos e associações requeridos, seria suficiente a inserção de pontas de prova nos nós n que contivessem definição de variáveis ou que constituissem arcos primitivos [CHU87]. Além disso, a escrita de todos os nós pode facilitar a implementação de outras funções; por exemplo, a determinação do número de vezes que um determinado comando foi executado [CHA91].

Por razões óbvias, a instrumentação deve ser tal que reflita

a semântica dos comandos da LI e, ao mesmo tempo, viabilize a correta avaliação dos caminhos efetivamente executados; deve ser observado, também, que a instrumentação está fortemente restrita ao modelo de fluxo de controle adotado. Soluções particulares foram tomadas para tratamento de comandos como "DO", "IF lógico", "STOP" e outros. Essas soluções são mostradas no decorrer deste capítulo.

Não obstante o salientado por Maldonado [MAL91] quanto à não modificação nos comandos originais do programa fonte, na presente configuração para a linguagem FORTRAN, este objetivo não pôde ser totalmente alcançado. Comandos como o "IF lógico", "ELSEIF" e "CONTINUE" são apresentados na versão instrumentada de uma forma diferente daquela apresentada no código fonte.

As localizações das pontas de prova dependem dos comandos de fluxo de controle da linguagem da unidade fonte que foi traduzida para LI. Neste capítulo são descritas as localizações das pontas de prova para monitorar os nós gerados pelos diversos comandos da LI.

Antes da inserção das pontas de prova propriamente ditas é necessário inserir alguns comandos preliminares, dependendo da linguagem da unidade em teste [CHA91].

No caso da configuração da POKE-TOOL para a linguagem FORTRAN-77, são inseridos no programa fonte:

a) Linhas de comentário para a identificação da unidade instrumentada;
b) Declarações de variáveis e comandos para a abertura (OPEN) e fechamento (CLOSE) do arquivo PATH.TES necessários para a instrumentação;
c) Chamadas à subrotina PROVA sempre que um novo nó é identificado no programa fonte;
d) Linha de comentário indicadora do número do nó a que o comando da próxima linha pertence;

e) A subrotina PROVA que efetivamente serve como "ponta de prova" nesta instrumentação, escrevendo o número do nó no arquivo PATH.TES.

As Figuras 4.1 e 4.2 mostram, respectivamente, como a unidade fonte é modificada para a instrumentação e o código da subrotina PROVA que é adicionado.

Em suma, o modelo de instrumentação visa inserir chamadas à subrotina PROVA passando como parâmetro o número do nó monitorado. Quando o número de nó passado é 999 está sendo indicado o término da execução da unidade instrumentada. A unidade em teste instrumentada recebe o nome de TESTEPROG.FOR. Uma vez compilada e executada, a unidade instrumentada gera o arquivo PATH.TES que armazena todos os caminhos exercitados.

4.3 ESPECIFICAÇÃO PARA A APLICAÇÃO DOS MODELOS DE FLUXO DE CONTROLE E INSTRUMENTAÇÃO PARA O FORTRAN-77

Apresentam-se nesta seção as soluções adotadas para a implementação dos critérios Potenciais Usos no ambiente POKE-TOOL para a linguagem FORTRAN. São mostradas os comandos que causam desvios de fluxo de controle, suas sintaxe e semântica, e o mapeamento para a LI; é mostrado também o grafo de fluxo de controle, a instrumentação e observações pertinentes. No próximo capítulo apresenta-se a solução de implementação adotada.

Quando da configuração da POKE-TOOL para a linguagem FORTRAN-77, encontraram-se dificuldades para o mapeamento fiel de comandos dessa linguagem para a LI. A LI não podia refletir a semântica de todos os comandos do FORTRAN de uma forma direta. Isso poderia implicar numa maior mudança no código fonte original quando da

```
*
*      TESTEPROG.FOR - PROGRAMA FONTE INSTRUMENTADO
*                      PARA TESTES
*
*
          ......................
*
*-------    PARA INSTRUMENTAÇÃO    ---------
*
       INTEGER IMPNO, TABNOS
       DIMENSION TABNOS(10)
       COMMON IMPNO, TABNOS
*
*-----------------------------------------
*
                ...................
*
*-------    PARA INSTRUMENTAÇÃO    ---------
*
       IMPNO = 0
       OPEN (20, FILE='PATH.TES', STATUS='OLD')
*
*-----------------------------------------
*
          ......................
       CALL PROVA (999)
       CLOSE (20, STATUS='KEEP')
```

Figura 4.1 - Comandos Inseridos na Unidade Fonte em Teste para Instrumentação.

```
COMENTARIOS :
C
C--------    PARA INSTRUMENTAÇÃO    --------
C
C          SUBROTINA PONTA DE PROVA
C
C-----------------------------------------

      SUBROUTINE PROVA(NUM)

      INTEGER IMPNO,
     ,        TABNOS
      DIMENSION TABNOS(10)
      COMMOM IMPNO,
     ,       TABNOS

      IMPNO = IMPNO + 1
      TABNOS(IMPNO) = NUM
      IF ( NUM .EQ. 999 )
                THEN
                WRITE (20,*) (TABNOS(I),I=1,IMPNO-1)
                IMPNO = 0
      ELSEIF (IMPNO .EQ. 10)
     ,          THEN
                WRITE (20,*) (TABNOS(I),I=1, IMPNO)
                IMPNO = 0
      ENDIF
      RETURN
      END
```

Figura 4.2 - Subrotina PROVA.

instrumentação. Novos átomos foram então adicionados à LI para o tratamento dos desvios de fluxo encontrados nos comando READ (com cláusula de ERR e/ou END), GOTO assinalado, GOTO computado e IF aritmético. Portanto, agregou-se ao conjunto de átomos da LI, os átomos $GT e $ROTALL. Esses átomos são descritos na Figura 4.3.

Ao se descrever os modelos de fluxo de controle e instrumentação adotados para a instanciação dos comandos FORTRAN-77 para a LI os seguintes recursos foram utilizados:

- Apresentação do grafo de fluxo de controle do comando sendo mapeado. Neste grafo, os nós são notados com um círculo identificado por seu número. Dois círculos concentricos representam sub-grafos, que deverão ser expandidos. Os arcos orientados representam os fluxos de controle;
- Apresentação da equivalência sintática entre o comando FORTRAN-77 e o comando em LI. Note-se que essa equivalência nem sempre é de um para um. Isto é, algums comandos FORTRAN são mapeados para mais de um comando LI;
- Apresentação da tradução para a LI de um exemplo de comando. Essa tradução segue o descrito no Apêndice A. Aos comandos em FORTRAN-77 são associados os átomos em LI correspondentes. A cada átomo da LI são acrescentados números identificadores que relacionam-se com o programa fonte. Na tradução, após cada átomo são gerados três números que representam, respectivamente:
  - 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);
  - o número da linha onde se encontra o átomo.

  A utilidade dos ponteiros dos átomos é possibilitar o acesso ao código fonte associado ao átomo da LI;
- Apresentação da instrumentação do exemplo de comando.

COMANDO $GT

<gt> ::= <gt-atm> <cond-atm> <inicio-bloco> <rotc-atm>
[<rotc-atm>] [<rotd-atm>] <final-bloco> |
<gt-atm> <cond-atm> <inicio-bloco> <rotall-atm>
<final-bloco>

onde:

<gt-atm>	=	$GT	<inicio>	<comprimento>	<linha>
<cond-atm>	=	$C	<inicio>	<comprimento>	<linha>
<inicio-bloco>	=	{	<inicio>	<comprimento>	<linha>
<fim-bloco>	=	}	<inicio>	<comprimento>	<linha>
<rotc-atm>	=	LABEL	<inicio>	<comprimento>	<linha>
<rotd-atm>	=	$ROTD	<inicio>	<comprimento>	<linha>
<rotall-atm>	=	$ROTALL	<inicio>	<comprimento>	<linha>

Grafos:

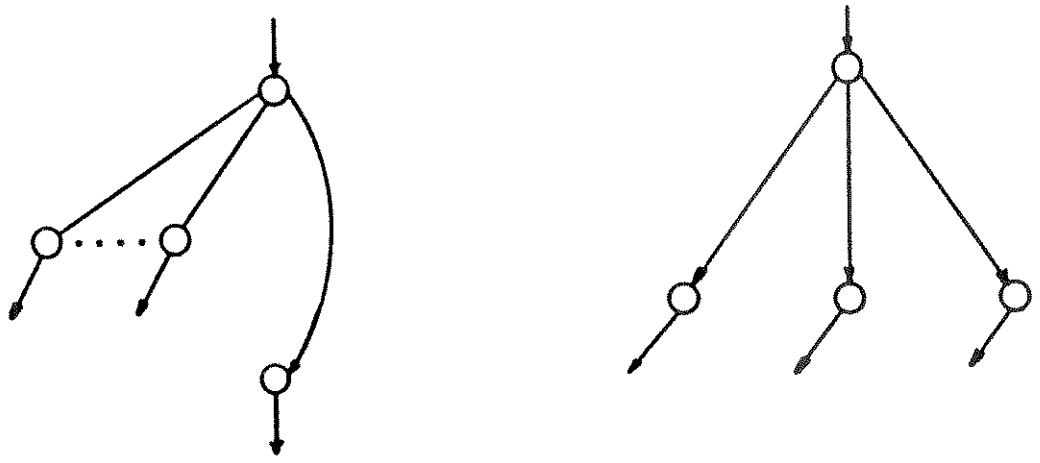


Figura 4.3 - Modelo de Fluxo de Controle Associado ao Comando $GT da LI.

4.3.1 COMANDOS DE ENTRADA/SAÍDA

Os comandos READ e WRITE da linguagem FORTRAN-77 são os comandos utilizados para a transferência de dados da entrada para a memória e da memória para a saída, respectivamente. O mapeamento desses comandos de entrada/saída se dá de duas maneiras distintas. Os comandos READ e WRITE que não contenham cláusulas de "END = label" e "ERR = label" são mapeados para $S. Já aqueles comandos de entrada/saída que possuem uma destas cláusulas ou ambas, são mapeados para $GT. O mapeamento dos comandos para a LI é mostrado a seguir.

COMANDOS: READ e WRITE

SINTAXE:

```
READ  ([unidade] [,IOSTAT=i] [,END=n1] [,ERR=n2] ) [lista_de_var]
WRITE ([unidade] [,IOSTAT=i] [,END=n1] [,ERR=n2] ) [lista_de_var]
```

SEMÂNTICA:

São comandos de entrada e saída que possuem cláusulas de final (END) e/ou erro (ERR). Estas cláusulas devem ser consideradas pois causam o desvio do fluxo do programa para os rótulos (n_i) indicados após os respectivos sinais de igual (=). A Figura 4.4 mostra o grafo de fluxo controle correspondente.

MAPEAMENTO PARA A LI :

Os comandos READ e WRITE são mapeados para comandos $GT da LI. As cláusulas END = $n1$ e/ou ERR = $n2$ são mapeadas para rótulos iguais à "ni". A equivalência sintática entre a linguagem FORTRAN

e a intermediária é mostrada na Tabela 4.1.

Para o átomo $GT as informações de início e comprimento apresentadas são as de toda a linha do comando READ. A informação de linha é a relativa ao início do comando.

No caso de aparecerem em um mesmo comando READ as duas cláusulas de desvio ("END =" e "ERR ="), estas são mapeadas para apenas um comando $C com número de predicados igual à 2.

O comando da LI -$ROTD- indica que o destino do fluxo do programa se dará para o comando seguinte ao comando READ, no caso de não ocorrer desvio relacionado com as cláusulas "END" e "ERR".

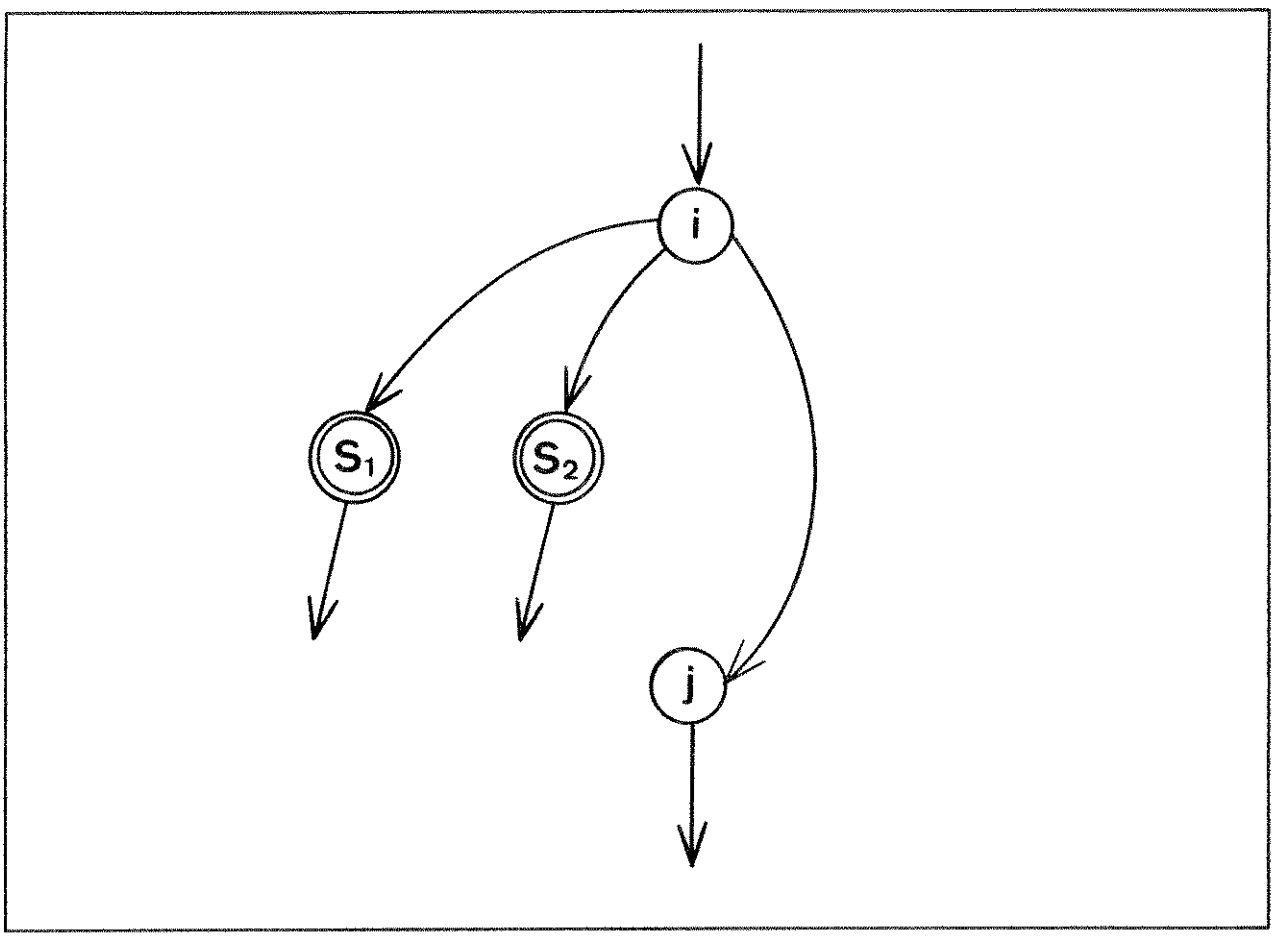


Figura 4.4 - Grafo de Fluxo de Controle: Comandos de E/S.

Tabela 4.1 - Equivalência Sintática: Comandos E/S.

READ ([unidade] [,IOSTAT=i] [,END=n1] [,ERR=n2]) [lista]	$GT
END =	$C
ERR =	$C
ni	LABEL

INSTRUMENTAÇÃO:

Para instrumentar os comandos de entrada/saída READ/WRITE, deve-se adicionar ao código fonte chamadas à subrotina PROVA antes do comando READ se o nó em que este se encontra ainda não tiver sido monitorado. Como os comandos destino das cláusulas "END" e "ERR" são comandos rotulados, esses provocarão chamadas à subrotina PROVA.

EXEMPLO :

READ (1, END = 50 , ERR = 80) A , B , C

A tradução para a LI do exemplo é mostrada na Figura 4.5. e a instrumentação do exemplo é mostrada na Figura 4.6.

Considera-se no exemplo, que o nó *k* (nó 12) contém o primeiro comando executável de *S1* e o nó *l* (nó 25) contém o primeiro comando executável de *S2*. Os nós representados por *i* (nó 5) *e j* (nó 6) referem-se, respectivamente, aos nós anterior e posterior ao comando READ.

```
READ (1,END=50, ERR=80) A,B,C
END =

50
80
```

```
$GT          50   41   3
$C(02)01     67    5   3
{             0    0   0
50           73    2   3
80           84    2   3
$ROTD         0    0   0
}             0    0   0
```

Figura 4.5. - Tradução para a LI do Exemplo de Comando de E/S.

```
          CALL PROVA (5)
          ..............
*  NO' 5  *
          READ ( 1, END = 50, ERR = 80 ) A, B, C
          CALL PROVA (6)
*  NO' 6  *
   ..............
   ..............
*  NO' 12  *
   50  CALL PROVA (12)
   ..............
   ..............
*  NO' 25  *
   80  CALL PROVA (25)
   ..............
   ..............
```

Figura 4.6. - Instrumentação do Exemplo de Comando de E/S.

4.3.2 COMANDO DE ITERACÃO:

Um elemento muito importante em projeto e codificação de programas é a estrutura usada para programar execução repetida de um bloco de instruções. A estrutura de repetição é implementada em FORTRAN pelo laço DO [DAV88].

COMANDO: DO

SINTAXE:

```
DO label [,] variavel= expressão, expressão [,expressão]
< comandos >
label  comando
```

SEMÂNTICA:

O comando DO é um comando executável de controle que permite que uma sequência de comandos seja executada de forma repetitiva enquanto o valor de uma variável de controle de fluxo está variando entre limites especificados. O número de vêzes em que os comandos dessa sequência são executados depende da variável de controle [HEH86]. A Figura 4.7 mostra o grafo de fluxo de controle correspondente ao comando DO.

MAPEAMENTO PARA A LI:

O mapeamento do comando DO não traz dificuldades. Sua tradução é feita para o comando $FOR da LI sem modificações. A Tabela 4.2 mostra a equivalência sintática entre as linguagens FORTRAN e LI para este comando.

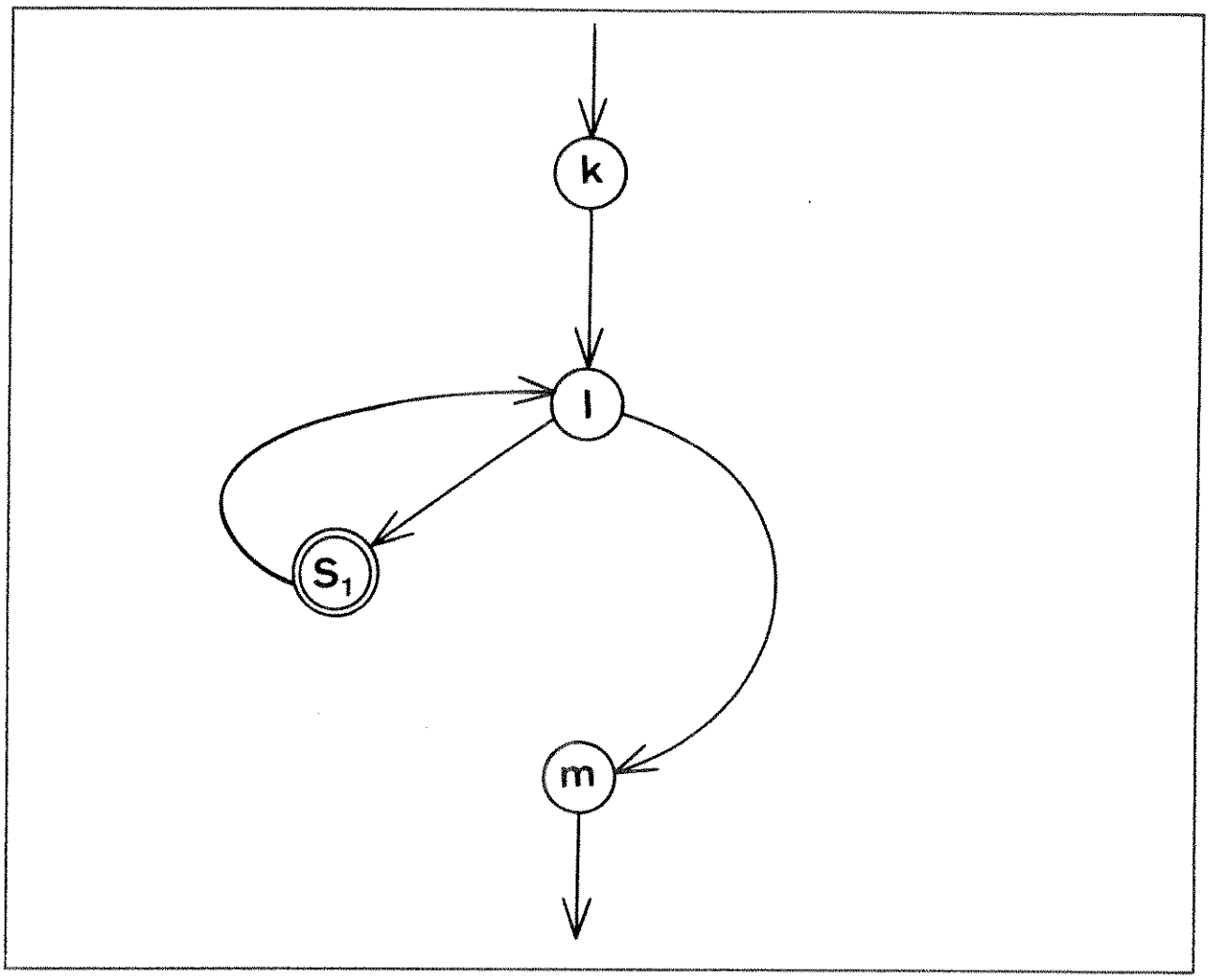


Figura 4.7. - Grafo de Fluxo de Controle : Comando DO.

Tabela 4.2. - Equivalência Sintática : comando DO

DO rótulo	$FOR
variável = expressão_1	$S
, expressão_2	$C
[, expressão_3]	$S

INSTRUMENTAÇÃO:

Para a instrumentação do comando DO deve-se inserir pontas de prova no código fonte como descrito para o comando "for" em [CHA91]. No grafo, mostrado na Figura 4.7, os nós de entrada e saída do subgrafo Si são i e j, respectivamente.

No início do nó k é inserida a ponta de prova k; no início do nó i as pontas de provas l e i; e no início do nó m, as pontas de provas l e m.

No nó j é também inserida a ponta de prova j.

EXEMPLO [CER87]:

```
      DO 10, M=2,N
         FAT = FAT * M
   10 CONTINUE
```

OBSERVAÇÃO:

Note-se que, na tradução do Exemplo (Figura 4.8), os ponteiros para o programa fonte utilizados para mapear o incremento (implicito) do contador do laço do comando (+1) apontam para toda a definição da variável de controle do laço (M=2,N). Esta soluçao de certa forma facilitou a implementação da geração do grafo *def* pois, no "parser" utilizado, não há a necessidade de se salvar o nome da variável definida no primeiro nó do laço, variável essa que também será definida no último nó desse mesmo laço.

DO 10	$FOR	93	5	5
M=2	$S02	100	3	5
,N	$C(01)01	103	2	5
	$S03	100	5	5
	[	0	0	0
FAT=FAT*M	$S04	117	13	6
10	10	135	2	7
CONTINUE	$S05	138	8	7
	]	0	0	0

Figura 4.8. - Tradução para a LI de um Exemplo de Comando de Iteração.

```
            CALL PROVA (1)
            ..............
*  NO'  1  2  4  *
         DO 10 M=2,N
            CALL PROVA(2)
            CALL PROVA(3)
*  NO'  3  *
            FAT = FAT * M
            CALL PROVA (4)
      10 CONTINUE
            CALL PROVA (2)
            CALL PROVA (5)
*  NO'  5  *
            ..............
```

Figura 4.9 - Instrumentação para a LI do Exemplo de Comando de Iteração.

4.3.3 COMANDOS DE TRANSFERÊNCIA:

Os comandos de transferência são comandos executáveis que transferem o fluxo de controle para um outro comando executável na mesma unidade de programa. São três os tipos desses comandos :

- GO TO incondicional,
- GO TO computado e
- GO TO assinalado.

A boa prática de programação sugere que esses comandos não sejam usados. O motivo para isso é que eles tornam difícil o acompanhamento da lógica de um programa, causando transtornos quando da depuração e manutenção.

COMANDO: GO TO incondicional

SINTAXE:

GO TO rótulo

SEMÂNTICA:

O comando GO TO incondicional transfere a execução para o comando identificado pelo rótulo. O comando indicado deve estar na mesma unidade de programa. A Figura 4.10 mostra o grafo de fluxo de controle representativo do comando GO TO incondicional.

MAPEAMENTO PARA A LI:

O comando GO TO incondicional é mapeado diretamente para o comando $GOTO da LI. Este comando é sempre o último comando

associado ao bloco i (nó i) ou seja, tem a ação de encerrar um bloco. A Tabela 4.3 mostra a equivalência sintática entre as linguagens FORTRAN e intermediária para o comando GO TO incondicional.

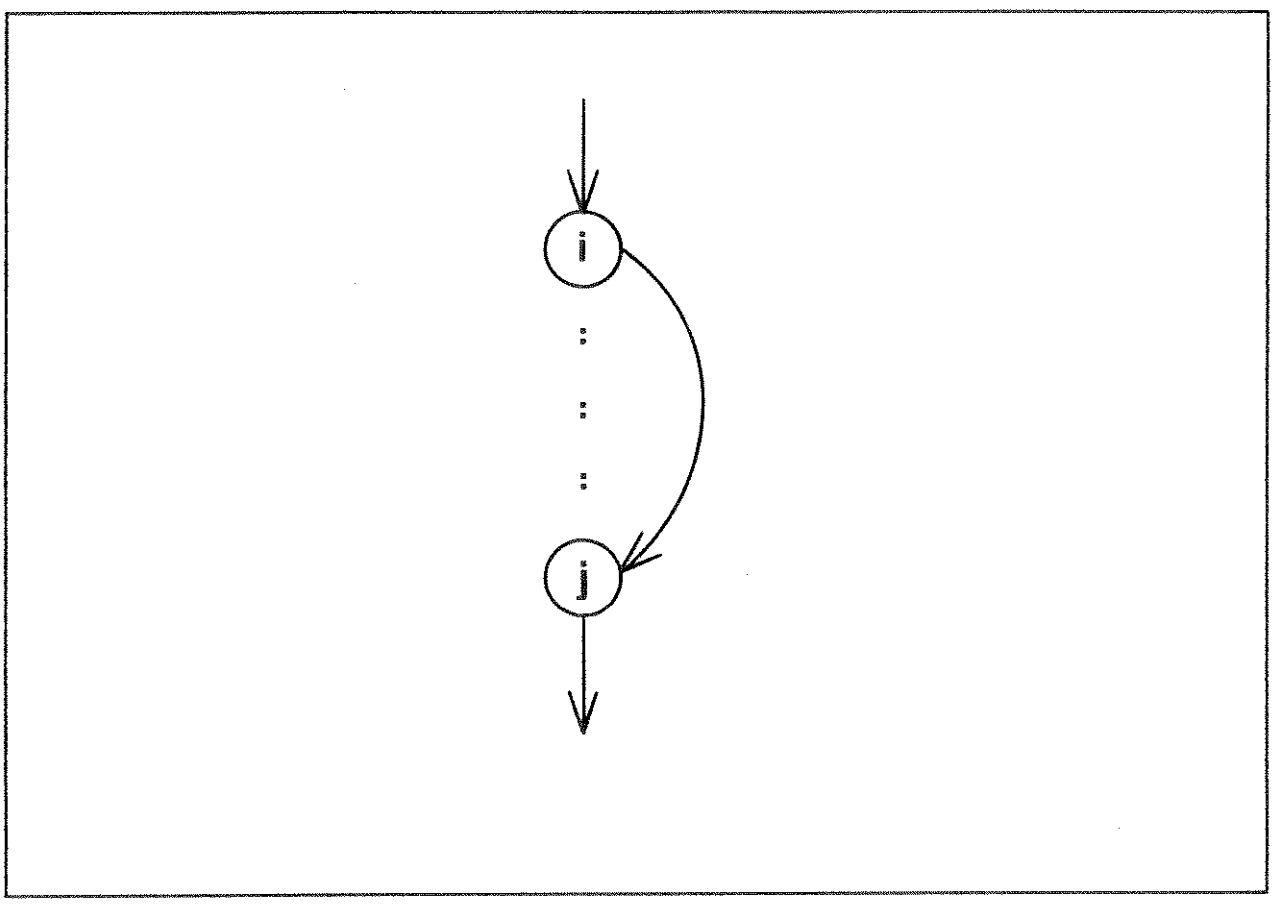


Figura 4.10 - Grafo de Fluxo de Controle: Comando GO TO Incondicional.

