GABRIEL DOS SANTOS

AUTORIA E INTERPRETAÇÃO TUTORIAL DE SOLUÇÕES ALTERNATIVAS PARA PROMOVER O ENSINO DE PROGRAMAÇÃO DE COMPUTADORES

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Alexandre Ibrahim Direne
Co-Orientador: Prof. Dr. André Luiz Pires Guedes

CURITIBA

2003

GABRIEL DOS SANTOS

AUTORIA E INTERPRETAÇÃO TUTORIAL DE SOLUÇÕES ALTERNATIVAS PARA PROMOVER O ENSINO DE PROGRAMAÇÃO DE COMPUTADORES

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Alexandre Ibrahim Direne
Co-Orientador: Prof. Dr. André Luiz Pires Guedes

CURITIBA

2003

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O presente trabalho aborda linguagens e ferramentas de autoria assim como Sistemas Tutores Inteligentes (STI) para apoiar o ensino de programação de computadores. Até então, nenhum trabalho apresentou um sistema de autoria para desenvolvimento de STIs nessa categoria e nem ferramentas independentes da linguagem ensinada. Além disso, as ferramentas existentes ou restringem demais a criatividade do aluno ou falham em prover um feedback com alto valor cognitivo agregado. Esse trabalho aborda a construção de ambas as ferramentas, procurando apresentar abordagens que tratem de ambas as limitações mencionadas. Ao final, são apresentadas algumas direções futuras que trabalhos futuros podem tomar a partir do estado atual de desenvolvimento deste trabalho.

ABSTRACT

This work approaches languages and authoring tools as well as Intelligent Tutoring Systems (ITS) to support teaching of computer programming. So far, no work presented an authoring system to develop tutors for programming and neither have anyone developed a language-independent tool. Besides, the existing tools either restrict too much the student's creativity or fail to provide a feedback useful for cognitive development. This work presents approaches to address both limitations. At the end, we present some of the possible directions a future work could follow from the present state of development in this work.

1

CAPÍTULO 1

INTRODUÇÃO

Ensinar, de uma forma geral, é uma tarefa de enorme complexidade, e os processos cognitivos envolvidos ainda são desconhecidos mesmo para profissionais humanos. Apesar de existirem metodologias, poucas são aceitas universalmente, e mesmo as mais consolidadas não formalizam um processo claro de procedimento de ensino, deixando muitos aspectos delegados ao instrutor.

Assim, a questão de como ensinar e qual a melhor maneira ainda reside muito no plano intuitivo e pragmático. Uma vez que depende da intuição e requer habilidades que extrapolam o domínio do conhecimento a ser tranferido, a tarefa de ensinar, é uma tarefa difícil, se é que é possível, de ser imitada por completo e de maneira eficiente dentro de um ambiente computacional. A tarefa de programação, em particular, demanda uma enorme carga cognitiva, uma vez que não se restringe apenas à capacidade de trabalhar com uma linguagem de programação[15], como veremos mais adiante nessa seção.

Dentro das tarefas envolvidas no processo de ensino em qualquer domínio, tão importante quanto transmitir o conhecimento, é avaliar quão bem o mesmo foi absovido, estimando o progresso do aprendiz. Tendo em mãos o perfil do aluno, torna-se possível determinar os parâmetros necessários para planejar o processo pedagógico de modo a atender as necessidades específicas do aprendiz.

O presente trabalho trata justamente de uma parte do processo de avaliação, transposto no domínio de programação como o diagnóstico da solução proposta pelo aluno. Embora o processo completo de avaliação deva, de fato, prover parâmetros para o sistema determinar a direção a ser tomada, neste trabalho nos restringimos a analisar a solução e relatar observações que sejam úteis ao aprendiz no sentido de entender quais conceitos não foram corretamente observados.

Ainda nesse capítulo, vamos nos aprofundar na questão da problemática de ensino de programação, formalizar os objetivos e ressaltar os aspectos que constituem as contribuições desse trabalho.

1.1 Complexidade do aprendizado de programação

A atividade de programar pode ser descrita como uma meta-atividade, uma vez que consiste em desenvolver sistemas que, por sua vez, prestarão auxílio ou desenvolverão outras atividades. Dessa forma, o conhecimendo de um programador não está restrito apenas ao domínio de programação, mas deve se estender para abranger o domínio das atividades fim dos sistemas a serem desenvolvidos.

A capacidade de analisar novos problemas, muitas vezes em domínio muito distintos da programação, figura entre as primeiras habilidades que se constata em um profissional competente. O processo de análise, consiste tanto na capacidade de decompor problemas, como de compor novas soluções a partir de conhecimento anterior. Do ponto de vista da elaboração de um programa, um programador deve saber organizar as idéias, tal como em uma linguagem natural, ao mesmo tempo que deve saber modelar o problema em estruturas e procedimentos que muito se relacionam com a matemática, em particular na questão algorítmica.

Podemos dividir a atividade sob dois aspectos: a aquisição de conhecimentos de princípios de programação e a perícia em programação [14]. O conhecimento de **princípio** abrange os detalhes do rígido formalistmo sintático e semântico das estruturas linguísticas de uma Linguagem de Programação (LP), ou seja, a compreensão da forma de escrita e dos resultados. O conhecimento de **perícia** abrange a habilidade necessária para integrar detalhes sintáticos e semânticos isolados em um planejamento maior que agregue vários fragmentos para compor um programa completo. Essa habilidade às vezes é também conhecida como habilidade tática[14], e considera justamente a habilidade de decompor problemas e de compor soluções e partir de fragmentos que resolvem problemas mais simples.

Por exemplo, a construção de um programa pode partir de um plano que integre vários fragmentos: um para realizar a leitura da memória secundária, um para tomar uma decisão, um para repetir uma operação aritmética de somatório, e um para imprimir o resultado. Observe que antes de iniciar o processo de construção, o aprendiz deve ser capaz de decompor o problema e entender que as partes sugeridas eram de fato necessárias.

Sistemas tutores inteligentes e ambientes exploratórios para aprendizagem de programação de computadores procuraram abordar estes dois tipos de conhecimento (princípio e perícia) por meio de diversas técnicas distintas[1]. Entretanto, nenhuma das contribuições de pesquisa de representação de soluções de problemas se preocupou em adotar abordagens genéricas de representação de soluções de problemas por meio de linguagens e ferramentas de autoria. A existência de tais ferramentas permitiria que autores de cursos sobre programação de computadores pudessem estender o material de ensino de acordo com a relevância das classes de problemas que deveriam ser abordados, além de prover comentários pertinentes à abordagem adotada para ensino.

Vários trabalhos contemplaram o diagnóstico de programa do aluno [1, 18, 20, 23, 24, 22, 35, 27] e foram contribuições importantes para a área. Porém, dadas as limitações dos métodos e ferramentas para o diagnóstico de programas de aprendizes, torna-se importante a possibilidade de dar espaço amplo à atuação humana, por meio de autoria, para apresentar variações de programas que solucionam um mesmo problema. Para tanto,

[1]No capítulo2 serão abordadas, de maneira crítica, várias dessas técnicas, com enfoque particular nos sistemas de diagnóstico de programas construídos por aprendizes.

torna-se essencial a re-engenharia das máquinas de ensino para que adotem mecanismos muito mais genéricos. No passado, dois sistemas que procuraram uma abordagem mais genérica foram dedicados ao treinamento da operação de dispositivos digitais[13] e à manipulação de expressões algébricas[10]. Todavia, nenhum deles abordou a dimensão de linguagens e ferramentas associadas à autoria e ao ensino de programação de computadores.

1.2 Objetivos Específicos e Contribuições

O trabalho tem como objetivos apresentar novos conceitos e as arquiteturas de duas ferramentas:

- A ferramenta de autoria, a ser utilizada por um autor de curso para produzir o material (nesse caso problemas propostos, soluções entre outras). Essa ferramenta dispõe dos seguintes recursos:
  - Inserção e Edição de questões propostas assim como de suas soluções de referência;
  - Inserção e Edição de padrões de erro assim como de suas explicações.
- A ferramenta de diagnóstico, a qual deverá ser capaz de analisar uma solução proposta pelo aluno e fazer comentários pertinentes quanto às impropriedades detectadas na solução. A ferramenta de diagnóstico deve atender às seguintes premissas:
  - Ser flexível o suficiente para aceitar soluções que não sejam idênticas às soluções de referência propostas pelo professor;
  - Disponibilizar meios de prover feedback que agregue um alto valor cognitivo;
  - Ser genérica quanto à linguagem, sendo capaz de tratar com qualquer linguagem desde que esta seja consistente com o paradigma imperativo.

Sendo assim, as principais contribuições desse trabalho são as de prover uma ferramenta de autoria para o ensino de programação de computadores e estender ferramentas de diagnóstico anteriores, permitindo incorporar feedback com alto valor cognitivo. Ainda, este trabalho surge como parte da tentativa de reavivar a pesquisa em STIs na área de programação, a qual têm sido relativamente pouco ativa nesses últimos dez anos.

CAPÍTULO 2

RESENHA LITERÁRIA

Nesse capítulo é realizado um levantamento de trabalhos relacionados, bem como uma avaliação crítica dos resultados dos trabalhos que procuraram tratar o mesmo problema envolvido nessa dissertação. Antes da leitura dessa resenha, vale ressaltar que, desde 1990, poucos trabalhos, se é que algum, envolveu STIs no domínio de programação, o que justifica o inexistência de referências a trabalhos muito recentes.

2.1 Sistemas Tutores Inteligentes

O conceito de Sistema Tutorial Inteligente surgiu da idéia de aplicar técnicas de Inteligência Artificial (IA) em programas com objetivos educacionais. Dessa forma, STIs são programas de computadores projetados para incorporar técnicas provenientes do campo de IA de modo a prover tutores que saibam *como* ensinar, *o que* ensinar e conheçam *quem* eles estão ensinando[30].

O projeto de desenvolvimento de tais sistemas se situa na intersecção das áreas de ciência da computação, psicologia cognitiva e pesquisa educacional. A área criada por essa intersecção é normalmente denominada *Ciência Cognitiva*.

Uma das formas mais utilizadas para subdividir um tutor inteligente, embora não seja uma visão indiscutível e nem mesmo adotada por todos os trabalhos, apresenta o STI através de quatro módulos, cada qual com sua especialidade:

- **Representação do Domínio**: O módulo de representação do domínio consiste em um conjunto de objetos, regras ou elementos que constituem um solucionador de problemas do domínio a ser tutorado.

- **Modelo do Estudante/Aprendiz**: O modelo do aprendiz é uma estrutura interna do sistema que representa o perfil do aprendiz, como até então deduzido pelo STI, com relação à proficiência no domínio tutorado.

- **Interface**: O módulo de interface é responsável por prover um meio ao aluno para interagir com o sistema, e para o sistema se apresentar ao aluno.

  Em alguns trabalhos[14], esse módulo é suprimido do modelo, uma vez que a maioria dos estudos se concentra nos outros 3 módulos.

- **Módulo Didático-Pedagógico**: O módulo didático pedagógico representa o especialista em ensinar. Esse módulo tem como responsabilidade gerenciar o sequenciamento de ensino de curto e médio prazos, avaliar o aluno e interagir, quando

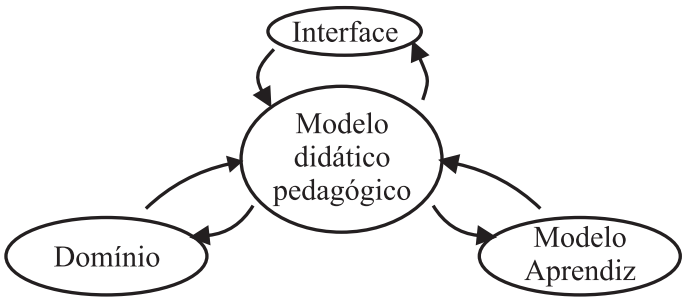


Figura 2.1: Arquitetura Geral de STIs

necessário, com todos os outros módulos do sistema, seja para tomar alguma decisão, como para atualizar os diversos modelos existentes.

Em alguns casos, esse módulo é visto em duas partes[14]: o módulo de ensino e o módulo de estratégias tutoriais. Nesse caso, o módulo de ensino é responsável por determinar quais estratégias devem ser aplicadas em cada momento.

Essa mesma arquitetura está claramente ilustrada na figura 2.1.

2.1.1 Diagnóstico Automático de Programas

Também conhecidos como diagnosticadores (*Bug Finders*), os sistemas de diagnóstico automáticos são ferramentas que têm como objetivo encontrar erros no programa provido por um aluno e classificá-los, com o objetivo de orientar o aluno quanto aos erros cometidos, além de ser uma ferramenta muito importante para o próprio STI no preenchimento do modelo do aprendiz.

Essa categoria de trabalhos é de grande interesse científico porque cobre vários dos mais importantes projetos de pesquisa nos quais os programas eram, na medida do possível, capazes de detectar discrepâncias no programa do aprendiz, tarefa que será tratada em parte pelo presente trabalho. Os sistemas de diagnóstico podem ser encaixados em três categorias[14]:

- **Solução de Referência (Specimen Answer):** Sistemas que utilizam um programa de referência para confrontar com a solução do aprendiz. Um exemplo de trabalho nesta área é o sistema LAURA[1].

- **Análise de Especificação (Specification Analysys):** Sistemas que avaliam o programa do aluno segundo a espeficicação do problema. Alguns exemplos de sistemas que utilizam análise de especificação são MYCROFT[18], AURAC[20], PUDSY[27] e PROUST[23, 24, 22, 35].

- **Diálogo de Depuração (Debugging Dialogue):** Sistemas dessa categoria entram em um diálogo com o aluno, tentando orientá-lo a encontrar o erro. Um

exemplo de sistema que cai nessa classe é o desenvolvido por Shapiro[34], para trabalhar com linguagens declarativas (em particular, com o Prolog). Não entraremos em detalhes desse tipo de sistema por não ser diretamente relavante para o foco deste trabalho.

2.1.1.1 Diagnóstico por Solução de Referência

LAURA foi um sistema único no seu ramo. Tal como sugere sua categorização, LAURA se utiliza de uma solução de referência (um programa) para diagnosticar o programa do aluno. Para tanto, o sistema construía o grafo de fluxo de controle de ambos os programas. O estágio que se segue é uma tentativa de transformar o grafo do aluno e casar com o grafo de referência. São construídas várias hipóteses sobre a correlação entre os nodos e arcos do grafo. Uma vez que seja encontrada uma correspondência mais aceitável, tomam lugar uma série de transformações, tais como permutar passos independentes da ordem de computação.

Uma vez que o sistema se utiliza de um processo sofisticado de casamentoo de padrões, a grande vantagem de LAURA estava na capacidade de aceitar programas corretos, mas que não fossem idênticos ao programa de referência. Essa capacidade é importante, uma vez que permite que o aluno exerça em parte sua criatividade e não seja coagido a seguir um caminho específico. Por outro lado, o grande defeito reside na capacidade de resposta do sistema. Para o caso de existência de erros, LAURA é capaz apenas de diagnosticar erros em baixo nível de abstração, ou seja, erros pequenos que não levam em conta o contexto principal de solução do problema abordado no exercício em questão. Porque o sistema trabalha de forma genérica, as mensagens de erro são construídas dinamicamente, e portanto o sistema é incapaz de considerar o contexto e apresentar informações relativas à capacidade de planejamento do aluno.

De qualquer forma, LAURA tem um papel fundamental dentro dessa proposta, uma vez que é o sistema que mais se assemelha à ferramenta de diagnóstico que nos propomos a desenvolver. Entraremos em mais detalhes no capítulo 7.

2.1.1.2 Diagnóstico por Análise de Especificação

Na classe de sistemas de diagnóstico por análise de especificação, existe uma maior variedade de sistemas. O grande atrativo de todo sistema dessa categoria é a capacidade de avaliar o programa do aluno em um contexto mais global, o que permite ao sistema prover explicações mais adaptadas ao desvio ocorrido no programa do aprendiz.

Mycroft, desenvolvido por Goldstein[18], foi um sistema desenvolvido para detectar e reparar erros em programas projetados para produzir desenhos com linhas retas na lingua-

gem LOGO[32]. O problema era especificado através de asserções sobre as propriedades geométricas do desenho que supostamente deveria ser produzido.