Tabela 4.3 - Equivalência Sintática: Comando GO TO Incondicional.

GO TO rótulo	$GOTO LABEL

EXEMPLO:

GO TO 100

Para este Exemplo, a tradução para a linguagem intermediária e a instrumentação são mostradas respectivamente nas Figuras 4.11 e 4.12.

```
GO TO          $GOTO  35   5   6
100            100    41   3   6
```

Figura 4.11 - Tradução para a LI do Exemplo de Comando GO TO Incondicional.

```
               CALL PROVA (10)
* NO' 10       *
               GO TO 100
               ..............
```

Figura 4.12 - Instrumentação do Exemplo de Comando GO TO Incondicional.

COMANDO: GO TO computado

SINTAXE:

GO TO (rótulo_1 [, rótulo_n]) [,] expressão

SEMÂNTICA:

O comando GO TO computado fornece uma maneira "aritmética" para transferir o fluxo de controle para um comando identificado a partir de uma lista de números de comandos, dependendo do valor corrente de uma expressão aritmética inteira [HEH86]. A expressão é avaliada e sendo seu valor igual a n , o controle é transferido para o comando cujo rótulo corresponde ao enésimo rótulo da lista. Se o valor da expressão estiver fora do intervalo [1, n], o GO TO computado é ignorado e o comando seguinte é executado. O grafo de fluxo de controle correspondente ao comando GO TO computado é apresentado na Figura 4.13.

MAPEAMENTO PARA A LI:

O comando GO TO computado é mapeado para o comando $GT da LI. As constantes que representam a lista de rótulos são mapeadas para LABEL. O átomo da LI $ROTD representa o desvio para o comando imediatamente seguinte ao comando GOTO computado. A Tabela 4.4 mostra a equivalência sintática entre a linguagen FORTRAN e a intermediária.

INSTRUMENTAÇÃO:

Para a instrumentação do comando GO TO computado faz-se a inserção de chamadas à subrotina PROVA no início dos blocos k e l e no início dos blocos iniciais de $S1$, $S2$, ... Sn..

EXEMPLO [CER87]:

```
GO TO (10, 20, 80, 10), I
```

As Figuras 4.14 e 4.15 mostram, respectivamente, a tradução para a LI e a instrumentação do Exemplo de comando GO TO computado.

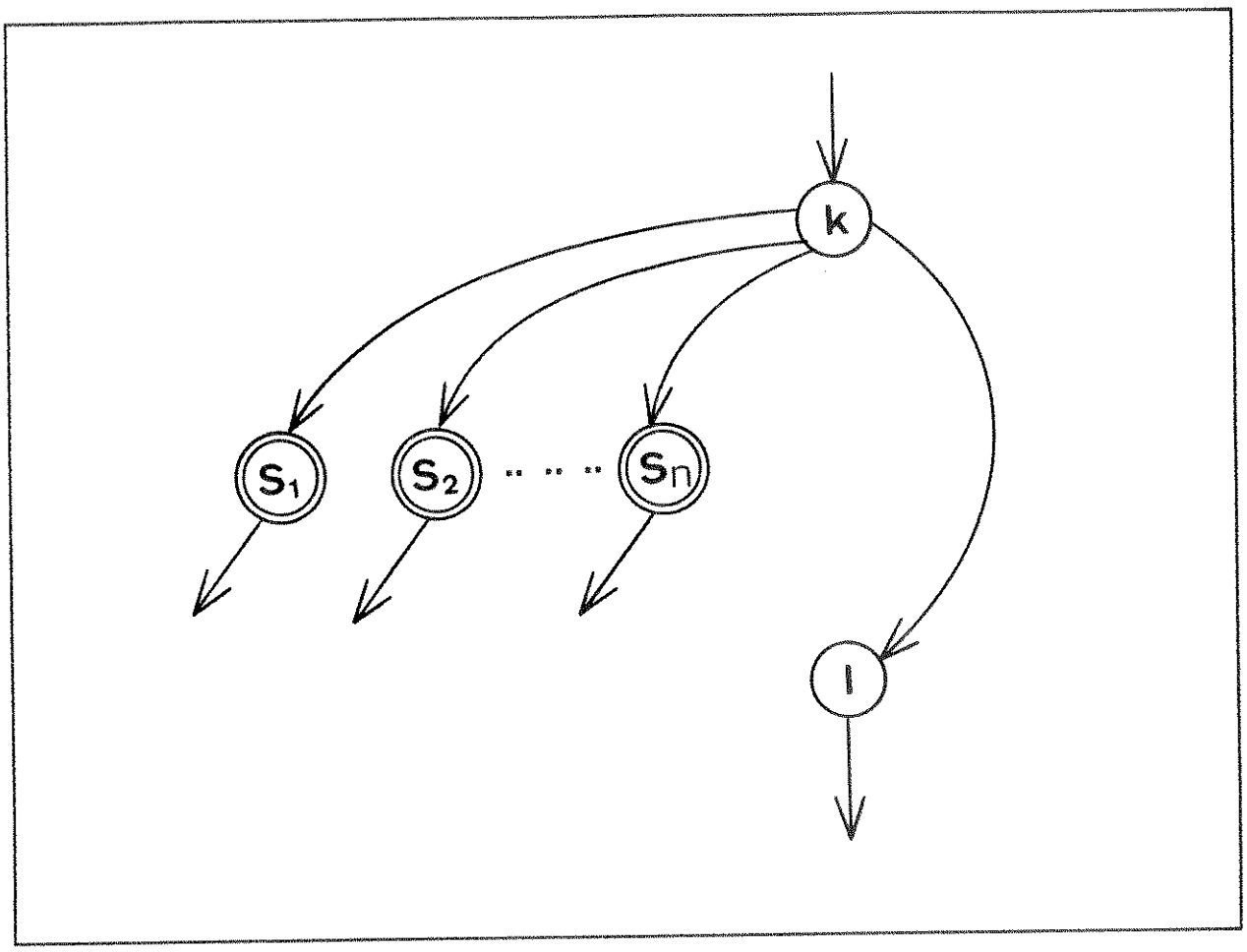


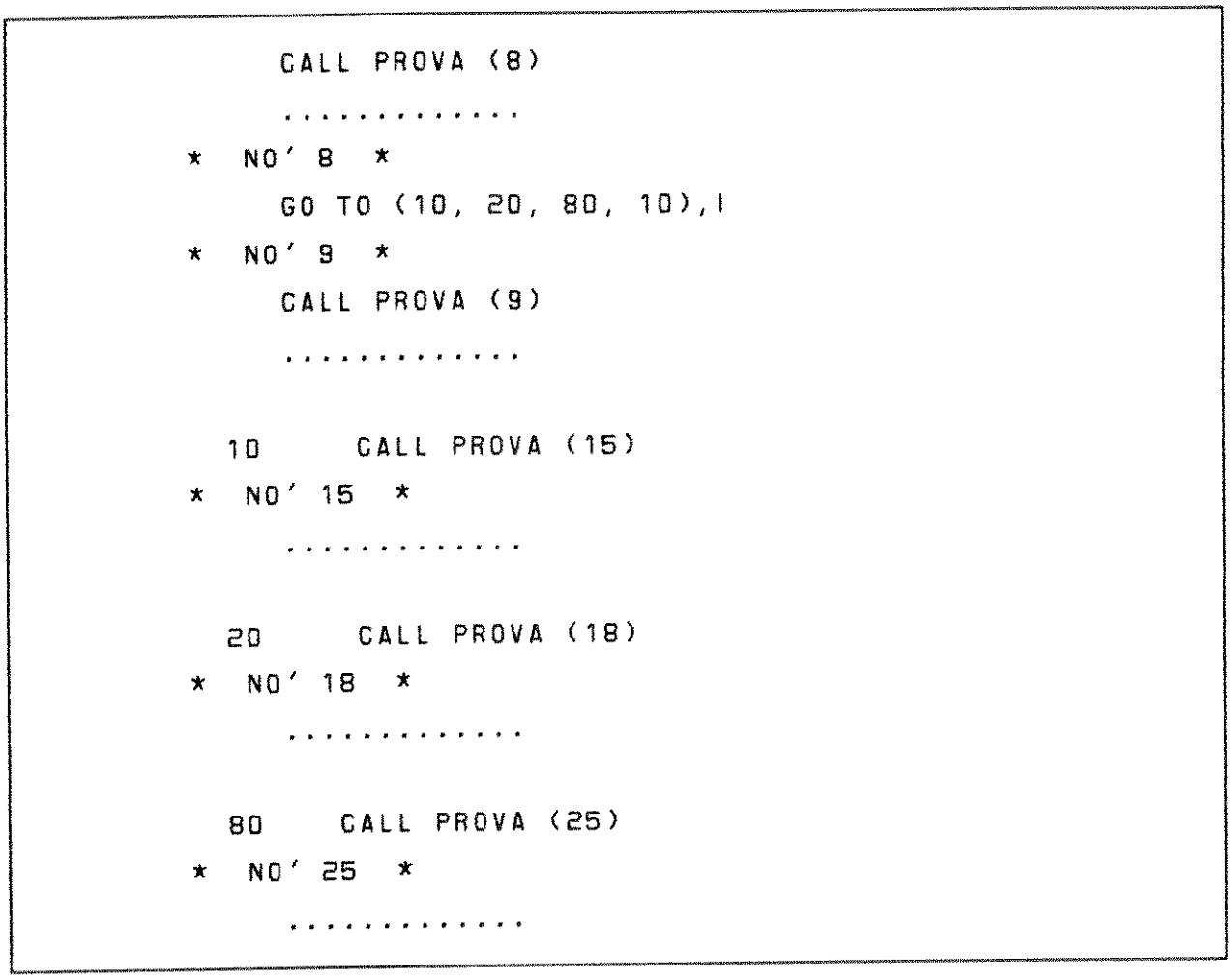
Figura 4.13 - Grafo de Fluxo de Controle: Comando GO TO Computado.

GO TO	$GT
(	[
rótulo	LABEL
)	]
[,] expressão	$C

Tabela 4.4 - Equivalencia Sintática: Comando GO TO Computado.

GO TO	$GT	63	5	4
,I	$C(01)01	85	2	4
(	[	69	1	4
10	10	70	2	4
20	20	74	2	4
80	80	78	2	4
10	10	82	2	4
	$ROTD	0	0	0
)	]	84	1	4

Figura 4.14 - Tradução para a LI do Exemplo de Comando GO TO Computado



```
      CALL PROVA (8)
      ..............
*   NO' 8   *
      GO TO (10, 20, 80, 10),I
*   NO' 9   *
      CALL PROVA (9)
      ..............

   10     CALL PROVA (15)
*   NO' 15   *
      ..............

   20     CALL PROVA (18)
*   NO' 18   *
      ..............

   80     CALL PROVA (25)
*   NO' 25   *
      ..............
```

Figura 4.15 - Instrumentação do Exemplo de GO TO Computado

COMANDO: GO TO assinalado

SINTAXE:

GO TO variável [,] (rótulo [, rótulo])

SEMÂNTICA:

Este comando transfere o fluxo de controle para um comando cujo número foi colocado em uma variável de controle através do comando ASSIGN. Portanto, o destino da transferência depende da última atribuição feita à variável pelo comando ASSIGN [HEHB6]. A Figura 4.16 mostra o grafo de fluxo de controle para este comando.

MAPEAMENTO PARA A LI:

O comando GO TO assinalado é mapeado para o comando $GT da Linguagem Intermediária, seguido pela lista de rótulos entre chaves. No caso de ser omitida a lista de rótulos, será considerada a possibilidade de desvios para todos os rótulos da unidade. Neste caso, a lista de rótulos será mapeada para $ROTALL. $ROTALL representa todos os rótulos definidos na unidade. A Tabela 4.5 mostra a equivalência sintática entre as linguagens FORTRAN e intermediária para o comando GO TO assinalado.

INSTRUMENTAÇÃO:

Para a instrumentação do comando GO TO assinalado, procede-se a inserção de chamada à subrotina PROVA no início do bloco k e no início dos blocos iniciais de $S1$, $S2$, ... Sn.

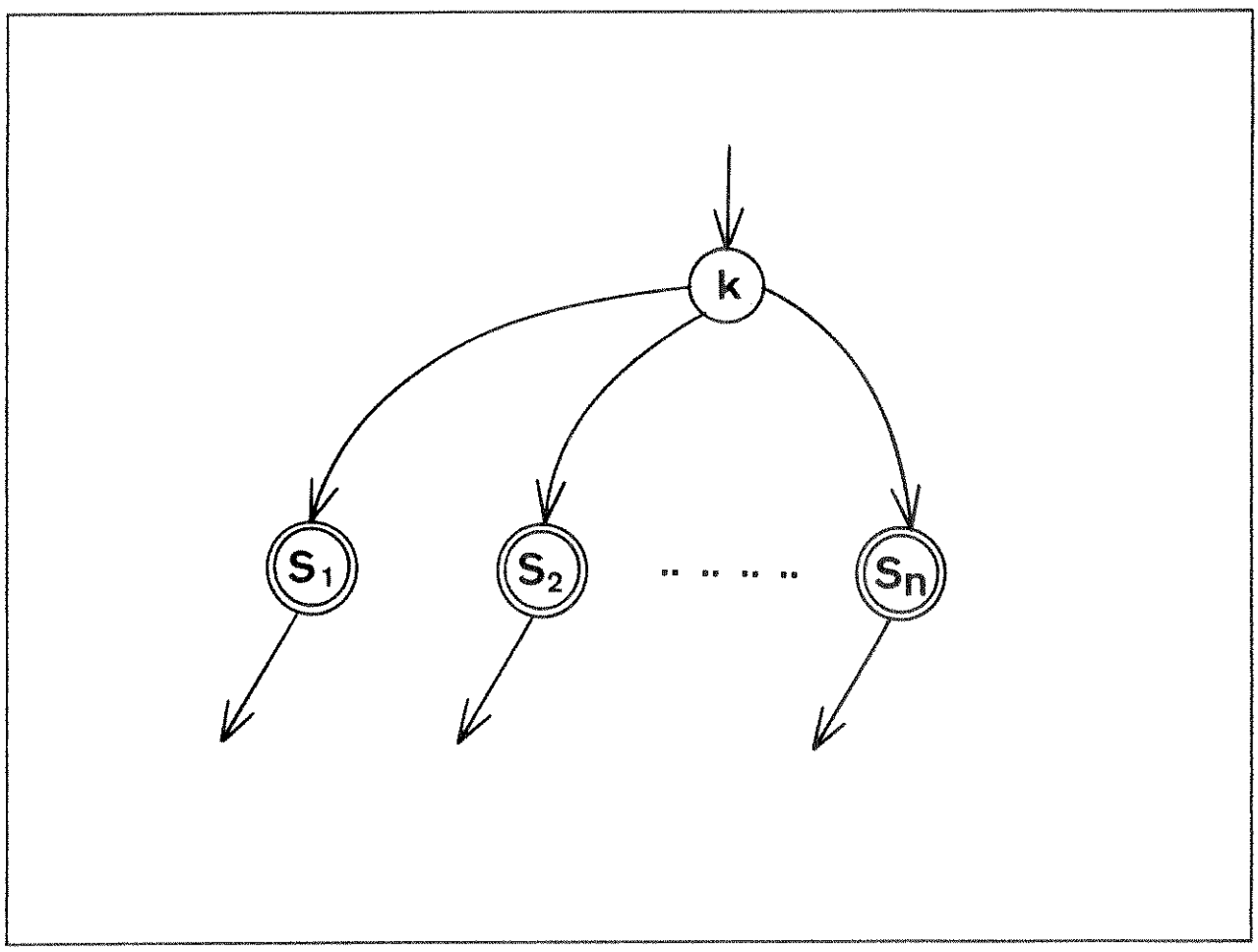


Figura 4.16 - Grafo de Fluxo de Controle: Comando GO TO Assinalado.

Tabela 4.5 - Equivalência Sintática: Comando GO TO Assinalado.

GO TO	$GT
variável	$C
	(
rótulo	LABEL
	)

EXEMPLO 1:

```
GO TO ITEM , ( 80 , 100 , 300 )
```

Nas Figuras 4.17 e 4.18 são mostrados respectivamente, o mapeamento para a LI e a instrumentação do Exemplo 1 de comando GO TO assinalado.

$GT	225	5	11
$C(01)04	231	4	11
(	238	1	11
80	240	2	11
100	245	3	11
300	251	3	11
)	261	1	11

Figura 4.17 - Tradução para a LI do Exemplo 1 de Comando GO TO Assinalado.

```
            CALL PROVA (4)
            ............
*   NO' 4   *
            GO TO ITEM  , ( 80 , 100 , 300 )
            ............

*   NO' 15  *
    80      CALL PROVA (15)
            ............

*   NO' 16  *
   100      CALL PROVA (16)
            ...........

*   NO' 18  *
   300      CALL PROVA (18)
            ............
```

Figura 4.18 - Instrumentação do Exemplo 1 de Comando GO TO Assinalado.

EXEMPLO 2:

GO TO ITEM

A Figura 4.19 mostra a tradução para a LI do Exemplo 2 de comando GO TO assinalado.

GO TO	$GT	130	5	25
ITEM	$C	136	4	25
	{	0	0	0
	$ROTALL	0	0	0
	}	0	0	0

Figura 4.19 - Tradução para a LI do Exemplo 2 de Comando GO TO Assinalado

OBSERVAÇÃO:

A instrumentação deste Exemplo não traz nenhuma novidade. O comando $ROTALL da LI associa ao comando $GT todos os rótulos da unidade em teste. Cada rótulo configura um novo nó e contém sempre uma ponta de prova.

4.3.4 COMANDOS DE DECISÃO:

Em FORTRAN, uma decisão é programada através do uso do comando IF. São três os tipos de comandos IF na linguagem. Eles são comandos executáveis que transferem o fluxo de controle ou executam outro comando (ou um bloco de comandos) dependendo de uma condição dada como resultado de uma expressão neles contida. Os três tipos de comandos IF são :

- IF aritmético,
- IF lógico e
- IF bloco.

COMANDO: IF aritmético

SINTAXE:

IF (expressão) rótulo1 , rótulo2 , rótulo3

SEMÂNTICA:

O comando IF aritmético transfere o fluxo de controle para um entre três comandos, cujos rótulos nele aparecem. A decisão para qual dos três comandos será o desvio depende do valor de uma expressão aritmética nele contida. O valor da expressão é testado e o controle é transferido para:

rótulo1 se expressão < 0
rótulo2 se expressão = 0
rótulo3 se expressão > 0

O uso do comando IF aritmético, apesar de permitido, não é recomendado, pois ele geralmente leva a programas pobremente estruturados que são de difícil entendimento. A Figura 4.20 mostra o grafo de fluxo de controle representativo do comando IF aritmético.

MAPEAMENTO PARA A LI:

O comando de decisão IF aritmético é mapeado para o comando $GT da linguagem intermediária. A Tabela 4.6 mostra a equivalência sintática entre o comando IF aritmético da linguagem FORTRAN e o respectivo comando da LI.

INSTRUMENTAÇÃO:

A instrumentação do comando IF aritmético consiste na inserção da chamada à subrotina PROVA no início do bloco k e no início dos blocos de entrada de $S1$, $S2$, e $S3$.

EXEMPLO [HEH86]:

```
IF ( N / 2 * 2 - N ) 10 , 50 , 10
```

Nas Figuras 4.21 e 4.22 são apresentadas, respectivamente, o mapeamento para a LI e a instrumentação do Exemplo de comando IF aritmético.

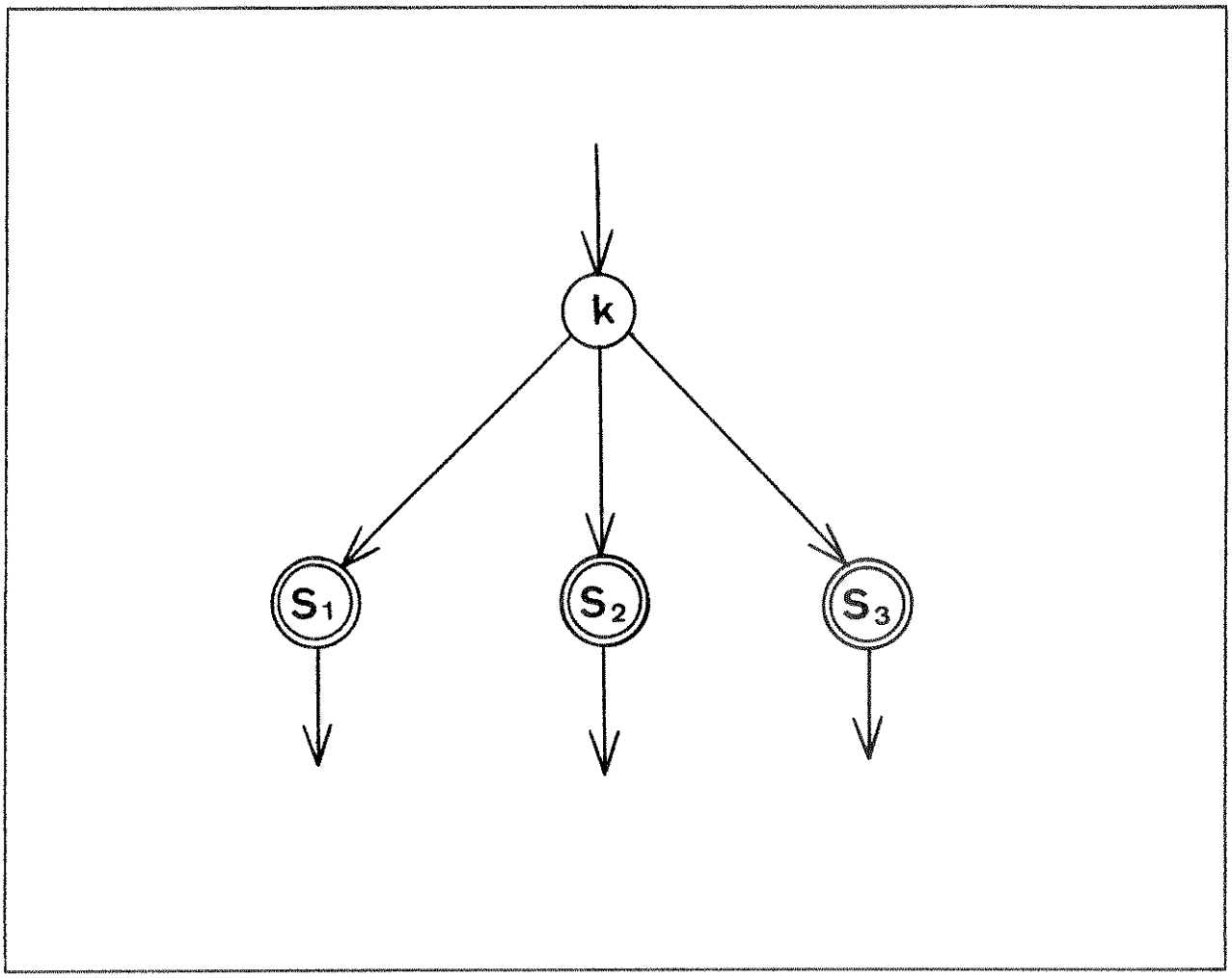


Figura 4.20 - Grafo de Fluxo de Controle: Comando IF Aritmético.

Tabela 4.6 - Equivalência Sintática: Comando IF Aritmético.

IF	$GT
expressão	$C
rótulo1	LABEL
rótulo2	LABEL
rótulo3	LABEL

$GT	46	2	3
$C(01)01	49	17	3
(	0	0	0
10	67	2	3
50	72	2	3
10	77	2	3
)	0	0	0

Figura 4.21 - Mapeamento para a LI do Exemplo de Comando IF Aritmético.

```
                CALL PROVA ( 12 )
*  NO' 12  *
                IF ( N / 2 * - 2 ) 10 , 50 , 10
                ..............

*  NO' 15  *
   10           CALL PROVA (15)
                ..............

*  NO' 17  *
   50           CALL PROVA (17)
                ..............
```

Figura 4.22 - Instrumentação do Exemplo de Comando IF Aritmético

COMANDO: IF lógico

SINTAXE:

IF (expressão) comando

SEMÂNTICA:

O comando IF lógico avalia uma expressão lógica/relacional e executa ou ignora um comando contido no próprio IF, dependendo do valor dessa expressão [HEH86]. A Figura 4.23 mostra o grafo de fluxo de controle correspondente ao comando IF lógico da linguagem FORTRAN.

MAPEAMENTO PARA A LI:

O comando IF é mapeado para o comando "if" da Linguagem Intermediária. A Tabela 4.7 mostra a equivalência sintática entre a linguagem FORTRAN e a intermediária.

INSTRUMENTAÇÃO:

Para a instrumentação deste comando, há a necessidade de modificação do código fonte. Esta modificação consiste na inserção de uma negação à expressão que é mapeada para $C. Esta foi a solução encontrada para a monitoração dos nós alternativos do comando IF.

De forma semelhante à instrumentação de outros comandos, insere-se também chamadas à subrotina PROVA imediatamente antes e após ao nó que contém o comando IF.