A grande vantagem do Mycroft reside na capacidade de gerar um plano a partir do programa do aluno que leve o programa a atingir os objetivos propostos. Caso não seja possível atingir o objetivo, **Mycroft** é capaz de sugerir e testar mudanças que corrijam o programa.

Por mais impressionante que fosse, o sistema se beneficiou muito devido à escolha do domínio de desenho, em particular, na capacidade de utilizar propriedades geométricas para especificar o problema. Esta característica simplificou muito a questão de determinar se o programa atinge o objetivo. Ainda, as estratégias de análise estão intrinsicamente ligadas à simplificidade da linguagem LOGO.

Infelizmente, essas simplificações são demasiadamente restritivas. Em particular, no contexto alvo desse trabalho, domínios genéricos de programação, a tarefa de determinar que o objetivo foi atingido é uma questão crucial e não pode estar atrelada a um domínio tão restrito como figuras geométricas.

Aurac é outro sistema que se beneficiava da simplicidade da linguagem que tutora. O Aurac é um sistema desenvolvido por Hasermer[20] para depurar programas escritos em uma liguagem chamada *Solo*, a qual era utilizada para manipular uma base de dados de relações.

O processo de diagnóstico passava por 3 fases de análise: casamento de produções que representam condições de erro, casamento de segmentos de código com um conjunto de clichês[1] armazenados em uma biblioteca e, por fim, análise do fluxo de dados.

Uma vez que todos os mecanismos de análise do Aurac consideram segmentos de código, portanto realizando uma análise de mais alto nível, o Aurac tinha condições de apresentar mensagens de erro significativas.

Mas novamente, a natureza restritiva da linguagem Solo é que viabilizou técnicas como a utilização de uma biblioteca de clichês. Em um domínio mais genérico, uma técnica como essa seria provavelmente inviável devido à variedade de fragmentos que o programador pode criar.

Pudsy, desenvolvido por Lukey[27], era um sistema que trabalhava a partir de uma especificação abstrata, capaz de eliminar uma limitada classe de erros de pequenos programas em Pascal. O sistema utilizava dois tipos de depuração.

Na primeira eram procuradas pistas no código que indicassem erros, através de uma busca *bottom-up*. Na segunda fase, o sistema realizava uma análise de fluxo de dados dos segmentos, gerando uma descrição do que seria a especificação do programa analisado.

[1] Clichês são fragmentos corretos de código utilizados com freqüência.

O ponto forte no Pudsy é trabalhar com uma linguagem de programação mais genérica e ainda assim ser capaz de não apenas diagnoticar erros, mas também de corrigí-los.

Por outro lado, o próprio Lukey apontou como alguns dos problemas a falta de flexibilidade, a incapacidade de avaliar o comportamento do programa em tempo de execução, além de suas abordagens serem um tanto *ad-hoc*. Além disso, apesar de estar dentro da classe de análise de especificação, Pudsy era limitado na capacidade de explicar os erros ocorridos[14], principalmente porque não tinha expectativas sobre que tipo de erro poderia encontrar.

Proust foi outro sistema de diagnóstico para programas em Pascal, desenvolvido por Johnson *et al*[23, 24, 22, 35], certamente figurando entre os mais impressionantes.

Nesse sistema, a especificação do programa é expressa através de um conjunto de objetivos. Cada objetivo é associado a um objeto que armazena informação declarativa sobre o mesmo, bem como relaciona um ou mais planos que atingem o objetivo proposto. Cada plano é associado a um gabarito (**template**) que pode ser usado para casamento com códigos em pascal, além da possibilidade de ter um conjunto de objetivos próprios, ou seja, apresentando assim uma definição recursiva.

A grande vantagem dessa abordagem é a capacidade de relacionar fragmentos de código aos objetivos presentes na especificação. Mais do que em qualquer outro sistema, o Proust é capaz de fazer comentários impressionantes sobre problemas de lógica no programa, inclusive sendo capaz de surgerir dados de entrada para os quais o programa analisado não seria bem sucedido.

Por outro lado, o sistema é muito sensível a pequenas diferenças e incapaz de aceitar programs equivalentes, dessa forma restringindo muito os programas do aluno. Ainda, o Proust é capaz de lidar apenas com um subconjunto do Pascal e, dentro desse contexto, apenas com um subconjunto de problemas.

2.1.2 Tipos de intervenção

Um sistema tutorial de programação pode ser categorizado segundo os momentos nos quais realiza uma intervenção. Veremos nessa seção um pouco de cada uma das duas abordagens utilizadas nos trabalhos publicados até então.

2.1.2.1 Intervenção durante o evento de programação

A primeira abordagem a ser utilizada em um sistema tutorial de programação foi chamada de análise **durante o evento**. Nessa abordagem, o sistema monitora o aprendiz durante o processo de criação do programa. No momento em que o sistema detecta que o aluno está se desviando do caminho, imediatamente o sistema realiza uma intervenção.

O sistema pioneiro foi o **Malt** de Koffman *et al*[26], sistema que beirava a classificação entre CAI e STI. O Malt foi um sistema construído para ensinar linguagem de máquina. A estratégia de ensino era muito simples: ele gerava um programa e apresentava ao aluno. A medida que o aluno digitava cada linha, existia a probabilidade do sistema intervir. A medida que o aluno era considerado mais experiente, a probabilidade diminuía.

O sistema determinava se o programa do aluno estava correto ou baseado no casamento direto com código que o sistema gerou ou com uma pequena simulação de execução do programa do estudante.

Dentro dessa classe, um dos sistemas mais famosos foi o **Greaterp**, o tutor de LISP desenvolvido por Anderson e seus colaboradores[6, 7, 33]. O GREATERP foi construído seguindo as teorias e arquiteturas do ACT e o ACTP [3] de Anderson, e seus aspectos cognitivos foram formalizados em um trabalho posterior[5].

O sistema mantinha dois conjuntos de produções: o primeiro representava o conjunto de códigos bem formados, ou seja, considerados corretos, enquanto o segundo representava um conjunto de erros, no qual, a cada produção, estava associada uma mensagem de erro. Este segundo conjunto é também conhecido como *bug catalogue*.

O sistema apresentava um problema para o qual o aluno deveria escrever um programa em LISP. A medida que o aluno digitava, para cada token completado o sistema tentava fazer o casamento do código com alguma produção, seja no conjunto de produções corretas como no *bug catalogue*. Em caso de casar com um erro, o sistema apresentava a mensagem associada imediatamente no momento em que o erro foi detectado. Em caso de acerto, o sistema simplesmente permitia que o aluno continuasse. Finalmente, em caso de nenhuma regra casar, o sistema parava e forçava o aluno a escolhe um entre os possíveis caminhos existentes (fossem de acerto ou de erro).

A grande vantagem do sistema de Anderson era a capacidade de resolver os problemas propostos, ou seja, o sistema "entendia" a tarefa que estava propondo ao aprendiz. Por outro lado, o sistema era extremamente rígido na sua metodologia de não permitir que o aluno errasse, o que é coerente com os princípios de não permitir que o aluno persista no erro, por medo de que ele aprenda o erro.

O maior problema do Greaterp e dos sistemas baseados no ACT em geral é que a estratégia de ensino, o modelo do estudante e o domínio de ensinos são muito dependentes entre si[40], o que significa que a estratégia pedagógica depende excessivamente do domínio. Se o domínio muda, seja por uma produção, a estratégia também é modificada, o que impede o sistema de ter um comportamento pedagógico modular e que possa ser reutilizado.

2.1.3 Intervenção posterior ao evento de programação

Nesse tipo de abordagem, o aluno fica livre para programar da forma como achar mais conveniente e seu programa o sistema só intervêm e comenta sobre o programa no momento em que o aluno assinalar que a solução está concluída.

A atuação de sistemas nessa categoria tem por trás uma teoria cognitiva diferenciada, na qual é permitido ao aluno prosseguir e errar e somente interferir em um momento futuro. Se esta é ou não a melhor abordagem, é motivo de debate.

Vários sistemas se encaixam nesta categoria, em particular todos os sistemas mencionados na seção 2.1.1. Laura, Aurac, Prost, Pudsy, Shapiro e Mycroft todos são sistemas ativados somente após o aprendiz ter concluído o programa.