Altera-se ainda o código fonte fazendo-se a substituição do comando mapeado para $S por um comando GO TO <rótulo>. Este rótulo é gerado durante a instrumentação e a ele é atribuido valor que se encontra entre 7301 e 7399, inclusive. Note-se que a unidade em

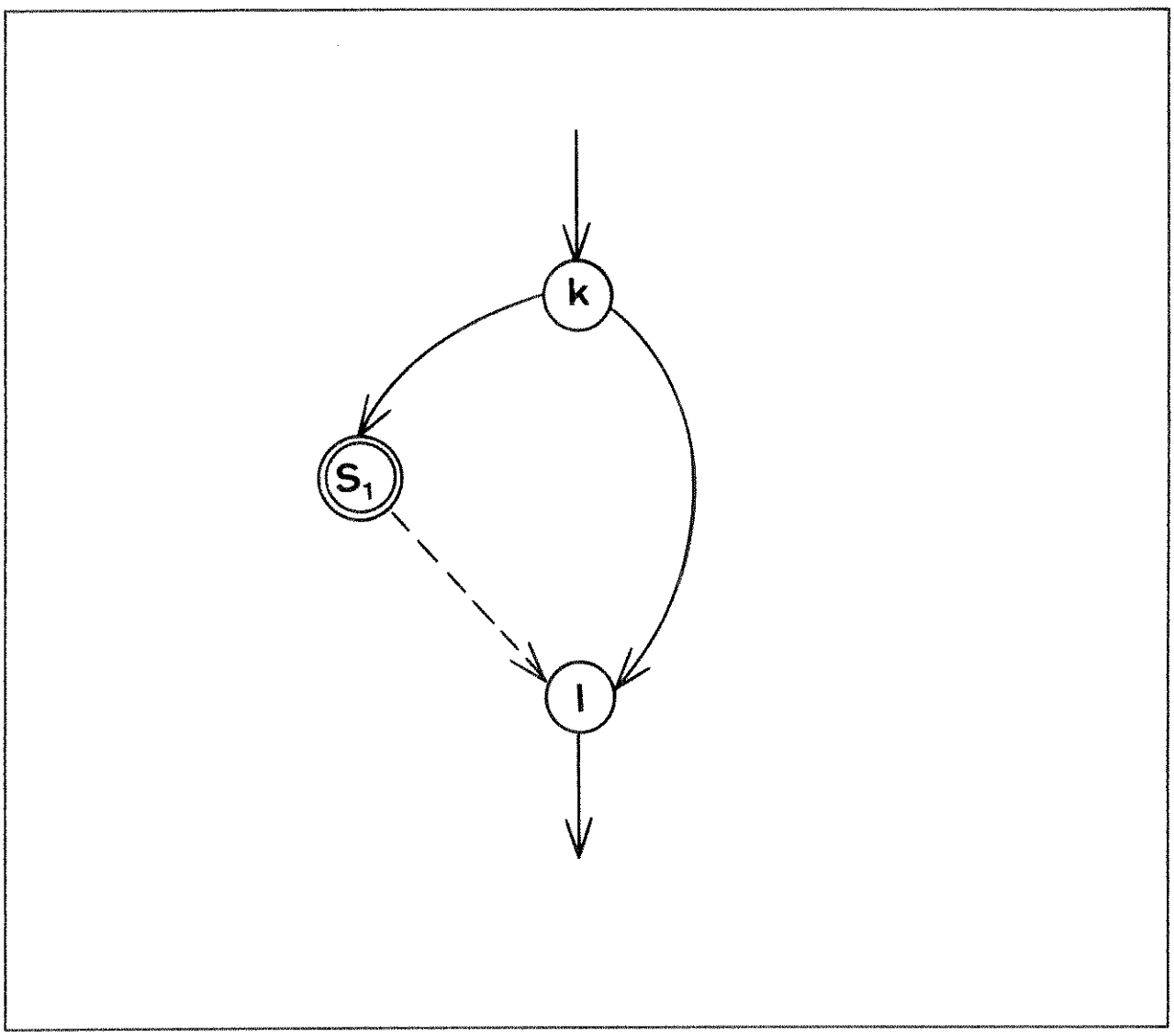


Figura 4.23 - Grafo de Fluxo de Controle: Comando IF Lógico.

Tabela 4.7 - Equivalência Sintática: Comando IF Lógico.

IF	$IF
(expressão)	$C
comando	⟨átomo⟩

teste não deverá ter rótulos dentro dessa faixa, quando submetida a teste.

Para completar a instrumentação, insere-se uma chamada à subrotina PROVA logo após o comando CONTINUE, destino do comando GO TO inserido para a instrumentação.

EXEMPLO [HEH86] :

```
IF ( A .AND. B ) RESTO = 0.0
```

Na Figura 4.24 é mostrado o mapeamento para a LI do Exemplo de comando IF lógico e na Figura 4.25 sua instrumentação.

$IF	89	2	5
$C(01)01	92	13	5
S03	106	11	5

Figura 4.24 - Mapeamento para a LI do Exemplo de Comando IF Lógico.

```
                CALL PROVA (2)
                ..........

      *  NO' 2  *
                IF ( .NOT. ( A .AND. B ) ) GO TO 7301
                CALL PROVA (3)
                RESTO = 0.0
      7301      CONTINUE
                CALL PROVA (4)
```

Figura 4.25 - Instrumentação do Exemplo de Comando IF Lógico.

COMANDO: IF bloco

SINTAXE:

```
IF ( expressão ) THEN
< comandos >
ENDIF
```

OU

```
IF ( expressão ) THEN
< comandos >
ELSE
< comandos >
ENDIF
```

OU

```
IF ( expressão ) THEN
< comandos >
ELSE IF ( expressão ) THEN
< comandos >
ELSE
< comandos >
ENDIF
```

SEMÂNTICA:

Dos comandos IF disponíveis em FORTRAN 77, os comandos IF bloco são os mais adequados para se construir programas estruturados.

Os comandos que devem ser executados se a expressão lógica for verdadeira seguem o THEN e são chamados de *bloco IF* ; os comandos que devem ser executados se a expressão lógica for falsa seguem o ELSE e são chamados *bloco ELSE*. O ELSE e o bloco ELSE

podem ser omitidos se nenhum processamento é para ocorrer se a expressão for falsa. O ELSE, quando usado, deverá ser o único comando na linha. Cada IF... THEN... [ELSE]...deve terminar com um ENDIF [DAV88].

O comando ELSEIF (ou ELSE IF) é usado para programar uma seqüência de testes dentro de um IF bloco. A expressão lógica é testada primeiro; se verdadeira, os comandos para a condição do IF são executados e os comandos restantes até o ENDIF são ignorados. Se falsa, a primeira expressão de ELSEIF é testada. Se ela for verdadeira, os comandos para o ELSEIF são executados e os comandos restantes até o ENDIF são ignorados. Se as expressões lógicas dos comandos IF e ELSEIF forem todas falsas, os comandos do bloco ELSE, se existir, são executados. O ELSEIF é parte do bloco IF e não exige um ENDIF separado. A Figura 4.26 mostra o grafo de fluxo de controle representativo do comando IF bloco.

MAPEAMENTO PARA A LI:

O comando IF é mapeado para o comando $IF da linguagem intermediária. O delimitador ELSE é mapeado para "}", "ELSE" e "{" da linguagem intermediária. Já os delimitadores THEN e ENDIF são mapeados para "{" e "}", respectivamente. A Tabela 4.8 apresenta a equivalência sintática entre os comandos das linguagens FORTRAN e LI.

OBSERVAÇÃO:

Para cada ELSE IF (ou ELSEIF) que a unidade de programa em teste possua, uma chave (}) é adicionada ao ser encontrado o respectivo ENDIF. Isto é feito pois o comando ELSEIF é mapeado para os átomos $ELSE e $IF da LI. Para este novo IF é necessária a inserção de um ENDIF. Durante a instrumentação, ao ser encontrada esta "}", um comando ENDIF é gerado. Esta solução visa possibilitar a monitoração do nó a que pertence o ELSEIF.

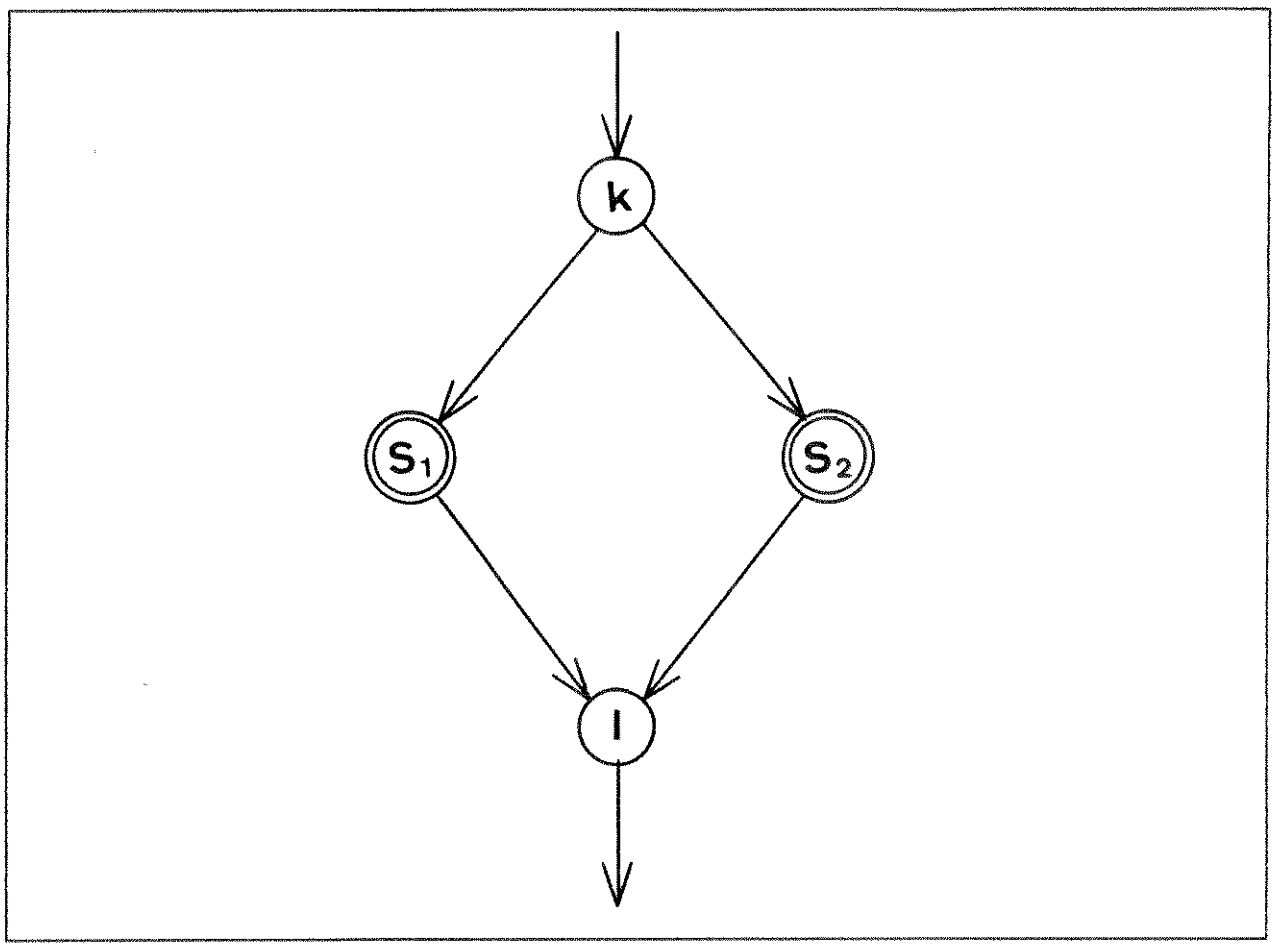


Figura 4.26 - Grafo de Fluxo de Controle: Comando IF Bloco.

Tabela 4.8 - Equivalência Sintática: Comando IF Bloco.

IF	$IF
(expressão)	$C
THEN	{
< comandos >	$S1
ELSE	},$ELSE,{
ELSEIF	},$ELSE,{,$IF
ENDIF	}

INSTRUMENTAÇÃO:

Sejam $i1$, $i2$,..., in os nós de entrada dos subgrafos $S1$, $S2$,..., Sn, respectivamente. No início dos blocos k, $i1$, $i2$,..., in, e l são inseridas chamadas à subrotina PROVA que adiciona pontas de prova à unidade em teste. Estas pontas de prova caracterizam a execução dos blocos, ou seja, pontas de prova k, $i1$, $i2$,...,in e l.

EXEMPLO [CER87]:

```
          IF (PONT2.NE.1)
$         THEN
            IF (PONT2.LE.6)
$           THEN
              NUMVOG = NUMVOG +1
            ELSE IF (PONT2.LE.27)
$                 THEN
                    NUMCOM = NUMCOM + 1
                  ELSE IF (PONT2.LE.37)
$                       THEN
                          NUMDIG = NUMDIG + 1
                        ELSE
                          NUMPOM = NUMPON + 1
            ENDIF
          ENDIF
```

A Figura 4.27 mostra o mapeamento para a LI do Exemplo de comando IF bloco. Ja' a Figura 4.28 mostra a instrumentação para o mesmo Exemplo.

$IF	1181	2	47
$C(01)04	1184	12	47
{	1209	4	48
$IF	1228	2	49
$C(01)05	1231	12	49
{	1258	4	50
$S13	1279	18	51
}	0	0	0
$ELSE	1312	4	52
{	0	0	0
$IF	1317	2	52
$C(01)06	1320	13	52
{	1353	4	53
S14	1379	19	54
}	0	0	0
$ELSE	1418	4	55
{	0	0	0
$IF	1423	2	55
$C(01)07	1426	13	55
{	1464	4	56
$S15	1495	19	57
}	0	0	0
$ELSE	1539	4	58
{	0	0	0
$S16	1570	19	59
}	1604	5	60
}	1622	5	0
}	1622	5	0
}	1622	5	61

Figura 4.27 - Mapeamento para a LI do Exemplo de Comando IF Bloco.

```
                CALL PROVA (11)
*  NO' 11    *
                IF (PONT2.NE.1)
     $          THEN
                   CALL PROVA (12)
*  NO' 12    *
                   IF (PONT2.LE.6)
     $             THEN
                      CALL PROVA (13)
*  NO' 13    *
                      NUMVOG = NUMVOG +1
*  NO' 14    *
                   ELSE
                      CALL PROVA (14)
*  NO' 14    *
                      IF (PONT2.LE.27)
     $                THEN
                         CALL PROVA (15)
*  NO' 15    *
                         NUMCOM = NUMCOM + 1
*  NO' 16    *
                         ELSE
                            CALL PROVA (16)
*  NO' 16    *
                            IF (PONT2.LE.37)
     $                      THEN
                               CALL PROVA (17)
*  NO' 17    *
                               NUMDIG = NUMDIG + 1
*  NO' 18    *
                            ELSE
                               CALL PROVA (18)
*  NO' 18    *
                               NUMPOM = NUMPON + 1
                            ENDIF
                         CALL PROVA (19)
*  NO' 19    *
                      ENDIF
                   CALL PROVA (20)
*  NO' 20    *
                   ENDIF
                CALL PROVA (21)
*  NO' 21    *
                ENDIF
```

Figura 4.28 - Instrumentação do Exemplo de comando IF bloco

4.3.5 INSTRUMENTAÇÃO DE COMANDOS ROTULADOS

Os números de comandos ou rótulos da linguagem FORTRAN consistem de um a cinco dígitos, um dos quais deve ser diferente de zero. Esses números são usados para identificar comandos.

A instrumentação de comandos rotulados pode se dar de três formas distintas:

a) *quando o comando rotulado é o comando terminal do laço do DO;*

Neste caso a linha de comando é apresentada sem alteração. Entretanto, imediatamente antes da linha de comando rotulada, insere-se uma *ponta de prova* que possibilita a monitoração do nó a que pertence o comando rotulado.

Exemplo [CAL89]:

```
      DO 10, I = 1,N
         READ *,NUM
10    ISUM = ISUM + NUM**2
```

Instrumentação do Exemplo:

```
                        CALL PROVA (2)
                .....................................
*     NO'  2  3  5 *
         DO 10 I = 1,N
                                CALL PROVA (3)
                                CALL PROVA (4)
*     NO'  4  *
         READ *,NUM
                                CALL PROVA (5)
*     NO'  5  *
10       ISUM = ISUM + NUM**2
                              CALL PROVA (3)
                              CALL PROVA (6)
*     NO'  6  *
```

Uma situação especial tratada é aquela na qual ocorre um desvio, dentro do laço do "DO", que promove a transferência do fluxo de controle para o terminal do laço. A monitoração do terminal do laço se dá antes da execução do comando de desvio.

Exemplo:

```
      DO 100 X = -5.4, -5.05, 0.1
      ..........................

      ..........................
      GO TO 100
      ..........................

      ..........................
  100 SOCOS = COS(X) + SOCOS
```

Instrumentação do Exemplo:

```
* NO' 4  5  20 *
      DO 100 X = -5.4, -5.05, 0.1
      CALL PROVA(5)
      CALL PROVA(6)
      ..........................
      ..........................
* NO' 9 *
      CALL PROVA(9)
      CALL PROVA(15)
      GO TO 100
      ..........................
      ..........................
      CALL PROVA(15)
* NO' 15 *
  100 SOCOS = COS(X) + SOCOS
      CALL PROVA(5)
      CALL PROVA(16)
```

b) *Quando o comando rotulado não é o comando terminal do laço do DO.*

Neste caso a instrumentação se dará através da modificação do código fonte. O comando que na unidade fonte aparece após o rótulo é substituido pela *ponta de prova* que monitora aquele nó e esse comando passa a figurar na linha seguinte.
Exemplo:

```
100 A = B + C
```

Instrumentação do Exemplo:

```
100 CALL PROVA(8)
    A = B + C
```

c) *Quando o comando rotulado é uma especificação FORMAT.*

Neste caso o comando rotulado não é monitorado. Ao contrário do que acontece com todos os outros comandos rotulados, o FORMAT é um comando não executável. Portanto não justifica que configure um novo nó. Quando da tradução para a LI, toda a linha de comando relativa à especificação FORMAT é representada pelo átomo $DCL.

Exemplo:

```
200 FORMAT (10X,5F8.2)
```

Instrumentação do Exemplo:

```
200 FORMAT (10X,5F8.2)
```

4.3.6 INSTRUMENTAÇÃO DO COMANDO STOP

O comando STOP é um comando executável que termina o processamento do programa. Ele pode, exibir tambem uma informação em terminal para auxiliar a identificação da razão da parada.

Para a instrumentação correta da unidade fonte, todas as chamadas à *ponta de prova* devem ser feitas antes do comando STOP que é o último comando executável. Portanto, no programa instrumentado, antes do comando STOP devem aparecer o CALL PROVA(nó) relativo a monitoração do comando STOP, o CALL PROVA(nó) relativo a monitoração do comando END e o comando que executa o fechamento do arquivo PATH.TES.

Exemplo:

```
      IF (NOME .EQ. 'FIM') STOP
```

Instrumentação do Exemplo:

```
      CALL PROVA(10)
*  NO' 10 *
      IF (.NOT.(NOME.EQ.'FIM')) GO TO 7304
      CALL PROVA(11)
      CALL PROVA(29)
      CALL PROVA(30)
      STOP
 7304 CONTINUE
```

A mesma solução de instrumentação dada ao comando STOP é dada ao comando RETURN que é responsável pelo término de execução de subrotinas.

4.4 CONSIDERAÇÕES FINAIS

Foi apresentada a especificação dos requisitos necessários para a aplicação dos critérios Potenciais Usos a unidades de programas escritas na linguagem FORTRAN-77.

Os critérios Potenciais Usos associam o tipo de ocorrência de variáveis à estrutura lógica do programa. Para configurar a POKE-TOOL para FORTRAN-77 foram caracterizados os modelos de fluxo de dados, instrumentação e fluxo de controle da linguagem.

Não foi trivial a tarefa de especificação dos modelos de implementação da POKE-TOOL para FORTRAN-77. Quando da configuração do modelo de fluxo de controle identificou-se a necessidade de se incluir ao conjunto de átomos da LI os átomos $GT e $ROTALL (Capítulo 2) para tratar, principalmente, os comandos não estruturados da linguagem. Na configuração do modelo de instrumentação foi necessária a modificação do código fonte para alguns comandos FORTRAN.

Os modelos especificados caracterizam de uma forma fiel a linguagem e possibilitam o tratamento correto de unidades de programas FORTRAN-77 pela POKE-TOOL. Esse modelos foram implementados como descrito no Capítulo 5.

CAPÍTULO
5

CONFIGURAÇÃO DA POKE-TOOL PARA A LINGUAGEM FORTRAN-77

Neste capítulo apresentam-se os principais aspectos da implementação da ferramenta POKE-TOOL/versão FORTRAN, para a aplicação dos critérios de testes estruturais Potenciais Usos (PU) [MAL91] a programas escritos na linguagem FORTRAN 77. A implementação foi desenvolvida segundo a relação de tarefas apresentadas no Capítulo 2 (Seção 2.5).

Como descrito no Capítulo 2, no processo de configuração obtém-se o grafo de fluxo de controle (grafo de programa) da unidade em teste e, como subprodutos, a versão em LI (Linguagem intermediária), a versão instrumentada da unidade e o grafo def da unidade sob teste. Para a execução deste trabalho é necessário o conhecimento das construções da linguagem e a capacidade para mapeamento destas para átomos da LI. A base para o mapeamento das instruções (comandos) da linguagem FORTRAN 77 para a LI é a identificação dos comandos que alteram o fluxo de controle. Isso é feito por meio da distinção dos comandos da linguagem em três classes, a saber: declarações, comandos sequenciais e comandos que alteram o fluxo de controle do programa. A maior parte dos comandos da linguagem intermediária pertencem à classe de comandos que alteram o fluxo de controle.

A tradução da linguagem FORTRAN para a LI não é inteiramente fiel, dada a abordagem adotada para a análise léxica e sintática na POKE-TOOL. A sensibilidade ao contexto, característica marcante da linguagem, não permite a representação completa da sua gramática por meio de uma BNF.

As palavras-chave da linguagem FORTRAN são usadas em menos de uma dezena de tipos de sentenças. As sentenças em FORTRAN são chamadas comandos. Um programa consiste de uma sequência de comandos.

No contexto desta dissertação, as palavras-chave da linguagem FORTRAN-77 são tratadas como "palavras reservadas", apesar dos compiladores assim não o considerarem.

O FORTRAN é uma linguagem simples, porém precisa. Cada comando tem uma forma muito bem definida.

Considerou-se neste trabalho que as unidades de programa submetidas a teste por esta versão da ferramenta já foram compiladas, sem erros, por um compilador FORTRAN-77. Os compiladores para a linguagem FORTRAN-77 são capazes de diagnosticar a maioria das violações de regras gramaticais: entretanto, algumas combinações sintáticas e semânticas podem não ser diagnosticadas por algum compilador, ou somente serem detectadas na fase de execução, como mencionado no Capítulo 3.

5.1 PASSOS NA GERAÇÃO DA LI

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

Os itens léxicos, reconhecidos e classificados, são tratados por um analisador sintático. Esse analisador 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 do programa fonte.

O reconhecimento da estrutura do programa é essencial para o processo de geração do grafo de programa, pois com este é possível a identificação: dos comandos que alteram o fluxo de execução, dos pontos de entrada e saída e construções do programa fonte que porventura estejam em desacordo com a especificação.

Feito o reconhecimento da estrutura do programa, o próximo passo é a representação das informações obtidas sobre o fluxo de execução do programa analisado. Essa representação deve possuir informações suficientes para a construção do grafo de programa.

5.2 ITENS SINTÁTICOS DO FORTRAN

Segundo Hehl [HEH86] os itens sintáticos básicos da linguagem FORTRAN-77 são: constantes, nomes simbólicos, variáveis, números de comandos, palavras-chave, operadores e caracteres especiais. Letras, dígitos e caracteres especiais são usados para formar itens sintáticos da linguagem.

De uma maneira sucinta, é a seguinte a descrição dos itens sintáticos do FORTRAN-77:

Constantes - são valores ou números que ocorrem num programa FORTRAN.

Nomes simbólicos - são representados por sequências de um a seis letras ou dígitos, o primeiro dos quais devendo ser uma letra.

Variáveis - são nomes simbólicos atribuídos a localizações específicas de memória para uma unidade de programa FORTRAN.

Números de comandos - consistem de um a cinco digitos, um dos

quais devendo ser diferente de zero. São usados como rótulos para identificação de comandos.

Palavras-chave - são sequências específicas de letras utilizadas para a construção dos comandos FORTRAN.

Operadores - são os operadores aritméticos, de caracter, relacionais e lógicos.

Caracteres especiais - podem ser operadores, ou terem significados especiais dependendo do contexto.

A construção de comandos em FORTRAN é feita utilizando-se o conjunto de seus caracteres que é constituído de : caracteres alfabéticos (letras), caracteres numéricos (dígitos) e caracteres especiais (= + - * / ( ) . , ' " $: ! > < % &).

Para a geração da Linguagem Intermediária representativa de unidades de programação escritas em FORTRAN-77, os programas fontes devem estar isentos de erros de sintaxe, isto é, passaram por um processo de compilação sem apresentar erros.

A tradução do programa fonte, em FORTRAN-77, para a linguagem intermediária é feita por meio de dois analisadores, um léxico e outro sintático; o primeiro baseado em autômatos finitos [SET81] [CHA91a], o segundo dirigido por um grafo sintático [WIR76] [CHA91b], ambos orientados por tabelas.

5.3 ANALISADOR LÉXICO PARA O FORTRAN-77

A análise léxica é o primeiro passo na compilação ou análise do código fonte de programas. A função do analisador léxico (AL) resume-se a varrer o programa fonte da esquerda para a direita, agrupando os símbolos de cada item léxico e determinando sua

classe [SETB1].

Devido ao seu aspecto multilinguagem, foi escolhido um algoritmo genérico de análise léxica para a POKE-TOOL.

Uma forma de se implementar um algoritmo genérico para a análise léxica pode ser representada por meio de uma máquina abstrata, chamada Autômato de Estados Finitos, abreviado por AF. Esse algoritmo interpreta um autômato finito cujas transições são organizadas numa estrutura de dados criada através de uma entrada do usuário. A interpretação do autômato é dirigida pela estrutura de dados que representa as transições e por caracteres lidos do programa fonte. Estão associadas às transições do autômato ações semânticas, que são executadas quando uma determinada transição ocorre. O objetivo da utilização de rotinas semânticas é a simplificação do autômato.

O analisador léxico é orientado por tabelas. No contexto da POKE-TOOL, para se implementar um analisador léxico para unidades de programas escritas em FORTRAN-77 ou qualquer outra linguagem procedural, são necessários uma tabela de transições léxicas, uma tabela de palavras reservadas e um conjunto de rotinas que estabelecem ações semânticas léxicas.

Na configuração do analisador léxico da POKE-TOOL para a linguagem FORTRAN-77 foram adotadas as classes: IDENTIFICADORES, CONSTANTES, CADEIA DE CARACTERES, OPERADORES LÓGICOS, OPERADORES RELACIONAIS, FORMAT, TEXTO, PALAVRAS RESERVADAS e SÍMBOLOS ESPECIAIS (e.g.,+,=,*,etc.).

Na classe IDENTIFICADORES representada no AL por IDENT estão os nomes simbólicos e variáveis. Excluem-se dessa classe as variáveis do tipo caracter, pois estas constituem a classe CADEIA DE CARACTERES. Em uma primeira avaliação, as palavras-chave também se encontram nesta classe. Após a consulta da tabela de palavras reservadas, por uma ação semântica, havendo casamento, a classe deste identificador passa a ser a própria palavra-chave.

Na classe CONSTANTES representada no AL por CONST estão todas as constantes consideradas pelo FORTRAN-77, inteiras, reais, dupla

precisão, complexas e hexadecimais.

Na classe CADEIA DE CARACTERES estão as constantes com tipo de dado caracter representadas no AL por CARACTER.

Os símbolos especiais tais como operadores aritméticos e o de caracter são representados em classes identificadas pelos próprios operadores, ou sejam : +, -, *, /, **, //.

Os operadores lógicos são representados pelos próprios : .AND. , .NOT. , .OR. , .EQV. e .NEQV. .

Da mesma forma, os operadores relacionais são representados por: .EQ. , .NE. , .LT. , .LE. , .GT. e .GE. .