2.2 Linguagens e ferramentas de autoria

Uma vez que um dos objetivos do trabalho proposto é o de desenvolver uma ferramenta de autoria para tutores de programação, não poderiámos deixar de mencionar outros trabalhos de autoria de STI existentes.

Um sistema de autoria de STI tem como objetivo prover uma ferramenta relativamente genérica para construção de sistemas tutores minimizando a necessidade de conhecimento fora do domínio a ser tutorado, dessa forma tornando a ferramenta acessível para especialistas do domínio.

O primeiro trabalho, e ao qual se deve muito no sentido de tomar conhecimento dos estado da arte em autoria de STI, é o desenvolvido por Murray[29], no qual é apresentada uma categorização de vários dos sistemas existentes. Embora o trabalho de Murray não contemple todos os sistemas de autoria já construídos, um aspecto importante é que este provê uma amostragem que abrange a área como um todo. Murray divide os sistemas de autoria segundo o tipo de tutores que produz. As categorias são as seguintes:

- Planejamento e sequenciamento de currículo;
- Estratégias tutoriais;
- Simulação de dispositivos e treinamento com equipamentos;
- Sistema especialista no domínio;
- Sistemas de propósito específico.

Uma observação importante é que a categorização apresentada não é necessariamente disjunta, ou seja, um sistema de autoria pode estar sob mais de uma categoria ao mesmo tmepo.

Uma vez que não é objetivo deste trabalho, não vamos detalhar cada categoria, a não ser a de *Sistemas Especialistas no Domínio*. Um Sistema Especialista no Domínio é, a grosso modo, o tipo de sistema que tem conhecimento e de alguma forma é capaz de interpretar o domínio a ser tutorado. Dois trabalhos interessantes se destacam nessa categoria, o **Demonstrat8** de Blessing[10] e o **RIDES** de Munro[28]. A característica comum de ambos os sistemas é a forma de autoria através de demonstração.

Em um sistema de autoria por demonstração, o autor pode criar conteúdo tutorial, ou seja, aspectos de estratégia pedagógica e do domínio, demonstrando como um determinado problema é solucionado. A autoria por demonstração, dentro do contexto de programação, pode ser um programa que o autor provê como exemplo de solução para um determinado problema.

O Demonstr8 trabalha principalmente com o domínio da aritmética e da álgebra. O objetivo do Demonstr8 é a autoria de sistemas tutores semelhantes aos ACT de Anderson. Para tanto, é utilizado um mecanismo de analogia muito semelhante ao ACT-R de Anderson[4], o qual a partir de uma série de exemplos de como resolver um determinado problema, é capaz de induzir as regras de produções necessárias. O Demonstr8 é relevante nesse contexto justamente porque este figura entre os sistemas de autoria de domínio mais semelhante ao de programação, justificado pela afinidade da aritmética e da álgebra com a tarefa de elaborar algoritmos.

O trabalho de Murray é uma importante evidência de que não existem sistemas de autoria para tutores de programação, ressaltando o interesse na construção do mesmo.

CAPÍTULO 3

NOTAÇÕES E CONCEITOS FUNDAMENTAIS

Neste capítulo são apresentados vários conceitos importantes necessários para compreender o presente trabalho como um todo. Outros conceitos chave mais voltados para o entendimento de um capítulo ou seção específicos serão introduzidos na medida do necessário.

3.1 Conceitos Formais

- Dada uma relação σ e dois elementos a e b, um fecho transitivo de uma relação σ, é uma relação denotada por σ^* tal qual se $a\ \sigma^*\ b$, então:
  - $a\ \sigma\ b$ ou
  - existe um elemento c tal que, $a\ \sigma\ c$ e $c\ \sigma^*\ b$.

3.2 Processos de Casamento, Padrões e Variáveis

- Um padrão é um elemento que pode ser composto de elementos variáveis, e descreve um conjunto de elementos possíveis, obtidos no momento de fixação de um valor para todas as variáveis que o compõem.
- Uma instância é uma variável valorada, ou seja, atribuída de um valor qualquer do domínio da mesma. Também é possível uma instância de um padrão, a qual é criada quando todas as variáveis que o compõem são valoradas.
- *Idenficação* é o processo de verificar que dois elementos são equivalentes em tipo, morfologia e função. Por exemplo, dado um programa, A, identificar uma instrução deste com outra em um programa B significa determinar que ambas as instruções tem o mesmo papel, em seus devidos contextos.
- *Casamento* de um padrão com um texto consiste no processo de determinar se o texto é uma instância desse padrão.
- *Casamento* de dois grafos significa identificar todas as instruções de um grafo no outro.

3.3 Escopos e blocos

O escopo de um programa consiste na região de ação do programa, na qual todas as instruções do programa são executadas. Por sua vez, o escopo de uma instrução é a região de ação da instrução.

Por exemplo, o escopo de uma iteração é a região na qual estão contidas todas as instruções que deverão ser repetidas se a condição da iteração for verdadeira (ou falsa, dependo do tipo de iteração). O escopo de uma condicional consiste da região com as instruções que serão executadas se uma determinada condição for verdadeira.

O conjunto de instruções contidas no escopo é o bloco de comandos da instrução.

3.4 Programas

- Texto fonte é o texto do programa escrito em uma linguagem de programação qualquer, em particular nesse trabalho em um subconjunto da linguagem "C";
- Código *inócuo* é aquele que, embora esteja no programa, não tem nenhum efeito e poderia até mesmo ser retirado sem prejuízo à semântica do programa;
- *Definição* de uma variável v em um programa é uma instrução qualquer que atribui valor a essa variável;
- *Referência* a uma variável v é uma expressão ou instrução que utiliza o valor de v;
- A condição de contorno de uma iteração, também conhecida como condição de parada, é a expressão que determina se o fluxo do programa deve adentrar o escopo;
- A condição de execução da condional é a expressão que determina quais ações condicionadas devem ser executadas;
- A condição de controle de um escopo é o termo usado para generalizar as condições anteriores;

3.5 Grafos Usados no Sistema

- O *Grafo Original* é o grafo que representa o código do programa antes de qualquer alteração ser realizada sobre o mesmo;
- O *Grafo de Trabalho* é aquele no qual as transformações são realizadas durante o processo de comparação;
- Vértice de Dado: é o termo usado neste trabalho para denotar o vértice que representa uma instrução que manipula dados;

- Vértice de Fluxo: é o termo usado neste trabalho para denotar o vértice que representa uma instrução que efetua um desvio ou controle de fluxo, como uma ramificação (branch), por exemplo;

3.6 Dependência de Dados e Dependência de Controle

Uma dependência é uma relação que indica que o resultado de uma determinada instrução influencia o resultado das instruções que dependem da mesma. Dizemos que uma instrução i *precede* uma instrução j se j depende de i. As dependências em um programa podem ser de dois tipos:

- Se existe uma *Dependência de Dados* entre i e j, ou seja i depende de j por dados, j realiza uma definição em alguma variável referenciada por i.

- Se existe uma *Dependência de Controle* entre i e j, ou seja i depende de j por controle, j controla um desvio condicional de controle que pode determinar se i é ou não executada;

3.7 Complexidade Computacional e Decidibilidade

Uma breve revisão dos conceitos relacionados à complexidade computacional é de grande importância para compreender, principalmente, o tamanho do problema que está sendo abordado nesse trabalho. Esse assunto é abordado na maioria dos livros de algoritmos, tal como nos livros de Szwarcfiter e Markenzon[36], Cormen[12], entre outros.

A complexidade computacional de um algoritmo, também chamada de ordem assintótica, é uma medida O calculada em função do tamanho da entrada n do problema que expressa a ordem de magnitude do tempo que o algoritmo leva para resolver o problema. Dado um problema qualquer a ser resolvido por meio de um algoritmo, um dos aspectos mais importantes a ser analisado é a complexidade para resolver esse problema. Dessa forma, quando mencionamos a complexidade de resolver um problema, de fato estamos falando da complexidade do que se espera do melhor algoritmo que resolva este problema.

As categorias de problemas relevantes para a compreensão desse trabalho são as seguintes:

- Complexidade **P**: problemas nessa categoria podem ser resolvidos por algoritmos de complexidade polinomial, ou seja, a expressão que estima a complexidade é um polinômio;

- Complexidade **NP**: é um conjunto de problemas que não são polinomiais, ou seja, a ordem é expressa por uma equação que cresce mais rápido que um polinômio, tal

como equações exponenciais ou fatoriais. Essa classe contém os problemas chamados *NP-Completos*, objeto de grande interesse de estudos.