Uma exceção ocorre no caso do comando FORMAT. Uma vez identificada a palavra-chave FORMAT, gera-se o item léxico FORMAT com classe FORMAT. Os itens léxicos que aparecem entre os parenteses após o comando FORMAT são agrupados e passam a ter a classe TEXTO.

O algoritmo utilizado, é o proposto por Setzer e Melo [SETB1], com algumas modificações propostas por Chaim descritas em [CHA91a].

No Apêndice C são mostradas a tabela de transições léxicas, a tabela de palavras reservadas, e as ações semânticas adotadas pelo analisador léxico e utilizadas para separar os itens léxicos das unidades de programas escritos em FORTRAN-77.

Os nomes dos arquivos utilizados para a análise léxica de unidades de programas em FORTRAN-77 são:

TABTRANS.FOR -> tabela de transições léxicas

ACSEM.C -> ações semânticas (codificadas na linguagem "C")

TABCHAVE.FOR -> tabela de palavras-chaves.

5.4 ANALISADOR SINTÁTICO PARA O FORTRAN-77

De maneira semelhante ao enfoque adotado no analisador

léxico, utiliza-se um algoritmo que dirige o processo de análise sintática através de uma estrutura de dados. Esse algoritmo é mais adequado para análise de linguagens procedimentais livres de contexto. Entretanto, para o caso do FORTRAN-77, o algoritmo é ainda utilizado e algumas restrições são consideradas. Essas restrições são descritas na Seção 5.6.

A forma utilizada para representar a gramática da linguagem adotada na POKE-TOOL é a de grafos sintáticos proposta por Wirth [WIR76] e adaptada por Chaim [CHA91a]. Essa notação permite a visualização e a representação do fluxo de controle durante o processo de análise sintática.

O grafo sintático utilizado para a análise de unidades de progamas escritos na linguagem FORTRAN é colocado na memória através da leitura de um arquivo - TABSIN.FOR - que contém a descrição da gramática a ser reconhecida. Esse arquivo é descrito utilizando-se a notação estendida de Backus-Naur apresentada em [SEB89]. Em [CHA91a] são apresentadas as ressalvas, limitações e diferenças encontradas entre o algoritmo proposto por Wirth e o implementado na POKE-TOOL.

O arquivo TABSIN.FOR, diferentemente da notação Backus-Naur, presta-se também para indicar as rotinas semânticas que são executadas quando um dado item de uma produção é reconhecido. Essas rotinas são chamadas rotinas semânticas [SET86].

As rotinas semânticas são as responsáveis pela identificação das variáveis que são definidas em um dado comando. Essas rotinas, analogamente às ações semânticas do analisador léxico , utilizam estruturas de dados e funções padronizadas da POKE-TOOL, que auxiliam no desenvolvimento do código propriamente dito dessas rotinas. Os padrões da POKE-TOOL são obedecidos para a geração das rotinas semânticas [CHA91a].

O arquivo que contém as rotinas semânticas para o FORTRAN-77 é um arquivo codificado na linguagem "C", "rotsem.c".

O método de análise sintática utilizado por Wirth é o da chamada análise sintática descendente. Esse método consiste na

tentativa de reconstruir os passos que geraram a sentença, do símbolo inicial até a sentença final.

O algoritmo de Wirth é considerado com um símbolo de "*lookahead*" e sem "*backtracking*". Essas características tornam o algorítmo genérico e também eficiente. Isso significa que todo passo da análise depende unicamente do estado atual da computação e do próximo símbolo a ser lido. Significa também que nenhum passo já executado precisa ser recuperado durante o processo de análise.

Em suma, o processo de reconhecimento da sentença é realizado tomando-se apenas o próximo símbolo da cadeia de entrada para decidir qual o próximo passo a ser dado. Além disso, não é necessário retornar a nenhum passo anterior para reconhecimento da sentença.

Tornava-se complicada a tarefa de com apenas um símbolo de "lookahead" e sem "backtracking", proceder-se à análise sintática de uma linguagem sensível ao contexto. Houve a necessidade de se introduzir um novo ítem léxico, o "NEW_LINE", para indicar o final de cada linha gerada, para a posterior análise pelo A.S. .

Essa solução facilitou bastante a construção da gramática -TABSIN.FOR (Apêndice D) - que foi, então, totalmente reformulada. Com o recurso de identificação de nova linha (ou fim da linha anterior), para a maioria dos comandos em FORTRAN, passou ser apenas necessária a análise do primeiro "token". Esta solução é esclarecida nos exemplos abaixo.

Para a linha de comando em FORTRAN

```
DATA PI / 3.1415926 /
```

os seguintes "tokens" são gerados pelo analisador léxico

DATA	DATA	105	4	5
PI	IDENT	110	2	5
/	/	113	1	5
3.1415926	CONST	115	9	5
/	/	125	1	5
	NEW_LINE	126	0	6

Para efeito de geração da L.I., ao ser identificada a palavra chave "DATA", já é possível afirmar que se trata de uma declaração e será traduzida para "$DCL". O "token" "NEW_LINE" serve para a identificação da posição onde termina a declaração.

Tem-se abaixo o trecho da gramática (TABSIN) que identifica o comando DATA como um comando sequencial:

```
data = 'DATA' 5 token_0 [ token_0 ] 'NEW_LINE' 2 .

token_0 = 'CONST' , 'IDENT' , 'CARACTER' , log_const ,
          '=' , '+' , '-' , '*' , '**' , '//' , '(' ,
          ')' , ',' , ':' , '<' , '>' , '/=' .
```

Note-se que, identificado o "token" 'DATA', qualquer "token" passível de aparecer no comando DATA é aceito até a identificação do token 'NEW_LINE' gerado pelo A.L.. Esta solução simplificou bastante a gramática que não necessita representar todas as construções permitidas para o comando DATA pela linguagem FORTRAN. Os números 5 e 2 indicam as ações semânticas que são chamadas quando da identificação dos tokens 'DATA' e 'NEW_LINE'.

Como resultado temos em L.I.:

```
$S        105        21        5
```

Outro exemplo mostra mais um tipo de comando da linguagem FORTRAN que teve sua identificação pela gramática simplificada. Na linha

```
REAL A(50,50), B(50), Z(50), T, RCOND
```

o analisador léxico gera os "tokens":

REAL	REAL	129	4	6
A	IDENT	134	1	6
(	(	135	1	6
50	CONST	136	2	6

,	,	138	1	6
50	CONST	139	2	6
)	)	141	1	6
.	.	.	.	.
.	.	.	.	.
.	.	.	.	.
RCOND	IDENT	154	5	6
	NEW_LINE	159	0	7

Apresenta-se abaixo o trecho da gramática que reconhece a declaração REAL, entre outras apresentadas no Apêndice D:

```
declaracao = dcl_aux [ token_0 ] fun_opt
fun_opt = 'FUNCTION' 'IDENT' [ token_0 ] 'NEW_LINE' 3 ,
          'NEW_LINE' 1 .
dcl_aux = 'REAL' 5 ,
          'EXTERNAL' 5 ,
.....................................
```

A declaração REAL será traduzida para a L.I. como:

$DCL	129	37	6

Mais um exemplo é apresentado para mostrar a solução dada para comandos do tipo READ, WRITE, PRINT. Nesses comandos, além de ser necessária a identificação da palavra-chave READ, também é necessária a busca das palavras-chave END e/ou ERR para decidir se a tradução da linha se dará para um $S ou $GT. O "token" "NEW_LINE" indica o limite dos tokens que pertencem à linha.

Na linha de comando

```
10      READ (*,20) LINHA
```

a análise léxica resultará em:

10	COMANDO	467	2	20
READ	READ	475	4	20
(	(	480	1	20
*	*	481	1	20

,	,	482	1	20
20	CONST	483	2	20
)	)	485	1	20
LINHA	IDENT	487	5	20
	NEW_LINE	492	0	21

o analisador sintático gerará:

10	467	2	20
$S02	475	17	20

Já para o comando READ contendo a cláusula END que é apresentado abaixo

```
10      READ( *, 20, END=200) LINHA
```

sua análise léxica resultará em

10	COMANDO	339	2	14
READ	READ	345	4	14
(	(	349	1	14
*	*	350	1	14
,	,	351	1	14
20	CONST	352	2	14
,	,	354	1	14
END	END	355	3	14
=	=	358	1	14
200	CONST	359	3	14
)	)	362	1	14
LINHA	IDENT	364	5	14
	NEW_LINE	369	0	15

e após a análise sintática teremos:

10	339	2	14
$GT	345	24	14
$C	355	4	14
(	0	0	0
200	359	3	14
$ROTD	0	0	0
)	0	0	0

O gramática utilizada para analisar sintáticamente os

comandos READ, WRITE e PRINT é a seguinte:

```
read_write = rwp_a1 token_rwp [ token_rwp ] 'NEW_LINE' 26 .
rwp_a1 = 'READ' 5 ,
         'WRITE' 5 ,
         'PRINT' 5 .
   token_rwp = io_err , io_end , token_1 .
       token_1 = 'CONST' , 'IDENT' , 'CARACTER' ,
                 '(' , ')' , '*' , ',' , '=' , ':' .
       io_end = 'END' 31 '=' 37 'CONST' 27 .
       io_err = 'ERR' 31 '=' 37 'CONST' 27 .
```

O Apendice D mostra toda a estrutura de dados que representa o grafo sintático da linguagem FORTRAN-77.

Os analisadores léxico e sintático compõem o módulo LI. Portanto, uma vez implementados, pode-se então realizar a tradução de unidades de programa escritas em FORTRAN-77 para uma nova versão escrita na Linguagem Intermediária.

A partir da versão em LI, o módulo **CHANOMAT** gera o grafo de fluxo de controle da unidade e uma nova versão, a NLI. Esta versão é aquela onde cada comando está associado ao nó correspondente.

5.5 MÓDULO POKERNEL

O responsável pelo restante da análise estática do código fonte é o módulo **POKERNEL**. O módulo POKERNEL tem as seguintes funções: Cálculo dos Arcos Primitivos, Extensão do Grafo de Fluxo de Controle, Instrumentação, Construção do Grafo (i) e Geração dos Descritores.

Das funções do módulo POKERNEL apenas a extensão do Grafo de Fluxo de Controle e a Instrumentação são dependentes da linguagem

alvo e, portanto, necessitam ser configuradas.

A configuração do módulo POKERNEL baseou-se nos modelos apresentados no Capítulo 4.

Na configuração do módulo POKERNEL para a linguagem FORTRAN-77 definiu-se um novo grafo sintático representativo da linguagem. Este grafo possui informações que provocam a chamada de rotinas que tratam a definição de variáveis existentes no código fonte, de acordo com o modelo de fluxo de dados descrito no Capítulo 4. Criaram-se novas rotinas e funções e também introduziram-se modificações em algumas das rotinas já existentes na versão configurada para a linguagem "C".

No Apêndice F são apresentadas, a título de ilustração algumas das rotinas desenvolvidas.

5.6 PRÉ-PROCESSADOR DA UNIDADE EM FORTRAN-77

A generalidade é uma das características desejáveis que ferramentas devem possuir para atender à crescente demanda de ambientes automatizados para desenvolvimento de software.

Procurando atingir este objetivo, a POKE-TOOL [CHA91] é dotada de aspectos multilinguagem, o que lhe confere a característica de generalidade.

Apesar da linguagem de programação FORTRAN-77 ser uma linguagem procedimental, ela tem significativas diferenças em relação a outras, também procedurais, como "C", "PASCAL", "ALGOL", etc. . FORTRAN-77 não é uma linguagem livre do contexto, além de permitir algumas construções que podem gerar ambiguidades.

A solução proposta para possibilitar a configuração da POKE-TOOL para o FORTRAN-77 envolve o pré-processamento das unidades de teste codificadas em FORTRAN-77.

O objetivo deste pré-processamento reside em relaxar as

restrições quanto à liberdade de codificação na linguagem FORTRAN-77 para sua tradução para LI. Assim, o programador não precisa se preocupar em escrever um programa "bem comportado". Como resultado deste pré-processamento, poder-se-ia submeter qualquer programa, que seja passível de compilação por um compilador que esteja de acordo com o padrão ANSI, a teste pela POKE-TOOL.

O pré-processamento de unidades escritas em FORTRAN-77 não faz parte do escopo deste trabalho. Portanto, considera-se que as unidades de testes aceitas por esta configuração da POKE-TOOL trata apenas de unidades codificadas de uma forma um pouco mais restrita.

Algumas soluções que seriam adotadas na implementação do pré-processador foram incorporadas aos analisadores léxico e sintático.

Dentre as soluções incorporadas aos analisadores está a indicação de nova linha, já apresentada. O FORTRAN-77 não possui o caracter ";", indicador de fim de linha como o da linguagem "C". A solução adotada resolve também o problema de identificação de linhas de continuação e linhas de comentário.

Uma unidade de programa está em condições de ser tratada pela presente configuração da POKE-TOOL se possuir as seguintes características:

i) constantes completamente definidas em uma única linha, isto é, não iniciam em uma linha e continuam em outra;

ii) palavras-chave consideradas como "palavras reservadas", não sendo permitido seu uso como identificadores;

iii) palavras-chave sempre seguidas de espaço em branco;

iv) as palavras UNIT, FILE, STATUS,..., que aparecem nos comandos READ, WRITE, OPEN,... consideradas como "palavras reservadas".

Pode-se notar que as restrições impostas devido à ausência de um pré-processador são pouco relevantes se boas práticas de programação forem utilizadas.

5.7 RESULTADO DA APLICAÇÃO DA POKE-TOOL A UNIDADES DE PROGRAMA EM FORTRAN-77

As Tabelas 5.1 e 5.2 apresentam o resultado da análise relativa à aplicação dos critérios Potenciais Usos, utilizando-se a versão FORTRAN-77 da ferramenta de teste POKE-TOOL. As unidades de programa em FORTRAN-77 selecionadas para este experimento foram retiradas de [CER87], [FAR86], [CAL89], e [RIC83] . Visou-se alcançar três objetivos principais:

i) contribuir para avaliar o trabalho de configuração para FORTRAN-77 da ferramenta;

ii) auxiliar na demonstração da factibilidade de aplicação dos critérios Potenciais Usos a programas em FORTRAN-77;

iii) investigar o esforço de aplicação de testes a programas em FORTRAN-77 com a utilização de uma ferramenta.

Na execução deste experimento procurou-se seguir o que foi definido em [MAL91] e que poderá vir a se tornar um padrão para a elaboração de "Benchmarks" para as configurações da POKE-TOOL.

Como salientado em [MAL91], são vários os pontos que influenciam o número de casos de teste requeridos por um critério baseado em fluxo de dados; por exemplo, o número de comandos de decisão tem uma forte influência, assim como as definições e (potenciais) usos ao longo do programa. Outros pontos são de relevância, tais como o tipo de comando (IF, DO,...), a ordem de ocorrência dos comandos, o número de definições e a sequência de suas ocorrências, entre outros. Por exemplo, pode-se elaborar programas com distribuição adequada de ocorrências de variáveis com o objetivo de maximizar o número de elementos requeridos, assim como, pode-se distribuir estas ocorrências de forma a minimizar esse número.

Na Tabela 5.1 as variáveis de controle são mostradas da seguinte forma: **C1** indica o número de comando de decisão da unidade, **C2** o número de variáveis utilizadas, **C3** o número de

Tabela 5.1: Variáveis de Controle Relativas ao Experimento

UNIDADE	# Comandos de Decisão / # nós (C1/C5)	# Variáveis Utilizadas (C2)	# Definições de Variáveis (C3)	# Nós com definição de Variáveis (C4)
CER80	4/21	6	18	14
CER100	4/17	6	13	10
CER119	4/13	7	13	6
COBAIA	1/7	3	4	3
FAR86	1/5	11	7	2
INCTAX	4/14	5	5	7
MATSOM	2/13	7	11	7
SINCOS	3/16	15	26	10
SUMSQR	2/9	4	6	4

Tabela 5.2: Variáveis Resposta e Associações Requeridas Relativas ao Experimento

UNIDADE	Cardinalidade	# Associações requeridas		
	CICT	CR1	CR2	CR3
CER80	2	67	67	43
CER100	2	60	60	50
CER119	10	42	42	30
COBAIA	1	8	8	6
FAR86	4	3	3	3
INCTAX	3	46	46	36
MATSOM	1	34	34	21
SINCOS	1	36	36	26
SUMSQR	3	12	12	9

definições de variáveis, C4 o número de nós com definição de variáveis e C5 o número de nós do grafo de programa. CCT é o conjunto de casos de teste.

Para a realização do experimento adotaram-se algumas diretrizes, concretizadas nas etapas descritas a seguir.

1) Leitura da descrição funcional da unidade, de detalhes e alternativas que nortearam a implementação das unidades a serem testadas.
2) Elaboração do conjunto inicial de casos de teste (CICT). A minimização dos conjuntos de casos de teste não é almejada. São selecionados casos de teste típicos da aplicação, atômicos, i.e., com propósito único, ao invés de selecionarem-se casos de teste que cobrissem várias características simultâneamente. Procurou-se identificar os casos especiais e valores limites das condições de entrada e saída; no entanto, não ficou caracterizada a aplicação explícita de nenhum critério de teste funcional, tais como "Equivalence Partitioning" ou "Boundary Value Analysis" [PREB7].
3) Implementação da unidade a ser testada utilizando a linguagem de programação FORTRAN-77.
4) Submissão do conjunto CICT à unidade a ser testada, sob controle da POKE-TOOL.
5) Análise de adequação do CICT em relação aos critérios Potenciais Usos (PU).
6) Geração e submissão de novos casos de testes a partir de informações oriundas da análise de cobertura feita pela POKE-TOOL, com o objetivo de cobrir as associações requeridas pelo critério todos-potenciais-usos. Casos de teste são gerados até que todas as asssociações executáveis requeridas sejam exercitadas. Se o conjunto de caminhos e associações ainda a serem exercitados for composto somente de caminhos não executáveis, dá-se por encerrado o teste da unidade de acordo com o critério em

questão: neste caso, diz-se que o Conjunto de Casos de Teste quase satisfaz ("almost satisfies") o critério [WEY90]. Procedimentos semelhantes são adotados com relação aos critérios todos-potenciais-usos/du e todos-potenciais-du-caminhos.

9) Coleta e Organização de Informações para a Análise dos resultados do experimento.

Como resultado imediato do experimento, apresenta-se na Tabela 5.2 uma síntese do que foi obtido, como segue:

- Conjunto Inicial dos Casos de Testes (CICT) e a análise da adequação do CICT para cada um dos critérios, ou seja, a análise de cobertura em relação a cada um dos critérios Potenciais Usos;
- Conjunto final de Casos de Testes para os Critérios **CR1** (CFCT/PU), **CR2** (CFCT/PUDU) e **CR3** (CFCT/PDU);
- Conjunto de associações requeridas para os Critérios **CR1**, **CR2** e **CR3**.

Um caminho completo é executável ou factível se existe um conjunto de valores que possa ser atribuído às variáveis de entrada do programa que causa a execução desse caminho; caso contrário, diz-se que ele é *não executável* [FRA87]. Um caminho é executável se ele for um subcaminho de um caminho completo executável, isto é, se ele for incluído em um caminho completo executável. Uma potencial associação é executável se existir um caminho completo executável que cubra essa associação; caso contrário é não executável.

A determinação da executabilidade de um caminho ou associação é indecidível [FRA87], e é de responsabilidade única do testador, uma vez que a POKE-TOOL, na versão atual, não fornece suporte nesta direção.

Na análise dos resultados obtidos, alguns pontos devem ser considerados quanto aos programas testados e quanto aos resultados obtidos, antes de proceder-se à análise propriamente dita desses resultados. A unidade COBAIA representa um tipo de rotina em

Tabela 5.3 Variáveis Resposta e Associações Requeridas Relativas ao Experimento para o Critério Todos-Potenciais-Usos

UNIDADE	Card.	CRITÉRIO TODOS-POTENCIAIS-USOS		
		Cardinalidade	# Associações	
	CICT	CFCT	não executadas	requeridas
CER80	2	2	10	67
CER100	2	6	5	60
CER119	10	29	7	42
COBAIA	1	1	1	8
FAR86	4	4	0	3
INCTAX	3	7	0	46
MATSOM	1	1	4	34
SINCOS	1	5	4	36
SUMSQR	3	3	1	12

OBS.: As associações não executadas foram identificadas como não executáveis, manualmente.

FORTRAN que não recebe dados externos: portanto, é passível apenas de uma entrada de casos de teste (por exemplo parâmetros pré-definidos). Já a unidade CER119, por se tratar de uma rotina de classificação, exigiu um grande número de casos de teste, pois a combinação dos dados de entrada é relevante para o resultado obtido.

Para os dados coletados, obteve-se a média ponderada M da razão entre o número de comandos de decisão t de um dado programa pelo número de casos de teste, nct, suficiente para satisfazer o critério selecionado (*análise do caso médio empírico*). Expressou-se então nct em função de M, ou seja, $nct = (1/M)t$, onde M é definido por

$$M = (1/n) \sum_{i=1}^{n} (ti/ncti)$$

e n é o número de programas considerados.

Para o *Critério Todos-Potenciais-Usos*, os resultados apresentados na Tabela 5.3 indicam que os programas considerados neste trabalho demandaram entre 0.5*t e 8*t casos de teste. O melhor caso foi o da unidade MATSOM com $nct = 0.5*t$, o pior caso foi o da unidade CER119 com $nct = 7.25*t$. Considerando-se a média ponderada M, tem-se $nct = 1.14*t$.

Maldonado [MAL91] relata os resultados da análise de um benchmark aplicado utilizando a POKE-TOOL/versão C; a título de comparação, para o Critério *Todos-Potenciais-Usos* o resultado obtido para a versão C foi $nct = 1.44t$.

Os resultados obtidos neste trabalho evidenciam que a utilização dos critérios Potenciais Usos em programas de interesse prático, escritos em FORTRAN, e' factível, da mesma forma que o observado para a linguagem C. Obviamente, o universo de programas utilizados é restrito, e o estabelecimento de um benchmark significativo com a finalidade de comprovar essas evidências

deverá ser conduzido.

Um ponto que deve ser destacado é que, a menos de 2 programas, todos os demais contêm caminhos não executáveis (o que pode ser observado na Tabela 5.3). Isso caracteriza a importância de se introduzir recursos para tratamento de caminhos não executáveis na POKE-TOOL/versão FORTRAN-77.

5.8 CONSIDERAÇÕES FINAIS

Neste Capítulo foram apresentados os principais aspectos da configuração da ferramenta POKE-TOOL que possibilitaram a aplicação dos critérios Potenciais Usos a programas em FORTRAN-77.

Foram apresentados os analisadores léxico e sintático que, com as tabelas descritas nos Apêndices C e D constituem o módulo LI. Tratou-se também de descrever os procedimentos necessários para a configuração do módulo POKERNEL. Os módulos LI e POKERNEL são dependentes da linguagem alvo da configuração.

Ao todo, foram desenvolvidas especificamente para esta configuração, 40 rotinas para o analisador léxico, 35 rotinas para o analisador sintático e 19 rotinas para o módulo POKERNEL. Várias outras rotinas foram modificadas. Todas as rotinas e funções desenvolvidas ou modificadas estão codificadas na linguagem "C". No Apêndice F são apresentados alguns exemplos das rotinas desenvolvidas.

Foram apresentadas na Seção 5.6 as restrições impostas às unidades a serem testadas por esta versão da ferramenta POKE-TOOL. Essas restrições poderão servir como base para a construção de um pré-processador para as unidades escritas em FORTRAN.

A validação parcial da versão FORTRAN da POKE-TOOL foi feita através da aplicação de testes às unidades descritas na Seção 5.7.

CAPÍTULO

6

CONCLUSÕES E TRABALHOS FUTUROS

Neste capítulo são apresentadas as conclusões resultantes das atividades de desenvolvimento desta Dissertação, suas principais contribuições e sugestões de trabalhos futuros.

6.1. CONCLUSÕES

Neste trabalho tratou-se, essencialmente, da configuração para a linguagem de programação FORTRAN-77 de uma ferramenta — a POKE-TOOL — que apóia o teste estrutural de software baseado em fluxo de dados, segundo os Critérios Potenciais Usos.

Como resultado, obteve-se uma ferramenta que consiste de um protótipo operacional que possibilita a aplicação de testes a unidades de programas em FORTRAN-77. As unidades de programa podem ser Programas, Subrotinas ou Funções, que serão testadas individualmente, ou ligadas com as outras unidades que compõem um programa FORTRAN.

As atividades de teste conduzidas manualmente são tão sujeitas a erros quanto as demais atividades do desenvolvimento de software. A POKE-TOOL/versão FORTRAN-77 é a primeira ferramenta que se tem notícia que automatiza o teste estrutural baseado em análise de fluxo de dados, para unidades de programa escritas nesta linguagem.

O aspecto multilinguagem da POKE-TOOL teve grande influência na rapidez do desenvolvimento e na confiabilidade dos resultados obtidos. Os analisadores léxico e sintático orientados por tabelas e a grande quantidade de rotinas já desenvolvidas facilitou bastante este trabalho.

Na atividade de configuração da ferramenta POKE-TOOL deve-se desenvolver recursos para a linguagem alvo que possibilitem a determinação dos elementos requeridos pelos critérios e dos elementos efetivamente executados em uma unidade. Isso é feito a partir dos modelos de Fluxo de Controle, de Fluxo de Dados e de Instrumentação. Neste trabalho foi descrita a instanciação desses modelos para a linguagem FORTRAN-77.

Na instanciação dos modelos de implementação dos Critérios Potenciais Usos para a linguagem FORTRAN-77, identificou-se algumas dificuldades, por exemplo: na instanciação do modelo de instrumentação para o FORTRAN-77 não foi possível evitar a modificação do código fonte, diferentemente do sugerido em [MAL91].

No desenvolvimento das atividades de teste e validação da presente configuração foi possível comprovar-se, em casos práticos, a dificuldade de se proceder à aplicação de critérios baseados em análise de fluxo de dados sem um suporte automatizado. Isso é devido à grande quantidade de associações e caminhos a serem testados. As informações fornecidas pela ferramenta constituem um excelente auxílio em termos de confiabilidade e eficiência nos testes. Observou-se que a aplicação dos Critérios Potenciais-Usos em programas escritos em FORTRAN-77 é factível,

como apresentado na Seção 5.7 do Capítulo 5.

Em continuidade a este trabalho seria interessante o desenvolvimento de atividades que visassem a efetiva validação do protótipo ora desenvolvido para assegurar a sua correção e aplicabilidade. Outra atividade que poderia ser executada em continuidade a esta, seria a inclusão à ferramenta de outros critérios de teste estrutural.

Durante as atividades de validação da ferramenta configurada para FORTRAN identificou-se que a caracterização automática dos caminhos não executáveis encontrados em uma unidade de programa — extensão para FORTRAN de [VER92] — seria de grande valia para garantir a qualidade e produtividade na atividade de teste. Esforços deveriam ser dirigidos no sentido de dotar a POKE-TOOL dessa facilidade.

A adaptação desta versão da POKE-TOOL para "dialetos" da linguagem FORTRAN não deverá ser uma tarefa difícil. Poucas deverão ser as modificações necessárias.

Esforços tambem deverão ser dirigidos para atender ao padrão FORTRAN-90 ISO (OIS 1539:1991) que possuí, entre outras características, procedimentos recursivos. Nenhuma das versões anteriores da POKE-TOOL apóia o tratamento de procedimentos recursivos.

BIBLIOGRAFIA

[ANS78] Americam National Standards Institute - "X3.9 - 1978 FORTRAN-77" - U.S.A. - 1978.

[ART88] Arthur, Lowell Jay - "Software Evolution" - Jonh Wiley & Sons, Inc. - U.S.A. - 1988.

[BOE81] Boehm, B.W. - "Software Engineering Economics" - Prentice Hall - U.S.A - 1981.

[BOR87] Borland International Inc. - "Turbo Pascal version 3.0 - Reference Manual" - U.S.A. - 1987.

[BRO75] Brooks, F. - "The Mythical Man-Month" - Addison-Wesley - U.S.A. - 1975.

[BUC84] Buckley, F.J. and Poston, R. -. " Software Quality Assurance" - IEEE Trans. Software Eng. - Vol. SE 10 No. 1 - U.S.A. - Jan. 1984.

[CAL89] Calderbank, V.J. - "Programming in Fortran - Third Edition" - Chapman and Hall - London - U.K. - 1989.

[CAR91] Carnassale, M., "GFC - Uma Ferramenta Multilinguagem para a Geração do Grafo de Programa" - Tese de Mestrado - DCA/FEE/UNICAMP, Campinas, SP, Fev. 1991.

[CER87] Cereda, R.L.D. e Maldonado, J.C. - "Introdução ao FORTRAN 77 para Microcomputadores" - Editora McGraw-Hill - São Paulo - S.P. - 1987.

[CHA91a] Chaim, M.L., Maldonado, J.C. e Jino, M. - "Manual de Configuração da POKE-TOOL" - Relatório Técnico DCA/RT/008/91 - DCA/FEE/UNICAMP - 1991.

[CHA91b] Chaim, M.L., "POKE-TOOL - "Uma Ferramenta para Suporte ao Teste Estrutural de Programas Baseado em Análise Fluxo de Dados", Tese de Mestrado - DCA/FEE/UNICAMP - Campinas, SP - Abril 1991.

[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 - Maio 1987, pp. 509-517.

[COL69] Cole, R. Wade - "Introduction to Computing" - Mc Graw-Hill Book, Inc. - U.S.A. - 1969.

[COW88] Coward, P.D. - "A Review of Software Testing" - Information and Software Technology - Butterworth & Co Ltd. - U.K. - 1988.

[DAV88] Davis, Gordon B.; Hoffmann, Thomas R. - "FORTRAN 77: A Structured, Disciplined Style" - McGraw-Hill Book Company - U.S.A. - 1988.

[DEU82] Deutsch, M.S. - "Software Verification and Validation" - Englewood Cliffs - Prentice-Hall - U.S.A. - 1982.

[EVA87] Evans, Michael W. e Marciniak, John J. - "Software Quality Assurance and Management" - John Wiley & Sons, Inc. - U.S.A. - 1987.

[FAR86] Farouit, S., Simon, D. - "FORTRAN Structuré e Méthodes Numériques" - Bordas - Paris - 1986.

[FOR92] Forte, G. - "Tools Fair: Out of the Lab, Onto the Shelf" - IEEE Software - U.S.A. - Maio - 1992.

[FRA85] Frankl, F.G. , Weyuker, E.J. - "Data Flow Testing Tool" - Proc. Softfair II - San Francisco - U.S.A. - Dez. 1985

[FRA87] Frankl, F.G. - "The Use of Data Flow Information for the Selection and Evaluation of Software Teste Data" - Ph.D. thesis - New York University - U.S.A. - 1987.

[FRA88] Frankl, F.G. and Weyuker, E. J. - "An Applicable Family of Data Flow Testing Criteria" - IEEE Trans. on Software Eng. - Vol. 14, No. 10 - Out. 1988 - pp. 1483-1498.

[GHE87] Ghezzi, Carlo, Jazayeri, M. - "Programming Languages Concepts" - John Wiley and Sons - 2d. edition - New York - U.S.A. - 1987.

[HAL91] Hall, P.A.V., "Relationship Between Specifications and Testing" - Information and Software Technology - Vol. 33, No.1 Jan. 1991

[HEH86] Hehl, Maximiliam Emil. - "Linguagem de Programação Estruturada FORTRAN 77" - McGraw-Hill - São Paulo - 1986.

[HER76] Herman, P.M. - "A Data Flow Analysis Approach to Program Testing" - The Australian Computer Journal - Vol. 8, No. 3 - Nov. 1976, pp 92-96.

[HOR84] Horowitz, Ellis - " Fundamentals of Programming Languages " - Springer/Verlag - U.S.A. - 1984.

[HOR87] Horowitz, Ellis - " Programming Languages - A Grand Tour / Old Languages With New Faces " - - U.S.A. - 1987.

[HOR92] Horgan, J.R. and Mathur, A.P. - "Assessing Testing Tools in Research and Education" - IEEE Software - Maio 1992.

[HORW87] Horwitz, S. , Demers, A. , Teitelbaum - "An Efficient General Iterative Algorithm for Dataflow Analysis" - Acta Informatica - Vol. 24 , pp.679-694 - 1987.

[HOW87] Howden, William E. - "Functional Program Testing and Analysis" , McGraw-Hill - U.S.A. - 1987.

[KER90] Kernighan, Brian W., Ritchie, Dennis M. - "C a Linguagem de Programação - Padrão ANSI" - Ed. Campus - R.J. - 1990

[KOR85] Korel, B. e Laski, J. W. - "A Tool for Data Flow Oriented Program Testing" in Proc. IEEE Softfair II, San Francisco, CA, U.S.A., - Dez. 1985.

[LEI92] Leitão Jr., Plínio de Sá - "Suporte ao Teste de Programas Cobol no Ambiente POKE-TOOL" - Tese de Mestrado - DCA/FEE/UNICAMP - Campinas - S.P. - Ago. 92.

[MAL88a] Maldonado, J.C., Chaim, M.L., Jino, M. - "Seleção de Casos de Teste Baseada em Fluxo de Dados através dos Critérios Potenciais Usos" - in Proc. II Simp. Bras. de Eng. de Software - Canela, RS, - Out. 1988, pp. 24-35.

[MAL88b] Maldonado, J.C., Chaim, M.L., Jino, M. - "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 - Dez. 1988.

[MAL89] Maldonado, J. C., Chaim, M.L., Jino, M. - "Arquitetura de uma Ferramenta de Teste de Apoio aos Critérios Potenciais Usos" - in Proc. XXII Congresso Nacional de Informática - São Paulo, SP - Set. 1989.

[MAL91] Maldonado, J.C. - "Critérios Potenciais Usos: Uma Contribuição ao Teste Estrutural de Software" - Tese de Doutorado - DCA/FEE/UNICAMP, Campinas, SP, - Julho 1991.

[MAR85] Marshall, Garry - " FORTRAN Para Micros " - Editora Campus Ltda. - R.J. - 1985.

[MET82] Metcalf, Michael - "Fortran Optimization" - Academic Press - London - 1982.

[MIC87] Microsoft Corporation - " FORTRAN 4.1 - Optimizing Compiler - Language Reference" - U.S.A. - 1987.

[MUS89] Musa, John D. e Ackerman, A. Frank - "Quantifying Software Validation: When to Stop Testing?"- IEEE Software - U.S.A. - Maio 1989.

[MYE79] Myers, G.J. - "The Art of Software Testing" - John Wiley & Sons - New York - U.S.A. - 1979.

[NTA88] Ntafos, S. C. - "A Comparison of Some Structural Testing Strategies" - IEEE Trans. Software Eng. - Vol. 14, No. 6 - Jun. 1988.

[OST76] Osterweil, L.J. , Fosdick, L.D. - "DAVE: a Validation Error Detection on System for FORTRAN Programs" - Software: Practice and Experience - Out. 1976.

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

[PRE87] Pressman, R.B., "Software Engineering: a Practitioner's Approach" - McGraw-Hill - New York - U.S.A. - 2d. edition - 1987.

[PRI90] Price, A.M. e Zorzo, A. - "Visualizando o Fluxo de Controle de Programas" in Proc. IV Simp. Bras. Eng. de Software - Águas de São Pedro - S.P. - Out. 1990.

[RAP82] Rapps, S. and Weyuker, E.J. - "Data Flow Analysis for Test Data Selection" - in Proc. Int. Conf. Software Eng. - Tokyo, Japan, pp. 272-278 - Sep. 1982.

[RAP85] Rapps, S. and Weyuker, E.J. - "Selecting Software Test Data Using Data Flow Information" - IEEE Trans. Software Eng. - Vol. SE-11, Abr. 1985, pp. 367-375.

[REI90] Reisman, S. - "Management and Integrated Tools" - IEEE Software - Vol. 7, No. 3 - Maio 1990.

[RIC83] Rice, John R. - "Numerical Methods, Software, and Analysis" - McGraw-Hill Book Company - Singapore - 1983.

[SEB89] Sebesta, W. Robert - "Concepts of Programming Languages" - Benjamin/Cummings Publishing Co. - U.S.A. - 1989.

[SET81] Setzer, V.W. e Melo, I.S.H., "A Construção de um Compilador" - 3a. edição - Editora Campus - R.J. - 1981.

[SCH92] Schneidewind, Norman F. e Keller, Ted W. - "Applying Reliability Models to the Space Shuttle" - IEEE Software - U.S.A. - Julho 1992.

[VER92] Vergílio, Sílvia R. - "Caminhos Não Executáveis: Caracterização, Previsão e Determinação para Suporte ao Teste de Programas" - Tese de Mestrado - DCA/FEE/UNICAMP - Campinas - S.P. - Brasil - Jan. 1992.

[WEY88] Weyuker, E. J. - "An Empirical Study of the Complexity of Data Flow Testing" - Proc. Second Workshop on Software Testing, Verification, and Analysis - Banff - Canada - Jul. 1988.

[WIR76] Wirth, N. - "Algorithms + Data Structures = Programs" - Englewood Cliffs - NJ - U.S.A. - Prentice-Hall - 1976.

[YOU89] Yourdon, Edward - "Revisões Estruturadas", tradução de "Structured Walktroughs" - Editora Campus Ltda - Rio de Janeiro - 1989.

APÊNDICE

A

A LINGUAGEM INTERMEDIÁRIA

Neste Apêndice é apresentada uma síntese da descrição da Linguagem Intermediária, a LI [CAR91,CHA91b], considerando-se apenas os comandos utilizados na configuração da ferramenta POKE-TOOL para a linguagem FORTRAN-77.

A linguagem intermediária tem como função principal identificar o fluxo de execução em um programa. Por isso, basicamente, a LI tem dois tipos de comandos : comandos sequenciais e comandos de controle de fluxo [CHA91].

A LI possui sintaxe e semântica muito simples. 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 unidade de programação em LI será utilizada 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 [CAR91].

Os comandos que resultam da tradução de uma unidade codificada em FORTRAN-77 para a LI guardam uma correspondência, direta ou indireta, com a unidade fonte de onde foram traduzidos. A cada elemento da LI é associado o endereço da porção do código

fonte correspondente.

Os comandos sequenciais da LI representam os comandos das linguagens procedimentais que não alteram o fluxo de execução, uma declaração de variável ou de 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.

São apresentados na Tabela A.1. todos os comandos originais da LI classificados por categorias. Nem todos os comandos apresentados foram utilizados nesta configuração. No Capítulo 2 são apresentados os comandos que foram inseridos por necessidade desta configuração.

A cada átomo da linguagem intermediária são acrescentados números identificadores que relacionam-se com o programa fonte. Na unidade traduzida, após cada átomo são gerados três números que representam, respectivamente:

- 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);
- o número da linha onde se encontra o átomo.

A utilidade dos ponteiros dos átomos é possibilitar o acesso ao código fonte associado ao átomo da LI.

Antes de apresentar os comandos da LI que representam os comandos da linguagem FORTRAN, é apresentada a estrutura dos átomos da LI.

No projeto da POKE-TOOL [CHA91b] foi utilizada a notação de Backus-Naur para a representação dos átomos da LI. Um átomo da LI possui a seguinte estrutura:

$\langle atm\_li \rangle ::= \langle atomo \rangle \langle inicio \rangle \langle comprimento \rangle \langle linha \rangle .$

onde

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

e

⟨*inicio*⟩ ::= NUM,
⟨*comprimento*⟩ ::= NUM e
⟨*linha*⟩ ::= NUM.

Os terminais acima representam a seqüência de caracteres indicada pelos próprios terminais; com exceção de $S que indica a seqüência de caracteres Sd^n$ onde d pertence a {0,1,...,9} e n>0, $C que indica a seqüência $C($d^n$)$d^n$, $NC que indica a seqüência $NC($d^n$)$d^n$, *NUM* que indica a seqüência d^n e *LABEL* que indica uma seqüência de letras e caracteres sempre começando por um caractere.

A utilidade desses ponteiros dos átomos é possibilitar o acesso ao código fonte associado ao átomo LI, o que vai ser necessário, por exemplo, para a extensão do grafo de fluxo de controle para a geração do *grafo def* e para a instrumentação da unidade em teste. Por exemplo, o comando na linguagem FORTRAN

IMPLICIT INTEGER (A-Z)

seria traduzido para a LI como

$DCL 139 22 11

o comando LI significa uma declaração que começa a 139 bytes do início do arquivo fonte, tem 22 bytes de comprimento e está

Declarações:	$DCL
Sequenciais:	$S
Seleção:	$IF
	$C
	$ELSE
Seleção Múltipla:	$CASE
	$CC
	$ROTC
	$ROTD
Iteração:	$FOR
	$C
	$WHILE
	$REPEAT
	$UNTIL
	$NC
Desvios Incondicionais:	$GOTO
	LABEL
	$BREAK
	$CONTINUE
	$RETURN

Tabela A.1. - Comandos da Linguagem Intermediária.

localizado na linha 11. Note-se que toda uma linha de comando em FORTRAN originou um átomo da LI.

Na convenção de Backus-Naur [SEB89] adotada aqui, os terminais serão representados em itálico, os não-terminais entre "<" e ">" e os meta-símbolos sublinhados.

Os comandos seqüenciais 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.

Os comandos de controle de fluxo são os seguintes:

(1) SELEÇÃO

<if> ::= *<if_atm><cond_atm><statement1>* |
<if_atm><cond_atm><statement1><else_atm><statement2>

onde

<if_atm> ::= *$IF<inicio><comprimento><linha>*,
<cond_atm> ::= *$C<inicio><comprimento><linha>*,
<else_atm> ::= *$ELSE<inicio><comprimento><linha>* e

<statementi> é o não-terminal que denota todos os possíveis comandos da LI, agrupados ou não; *<statementi>* será definido formalmente mais adiante. Este comando significa o "if" tradicional das linguagens do estilo ALGOL, ou seja, os "comandos" em *<statement1>* serão executados se *<cond_atm>* for verdadeiro e os "comandos" em *<statement2>* serão executados caso contrário.

Por exemplo, o bloco de comandos FORTRAN

```
IF (A.GT.B) THEN
    A = B
ELSE
    A = A + 2
ENDIF
```

seria traduzido para a LI como

$IF	319	2	16
$C(01)01	322	8	16
{	331	4	16
$S03	347	5	17
}	0	0	0
$ELSE	360	4	18
{	0	0	0
$S04	376	9	19
}	393	5	20

A solução para a ambigüidade que poderia ser gerada pelo encadeamento de *$IF* e *$ELSE* sem delimitadores de bloco "{" e "}" foi tomada inspirada na maioria das linguagens do estilo ALGOL, onde o *$ELSE* é associado com o mais recente *$IF* sem *$ELSE*, salvo o uso explícito de chaves que podem forçar a associação apropriada. Observe-se que $S3 representa o terceiro comando sequencial do programa em LI, ou seja, o número que segue os caracteres "$S" indica a ordem em que aparece o comando sequencial no programa. Os números que aparecem em $C(1)1 indicam, respectivamente, o número de predicados que possui a condição e a ordem de aparição.

(2) ITERAÇÃO

A LI fornece comandos para iteração tanto para um número fixo de repetições quanto para um número de repetições que depende de uma condição.

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

⟨*for*⟩ ::= ⟨*for_atm*⟩⟨s_1⟩⟨*cond_for-atm*⟩⟨s_2⟩⟨*statement*⟩

onde

⟨*for_atm*⟩ ::= $FOR⟨*início*⟩⟨*comprimento*⟩⟨*linha*⟩ e
⟨*cond_for_atm*⟩ ::= $C⟨*início*⟩⟨*comprimento*⟩⟨*linha*⟩.

O não-terminal ⟨*for-atm*⟩ indica o comando "for" da LI, o não terminal ⟨s_1⟩ representa a iniciação das variáveis de controle do "for" através de um comando sequencial, ⟨*cond_for_atm*⟩ representa a condição, ⟨s_2⟩ 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.

Os comandos de desvio incondicional provocam a mudança do fluxo de execução em um programa. A LI possui um comando de tranferência incondicional irrestrito do tipo "goto".

O comando "goto" da LI é definido como

⟨*goto*⟩ ::= ⟨*goto_atm*⟩⟨*label_atm*⟩

onde

⟨goto_atm⟩ ::= $GOTO⟨início⟩⟨comprimento⟩⟨linha⟩ e
⟨label_atm⟩ ::= LABEL⟨início⟩⟨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 é encontrado o comando "goto".

Até aqui foram descritos alguns comandos individuais da LI, utilizados na configuração da POKE-TOOL para FORTRAN. Entretanto, para concluir a apresentação da LI, ainda falta definir como se agrupam comandos na LI e como esses comandos são organizados em um programa. O não-terminal ⟨statement⟩, muito utilizado acima, representa um único comando da LI ou um agrupamento deles e é definido como

⟨statement⟩ ::= { {⟨statement⟩} } |
⟨dcl⟩ |
⟨s⟩ |
⟨if⟩ |
⟨case⟩ |
⟨for⟩ |
LABEL⟨statement⟩ |
⟨goto⟩ |
⟨return⟩.

Os programas em LI são definidos da seguinte maneira:

⟨program⟩ ::= {C ⟨dcl⟩|⟨s⟩} {⟨statement⟩}.

Um exemplo de tradução de um programa fonte para a LI é fornecido no Apêndice E.

A associação dos comandos do código fonte para a LI não é sempre direta. Considere o trecho de programa em FORTRAN:

```
      INTEGER FAT, N

      FAT = 1
      READ *, N
      DO 10, M = 2, N
      FAT = FAT * N
10    CONTINUE
```

seria traduzido para

$DCL	33	14	3
$S01	60	7	5
$S02	78	9	6
$FOR	98	5	7
$S03	105	5	7
$C(01)01	110	3	7
$S04	105	8	7
{	0	0	0
$S05	124	13	8
10	143	2	9
$S06	148	8	9
}	0	0	0

Observe-se que no comando "for" os ponteiros de "{" e "}"são iguais a 0, indicando que o átomo da LI não possui correspondência no arquivo fonte. Obviamente, a decisão de como mapear a linguagem da unidade para a LI é do usuário configurador da POKE-TOOL.

APÊNDICE

B

MODELOS DE IMPLEMENTAÇÃO DOS CRITÉRIOS POTENCIAIS USOS

Para a implementação dos critérios Potenciais Usos no ambiente da POKETOOL é necessário definir os Modelos de Fluxo de Dados, de Fluxo de Controle e de Instrumentação, específicos para a linguagem de programação alvo.

Apresentam-se neste Apêndice as características relevantes sobre esses modelos tendo como referência os descritos em [CHA91b].

B.1 MODELO DE FLUXO DE CONTROLE

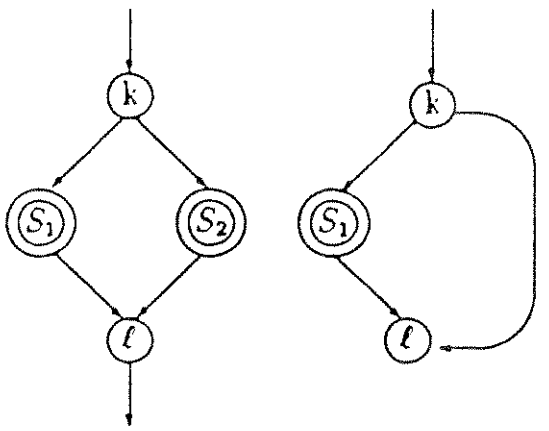
No modelo de fluxo de Controle adotado, um programa P é representado por um *grafo dirigido* $G(N,A,s)$, onde os blocos disjuntos de comandos correspondentes da LI são associados aos nós $n \in N$ e os possíveis fluxos de controle entre os blocos associados aos arcos $e \in A$; o nó s representa o nó de entrada. Esse grafo é usualmente conhecido como grafo de fluxo de controle ou grafo de programa.

Os comandos que compõem a linguagem intermediária (LI) implementam construções básicas de fluxo de controle como seqüência, seleção, iteração e desvio; a descrição da LI é mostrada no Apêndice B. As Figuras B.1, B.2, B.3 e B.4 representam as construções básicas de fluxo de controle adotadas para os comandos da linguagem LI. O grafo de fluxo de controle de uma unidade é obtido pela simples concatenação dessas construções básicas.

O grafo de fluxo de controle da unidade em teste é independente da linguagem de implementação da unidade; no entanto, na configuração da POKE-TOOL deve-se levar esse modelo em consideração pois, de certa forma, ele reflete os aspectos semânticos da LI e terá forte influência na instrumentação da unidade em teste.

Comandos de seleção

< if > ::= < if-atm >< cond-atm>< statement$_1$ > |
< if-atm><cond-atm><statement$_1$ >< else-atm ><statment$_2$ >



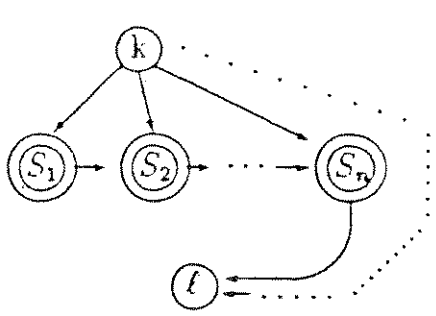
(1) – representa o subgrafo correspondente ao statement$_1$

(2) – representa o subgrafo correspondente ao statement$_2$.

(k) – associado a cond-atm.

instrumentação: Sejam i e j os nós de entrada dos subgrafos S_1 e S_2, respectivamente. No início dos blocos k, i, j e são inseridas pontas de prova que caracterizam a execução destes blocos, ou seja, pontas de prova k, i, j e .

< CASE >::= < case-atm >< case-cond-atm > { { (< rotc-atm > | < rotd-atm >)
{ < statement > } } }



(i) i=1, ..., n representa os subgrafos correspondentes ao statement, relativos a cada uma das alternativas da seleção múltipla

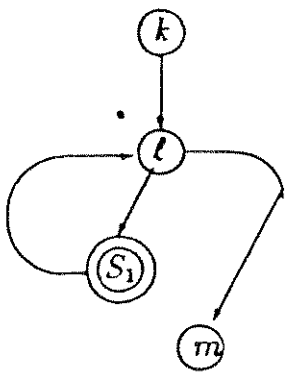
(k) – associado a case-cond-atm.

instrumentação:
Sejam $i_1, i_2, ..., i_n$ os nós de entrada dos subgrafos $S_1, S_2, ..., S_n$, respectivamente. No início dos blocos k, $i_1, i_2, ..., i_n$, l são inseridas pontas de prova que caracterizam a execução destes blocos, ou seja, pontas de prova k, $i_1, i_2, ..., i_n$ e l.

Figura B.1: Modelo de Fluxo de Controle e Instrumentação associado aos comandos da Linguagem LI: Comandos de Seleção.

Comandos de Iteração

< while > ::= < while-atm >< cond-while-atm>< statement>

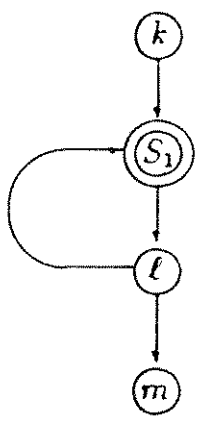


(S1) - representa o subgrafo correspondente a < *statement* >, ou seja, ao corpo do laço

(ℓ) - associado a < cond-while-atm>

instrumentação: seja i o nó de entrada do subgrafo S_1. No nó k é inserida a ponta de prova k; no nó i as pontas de prova l e i e no nó m, as pontas de prova l e m.

<repeat-until > ::= < repeat-atm >< statement >< until-atm>< cond-until-atm>



(S1) - representa o subgrafo correspondente a < statement >

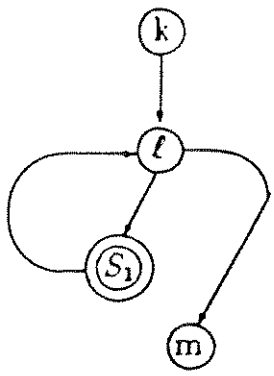
(ℓ) - associado a < cond-until-atm>

instrumentação: seja i o nó de entrada do subgrafo S_1 e j o nó saída de S_1. No início dos nós i, j, k e m são inseridas as pontas de provas i, j, k e m. Adicionalmente, no fim do nó j, é inserida a ponta de prova l.

Figura B.2: Modelo de Fluxo de Controle e Instrumentação associado aos comandos da Linguagem LI: Comandos de Iteração — "while" e "repeat".

Comandos de Iteração

< for > ::= < for-atm >< S_1 >< cond-for-atm>< S_2 >< statement>



(S_1) - representa o subgrafo correspondente a < *statement* >, ou seja, ao corpo do laço

(k) - associado aos comandos de iniciação da variável de controle do "for", ou seja, aos comandos representados por < S_1 >.

(j) - associado aos comandos que alteram a variável de controle, ou seja, aos comandos representados por < S_2 >, onde o nó j é o nó saída do subgrafo S_1.

(ℓ) - associado a < cond-for-atm>

instrumentação:

sejam *i* e *j* os nós de entrada e saída, respectivamente, do subgrafo < S_1 >.
No início do nó k, é inserida a ponta de prova k; no início do nó i as pontas de provas l e i e no início do nó m, as pontas de provas l e m.
No nó j é também inserida a ponta de prova j.

Figura B.3: Modelo de Fluxo de Controle e Instrumentação associado aos comandos da Linguagem LI: Comando de Iteração "for".