A maioria dos problemas abordados nesse trabalho estão na classe dos NP-Completos, cobertos com abrangência por Garey e Johnson[17] e Papadimitriou[31]. Formalmente, um problema C é NP-Completo se está em NP e se todos os outros problemas NP-Completos são redutíveis para C. Uma redução significa que, para um dado problema L, existe um algoritmo em tempo polinomial que transforma instâncias de L em instâncias de C.

Um assunto relacionado à complexidade é o de *Decibilidade*, mais precisamente *Indecidibilidade*. Um problema Indecidível é aquele para o qual não existe um algoritmo que receba uma instância do problema e dê uma resposta *sim* ou *não*[21]. De fato, um problema indecidível não é sequer computável, ou seja, é provado que não existe algoritmo que o solucione.

O problema de determinar se um algoritmo arbitrário qualquer pára foi o primeiro a ser provado como sendo indecidível, por Turing[38]. A prova de que um problema é indecidível pode ser através de uma redução do problema para qualquer um que seja comprovadamente indecidível, em particular, o da parada.

3.8 Satisfação de Restrição

Dado um conjunto de variáveis, cada qual com seu respectivo domínio, e um conjunto de restrições sobre estas variáveis, o problema de Satisfação de Restrição[37] consiste em instanciar todas as variáveis de modo que nenhuma restrição seja violada. O próprio SAT[17, 31] é um problema de satisfação de restrições (daí no nome, Satisfiability), no qual um conjunto de variáveis booleanas deve receber um valor, verdadeiro ou falso, de modo que todas as expressões lógicas associadas resultem em verdadeiro.

Sabendo que o SAT é um problema de Satisfação de Restrição, é natural concluir que os problemas dessa classe são NP-Completos e, de fato, em trabalhos como o de Tsang [37], essa afirmação é verificada.

O que torna essa classe de problemas interessante é que existem algoritmos de programação por restrição [37] bastante eficientes para solucionar esse problema. Algumas das heurísticas mais usadas são a redução de domínio à medida que as variáveis são instanciadas, a ordem de instanciação das variáveis e verificação heurística adiantada.

A estratégia de instanciação consiste em escolher uma variável v qualquer do problema e atribuir um dos valores de seu domínio. No momento em que isso acontece, todas as restrições são verificadas tendo em vista que esta variável já tem valor. No momento em que as restrições são verificadas, automaticamente o domínio das variáveis envolvidas em cada restrição é reduzido em função da instanciação. Se alguma variável ficar com domínio vazio, o algoritmo considera que falhou e desinstancia v e tenta outro valor. Através desse

procedimento, o algoritmo rapidamente identifica se um determinado ramo na árvore de busca vai ou não ser frutífero.

Observe que, segundo a estratégia proposta para instanciação, a ordem de instanciação pode fazer diferença quanto à eficiência do algoritmo. A questão é determinar uma ordem que reduza ao máximo o número de tentativas de instanciação. Com esse fim, é usada uma ordenação heurística, na qual as variáveis mais restritas, ou seja, que influenciam um número maior de restrições e que têm o menor domínio, são instanciadas com antecedência. Não é necessário ordenar explicitamente, mas apenas determinar em cada momento qual será a próxima variável a ser instanciada. A heurística está na determinação do quão restrita é uma certa variável.

Por fim, outra heurística popular, é a verificação adiantada, que consiste em fazer uma pré-avaliação do impacto de uma instanciação reduzindo ao máximo o tamanho dos domínios das variáveis. Ao reduzir o tamanho do domínio, há benefício em pelo menos dois aspectos: se uma variável ficar sem domínio, imediatamente é determinado que este caminho é inviável e o próprio tamanho do domínio pode ser levado em conta na ordem de instanciação também.

Existem variações possíveis tanto em cima da estratégia de instanciação como das heurísticas, mas todas seguem as mesmas regras gerais. A grande vantagem proporcionada por estas heurísticas é que, ao determinar o mais cedo possível que um caminho não leva à solução, uma grande fatia das hipóteses são eliminadas em função do corte de uma determinada árvore.

CAPÍTULO 4

ARQUITETURA E MODELOS

Até então, discorremos sobre a complexidade do problema e fizemos uma breve revisão crítica de trabalhos que abordaram problemas semelhantes ou relacionados. O objetivo deste capítulo é delinear a arquitetura geral da abordagem solucionadora e detalhar a representação utilizada.

4.1 Arquitetura Geral

A arquitetura geral do sistema apresenta a relação das ferramentas envolvidas, tal como como ilustrada na Figura 4.1. Nessa figura é possível observar a ferramenta de autoria e a de interpretação tutorial, os atores que interagem com cada uma dessas ferramentas respectivamente (professor/autor e aprendiz) e os componentes criados e ou editados pela ferramenta de autoria, os quais são armazenados e utilizados como entrada na ferramenta de interpretação. Figuram entre estes componentes um conjunto de Classes de Erros do Aluno (CEA) e um conjunto de problemas propostos de programação, por sua vez constituídos do respectivo Enunciado de Problema Proposto (EPP) e de um conjunto de alternativas de soluções de referência para este problema (SRP). Dessa forma, o fluxo geral do sistema consiste na atuação do autor para, através da ferramenta de autoria, criar e armazenar material de curso, enquanto o aprendiz ativa a ferramenta de intrepretação e tenta resolver os problemas propostos no material.

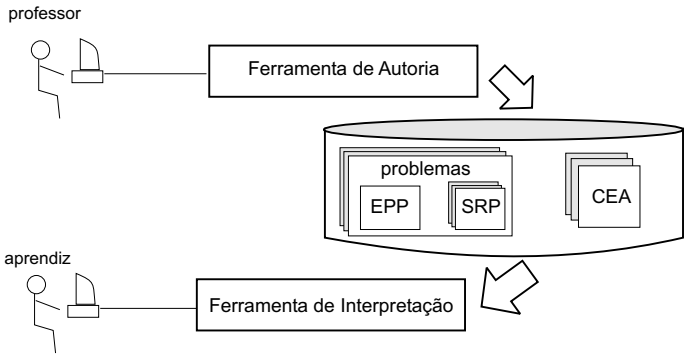


Figura 4.1: Arquitetura geral do Sistema

```
main()
{
   int  b;

   for( f=0; 10−f ;  f=f+1){
      write ( f );
   }

   write ( f );
}
```

Figura 4.2: Exemplo de grafo de fluxo de dados representando um programa

4.2 Representação

As soluções, representadas pelos programas escritos pelo autor de curso e pelo aprendiz, são representadas ao longo do programa por três modelos distintos, o código fonte na forma texto, através de um Grafo de Fluxo de Controle e por uma linguagem intermediária usada no processo de casamento de padrão. Nesta seção serão apresentadas as estruturas e um breve contexto para ilustrar a finalidade de cada uma.

4.2.1 Programa Fonte

O Programa Fonte é a forma original da solução. A linguagem objeto neste trabalho é um subconjunto da linguagem "C", mas poderia ser qualquer outra linguagem imperativa, desde que dentro das limitações do interpretador. O uso dessa representação como meio de entrada de soluções é a aplicação mais natural, mas também é essencial no processo de explicação (**feedback**) do sistema. No momento em que o aprendiz recebe as informações de diagnóstico, é importante que sejam usados os programas originais, apontando os elementos incorretos e deconhecidos claramente. Na Figura 4.2 é apresentado um exemplo de programa que será usado como base para demonstrar as outras formas de representação.

4.2.2 Grafo de Fluxo de Controle (GFC)

O grafo é a representação utilizada pelo interpretador tutorial como meio de análise, comparação e detecção de impropriedades. Embora a representação seja inicialmente através de um GFC, uma gama muito maior de informações é usada e o grafo mais importante do ponto de vista de comparação é na verdade o que representa as dependências dos vértices, como será visto claramente no Capítulo 7.