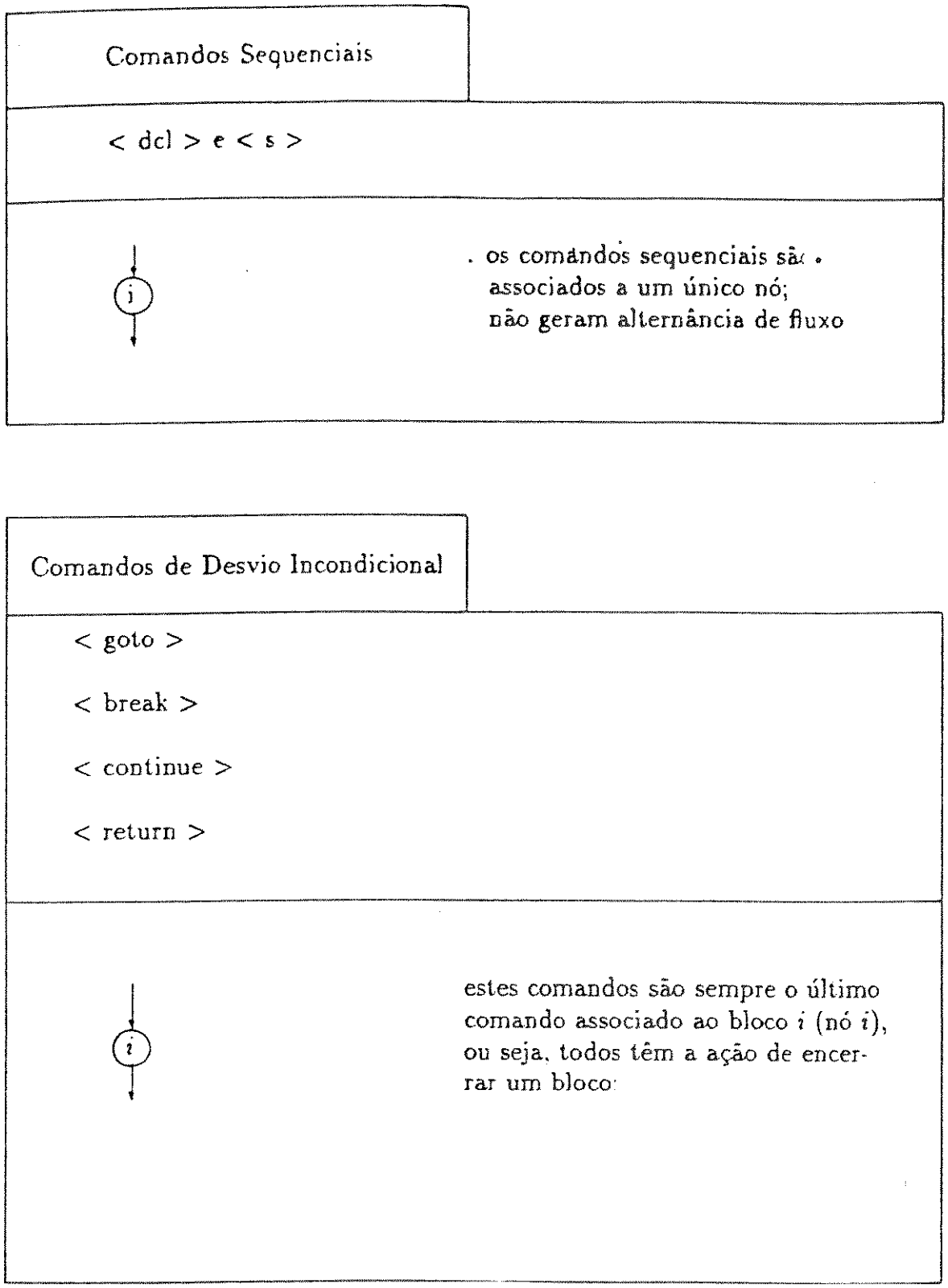


Figura B.4: Modelo de Fluxo de Controle e Instrumentação associado aos comandos da Linguagem LI: Comandos Seqüenciais e de Desvio Incondicional.

B.2 MODELO DE INSTRUMENTAÇÃO

A instrumentação fornece facilidades para a análise posterior à execução dos casos de teste, como a análise de adequação de um dado conjunto de casos de teste. Para tanto, este módulo modifica, com inserção de código fonte, a própria unidade em teste, gerando uma nova versão do programa, usualmente denominada *versão instrumentada*: no caso da POKE-TOOL, para a linguagem FORTRAN, essa versão é denominada TESTEPROG.FOR. No Apêndice E apresenta-se a versão instrumentada de um exemplo.

A instrumentação consiste essencialmente em inserir pontas de prova nos blocos de comandos correspondentes a cada nó do grafo de programa da unidade em teste, possibilitando a identificação do caminho executado pelo caso de teste fornecido. Uma ponta de prova consiste basicamente em um comando de escrita do número do nó em um arquivo (arquivo PATH.TES, na versão atual da POKE-TOOL), produzindo um "trace" do caso de teste fornecido; esta informação é imprescindível para a função de avaliação da POKE-TOOL.

Obviamente, a instrumentação deve ser tal que reflita a semântica dos comandos da LI e ao mesmo tempo viabilize a correta avaliação dos comandos efetivamente executados; observa-se também que a instrumentação está fortemente restrita ao modelo de fluxo de controle adotado. As Figuras B.1, B.2, B.3, B.4 ilustram a instrumentação associada aos comandos da LI.

B.3 MODELO DE FLUXO DE DADOS

No contexto de teste de software, a análise de fluxo de dados usualmente é utilizada para estender o grafo de programa pela

associação de tipos de ocorrências de variáveis aos elementos deste grafo que, posteriormente, é utilizado para determinação dos caminhos e associações a serem requeridos.

No caso dos critérios Potenciais Usos é suficiente associar a cada nó *i* do grafo de programa o conjunto de variáveis definidas no bloco de comandos correspondente; esta extensão denomina-se *grafo def*.

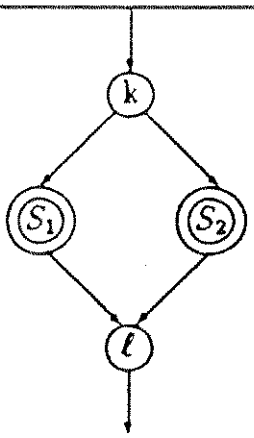
Para a correta geração do grafo def é necessário precisar o que considera-se uma definição de variável; tal consideração foi discutida no Capítulo 2. Adicionalmente, a passagem de valores entre procedimentos através da passagem de parâmetros pode ser feita por: *valor*, *referência* ou *nome* [GHE87]. Se a variável é passada por referência ou por nome considera-se que seja um parâmetro de saída. As definições decorrentes de possíveis definições em chamadas de procedimentos são distinguidas das demais e são ditas *definidas por referência*; esta distinção é utilizada na geração dos grafo(i) [CHA91b], ou seja, na determinação dos caminhos livres de definição para as variáveis definidas em *i*.

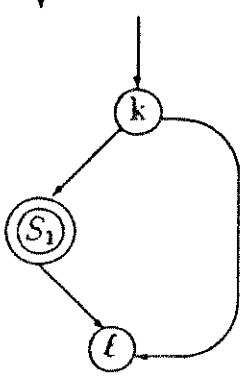
No modelo de fluxo de dados adotado, considera-se que no nó de entrada ocorre uma definição dos parâmetros e das variáveis globais que ocorrem na unidade em teste.

As Figuras B.5, B.6, e B.7 sintetizam os conceitos e hipóteses básicos para a determinação do grafo def a partir do modelo de fluxo de controle dos principais comandos da LI. Para os comandos seqüenciais a extensão é obvia, uma vez que esses comandos estão sempre associados a um único nó.

Comandos de Seleção

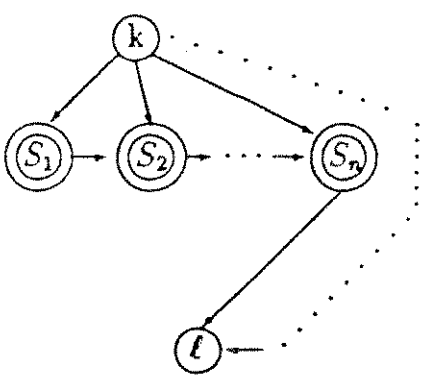
< if > ::= < if-atm >< cond-atm>< statement$_1$ > |
< if-atm><cond-atm><statement$_1$ >< else-atm ><statement$_2$ >





Ao nó *k* são atribuídos o conjunto de variáveis definidas no bloco de comandos associado a este nó e o conjunto de variáveis definidas na condição associada a <cond-atm>. Ao nó *l* é atribuído o conjunto de variáveis definidas no bloco de comandos associado a este nó. A extensão dos subgrafos S_1 e S_2 depende dos comandos associados a < statement$_1$ > e < statement$_2$ >, respectivamente.

<case>::=<case-atm>< case-cond-atm>{{ [<rotc-atm>| <rotd-atm>) {<statement>} }}

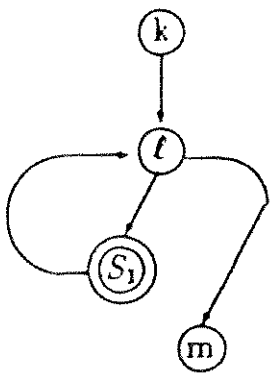


Ao nó *k* são atribuídos o conjunto de variáveis definidas no bloco de comandos associados a este nó e o conjunto de variáveis definidas na condição do case associada a < case-cond-atm >. Ao nó *l* é atribuído o conjunto de variáveis definidas no bloco de comandos associado a a este nó. A extensão dos subgrafos S_n depende dos comandos associados a cada uma das alternativas da seleção múltipla.

Figura B.5: Diretrizes para a Expansão do Grafo de Programa para a Obtenção do Grafo Def: Comandos de Seleção.

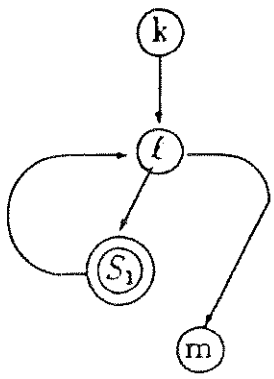
Comandos de iteração

< while > ::= < while-atm >< cond-while-atm>< statement>



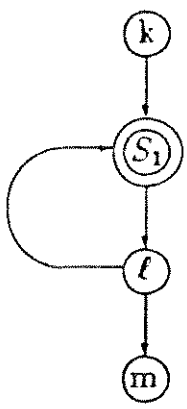
Ao nó k é associada nenhuma definição de variável devido ao comando while , somente devido aos demais comandos eventualmente associados ao nó k. Ao nó é associado conjunto de variáveis definidas na condição associada a cond-while-atm . Ao nó m somente é atribuído o conjunto de variáveis definidas devido a outros comandos associados a este nó. A extensão do subgrafo S_1 depende dos comandos do corpo do laço, ou seja, associados a statement .

< for > ::= < for-atm >< S_1 ><cond-for-atm>< S_2 >< statement>



Seja x o nó de saída de S_1. Ao nó k é associado o conjunto de variáveis definidas pelos comandos representados por S_1 . Aos nós e m idem ao comando while . Ao nó x é atribuído o conjunto de variáveis definidas pelos comandos representados por S_2 e, se for o caso, o conjunto de variáveis definidas por outros comandos associados ao nó x. A extensão do subgrafo S_1 depende dos comandos do corpo do laço associados a statement.

<repeat-until> ::= <repeat-until><statement>< until-atm><cond-until-atm>



Ao nó k não é associada nenhuma definição de variável devido ao comando repeat-until , somente devido aos demais comandos eventualmente associados ao nó k. Ao nó l é associado o conjunto de variáveis definidas na condição associada a <cond-until-atm>. Ao nó *m*, somente é atribuído o conjunto de variáveis definidas devido a outros comandos eventualmente associados a este nó. A extensão do subgrafo S_1 depende dos comandos do corpo do laço, ou seja, associados a < statement >.

Figura B.6: Diretrizes para a Expansão do Grafo de Programa para a Obtenção do Grafo Def: Comandos de Iteração.

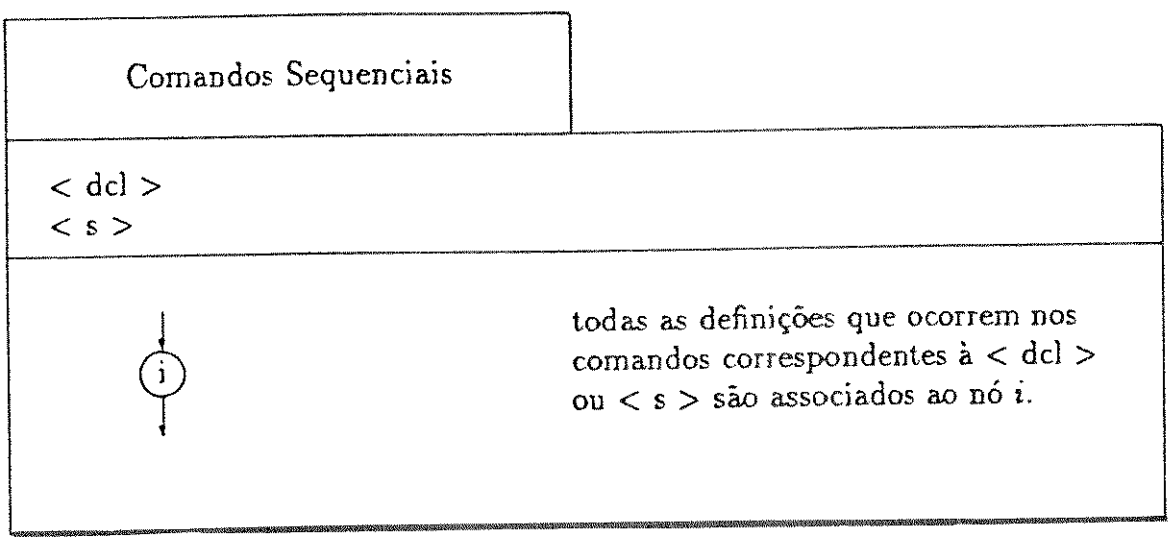


Figura B.7: Diretrizes para a Expansão do Grafo de Programa para a Obtenção do Grafo Def: Comandos Seqüenciais.

APÊNDICE

C

TABELAS DO ANALISADOR LÉXICO PARA FORTRAN-77

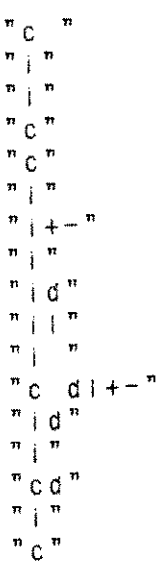
Este apêndice apresenta a tabela de transições léxicas definida para a linguagem FORTRAN-77. Cada linha da tabela (transição do autômato) é composta de cinco campos: argumento de comparação com o caracter lido, estado corrente, próximo estado, número da ação semântica que será executada e índice relativo ao próximo estado. O caracter "ℓ" representa o conjunto de letras do alfabeto, "∂" o conjunto de numerais. O caracter "≟" indica "igual a" e o caracter "⊂" indica "complemento de". Como exemplo, considere a sexta linha da tabela: caso o caracter lido seja uma letra, ocorrerá uma transição do estado 0 para o estado 10, a ação semântica 1 será executada. A tabela indica ainda que as transições do estado 10 localizam-se após a quadragésima sétima linha da tabela. Existe apenas um estado inicial (0) e um estado final (1000). As transições que têm o mesmo estado corrente estão agrupadas consecutivamente.

C.1 - TABELA DE TRANSIÇÕES LÉXICAS

" ¡ "	0	0	0	0
" ¡ & "	0	15	31	63
" ¡ "	0	4	33	27
" c d ¡ . + - * / # = : $, ( ) % @ ¡ & ¡ "	0	1000	24	1000
" ¡ d "	0	1	1	15
" ¡ ¡ "	0	10	1	47
" ¡ @ "	0	1000	29	1000
" ¡ . "	0	11	1	50
" ¡ + - * / "	0	13	14	57
" ¡ = : , ( ) $ "	0	14	14	61
" ¡ "	0	23	1	81
" ¡ # "	0	16	9	66
" ¡ ! "	0	31	30	100
" ¡ "	0	1000	0	1000
" ¡ ¡ "	0	1000	34	1000
" ¡ "	1	1000	3	1000
" ¡ d "	1	1	2	15
" ¡ "	1	1000	0	1000
" ¡ . "	1	3	2	23
" ¡ ¡ "	1	32	26	102
" c d ¡ . "	1	1000	3	1000
" ¡ "	2	1000	22	1000
" c "	2	1000	25	1000
" ¡ "	3	4	0	27
" ¡ d "	3	1	2	15
" ¡ ¡ "	3	32	26	102
" c d ¡ "	3	1000	3	1000
" ¡ "	4	1000	19	1000
" ¡ "	4	1000	34	1000
" ¡ * ¡ "	4	1000	28	1000
" ¡ d "	4	1000	11	1000
" ¡ "	4	5	0	33
" c d ¡ * "	4	1000	21	1000
" ¡ "	5	6	0	36
" ¡ d "	5	1000	11	1000
" c d "	5	1000	21	1000
" ¡ "	6	7	0	39
" ¡ d "	6	1000	11	1000
" c d "	6	1000	21	1000
" ¡ "	7	8	0	42
" ¡ d "	7	1000	11	1000
" c d "	7	1000	21	1000
" ¡ "	8	9	0	45
" ¡ d "	8	1000	11	1000
" c d "	8	1000	21	1000

```
"|  "
"c  "
"|d|"
"|"
"cd|"
"|  "
"||"
"|d"
"||"
"|."
"|"
"c|."
"|*"
"|/"
"|"
"c*/"
"|"
"c"
"|d"
"|  "
"c  d"
"c"
"|"
"|"
"c  "
"|  "
"|  "
"|d"
"|  "
"|d"
"|  "
"|d"
"|  "
"|d"
"|  "
"c  "
"c"
"|"
"|"
"|"
"|  "
"|d"
"|  "
"|d"
"|  "
"|d"
"|  "
"|d"
"|  "
"|d"
"|  "
```

9	1000	32	1000
9	0	0	0
10	10	2	47
10	1000	15	1000
10	1000	10	1000
11	11	0	50
11	12	2	53
11	1	2	15
12	12	2	53
12	34	4	111
12	1000	0	1000
12	1000	3	1000
13	1000	12	1000
13	1000	12	1000
13	1000	0	1000
13	1000	13	1000
14	1000	0	1000
14	1000	3	1000
15	15	2	63
15	1000	7	1000
15	1000	23	1000
16	16	17	66
16	17	0	68
17	17	33	68
17	1000	8	1000
17	18	0	71
18	19	0	73
18	1000	8	1000
19	20	0	75
19	1000	8	1000
20	21	0	77
20	1000	8	1000
21	22	0	79
21	1000	8	1000
22	1000	6	1000
22	16	0	66
23	23	2	81
23	30	2	97
23	24	0	84
24	24	33	84
24	25	0	87
24	1000	5	1000
25	26	0	89
25	1000	5	1000
26	27	0	91
26	1000	5	1000
27	28	0	93
27	1000	5	1000
28	29	0	95
28	1000	5	1000
29	0	0	0



29	23	0	81
30	23	0	81
30	1000	20	1000
30	1000	35	1000
31	31	17	100
31	1000	18	1000
32	32	2	102
32	1000	0	1000
32	33	2	108
32	1000	27	1000
32	32	0	102
32	1000	3	1000
33	33	2	108
33	1000	0	1000
33	1000	3	1000
34	1000	0	1000
34	1000	3	1000

C.2 - TABELA DE PALAVRAS-CHAVE

ASSIGN	10
BACKSPACE	20
BLOCK	30
CALL	40
CHARACTER	50
CLOSE	60
COMMON	70
COMPLEX	80
CONTINUE	90
DATA	100
DIMENSION	110
DO	120
DOUBLE	130
ELSE	135
ELSEIF	140
END	145
ENDFILE	150
ENDIF	155
ENTRY	160
EQUIVALENCE	170
ERR	175
EXTERNAL	180
FORMAT	190
FUNCTION	200
GO	210
GOTO	215
IF	220
IMPLICIT	230
INQUIRE	240
INTEGER	250
INTRINSIC	260
LOGICAL	270
OPEN	280
PARAMETER	290
PAUSE	300
PRECISION	305
PRINT	310
PROGRAM	320
READ	330
REAL	340
RETURN	350

REWIND	360
SAVE	370
STOP	380
SUBROUTINE	390
THEN	395
TO	400
WRITE	410

APÊNDICE

D

DESCRIÇÃO SINTÁTICA DA LINGUAGEM FORTRAN-77

Neste Apêndice é mostrada a especificação gramatical da linguagem FORTRAN-77 utilizada na configuração do analisador sintático da POKE-TOOL; sua notação baseia-se na notação estendida de Backus-Naur. A diferença básica entre a notação de Backus-Naur e a notação aceita pela POKE-TOOL consiste nos meta-símbolos: "=" indica a atribuição de uma regra gramatical; "," representa a ocorrência de sentenças alternativas; "[" e "]" limitam uma estrutura que pode ocorrer zero ou mais vezes; e "*" designa que o reconhecimento da regra gramatical é opcional. Os elementos entre plicas "'" indicam os símbolos terminais, ou seja, as classes de itens léxicos caracterizados na análise léxica; os demais elementos auxiliam a definição gramatical da linguagem. Ainda, a notação aceita pela POKE-TOOL requer que cada regra sintática seja terminada por um ponto ".".

D.1 DESCRIÇÃO SINTÁTICA DA LINGUAGEM FORTRAN-77

```
unidade = linha [ linha ] 'END' 4 .
  linha = nome_unidade , lin_coment , lin_brco , lin_cmd .
        lin_cmd = comandos  [ comandos ] .
        lin_coment = 'COMENT' 'NEW_LINE' [ 'COMENT' 'NEW_LINE' ] .
        lin_brco = 'LIN/BRCO' 'NEW_LINE' [ 'LIN/BRCO' 'NEW_LINE' ] .
nome_unidade = programa , subrotina , funcao .
 programa  = 'PROGRAM' 5 'IDENT' 3  'NEW_LINE' .
 subrotina = 'SUBROUTINE' 5 'IDENT' [ token_0 ]  'NEW_LINE' 3 .
 funcao    = 'FUNCTION' 5 'IDENT' [ token_0 ]  'NEW_LINE' 3 .
comandos = cmd_aux1  cmd_aux2  .
 cmd_aux1 = 'COMANDO' 36 , * .
 cmd_aux2 = format , declaracao  , sequencial , desvio ,
                  iteracao , if , entr_saida .
declaracao =  dcl_aux [ token_0 ] fun_opt .
           fun_opt = 'FUNCTION' 'IDENT' [ token_0 ] 'NEW_LINE' 3 ,
                     'NEW_LINE' 1 .
    dcl_aux = 'REAL' 5 ,
       'EXTERNAL' 5 ,
                     'INTRINSIC' 5 ,
                     'PARAMETER' 5 ,
                     'CHARACTER' 5 ,
                     'COMPLEX' 5 ,
                     'DOUBLE' 5 'PRECISION' ,
                     'IMPLICIT' 5 tipo ,
                     'INTEGER' 5 ,
                     'LOGICAL' 5 ,
                     'COMMON' 5 ,
                     'DIMENSION' 5 ,
                     'BLOCK' 5 'DATA' ,
                     'FUNCTION' 5 ,
                     'ENTRY' 5 ,
                     'SAVE' 5  .
tipo =  'INTEGER' ,
        'REAL' ,
        'DOUBLE' 'PRECISION' ,
        'CHARACTER' ,
        'COMPLEX' ,
        'LOGICAL' .
token_0  =  'CONST' , 'IDENT' , 'CARACTER' , log_const ,
      '=' , '+' , '-' , '*' , '**' , '/' , '(' ,
          ')' , ',' , ':' , '<' , '>' , '//' .
sequencial =  cmd_atrib , chama_sub , dat_par .
desvio = continue , else , elseif , endif , goto , pause , stop ,
         return .
iteracao = 'DO' 5  'CONST' 19  comma_opt  'IDENT' 5  '='
           arit_expr ',' 12 arit_expr exp2_opt 'NEW_LINE' 14 .
```

```
          exp2_opt =  ',' 13 arit_expr , * .
entr_saida = read_write , cond_arq , posi_arq .
format =  'FORMAT' 5  'TEXTO...'  'NEW_LINE' 2 .
dat_par = data .
cmd_atrib = atrib  , assign .
          atrib =  'IDENT' 5  token_0  [ token_0 ] 'NEW_LINE' 2 .
assign = 'ASSIGN' 5 'CONST' 'TO' 'IDENT' 'NEW_LINE' 2 .
chama_sub =  'CALL' 5 'IDENT' [ token_0 ]  'NEW_LINE' 2 .
   call_aux = call_aux1 , * .
          call_aux1 = '(' call_aux2 ')' .
   call_aux2 = arg_call [ ',' arg_call ] .
   arg_call = expr , nome , call_label .
          call_label = '*' 'CONST' .
read_write =  rwp_a1  token_rwp [ token_rwp ]  'NEW_LINE' 26 .
      rwp_a1 = 'READ' 5 ,
        'WRITE' 5 ,
        'PRINT' 5 .
 token_rwp = io_err , io_end , token_1 .
           token_1 = 'CONST' , 'IDENT' , 'CARACTER' ,
                     '(' , ')' , '*' , ',' , '=' , ':' .
       io_end = 'END' 31 '=' 37 'CONST' 27 .
         io_err = 'ERR' 31 '=' 37 'CONST' 27 .
cond_arq = open , close , inquire .
         open =    'OPEN'    5  c_arq_aux  'NEW_LINE' 26 .
         close =   'CLOSE'   5  c_arq_aux  'NEW_LINE' 26 .
         inquire = 'INQUIRE' 5  c_arq_aux  'NEW_LINE' 26 .
         c_arq_aux =  token_c_arq  [ token_c_arq ] .
         token_c_arq = io_err , token_1 .
posi_arq = backspace , endfile , rewind .
  backspace =  'BACKSPACE'  5  p_arq_aux  'NEW_LINE' 26 .
  endfile   =  'ENDFILE'    5  p_arq_aux  'NEW_LINE' 26 .
  rewind    =  'REWIND'     5  p_arq_aux  'NEW_LINE' 26 .
         p_arq_aux = token_p_arq  [ token_p_arq ] .
         token_p_arq = io_err , token_1 .
equivalence = 'EQUIVALENCE' equiv_aux [ ',' equiv_aux ] .
            equiv_aux = '(' equiv_ent ',' equiv_ent
                        [ ',' equiv_ent ] ')' .
            equiv_ent = ident_aux .
data = 'DATA' 5 token_0 [ token_0 ] 'NEW_LINE' 2 .
     data_ist = 'NEW_LINE' , data_aux , ',' data_aux .
     data_aux = name_list '/' const_list  '/' .
     name_list =  data_names [ ',' data_names ] .
     const_list = data_const [ ',' data_const ] .
     data_names = ident_aux  , dat_do_ist .
     ident_aux = 'IDENT' 5  data_a1 .
     data_a1 = data_a2 , * .
     data_a2 =  '(' data_a3 ')' [ '(' data_a3 ')' ] .
     data_a3 = data_a4 [ ':' data_a4 ] .
     data_a4 = data_a5 , * .
     data_a5 = arit_expr [ ',' arit_expr ] .
     dat_do_ist = '(' data_a6  'IDENT'  '=' data_a7  ')' .
```

```
      data_a6 = data_a8 [ ',' data_a8 ] .
      data_a7 = arit_expr [ ',' arit_expr ] .
      data_a8 = arr_e_name  , dat_do_ist .
      data_const = data_a9 , log_char .
      data_a9 = signal_opt  data_a10  data_a11 .
      data_a10 = 'CONST'  , 'IDENT' .
      data_a11 = data_a12 , * .
      data_a12 = '*'  data_a13 .
      data_a13 = data_a10 , log_char , cplex_const .
      log_char = 'CARACTER' , log_const .
      cplex_const = '(' signal_opt 'CONST' ','
                        signal_opt 'CONST' ')' .

continue = 'CONTINUE' 11  'NEW_LINE' 38 .
goto = goto_ax1  goto_ax2  .
      goto_ax1 =  'GO' 5 'TO' 22 , 'GOTO' 5 .
      goto_ax2 = goto_incond  'NEW_LINE'     ,
                 goto_comput  'NEW_LINE' 17 ,
                 goto_assin   'NEW_LINE' 23 .
goto_incond = 'CONST' 16  .
goto_comput = '(' 20 'CONST' 20 [ ','  'CONST' 20 ] ')' 20
       [ ',' ] arit_expr  .
goto_assin = 'IDENT' 18  [ ',' ]  [ '(' 20  'CONST' 20
             [ ','  'CONST' 20 ] ')' 21 ] .
if = 'IF' 5 if_aux0 .
  if_aux0 = '(' 8 if_aux1 [ if_token  if_aux1 ] if_aux3  .
    if_aux1 = if_aux2  [ if_aux2 ] ')' 28  [ ')' 28 ] .
    if_aux2 = if_token ,
              'CONST' ,
              'IDENT' .
    if_aux3 = if_aux4 , if_logico .
    if_aux4 = if_aritm ,
              if_bloco ,
              if_token .
    if_token =  'CARACTER' , '(' , arit_oper ,
          log_const , log_oper , oper_rel , '.NOT.' , ',' , ':' .
if_aritm =  'CONST' 32 ',' 'CONST' 24 ',' 'CONST' 33 'NEW_LINE' .
if_bloco =  'THEN' 34 'NEW_LINE' .
if_logico = cmd_aux3  .
          cmd_aux3 = sequencial , desvio , if , entr_saida  .
else = 'ELSE' 29 elsif .
   elsif = 'NEW_LINE' , 'IF' 40 if_aux0 .
elseif = 'ELSEIF' 39 if_aux0 .
endif = 'ENDIF' 30  'NEW_LINE' .
pause = 'PAUSE' 27 motivo 'NEW_LINE' 38 .
stop = 'STOP' 5 motivo 'NEW_LINE' 6 .
return = 'RETURN' 5 motivo 'NEW_LINE' 6 .
     motivo = 'CONST' 7 , 'CARACTER' 7 , * .
io_ist = io_ist_1  [ ',' io_ist_1 ] .
   io_ist_1 = sig_exp io_ist_2 [ oper_rel sig_exp_opt io_ist_2 ] ,
                    io_ist_2  [  oper_rel  sig_exp_opt  io_ist_2 ] .
   io_ist_2 = exp_a1  [  '//'  exp_a1 ] .
```

```
   exp_a1 = 'CONST' , 'CARACTER' , ident_a1 , io_lst_3 ,
            log_const .

          ident_a1 = 'IDENT' exp_a2 .
            exp_a2 = exp_a3 , imp_do , * .
          imp_do = '=' arit_expr  [  ',' arit_expr ] ')' .
          exp_a3 = '('  exp_a4 .
          exp_a4 = exp_a5 , exp_a6 .
          exp_a6 = ':'  [ arit_expr ]  ')' .
          exp_a5 = sig_exp  io_lst_2 [ ',' io_lst_2 ] exp_a7 ,
                            io_lst_2 [ ',' io_lst_2 ] exp_a7 .
          exp_a7 = ')'  , exp_a6 .
          io_lst_3 =  '(' io_lst  ')' .
lista_nome = nome [ ',' nome ] .
           nome = 'IDENT'  nome_array .
           nome_array =  '(' dim_aux ')'  , * .
array_dcl = 'IDENT' ar_dcl_aux .
          ar_dcl_aux = '(' dim_aux ')' .
   dim_aux = dim_aux1 [ ',' dim_aux1 ] .
   dim_aux1 = '*' , dim_aux2 .
          dim_aux2 = dim_b_expr  dim_aux3 .
          dim_aux3 = dim_aux4 , * .
          dim_aux4 = ':'  dim_aux5 .
          dim_aux5 = dim_b_expr , '*' .
arr_e_name = 'IDENT' '(' arit_expr [ ',' arit_expr ] ')' .
v_ar_name = 'IDENT' var_array_aux .
              var_array_aux = '(' arit_list ')'  , * .
arit_list = arit_expr [ ',' arit_expr ] .
dim_b_expr = signal_opt  dim_elem  [ arit_oper  dim_elem ] .
       dim_elem = 'CONST' , 'IDENT' , '(' dim_b_expr ')' .
arit_expr = signal a_exp_a1  [ arit_oper  a_exp_a1 ] ,
                   a_exp_a1  [ arit_oper  a_exp_a1 ] .
          a_exp_a1 = 'CONST' , identa1 , arit_aux .
          arit_aux = '(' arit_expr ')' .
          identa1 = 'IDENT' identa2 .
          identa2 = identa3 , * .
          identa3 = '(' identa4 ')' .
          identa4 = expr_lst , * .
char_expr = char_a1  [ '//' char_a1 ] .
          char_a1 = 'CARACTER' , char_a2 , char_a3 .
          char_a2 = 'IDENT' identa2 .
          char_a3 = '('  char_expr  ')' .
          expr_lst = expr [ ',' expr ] .
log_expr = '.NOT.' log_a1 [ log_oper  not_opt  log_a1 ] ,
                   log_a1 [ log_oper  not_opt  log_a1 ] .
         not_opt = '.NOT.' , * .
         log_a1 = log_const , arit_rel , char_rel , log_a2 .
         arit_rel = arit_expr  a_rel_aux .
         a_rel_aux = rel_aux , * .
         rel_aux = oper_rel  arit_expr .
         char_rel = char_expr  oper_rel  char_expr .
         log_a2 = '(' log_expr ')' .
```

```
expr =  sig_exp  name_aux  [ expr_oper  name_aux ] ,
   name_aux  [ expr_oper  name_aux ] .
     sig_exp_opt = '+' , '-' , '.NOT.' , * .
     sig_exp =  '+' , '-' , '.NOT.' .
     name_aux =  'CONST'  , ident_aux  , expr_aux  ,
                 'CARACTER' , log_const .
     expr_aux = '(' expr ')' .
signal = '+'  , '-' .
signal_opt = '+'  , '-'  , * .
comma_opt = ','  , * .
log_const = '.TRUE.'  , '.FALSE.' .
arit_oper = '+' , '-' , '*' , '/' , '**' .
log_oper = '.AND.'  , '.OR.'  , '.EQV.'  , '.NEQV.' .
expr_oper = arit_oper  , '//'  , log_oper  , oper_rel .
oper_rel = '.LT.' , '.LE.' , '.EQ.' , '.NE.' , '.GT.' , '.GE.' .
```