Um Grafo de Fluxo de Controle[2] (GFC) é um grafo direcionado e conexo utilizado como representação de um programa, em particular do fluxo de execução, no qual os vértices do grafo representam computações e os arcos do grafo representam o fluxo do

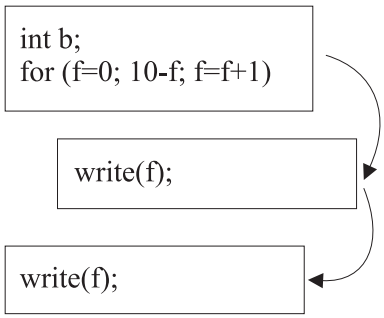


Figura 4.3: O programa da figura 4.2 representado como um GFC tal como descrito em Aho[2]

controle. Em outras palavras, os GFCs indicam qual passo de computação deve ser executado, possivelmente condicionado pelo resultado de uma expressão. Nesse caso o termo *controle* se refere ao processo de decidir qual a próxima instrução a ser executada em cada instante, considerando as condições atuais de memória do programa. Dessa forma, em um programa imperativo as alternativas de controle se resumem a:

- Se o programa foi iniciado neste instante, o próximo vértice a ser processado é aquele que representa a computação do passo inicial[1].
- Dado o vértice atualmente processado, se existe apenas um arco partindo deste vértice, o próximo passo de execução é justamente o vértice de destino desse arco.
- Se houver mais de um arco partindo do vértice atual, então necessariamente dados N arcos, $N-1$ arcos estarão associados a uma condição a ser satisfeita, e um arco não estará associado a nenhuma condição (chamado de passo incondicional). O próximo passo a ser processado é decidido da seguinte forma: percorra os arcos de forma que o passo incondicional seja o último. O primeiro arco que tiver sua condição satisfeita, fará com que o próximo passo seja o seu vértice de destino. No caso do arco incondicional, equivale a dizer que não é necessário satisfazer nenhuma condição para que o fluxo seja desviado para o destino deste.
- Se não houver nenhum vértice de destino, não existe próximo passo.

Tomando as regras acima como genéricas na interpretação de qualquer GFC, estas serão integradas a um interpretador mais completo, o qual será capaz de comparar dois ou mais grafos e produzir mensagens sobre discrepâncias entre eles. Na Figura 4.3 podemos ver um exemplo de GFC usando a estrutura proposta por Aho[2].

[1] O vértice que representa o passo inicial em um GFC pode ser explicitamente indicado, ou ainda pode ser deduzido como sendo o vértice que não é destino de nenhum arco do grafo

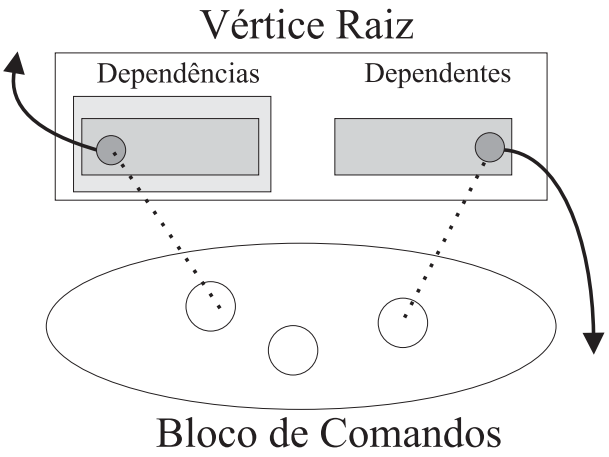


Figura 4.4: A representação de um vértice raiz

4.2.2.1 Peculiaridades do GFC

Embora tenha-se adotado uma representação coerente com aquela apresentada por Aho *et al*[2], existem algumas peculiaridades importantes de nossa representação de GFC, descritas a seguir:

- Um nodo para cada comando: na representação sugerida por Aho, um vértice pode conter um bloco com um mais comandos. Na representação adotada, cada vértice representa exatamente um comando.

  A justificativa é que dessa forma, torna-se mais fácil de projetar algoritmos que façam permutações e transformações em geral no programa representado.

- Expressões como árvores de derivação: em grafos de fluxo de controle tradicionais, uma expressão é um comando e portando seria representada apenas como um nodo (ou parte de) no grafo. Na nossa abordagem, uma expressão é representada como uma árvore de derivação, de forma que o nodo cabeça dessa árvore é incorporado no fluxo de controle tradicional.

  Nesta forma, várias transformações e equivalências algébricas, tais como comutatividade e associatividade, são aplicadas na ordem necessária do fluxo sobre a expressão.

- Os blocos de instruções têm várias informações importantes resumidas em um único vértice, a raiz do escopo, que é um vértice criado sem instrução, apenas para representar o escopo. Observe que o vértice raiz não contém em sua estrutura o subgrafo do escopo inteiro – ele apenas o representa. Com essa representação, o escopo pode ser tratado como se fosse um vértice apenas quando for de interesse, ou como parte do GFC normalmente. A Figura 4.4 ilustra um vértice raiz. Observe que na figura as arestas de dependência de dependentes podem ter uma relação que indica que

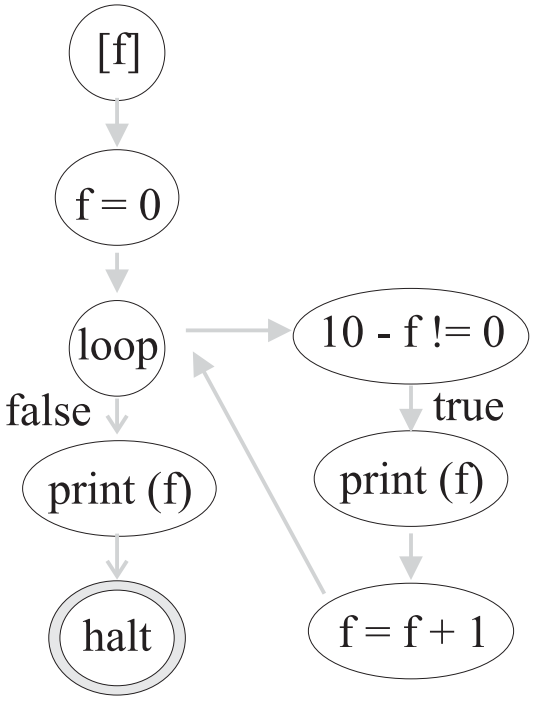


Figura 4.5: O programa da figura 4.2 representado como um GFC

a relação em questão é causada por um vértice específico do escopo enraizado por este vértice.

- O escopo do primeiro vértice é o programa inteiro, no qual pode haver instruções que possuem seus próprios escopos.
- Embora não seja uma peculiaridade que afete a estrutura do GFC em si, é interessante observar que cada vértice possui informações utilizadas pelo sistema para manter aspectos relacionados ao estado da avaliação do programa. Na seção 4.2.2.4 estas informações são meticulosamente apresentadas.

4.2.2.2 Grafo de Dependência de Dados (GDD)

Além dos aspectos exclusivos de um GFC, tal como detalhado na seção anterior, a representação por grafo representa também as dependências de dados e controle do grafo. Denomina-se Grafo de Dpendência de Dados (GDD) o grafo direcionado cujo conjunto de vértices V é o mesmo do GFC do programa e cujas arestas do conjunto E representam a existência de dependência entre um par de nodos, ou seja, a aresta $uv \in E$ se u depende de v. A figura 4.6 apresenta o GDD separado. O GDD não é construído durante a compilação como o GFC, mas no passo de normalização descrito no Capítulo 7.

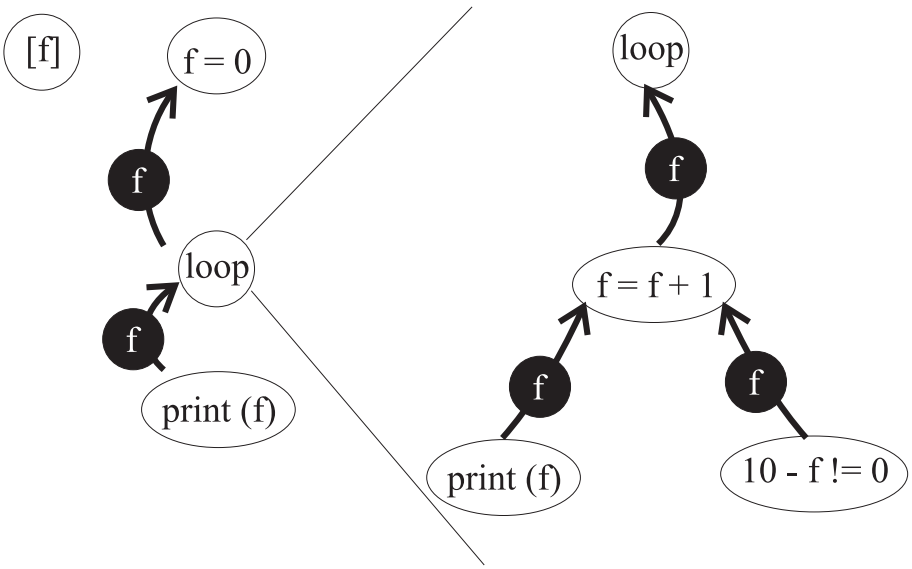


Figura 4.6: A figura acima ilustra o mesmo grafo da figura 4.5, mas apenas com as dependências de dados

4.2.2.3 Relação com o Grafo Original

Uma vez que os GFCs que representam os programas deverão ser transformados e distorcidos durante o processo de comparação e detecção de erros, é importante que ao final o sistema seja capaz de relacionar os resultados do diagnóstico ao programa original. Dessa forma, o processo mantém um par de grafos, aquele que representa a compilação original e o grafo que é usado no processamento. Para que a relação não seja perdida, cada vértice do grafo de trabalho possui uma aresta ligando-o ao grafo original.

4.2.2.4 Estrutura do Vértice

A partir do momento que conhecemos as várias estruturas que envolvem os vértices, vale a pena resumir como um vértice é representado, inclusive apresentando alguns dos atributos que não foram mencionados. A estrutura geral de um vértice usado nos grafos mencionados é apresentada na Figura 4.7. A descrição do vértice e suas diferentes partes é a seguinte:

- Sequência - arestas que indicam a sequência do grafo de fluxo de controle.
- Condição - é a condição para que essa aresta seja usada como caminho na execução de um programa.
- Dependências - arestas de dependências, classificadas em dois conjuntos, dependências de dados e de controle;
- Dependentes - arestas que relacionao os vértices que dependem deste;

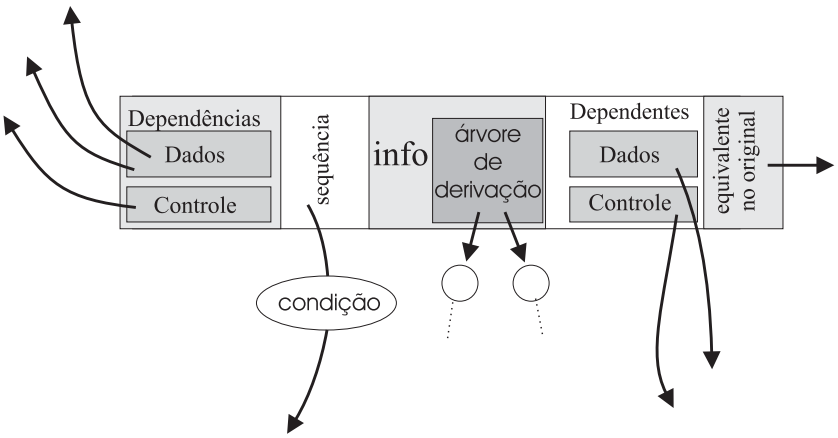


Figura 4.7: Estrutura Geral do Vértice

- Equivalente Original - arestas que indicam qual ou quais vértices no grafo original esse vértice representa. Quando há composição, por exemplo, é possível que um vértice no grafo de trabalho represente mais de um vértice no original.
- Info - consiste do seguinte conjunto de informações:
  - Tipo de Instrução - tipo de instrução que o vértice representa (ramificação, expressão, leitura, etc);
  - Instrução - é a instrução literal que o vértice representa;
  - Árvore de Derivação - se o vértice é uma expressão, a árvore de derivação da expressão é representada;
  - Resumo de escopo - informações necessárias para que, no caso deste vértice representar um bloco de comandos, seja possível mapear as dependências, sejam aquelas causadas por este nodo como aquelas que este sofre, dos vértices internos para os externos e vice-versa.
    - Pendências externas - se o vértice é raiz de uma escopo, as pendências externas representam todas as dependências que não foram resolvidas dentro do próprio escopo;
    - Definições internas - representam todas as definições realizadas no bloco e que podem causar dependência para os vértices posteriores.

4.2.3 Representação Intermediária

Alguns dos mecanismos desenvolvidos nesse trabalho estão limitados a processar apenas com textos, de modo que nem sempre o GFC das soluções de entrada são suficientes para dar prosseguimento ao diagnóstico. Dessa forma, é necessária a definição de uma

linguagem textual capaz de representar adequadamente o GFC, sem perda de semântica. Neste trabalho, é utilizada uma linguagem semelhante à linguagem "C" para representar o resultado do que pode ser chamado de enegenharia reversa do GFC para um programa fonte. A linguagem intermediária representa as seguintes primitivas:

- Condicionais: Todos os condicionais são representados através da primitiva *switch*, cuja forma é exatamente a seguinte:

```
switch(<condicao>) {
    case <resultado1>: {
        // bloco de comandos do resultado1
    }
    case <resultado2>: {
        // bloco de comandos do resultado2
    }
    ...
    default : {
        // se nenhum dos resultados for aceito
    }
}
```

- Iterações: As iterações podem ser representadas por meio de duas primitivas, *while* e *do...while*, a primeira utilizada para teste de condição de contorno no início da iteração e a outra para teste no final. A sintaxe é a seguinte:

```
while(condicao ) {
    // bloco de comandos da iteracao
}

// ou ainda

do {
    // bloco de comandos da iteracao
} while(condicao );
```

- Literal: Sempre que uma primitiva não estiver em alguma das anteriores, a representação intermediária da mesma é o texto literal da instrução, sem nenhuma alteração, salvo pelos caracteres
, `=`, `:` e `?`. Sempre que estes caracteres surgirem como parte do texto a ser gerado, estes serão precedidos pelo caracter
. É possível desabilitar esse processo mediante a uma especificação explícita de que não deve ocorrer.

```
[ f ];
f=0;
while( 10−f  != 0 ) {
      print ( f );
      f = f + 1;
}
print ( f );
```

Figura 4.8: GFC da Figura 4.5 na forma intermediária

Observe que existem poucas primitivas na linguagem intermediária, até porque esta nunca representará um programa que deverá ser executado. O aspecto importante a ser representado nessa linguagem são os subgrafos relacionados aos escopos, de modo que qualquer outro aspecto sintático é irrelevante para os mecanismos que dependerão dessa linguagem. Por fim, outra questão importante com relação a esta representação é que cada linha gerada no código intermediário é relacionada ao vértice original por meio de uma tabela de mapeamento, denominada *relacionamento vértice-linha*. Esta tabela voltará a ser discutida na Seção 7.6. Na Figura 4.8, podemos observar o GFC da Figura 4.5 na forma intermediária. Nesse caso, a representação é muito semelhante ao programa original, até porque ambos os programas herdam características sintáticas da linguagem "C".

CAPÍTULO 5

AUTORIA

A ferramenta de autoria deve ser utilizada para criar e estender o conjunto de problemas que pode ser proposto ao aprendiz. A ferramenta de autoria descrita é relativamente simples, mas cumpre o mínimo necessário para trabalhar como contrapartida do interpretador. A priori, para fins de simplificação, a ferramenta está restrita a trabalhar com programação em um subconjunto da linguagem "C", embora a ferramenta de interpretação seja independente de linguagem.

Um dos aspectos mais relevantes de dispor de uma ferramenta de autoria é a simplificação do processo de criação e extensão de cursos que serão "ministrados" pelo interpretador tutorial, consequentemente reduzindo custos e tempo, tornando todo o processo de desenvolvimento de um STI mais acessível.

A grosso modo, a ferramenta apresentada cobre a autoria de dois elementos, os problemas a serem propostos ao aluno e os padrões de erros. Nessa visão, o erro não está necessariamente vinculado a um problema proposto, mas veremos que criar uma associação entre o erro e o problema proposto pode enriquecer o valor cognitivo do feedback relacionado.

5.1 Primitivas na Autoria de Questões Propostas

Uma questão proposta nada mais é do que um problema e seu respectivo enunciado, o qual deverá ser apresentado pelo interpretador tutorial.