APÊNDICE

E

UM EXEMPLO COMPLETO

Nêste Apêndice apresenta-se um exemplo completo de aplicação de teste com a presente configuração da POKE-TOOL. A unidade submetida a teste é a INCTAX.FOR [CAL89]. Esta unidade faz o cálculo do imposto de renda a pagar, de um conjunto de salários, segundo critérios do fisco inglês.

E.1 - CÓDIGO FONTE

```
   PROGRAM INCTAX
*****************************************************************************
*                                                                           *
*                          PROGRAMA INCTAX                                  *
*                                                                           *
*     Programa para calcular o imposto de renda cobrado na Inglaterra       *
*     para um conjunto de salarios a uma taxa de 25% para as primeiras      *
*     750 Libras, 30% para as 5000 proximas libras e 45% para o resta-      *
*     nte. O programa primeiro le o numero de salarios.                     *
*                                                                           *
*****************************************************************************

    PRINT *,'Entre o numero de salarios: '
    READ *, N
    DO 10, I = 1,N
      PRINT *, 'Entre salario:'
      READ *, SALARY
      IF (( SALARY .GE. 1.0) .AND. (SALARY .LE. 20000.0)) THEN
       IF ( SALARY .LE. 750.0 ) THEN
        TAX = SALARY / 4.0
       ELSE
        TAX = 750.0 / 4.0
        REM = SALARY - 750.0
        IF (REM .LE. 5000.0) THEN
          TAX = TAX + (REM * 0.3)
        ELSE
          TAX = TAX + (5000.0 * 0.3)
          REM = REM - 5000.0
          TAX = TAX + (0.45 * REM)
        ENDIF
       ENDIF
       PRINT *,'O imposto pago pelo Salario ', SALARY, '  e'' ', TAX
      ELSE
       PRINT *,'*** Erro - Salario invalido ', SALARY
      ENDIF
   10  CONTINUE
     STOP
     END
```

E.2 - INFORMAÇÕES ESTÁTICAS

Nesta seção são organizadas as informaçoes estáticas geradas pela POKE-TOOL, para o programa INCTAX.FOR; estas informações estão contidas em arquivos gerados pela POKE-TOOL, durante a fase estática e são reproduzidos a seguir, tal como apresentados pela ferramenta.

ARQUIVO INCTAX.LI - Tradução da unidade em FORTRAN para a LI

```
{          4      14       1
$S01       755      38      13
$S02       798       9      14
$FOR       812       5      15
$S03       819       5      15
$C(01)01   824       2      15
$S04       819       7      15
{          0       0       0
$S05       833      25      16
$S06       865      14      17
$IF        886       2      18
$C(01)02   889      48      18
{          938       4      18
$IF        950       2      19
$C(01)03   953      21      19
{          975       4      19
$S07       988      18      20
}          0       0       0
$ELSE      1014       4      21
{          0       0       0
$S08       1027      17      22
$S09       1053      20      23
$IF        1082       2      24
$C(01)04   1085      17      24
{          1103       4      24
$S10       1118      23      25
}          0       0       0
$ELSE      1150       4      26
{          0       0       0
$S11       1165      26      27
$S12       1202      18      28
$S13       1231      24      29
}          1264       5      30
}          1277       5      31
```

```
$S14        1290      61      32
}           0       0       0
$ELSE       1358       4      33
{           0       0       0
$S15        1370      46      34
}           1423       5      35
10          1433       2      36
$S16        1437       8      36
}           0       0       0
$RETURN     1451       4      37
}           1461       3      38
```

ARQUIVO INCTAX.NLI - Unidade em LI com indicação de nós.

```
{                      1      4     14      1
$S01                   1    755     38     13
$S02                   1    798      9     14
$FOR                   1    812      5     15
$S03                   1    819      5     15
$C(01)01               2    824      2     15
$S04                  12    819      7     15
{                      3      0      0      0
$S05                   3    833     25     16
$S06                   3    865     14     17
$IF                    3    886      2     18
$C(01)02               3    889     48     18
{                      4    938      4     18
$IF                    4    950      2     19
$C(01)03               4    953     21     19
{                      5    975      4     19
$S07                   5    988     18     20
}                      5      0      0      0
$ELSE                  6   1014      4     21
{                      6      0      0      0
$S08                   6   1027     17     22
$S09                   6   1053     20     23
$IF                    6   1082      2     24
$C(01)04               6   1085     17     24
{                      7   1103      4     24
$S10                   7   1118     23     25
}                      7      0      0      0
$ELSE                  8   1150      4     26
{                      8      0      0      0
$S11                   8   1165     26     27
$S12                   8   1202     18     28
$S13                   8   1231     24     29
}                      8   1264      5     30
```

}	9	1277	5	31
$S14	10	1290	61	32
}	10	0	0	0
$ELSE	11	1358	4	33
{	11	0	0	0
$S15	11	1370	46	34
}	11	1423	5	35
10	12	1433	2	36
$S16	12	1437	8	36
}	12	0	0	0
$RETURN	13	1451	4	37
}	14	1461	3	38

ARQUIVO INCTAX.GFC - Grafo de fluxo de controle da unidade.

```
14

1
    2 0
2
    3 13 0
3
    4 11 0
4
    5 6 0
5
    10 0
6
    7 8 0
7
    9 0
8
    9 0
9
    10 0
10
    12 0
11
    12 0
12
    2 0
13
    14 0
14
    0
```

ARQUIVO ARCPRIM.TES

```
ARCOS PRIMITIVOS DO MODULO inctax.for

arco (  2,13) e' primitivo
arco (  3,11) e' primitivo
arco (  4,  5) e' primitivo
arco (  6,  7) e' primitivo
arco (  6,  8) e' primitivo
```

ARQUIVO TESTEPROG.FOR

```
*
*     TESTEPROG.FOR - PROGRAMA FONTE INSTRUMENTADO
*                     PARA TESTES
*
*
      PROGRAM INCTAX
*
*-------    PARA INSTRUMENTACAO    ---------
*
      INTEGER IMPNO, TABNOS
      DIMENSION TABNOS(10)
      COMMON IMPNO, TABNOS
*
*-----------------------------------------
*
*
*-------    PARA INSTRUMENTACAO    ---------
*
      IMPNO = 0
      OPEN (20, FILE='PATH.TES', STATUS='OLD')
*
*-----------------------------------------
*
                    CALL PROVA (1)
*  NO'  1  *
      PRINT *,'Entre o numero de salarios: '
*  NO'  1  *
      READ *, N
*  NO'  1  2 12 *
```

```
      DO 10 I = 1,N
                        CALL PROVA (2)
                        CALL PROVA (3)
*  NO' 3  *
      PRINT *, 'Entre salario:'
*  NO' 3  *
      READ *, SALARY
*  NO' 3  *
      IF(( SALARY .GE. 1.0) .AND. (SALARY .LE. 20000.0))
     $       THEN
                        CALL PROVA (4)
*  NO' 4  *
      IF( SALARY .LE. 750.0 )
     $       THEN
                        CALL PROVA (5)
*  NO' 5  *
      TAX = SALARY / 4.0
*  NO' 6  *
      ELSE
                        CALL PROVA (6)
*  NO' 6  *
      TAX = 750.0 / 4.0
*  NO' 6  *
      REM = SALARY - 750.0
*  NO' 6  *
      IF(REM .LE. 5000.0)
     $       THEN
                        CALL PROVA (7)
*  NO' 7  *
      TAX = TAX + (REM * 0.3)
*  NO' 8  *
      ELSE
                        CALL PROVA (8)
*  NO' 8  *
      TAX = TAX + (5000.0 * 0.3)
*  NO' 8  *
      REM = REM - 5000.0
*  NO' 8  *
      TAX = TAX + (0.45 * REM)
      ENDIF
                        CALL PROVA (9)
      ENDIF
                        CALL PROVA (10)
*  NO' 10  *
      PRINT *,'O Imposto pago pelo Salario ', SALARY, '  e'' ', TAX
*  NO' 11  *
      ELSE
                        CALL PROVA (11)
*  NO' 11  *
      PRINT *,'*** Erro - Salario Invalido ', SALARY
```

```
      ENDIF
                     CALL PROVA (12)
*  NO' 12  *
   10       CONTINUE
                     CALL PROVA (2)
                     CALL PROVA (13)
                     CALL PROVA (14)
                     CALL PROVA(999)
      CLOSE (20, STATUS='KEEP')
*  NO' 13  *
      STOP
*  NO' 14  *
      END

COMENTARIOS :
C
C----------    PARA INSTRUMENTACAO    ---------
C
C              SUBROTINA PONTA DE PROVA
C
C-----------------------------------------------

      SUBROUTINE PROVA(NUM)

      INTEGER IMPNO,
     '        TABNOS
      DIMENSION TABNOS(10)
      COMMON IMPNO,
     '       TABNOS

      IMPNO = IMPNO + 1
      TABNOS(IMPNO) = NUM
      IF ( NUM .EQ. 999 )
     '           THEN
                 WRITE (20,*) (TABNOS(I),I=1,IMPNO-1)
                 IMPNO = 0
      ELSEIF (IMPNO .EQ. 10)
     '           THEN
                 WRITE (20,*) (TABNOS(I),I=1,IMPNO)
                 IMPNO = 0
      ENDIF

      RETURN
      END
```

ARQUIVO GRAFODEF.TES

VARIAVEIS DEFINIDAS nos NOS do modulo inctax.for

Variaveis Definidas = Vars Defs
Variaveis possivelmente definidas por Referencia = Vars Refs

```
NO'  1
Vars defs:  N I
Vars refs:
NO'  2
Vars defs:
Vars refs:
NO'  3
Vars defs:  SALARY
Vars refs:
NO'  4
Vars defs:
Vars refs:
NO'  5
Vars defs:  TAX
Vars refs:
NO'  6
Vars defs:  TAX REM
Vars refs:
NO'  7
Vars defs:  TAX
Vars refs:
NO'  8
Vars defs:  TAX REM
Vars refs:
NO'  9
Vars defs:
Vars refs:
NO' 10
Vars defs:
Vars refs:
NO' 11
Vars defs:
Vars refs:
NO' 12
Vars defs:  I
Vars refs:
NO' 13
```

```
Vars defs:
Vars refs:
NO' 14
Vars defs:
Vars refs:
```

ARQUIVO PUASSOC.TES

ASSOCIACOES REQUERIDAS PELOS CRITERIOS TODOS POT-USOS E POT-USOS/DU

```
Associacoes requeridas pelo Grafo( 1)

1) <1,(2,13),{ N, I }>
2) <1,(3,11),{ N, I }>
3) <1,(6,8),{ N, I }>
4) <1,(6,7),{ N, I }>
5) <1,(12,2),{ N }>
6) <1,(4,5),{ N, I }>


Associacoes requeridas pelo Grafo( 3)

7) <3,(3,11),{ SALARY }>
8) <3,(6,8),{ SALARY }>
9) <3,(6,7),{ SALARY }>
10) <3,(2,13),{ SALARY }>
11) <3,(2,3),{ SALARY }>
12) <3,(4,5),{ SALARY }>


Associacoes requeridas pelo Grafo( 5)

13) <5,(2,13),{ TAX }>
14) <5,(11,12),{ TAX }>
15) <5,(3,11),{ TAX }>
16) <5,(6,8),{ TAX }>
17) <5,(9,10),{ TAX }>
18) <5,(6,7),{ TAX }>
19) <5,(4,5),{ TAX }>


Associacoes requeridas pelo Grafo( 6)

20) <6,(6,8),{ TAX, REM }>
21) <6,(2,13),{ REM }>
```

```
22) <6,(11,12),{ REM }>
23) <6,(3,11),{ REM }>
24) <6,(4,6),{ REM }>
25) <6,(5,10),{ REM }>
26) <6,(4,5),{ REM }>
27) <6,(6,7),{ TAX, REM }>


Associacoes requeridas pelo Grafo( 7)

28) <7,(2,13),{ TAX }>
29) <7,(11,12),{ TAX }>
30) <7,(3,11),{ TAX }>
31) <7,(4,6),{ TAX }>
32) <7,(5,10),{ TAX }>
33) <7,(4,5),{ TAX }>


Associacoes requeridas pelo Grafo( 8)

34) <8,(2,13),{ TAX, REM }>
35) <8,(11,12),{ TAX, REM }>
36) <8,(3,11),{ TAX, REM }>
37) <8,(4,6),{ TAX, REM }>
38) <8,(5,10),{ TAX, REM }>
39) <8,(4,5),{ TAX, REM }>


Associacoes requeridas pelo Grafo(12)

40) <12,(2,13),{ | }>
41) <12,(11,12),{ | }>
42) <12,(3,11),{ | }>
43) <12,(6,8),{ | }>
44) <12,(6,7),{ | }>
45) <12,(10,12),{ | }>
46) <12,(4,5),{ | }>
```

ARQUIVO PDUPATHS.TES

CAMINHOS REQUERIDOS PELO CRITERIO TODOS POT-DU-CAMINHOS

Caminhos requeridos pelo Grafo(1)

1) 1 2 13 14
2) 1 2 3 11 12 2
3) 1 2 3 4 6 8 9 10 12 2
4) 1 2 3 4 6 7 9 10 12 2
5) 1 2 3 4 5 10 12 2

Caminhos requeridos pelo Grafo(3)

6) 3 11 12 2 13 14
7) 3 11 12 2 3
8) 3 4 6 8 9 10 12 2 13 14
9) 3 4 6 8 9 10 12 2 3
10) 3 4 6 7 9 10 12 2 13 14
11) 3 4 6 7 9 10 12 2 3
12) 3 4 5 10 12 2 13 14
13) 3 4 5 10 12 2 3

Caminhos requeridos pelo Grafo(5)

14) 5 10 12 2 13 14
15) 5 10 12 2 3 11 12
16) 5 10 12 2 3 4 6 8 9 10
17) 5 10 12 2 3 4 6 7 9 10
18) 5 10 12 2 3 4 5

Caminhos requeridos pelo Grafo(6)

19) 6 8
20) 6 7 9 10 12 2 13 14
21) 6 7 9 10 12 2 3 11 12
22) 6 7 9 10 12 2 3 4 6
23) 6 7 9 10 12 2 3 4 5 10

Caminhos requeridos pelo Grafo(7)

24) 7 9 10 12 2 13 14
25) 7 9 10 12 2 3 11 12

```
26) 7 9 10 12 2 3 4 6
27) 7 9 10 12 2 3 4 5 10


Caminhos requeridos pelo Grafo( 8)

28) 8 9 10 12 2 13 14
29) 8 9 10 12 2 3 11 12
30) 8 9 10 12 2 3 4 6
31) 8 9 10 12 2 3 4 5 10


Caminhos requeridos pelo Grafo(12)

32) 12 2 13 14
33) 12 2 3 11 12
34) 12 2 3 4 6 8 9 10 12
35) 12 2 3 4 6 7 9 10 12
36) 12 2 3 4 5 10 12
```

ARQUIVO DES_PU.TES

```
DESCRITORES PARA O CRITERIO TODOS POT-USOS


N = 1 2 3 4 5 6 7 8 9 10 11 12 13 14


Descritores para o Grafo( 1)

Ni = 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Nt = 2 14

1) N* 1 Nnv* 2 [ Nnv* 2 ]* 13
Nnv = 2 3 4 5 6 7 8 9 10 11 13 14

2) N* 1 Nnv* 3 [ Nnv* 3 ]* 11
Nnv = 2 3 4 5 6 7 8 9 10 11 13 14

3) N* 1 Nnv* 6 [ Nnv* 6 ]* 8
Nnv = 2 3 4 5 6 7 8 9 10 11 13 14

4) N* 1 Nnv* 6 [ Nnv* 6 ]* 7
Nnv = 2 3 4 5 6 7 8 9 10 11 13 14
```

```
5) N* 1 Nnv* 12 [ Nnv* 12 ]* 2
Nnv = 2 3 4 5 6 7 8 9 10 11 12 13 14

6) N* 1 Nnv* 4 [ Nnv* 4 ]* 5
Nnv = 2 3 4 5 6 7 8 9 10 11 13 14


Descritores para o Grafo( 3)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14
Nt = 3 14

7) N* 3 [ Nnv* 3 ]* 11
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14

8) N* 3 Nnv* 6 [ Nnv* 6 ]* 8
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14

9) N* 3 Nnv* 6 [ Nnv* 6 ]* 7
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14

10) N* 3 Nnv* 2 [ Nnv* 2 ]* 13
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14

11) N* 3 Nnv* 2 [ Nnv* 2 ]* 3
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14

12) N* 3 Nnv* 4 [ Nnv* 4 ]* 5
Nnv = 2 4 5 6 7 8 9 10 11 12 13 14


Descritores para o Grafo( 5)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14
Nt = 5 10 12 14

13) N* 5 Nnv* 2 [ Nnv* 2 ]* 13
Nnv = 2 3 4 6 7 8 9 10 11 12 13 14

14) N* 5 Nnv* 11 [ Nnv* 11 ]* 12
Nnv = 2 3 4 6 7 8 9 10 11 12 13 14

15) N* 5 Nnv* 3 [ Nnv* 3 ]* 11
Nnv = 2 3 4 6 7 8 9 10 11 12 13 14

16) N* 5 Nnv* 6 [ Nnv* 6 ]* 8
Nnv = 2 3 4 6 7 8 9 10 11 12 13 14
```

17) N* 5 Nnv* 9 [Nnv* 9]* 10
Nnv = 2 3 4 6 7 8 9 10 11 12 13 14

18) N* 5 Nnv* 6 [Nnv* 6]* 7
Nnv = 2 3 4 6 7 8 9 10 11 12 13 14

19) N* 5 Nnv* 4 [Nnv* 4]* 5
Nnv = 2 3 4 6 7 8 9 10 11 12 13 14

Descritores para o Grafo(6)

Nl = 2 3 4 5 6 7 8 9 10 11 12 13 14
Nt = 6 8 10 12 14

20) N* 6 [Nnv* 6]* 8
Nnv = 2 3 4 5 9 10 11 12 13 14

21) N* 6 Nnv* 2 [Nnv* 2]* 13
Nnv = 2 3 4 5 7 9 10 11 12 13 14

22) N* 6 Nnv* 11 [Nnv* 11]* 12
Nnv = 2 3 4 5 7 9 10 11 12 13 14

23) N* 6 Nnv* 3 [Nnv* 3]* 11
Nnv = 2 3 4 5 7 9 10 11 12 13 14

24) N* 6 Nnv* 4 [Nnv* 4]* 6
Nnv = 2 3 4 5 7 9 10 11 12 13 14

25) N* 6 Nnv* 5 [Nnv* 5]* 10
Nnv = 2 3 4 5 7 9 10 11 12 13 14

26) N* 6 Nnv* 4 [Nnv* 4]* 5
Nnv = 2 3 4 5 7 9 10 11 12 13 14

27) N* 6 [Nnv* 6]* 7
Nnv = 2 3 4 5 9 10 11 12 13 14

Descritores para o Grafo(7)

Nl = 2 3 4 5 6 7 9 10 11 12 13 14
Nt = 6 10 12 14

28) N* 7 Nnv* 2 [Nnv* 2]* 13
Nnv = 2 3 4 5 9 10 11 12 13 14

29) N* 7 Nnv* 11 [Nnv* 11]* 12
Nnv = 2 3 4 5 9 10 11 12 13 14

30) N* 7 Nnv* 3 [Nnv* 3]* 11
Nnv = 2 3 4 5 9 10 11 12 13 14

31) N* 7 Nnv* 4 [Nnv* 4]* 6
Nnv = 2 3 4 5 9 10 11 12 13 14

32) N* 7 Nnv* 5 [Nnv* 5]* 10
Nnv = 2 3 4 5 9 10 11 12 13 14

33) N* 7 Nnv* 4 [Nnv* 4]* 5
Nnv = 2 3 4 5 9 10 11 12 13 14

Descritores para o Grafo(8)

Ni = 2 3 4 5 6 8 9 10 11 12 13 14
Nt = 6 10 12 14

34) N* 8 Nnv* 2 [Nnv* 2]* 13
Nnv = 2 3 4 5 9 10 11 12 13 14

35) N* 8 Nnv* 11 [Nnv* 11]* 12
Nnv = 2 3 4 5 9 10 11 12 13 14

36) N* 8 Nnv* 3 [Nnv* 3]* 11
Nnv = 2 3 4 5 9 10 11 12 13 14

37) N* 8 Nnv* 4 [Nnv* 4]* 6
Nnv = 2 3 4 5 9 10 11 12 13 14

38) N* 8 Nnv* 5 [Nnv* 5]* 10
Nnv = 2 3 4 5 9 10 11 12 13 14

39) N* 8 Nnv* 4 [Nnv* 4]* 5
Nnv = 2 3 4 5 9 10 11 12 13 14

Descritores para o Grafo(12)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14
Nt = 12 14

40) N* 12 Nnv* 2 [Nnv* 2]* 13
Nnv = 2 3 4 5 6 7 8 9 10 11 13 14

41) N* 12 Nnv* 11 [Nnv* 11]* 12
Nnv = 2 3 4 5 6 7 8 9 10 11 13 14

42) N* 12 Nnv* 3 [Nnv* 3]* 11
Nnv = 2 3 4 5 6 7 8 9 10 11 13 14

43) N* 12 Nnv* 6 [Nnv* 6]* 8
Nnv = 2 3 4 5 6 7 8 9 10 11 13 14

44) N* 12 Nnv* 6 [Nnv* 6]* 7
Nnv = 2 3 4 5 6 7 8 9 10 11 13 14

45) N* 12 Nnv* 10 [Nnv* 10]* 12
Nnv = 2 3 4 5 6 7 8 9 10 11 13 14

46) N* 12 Nnv* 4 [Nnv* 4]* 5
Nnv = 2 3 4 5 6 7 8 9 10 11 13 14

Numero Total de Descritores = 46

ARQUIVO DES_PUDU.TES

DESCRITORES PARA O CRITERIO TODOS POT-USOS/DU

N = 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Descritores para o Grafo(1)

Ni = 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Nt = 2 14

1) N* 1 Nnvlf* 2 13
Nnvlf = 2 3 4 5 6 7 8 9 10 11 13 14

2) N* 1 Nnvlf* 3 11
Nnvlf = 2 3 4 5 6 7 8 9 10 11 13 14

3) N* 1 Nnvlf* 6 8
Nnvlf = 2 3 4 5 6 7 8 9 10 11 13 14

4) N* 1 Nnvlf* 6 7

```
Nnvlf = 2 3 4 5 6 7 8 9 10 11 13 14

5) N* 1 Nnvlf* 12 2
Nnvlf = 2 3 4 5 6 7 8 9 10 11 12 13 14

6) N* 1 Nnvlf* 4 5
Nnvlf = 2 3 4 5 6 7 8 9 10 11 13 14


Descritores para o Grafo( 3)

Nl = 2 3 4 5 6 7 8 9 10 11 12 13 14
Nt = 3 14

7) N* 3 11
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14

8) N* 3 Nnvlf* 6 8
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14

9) N* 3 Nnvlf* 6 7
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14

10) N* 3 Nnvlf* 2 13
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14

11) N* 3 Nnvlf* 2 3
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14

12) N* 3 Nnvlf* 4 5
Nnvlf = 2 4 5 6 7 8 9 10 11 12 13 14


Descritores para o Grafo( 5)

Nl = 2 3 4 5 6 7 8 9 10 11 12 13 14
Nt = 5 10 12 14

13) N* 5 Nnvlf* 2 13
Nnvlf = 2 3 4 6 7 8 9 10 11 12 13 14

14) N* 5 Nnvlf* 11 12
Nnvlf = 2 3 4 6 7 8 9 10 11 12 13 14

15) N* 5 Nnvlf* 3 11
Nnvlf = 2 3 4 6 7 8 9 10 11 12 13 14

16) N* 5 Nnvlf* 6 8
```

```
Nnvlf = 2 3 4 6 7 8 9 10 11 12 13 14

17) N* 5 Nnvlf* 9 10
Nnvlf = 2 3 4 6 7 8 9 10 11 12 13 14

18) N* 5 Nnvlf* 6 7
Nnvlf = 2 3 4 6 7 8 9 10 11 12 13 14

19) N* 5 Nnvlf* 4 5
Nnvlf = 2 3 4 6 7 8 9 10 11 12 13 14


Descritores para o Grafo( 6)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14
Nt = 6 8 10 12 14

20) N* 6 8
Nnvlf = 2 3 4 5 9 10 11 12 13 14

21) N* 6 Nnvlf* 2 13
Nnvlf = 2 3 4 5 7 9 10 11 12 13 14

22) N* 6 Nnvlf* 11 12
Nnvlf = 2 3 4 5 7 9 10 11 12 13 14

23) N* 6 Nnvlf* 3 11
Nnvlf = 2 3 4 5 7 9 10 11 12 13 14

24) N* 6 Nnvlf* 4 6
Nnvlf = 2 3 4 5 7 9 10 11 12 13 14

25) N* 6 Nnvlf* 5 10
Nnvlf = 2 3 4 5 7 9 10 11 12 13 14

26) N* 6 Nnvlf* 4 5
Nnvlf = 2 3 4 5 7 9 10 11 12 13 14

27) N* 6 7
Nnvlf = 2 3 4 5 9 10 11 12 13 14


Descritores para o Grafo( 7)

Ni = 2 3 4 5 6 7 9 10 11 12 13 14
Nt = 6 10 12 14

28) N* 7 Nnvlf* 2 13
```

```
Nnvlf = 2 3 4 5 9 10 11 12 13 14

29) N* 7 Nnvlf* 11 12
Nnvlf = 2 3 4 5 9 10 11 12 13 14

30) N* 7 Nnvlf* 3 11
Nnvlf = 2 3 4 5 9 10 11 12 13 14

31) N* 7 Nnvlf* 4 6
Nnvlf = 2 3 4 5 9 10 11 12 13 14

32) N* 7 Nnvlf* 5 10
Nnvlf = 2 3 4 5 9 10 11 12 13 14

33) N* 7 Nnvlf* 4 5
Nnvlf = 2 3 4 5 9 10 11 12 13 14


Descritores para o Grafo( 8)

Ni = 2 3 4 5 6 8 9 10 11 12 13 14
Nt = 6 10 12 14

34) N* 8 Nnvlf* 2 13
Nnvlf = 2 3 4 5 9 10 11 12 13 14

35) N* 8 Nnvlf* 11 12
Nnvlf = 2 3 4 5 9 10 11 12 13 14

36) N* 8 Nnvlf* 3 11
Nnvlf = 2 3 4 5 9 10 11 12 13 14

37) N* 8 Nnvlf* 4 6
Nnvlf = 2 3 4 5 9 10 11 12 13 14

38) N* 8 Nnvlf* 5 10
Nnvlf = 2 3 4 5 9 10 11 12 13 14

39) N* 8 Nnvlf* 4 5
Nnvlf = 2 3 4 5 9 10 11 12 13 14


Descritores para o Grafo(12)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14
Nt = 12 14

40) N* 12 Nnvlf* 2 13
```

```
Nnvlf = 2 3 4 5 6 7 8 9 10 11 13 14

41) N* 12 Nnvlf* 11 12
Nnvlf = 2 3 4 5 6 7 8 9 10 11 13 14

42) N* 12 Nnvlf* 3 11
Nnvlf = 2 3 4 5 6 7 8 9 10 11 13 14

43) N* 12 Nnvlf* 6 8
Nnvlf = 2 3 4 5 6 7 8 9 10 11 13 14

44) N* 12 Nnvlf* 6 7
Nnvlf = 2 3 4 5 6 7 8 9 10 11 13 14

45) N* 12 Nnvlf* 10 12
Nnvlf = 2 3 4 5 6 7 8 9 10 11 13 14

46) N* 12 Nnvlf* 4 5
Nnvlf = 2 3 4 5 6 7 8 9 10 11 13 14

Numero Total de Descritores = 46
```

ARQUIVO DES_PDU.TES

```
DESCRITORES PARA O CRITERIO TODOS POT-DU-CAMINHOS

N = 1 2 3 4 5 6 7 8 9 10 11 12 13 14


Descritores para o Grafo( 1)

Ni = 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Nt = 2 14

 1) N* 1 2 13
 2) N* 1 Nlf* 3 11 12 2
 3) N* 1 Nlf* 6 8 Nlf* 12 2
 4) N* 1 Nlf* 6 7 Nlf* 12 2
 5) N* 1 Nlf* 4 5 Nlf* 12 2


Descritores para o Grafo( 3)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14
Nt = 3 14
```

```
 6) N* 3 11 Nlf* 2 13
 7) N* 3 11 Nlf* 2 3
 8) N* 3 Nlf* 6 8 Nlf* 2 13
 9) N* 3 Nlf* 6 8 Nlf* 2 3
10) N* 3 Nlf* 6 7 Nlf* 2 13
11) N* 3 Nlf* 6 7 Nlf* 2 3
12) N* 3 4 5 Nlf* 2 13
13) N* 3 4 5 Nlf* 2 3


Descritores para o Grafo( 5)

Nl = 2 3 4 5 6 7 8 9 10 11 12 13 14
Nt = 5 10 12 14

14) N* 5 Nlf* 2 13
15) N* 5 Nlf* 3 11 12
16) N* 5 Nlf* 6 8 9 10
17) N* 5 Nlf* 6 7 9 10
18) N* 5 Nlf* 4 5


Descritores para o Grafo( 6)

Nl = 2 3 4 5 6 7 8 9 10 11 12 13 14
Nt = 6 8 10 12 14

19) N* 6 8
20) N* 6 7 Nlf* 2 13
21) N* 6 7 Nlf* 3 11 12
22) N* 6 7 Nlf* 4 6
23) N* 6 7 Nlf* 4 5 10


Descritores para o Grafo( 7)

Nl = 2 3 4 5 6 7 9 10 11 12 13 14
Nt = 6 10 12 14

24) N* 7 Nlf* 2 13
25) N* 7 Nlf* 3 11 12
26) N* 7 Nlf* 4 6
27) N* 7 Nlf* 4 5 10


Descritores para o Grafo( 8)

Ni = 2 3 4 5 6 8 9 10 11 12 13 14
Nt = 6 10 12 14
```

```
28) N* 8 Nif* 2 13
29) N* 8 Nif* 3 11 12
30) N* 8 Nif* 4 6
31) N* 8 Nif* 4 5 10


Descritores para o Grafo(12)

Ni = 2 3 4 5 6 7 8 9 10 11 12 13 14
Nt = 12 14

32) N* 12 2 13
33) N* 12 Nif* 3 11 12
34) N* 12 Nif* 6 8 Nif* 10 12
35) N* 12 Nif* 6 7 Nif* 10 12
36) N* 12 Nif* 4 5 10 12

Numero Total de Descritores = 36
```

E.3 - INFORMAÇÕES DINÂMICAS

Nesta seção estão organizadas as informações dinâmicas geradas pela POKE-TOOL durante uma seção de trabalho, relativa ao teste da unidade INCTAX.FOR.

As informações dinâmicas, da mesma forma que as estáticas já apresentadas, são armazenadas em arquivos específicos gerados pela própria POKE-TOOL.

Apresenta-se nesta seção, primeiramente as entradas e saídas relativas aos testes executados. Os arquivos de entrada correspondem às entradas fornecidas como argumentos da linha de chamada, entradas via teclado.

O conjunto de casos de teste foi elaborado inicialmente, de maneira informal e não sistemática. O objetivo era exercitar os aspectos funcionais da unidade em teste, sem a aplicação de uma técnica funcional definida *a priori*. Entretanto, para finalizar os testes, foram consultados os resultados parciais fornecidos pela

ferramenta. Esses resultados apoiaram a formulação de casos de teste que permitiram alcançar a cobertura de 100% nos critérios Potenciais Usos.

ARQUIVOS DE ENTRADA

ARQUIVO INC1.TEC

```
1
6000
```

ARQUIVO INC2.TEC

```
1
4000
```

ARQUIVO INC3.TEC

```
1
749
```

ARQUIVO INC4.TEC

```
2
749
20001
```

ARQUIVO INC5.TEC

```
2
19999
20001
```

ARQUIVO INC6.TEC

```
17
0
100
749
750
751
1000
4999
5000
5001
```

```
7000
19999
4500
30000
4300
600
19000
500
```

ARQUIVO INC7.TEC

```
0
```

ARQUIVOS DE SAÍDA

ARQUIVO OUTPUT1.TES

```
Entre o numero de salarios:
Entre salario:
O imposto pago pelo Salario      6000.000000  e'      1800.000000
Stop - Program terminated.
```

ARQUIVO OUTPUT2.TES

```
Entre o numero de salarios:
Entre salario:
O imposto pago pelo Salario      4000.000000  e'      1162.500000
Stop - Program terminated.
```

ARQUIVO OUTPUT3.TES

```
Entre o numero de salarios:
Entre salario:
O imposto pago pelo Salario       749.000000  e'       187.250000
Stop - Program terminated.
```

ARQUIVO OUTPUT4.TES

```
Entre o numero de salarios:
Entre salario:
O imposto pago pelo Salario       749.000000  e'       187.250000
Entre salario:
*** Erro - Salario Invalido     20001.000000
Stop - Program terminated.
```

ARQUIVO OUTPUT5.TES

```
Entre o numero de salarios:
Entre salario:
O imposto pago pelo Salario     19999.000000  e'      8099.550000
Entre salario:
*** Erro - Salario invalido     20001.000000
Stop - Program terminated.
```

ARQUIVO OUTPUT6.TES

```
Entre o numero de salarios:
Entre salario:
*** Erro - Salario invalido     0.000000E+00
Entre salario:
O imposto pago pelo Salario       100.000000  e'       25.000000
Entre salario:
O imposto pago pelo Salario       749.000000  e'      187.250000
Entre salario:
O imposto pago pelo Salario       750.000000  e'      187.500000
Entre salario:
O imposto pago pelo Salario       751.000000  e'      187.800000
Entre salario:
O imposto pago pelo Salario      1000.000000  e'      262.500000
Entre salario:
O imposto pago pelo Salario      4999.000000  e'     1462.200000
Entre salario:
O imposto pago pelo Salario      5000.000000  e'     1462.500000
Entre salario:
O imposto pago pelo Salario      5001.000000  e'     1462.800000
Entre salario:
O imposto pago pelo Salario      7000.000000  e'     2250.000000
Entre salario:
O imposto pago pelo Salario     19999.000000  e'     8099.550000
Entre salario:
O imposto pago pelo Salario      4500.000000  e'     1312.500000
Entre salario:
*** Erro - Salario invalido     30000.000000
Entre salario:
O imposto pago pelo Salario      4300.000000  e'     1252.500000
Entre salario:
O imposto pago pelo Salario       600.000000  e'      150.000000
Entre salario:
O imposto pago pelo Salario     19000.000000  e'     7650.000000
Entre salario:
O imposto pago pelo Salario       500.000000  e'      125.000000
Stop - Program terminated.
```

ARQUIVO OUTPUT7.TES

```
Entre o numero de salarios:
Stop - Program terminated.
```

ARQUIVOS DE NÓS EXECUTADOS PELOS CASOS DE TESTE

CASO 1

```
 1   2  13  14
```

CASO 2

```
 1   2   3   4   6   8
 9  10  12   2
13  14
```

CASO 3

```
 1   2   3   4   6   7
 9  10  12   2
13  14
```

CASO 4

```
 1   2   3   4   5  10
12   2  13  14
```

CASO 5

```
 1   2   3   4   5  10
12   2   3  11
12   2  13  14
```

CASO 6

```
 1   2   3   4   6   8
 9  10  12   2
 3  11  12   2  13  14
```

CASO 7

```
 1   2   3  11  12   2
 3   4   5  10
12   2   3   4   5  10
12   2   3   4
```

```
 5   10   12    2    3    4
 6    7    9   10
12    2    3    4    6    7
 9   10   12    2
 3    4    6    7    9   10
12    2    3    4
 6    7    9   10   12    2
 3    4    6    7
 9   10   12    2    3    4
 6    8    9   10
12    2    3    4    6    8
 9   10   12    2
 3    4    6    7    9   10
12    2    3   11
12    2    3    4    6    7
 9   10   12    2
 3    4    5   10   12    2
 3    4    6    8
 9   10   12    2    3    4
 5   10   12    2
13   14
```

CASO 8

```
 1    2   13   14
```

AVALIAÇÃO DE COBERTURA EM RELAÇÃO AO CRITÉRIO TODOS POTENCIAIS USOS

ASSOCIACOES DO CRITERIO TODOS POT-USOS nao executadas:

Cobertura Total = 100.000000
Media da Cobertura dos Grafo(i) = 100.000000

ASSOCIACOES DO CRITERIO TODOS POT-USOS executadas:

<1,(2,13),{ N, I }>
<1,(3,11),{ N, I }>
<1,(6,8),{ N, I }>
<1,(6,7),{ N, I }>
<1,(12,2),{ N }>
<1,(4,5),{ N, I }>
<3,(3,11),{ SALARY }>
<3,(6,8),{ SALARY }>
<3,(6,7),{ SALARY }>
<3,(2,13),{ SALARY }>
<3,(2,3),{ SALARY }>
<3,(4,5),{ SALARY }>
<5,(2,13),{ TAX }>
<5,(11,12),{ TAX }>
<5,(3,11),{ TAX }>
<5,(6,8),{ TAX }>
<5,(9,10),{ TAX }>
<5,(6,7),{ TAX }>
<5,(4,5),{ TAX }>
<6,(6,8),{ TAX, REM }>
<6,(2,13),{ REM }>
<6,(11,12),{ REM }>
<6,(3,11),{ REM }>
<6,(4,6),{ REM }>
<6,(5,10),{ REM }>
<6,(4,5),{ REM }>
<6,(6,7),{ TAX, REM }>
<7,(2,13),{ TAX }>
<7,(11,12),{ TAX }>
<7,(3,11),{ TAX }>
<7,(4,6),{ TAX }>
<7,(5,10),{ TAX }>
<7,(4,5),{ TAX }>
<8,(2,13),{ TAX, REM }>
<8,(11,12),{ TAX, REM }>

```
<8,(3,11),{ TAX, REM }>
<8,(4,6),{ TAX, REM }>
<8,(5,10),{ TAX, REM }>
<8,(4,5),{ TAX, REM }>
<12,(2,13),{ | }>
<12,(11,12),{ | }>
<12,(3,11),{ | }>
<12,(6,8),{ | }>
<12,(6,7),{ | }>
<12,(10,12),{ | }>
<12,(4,5),{ | }>

Cobertura Total = 100.000000

Media da Cobertura dos Grafo(i) = 100.000000
```

AVALIAÇÃO DE COBERTURA EM RELAÇÃO AO CRITÉRIO TODOS POTENCIAIS-DU-CAMINHOS

```
POTENCIAIS-DU-CAMINHOS que nao foram executados:
Caminhos:
Cobertura Total = 100.000000
Media da Cobertura dos Grafo(i) = 100.000000

POTENCIAIS-DU-CAMINHOS que foram executados:

Caminhos:
1 2 13 14
1 2 3 11 12 2
1 2 3 4 6 8 9 10 12 2
1 2 3 4 6 7 9 10 12 2
1 2 3 4 5 10 12 2
3 11 12 2 13 14
3 11 12 2 3
3 4 6 8 9 10 12 2 13 14
3 4 6 8 9 10 12 2 3
3 4 6 7 9 10 12 2 13 14
3 4 6 7 9 10 12 2 3
3 4 5 10 12 2 13 14
3 4 5 10 12 2 3
5 10 12 2 13 14
5 10 12 2 3 11 12
```

```
5 10 12 2 3 4 6 8 9 10
5 10 12 2 3 4 6 7 9 10
5 10 12 2 3 4 5
6 8
6 7 9 10 12 2 13 14
6 7 9 10 12 2 3 11 12
6 7 9 10 12 2 3 4 6
6 7 9 10 12 2 3 4 5 10
7 9 10 12 2 13 14
7 9 10 12 2 3 11 12
7 9 10 12 2 3 4 6
7 9 10 12 2 3 4 5 10
8 9 10 12 2 13 14
8 9 10 12 2 3 11 12
8 9 10 12 2 3 4 6
8 9 10 12 2 3 4 5 10
12 2 13 14
12 2 3 11 12
12 2 3 4 6 8 9 10 12
12 2 3 4 6 7 9 10 12
12 2 3 4 5 10 12

Cobertura Total = 100.000000

Media da Cobertura dos Grafo(i) = 100.000000
```

AVALIAÇÃO DE COBERTURA EM RELAÇÃO AO CRITÉRIO TODOS POTENCIAIS USOS/DU

```
POTENCIAIS-DU-CAMINHOS que nao foram executados:

Caminhos:

Cobertura Total = 100.000000

Media da Cobertura dos Grafo(i) = 100.000000


POTENCIAIS-DU-CAMINHOS que foram executados:

Caminhos:
1 2 13 14
1 2 3 11 12 2
1 2 3 4 6 8 9 10 12 2
1 2 3 4 6 7 9 10 12 2
1 2 3 4 5 10 12 2
3 11 12 2 13 14
3 11 12 2 3
3 4 6 8 9 10 12 2 13 14
```

```
3 4 6 8 9 10 12 2 3
3 4 6 7 9 10 12 2 13 14
3 4 6 7 9 10 12 2 3
3 4 5 10 12 2 13 14
3 4 5 10 12 2 3
5 10 12 2 13 14
5 10 12 2 3 11 12
5 10 12 2 3 4 6 8 9 10
5 10 12 2 3 4 6 7 9 10
5 10 12 2 3 4 5
6 8
6 7 9 10 12 2 13 14
6 7 9 10 12 2 3 11 12
6 7 9 10 12 2 3 4 6
6 7 9 10 12 2 3 4 5 10
7 9 10 12 2 13 14
7 9 10 12 2 3 11 12
7 9 10 12 2 3 4 6
7 9 10 12 2 3 4 5 10
8 9 10 12 2 13 14
8 9 10 12 2 3 11 12
8 9 10 12 2 3 4 6
8 9 10 12 2 3 4 5 10
12 2 13 14
12 2 3 11 12
12 2 3 4 6 8 9 10 12
12 2 3 4 6 7 9 10 12
12 2 3 4 5 10 12

Cobertura Total = 100.000000

Media da Cobertura dos Grafo(i) = 100.000000
```

APÊNDICE

F

EXEMPLOS DE ROTINAS DESENVOLVIDAS

Neste Apêndice são apresentadas, para servirem de exemplo, algumas das rotinas desenvolvidas na configuração dos módulos da POKE-TOOL. Do módulo LI ações semânticas léxicas (do analisador léxico) e rotinas semânticas (do analisador sintático). Do módulo Pokernel algumas funções.

F.1 - MÓDULO LI - ANALISADOR LÉXICO

```
void a1()
{
 tamanho = 0;
 saida.inicio = offset;
 saida.linha = linha;
 saida.label [tamanho] = ch;
 saida.label [++tamanho] = '\0';
 strcpy ( saida.classe, "CONST" );
 saida.comp = tamanho;
}
```

```
void a2()
{
 if(tamanho <= 18)
   {
     saida.label [tamanho] = ch;
     saida.label [++tamanho] = '\0';
   }
 else
    ++ tamanho;
 saida.comp = tamanho;
 saida.linha = linha;
}



void a10()
{
 char vetor [25];
 dev_ch();
 strcpy (vetor, saida.label);
 if (pesq_tab(vetor) == 0)
     strcpy (saida.classe, "IDENT");
 else strcpy (saida.classe,saida.label);
 if (strcmp (vetor,"FORMAT" ) ==0 )
     {
       ch = '#';
       dev_ch();
     }
 saida.comp = tamanho;
}



void a17()
  {
    ++ tamanho;
  }



void a25()
   {
      printf ("ARQUIVO FONTE NAO COMPATIVEL. LNO= %u",linha);
   }
```

F.2 - MÓDULO LI - ANALISADOR SINTÁTICO

```
void rot5_sem()     /*  salva inicio, comprimento, linha  */
{
    if ( if_logico == 1 )
       {
          recup_apont (0)
          grava_li ( 1, "$IF" ) ;
          recup_apont (1)
          grava_li ( 3 ,"$C" ) ;
          -- top_if_pilha ;
       }
    le_apont
    top_pilha = 0 ;
    salva_apont (top_pilha)
    if ( strcmp ( saida.label , "IF" ) == 0 )
        {
          if_logico = 1 ;
          strcpy(pilha_if[top_if_pilha++],saida.label) ;
        }
    else
          if_logico = 0 ;
}

void rot11_sem()     /*  Trata comando CONTINUE    */
{
  int i ;
    le_apont
    grava_li ( 2, "$S" ) ;
    for ( i = 0 ; i < n_labels ; i++ )
        if ( strcmp ( do_labels [i] , pilha->label [0] ) == 0 )
                {
                  /*   e' terminal de "DO"    */
                  recup_apont (0)
                  grava_li ( 1, "}" ) ;
                }
}

void rot15_sem()     /*  <$C> para Condicao do FOR  */
{
 comp_token = comp_ant ;
 grava_li(3,"$C") ;
 pred = 1 ;
 calcula_posicao ();
}
```

```
void rot17_sem()    /*    <$GT1> para GOTO computado    */
{
    final = saida.inicio ;
    comp = saida.comp ;
    recup_apont (top_pilha)
    comp_token = ( (int) ( final - ini_token )) ;
    -- comp_token ;
    ++ ini_token ;
    ++ top_pilha ;
    salva_apont (top_pilha)
    recup_apont (0)
    grava_ii ( 1, "$GT" ) ;
    recup_apont (top_pilha)
    grava_ii ( 3, "$C" ) ;
    recup_apont (1)
    grava_ii ( 1, "{" ) ;
    for ( j = 2 ; j < top_pilha - 1 ; j++ )
      {
        recup_apont (j)
        grava_ii ( 1, pilha->label[j] ) ;
      }
    zera_apont
    grava_ii ( 1, "$ROTD" ) ;
    recup_apont (top_pilha-1)
    grava_ii ( 1, "}" ) ;
}
```

F.3 - MÓDULO POKERNEL

```
/*****************************************************************************/
/* void in_label_monitor (pnewsymbol,num_no)                                 */
/* Autor: Rubens Pontes da Fonseca                                           */
/* Data: 10/06/92                                                            */
/* Versao: 1.0                                                               */
/*                                                                           */
/* FUNCAO: Monitora no' de FORMAT antes de copia-lo.                         */
/*                                                                           */
/* ENTRADAS: Rotulo e numero do no' a ser monitorado.                        */
/*                                                                           */
/* SAIDA: nenhuma.                                                           */
/*                                                                           */
/* VARIAVEIS GLOBAIS UTILIZADAS: tab_counter, num_no, do_loop.               */
/*                                                                           */
/*****************************************************************************/
void in_label_monitor(simb,num)
struct symbol * simb;
int num;

{
  if(info_no[num].to_monitor != ALREADY)
    {
      ajusta_ponta_de_prova(tab_counter);
      fprintf(mod_arq,"CALL PROVA (%d)",num);
      info_no[num].to_monitor = ALREADY;
    }
 write_no(num);
 fprintf(mod_arq,"%s",&nulo[strlen(simb->simbolo)]);/* escreve brancos
                          anteriores ao rotulo */
 copy_command(simb);
  return;
}
```

```
/*************************************************************************/
/* int do_monitor (struct symbol *)                                      */
/* Autor: Rubens Pontes da Fonseca                                       */
/* Data: 08/07/92                                                        */
/* Versao: 1.0                                                           */
/*                                                                       */
/* FUNCAO: Identifica e salva o comando terminal de um laco de DO.       */
/*                                                                       */
/* ENTRADAS:                                                             */
/*                                                                       */
/* SAIDA: nenhuma.                                                       */
/*                                                                       */
/* VARIAVEIS GLOBAIS UTILIZADAS: nenhuma.                                */
/*                                                                       */
/*************************************************************************/
int do_monitor(simb)
struct symbol * simb;
  {
    char c;
    int i;
    int label;
    long auxcomprimento;
    char temp[10]= "          ";
    auxcomprimento = simb->comprimento;
    fseek(arqfonte,simb->inicio-1,SEEK_SET);
    i = 1;
    while( auxcomprimento > 0 )
       {
         c = getc (arqfonte);
         if (isdigit(c))
           {
            temp[i] = c;
            ++i;
           }
         --auxcomprimento;
       }
       temp[i+1] =  0x00;
       label = atoi ( temp );
       return (label);
  }
```

```
/*****************************************************************************/
/*                                                                           */
/* int  verifica_destino_goto (pnewsymbol)                                   */
/* Autor: Rubens Pontes da Fonseca.                                          */
/* Data: 28/07/92                                                            */
/* Versao: 1.0                                                               */
/*                                                                           */
/* Funcao: verifica destino de GO TO dentro de um DO por meio de             */
/*         consulta `a pilha de label's.                                     */
/*                                                                           */
/* Entradas: no' sucessor .                                                  */
/*                                                                           */
/* Saida: Nenhuma.                                                           */
/*                                                                           */
/*****************************************************************************/

 int verifica_destino_goto(pnewsymbol)
 struct symbol * pnewsymbol;

 {
  int k;
  int rotulo;
  if(apont_pilha!=0)
    {
      for( k=apont_pilha; k>0; --k)
 {
           rotulo = atoi(pnewsymbol->simbolo);
    if(rotulo == pilha_label[k])
       return(1);
 }
    }
 return(0);
 }
```