Quando se fala em problemas no domínio de programação, é natural considerar que para cada enunciado, existe uma infinidade de soluções corretas, ou seja, pode-se dizer que existe um espaço de soluções. Dificilmente seria possível representar todas as soluções possíveis e por conta da indecidibilidade do problema de programação, não existe mecanismo genérico capaz de comparar quaisquer soluções arbitrárias. Ainda assim, é importante que sejam representadas mais de uma alternativa, seguindo a premissa de que o aluno deve ter a liberdade para criar. Dessa forma, uma questão proposta é representada da seguinte forma:

- Enunciado - é um texto comum descrevendo o problema a ser solucionado pelo aprendiz;
- Domínio de Soluções de Referência - Como o próprio nome sugere, representa um conjunto de soluções aceitáveis para o problema proposto.

  - Uma solução de referência é um programa que corresponde a uma possível resposta para o problema.

Para que a questão de autoria seja factível e não exija demais do autor ao mesmo tempo sem sacrificar demais as alternativas do aluno, parte-se da premissa que o interpretador tutorial foi construído de modo a aceitar variações de implementações semelhantes. Dessa forma, um pequeno conjunto de soluções distintas e representativas é capaz de cobrir um espaço considerável, provavelmente cerceando a maioria das soluções providas por algum aprendiz. Essa questão voltará a ser abordada de forma mais objetiva no capítulo 7.

De posse da representação de uma questão, torna-se simples determinar as principais primitivas da ferramenta de autoria, tal como podemos observar a seguir:

- Inserção/Edição de problemas
  - Edição do enunciando - por meio de um editor de textos simples;
  - Inserção/edição de soluções de referência:
    * Edição do código fonte - por meio de um editor um pouco mais sofisticado;
    * Teste da Solução - a qualquer momento, o autor pode invocar o compilador para testar a sintaxe e até mesmo detectar erros de variáveis ociosas, uso de variável com valor indefinido, entre outros[1];
    * Remoção da Solução.
- Associar padrões de erro[2] ao problema - pode existir uma lista de padrões de erro específicos do problema proposto.

Além das primitivas de edição e autoria de fato, a ferramenta inclui primitivas que auxiliam o autor a testar o seu material. As primitivas são as seguintes:

- Verificação da sintaxe, a qual é realizada pelo compilador[3] que transforma os programas fontes em GFCs;
- Uso do módulo de análise de dependência de dados do interpretador para encontrar problemas com variáveis não referenciadas e potenciais iterações infinitas;

[1] No Capítulo 7 são apresentados os tipos de erro que podem ser detectados sem qualquer comparação
[2] Os padrões de erro serão vistos ainda nesse capítulo, na seção 5.2
[3] A ferramenta é descrita no próximo capítulo

5.2 LAPIDAE – Linguagem de Autoria de Padrões Interpretáveis Direcionados por Acerto ou Erro

Um padrão interpretável direcionado é definido nesse trabalho como sendo um padrão que representa um erro ou um acerto em um programa de entrada provido pelo aluno. Neste trabalho, um padrão é composto de três partes, as quais serão apresentadas com detalhes no decorrer dessa seção.

A linguagem em si, é parte da ferramenta de autoria, mas os mecanismos que a processam são utilizados por ambas as ferramentas, de autoria e de diagnóstico. Na ferramenta de autoria o mecanismo de correspodência é usado para que o autor possa testar o padrão e verificar se os casamentos estão de acordo com o esperado, enquanto na ferramenta de diagnóstico este mecanismo é usado efetivamente para encontrar um erro ou acerto[4].

Sendo parte da ferramenta de autoria, uma questão a ser considerada é o quanto ela facilita o trabalho de construção de material de curso. No contexto de autoria de cursos de programação, o público alvo são professores e especialistas da área de programação. Por conta desta afinidade, é factível esperar que os autores envolvidos sejam capazes de lidar uma linguagem de padrões, seja pela semelhança quanto pelo fato da linguagem ser menos complexa do que as linguagens com as quais o autor deve estar acostumado a lidar.

5.2.1 A linguagem - Definições

A linguagem criada para especificação de padrões é muito parecida com aquela utilizada pelo POP-11[8] com a mesma finalidade, a não ser pela forma de especificação das entradas, que ao invés de listas de strings utiliza uma string apenas, tanto para representar o padrão como para representar o texto a ser testado para casamento. Um padrão é uma sequência de meta-símbolos os quais devem ser utilizados para determinar que tipo de texto *satistaz* o padrão, ou ainda, as *restrições* que o padrão impõe para que algum texto de entrada seja aceito.

Os meta-símbolos que compõem a linguagem para especificação do padrão podem ser classificados de duas formas, baseado no tipo de casamento ou na capacidade de instanciar uma variável com o valor do casamento. Quanto ao tipo de casamento, o meta-símbolo pode especificar casamento exato, com uma string qualquer ou com uma sequência de zero a qualquer número de strings. Já do ponto de vista de instanciação, um meta-símbolo pode especificar se o segmento de texto casado deve ser atribuído a uma variável para uso futuro. Os meta-símbolos são descritos a seguir:

- Meta-símbolo literal ou constante: é uma string texto que deve casar com um valor específico no texto. Pode-se dizer que esses elementos é que delineiam a forma geral

[4]No Capítulo 7 esse procedimento será estudado com detalhes

de um padrão. A representação desse meta-símbolo no padrão é qualquer string que não comece coms os caracteres '?' e '='.

- meta-símbolo de casamento unitário: indica que uma string qualquer pode ser casada com este símbolo. Para especificar que o valor casado deve ser atribuído a alguma variável, representa-se como $?NomeVarialARecebeValor$. Caso contrário, utiliza-se o símbolo =.

- meta-símbolo de casamento variável: indica que qualquer sequência de strings, inclusive uma sequência vazia, pode ser casada com este símbolo. Para especificar que o valor casado deve ser atribuído a alguma variável, representa-se como $??NomeVarialARecebeValor$. Caso contrário, utiliza-se o símbolo ==.

- meta-símbolo de escape: o cactere "
  "deve preceder um caracter que seja parte de algum dos símbolos especiais para indicar que este deve ser tratado literalmente. Ou seja, os caracteres '?', '=' e '
  ' somente serão reconhecidos como parte da string de fato se forem especificados como '
  ?', '
  =' e '

  ', respectivamente.

Tendo em mãos a descrição dos meta-símbolos, um exemplo de padrão é o seguinte:

```
== O rato ?acao a roupa do = de ??localizacao.
```

O padrão acima determina que os textos aceitos (casados) podem ter qualquer quantidade de strings seguidas de "O rato", depois por qualquer string, e assim por diante. Alguns exemplos de entradas e se casam ou não com o padrão apresentado:

- O rato roeu a roupa do rei de Roma na Itália. *[aceita]*

- Dizem que O rato roeu a roupa do rei de Roma. *[aceita]*

- O rato mastigou avidamente a roupa do rei de Roma. *[recusa]*

Um meta-símbolo vinculado a uma variável pode ser repetido para representar padrões mais complexos. Essa repetição tem como objetivo forçar a correspondência por meio da presença de segmentos idênticos no texto de entrada.

Completando os aspectos incorporados no padrão, estão as funções restritivas, as quais podem ser aplicadas sobre qualquer meta-símbolo (exceto o literal), restringindo as palavras (ou sequência de) que podem ser casadas com o mesmo. Sintaticamente, uma restrição é adicionada da seguinte forma:

Padrão entrada	Padrão saída	Explicação
if (?? condicao : isExpression ()) ? corpoIf : isStatementBlock ()	**while** (?? condicao) ? corpoIf	*Ao invés da utilização de uma iteração, você utilizou um condicional.*

Figura 5.1: Exemplos de padrões de erro.

```
<meta-simbolo>:funcao(<parametros>)
```

Uma função restritiva pode ser qualquer uma, desde expressões regulares até teste sobre a conformidade do elemento com algum aspecto sintático da linguagem de programação envolvida, tal como expressões aritméticas ou até mesmo funções que verificam se a string está em conformidade com alguma gramática específica. Através de funções restritivas torna-se mais fácil controlar o processo de casamento de modo a evitar padrões demasiado abrangentes. Alguns exemplos de funções restritivas possíveis:

- `regex(<R>)` : o meta-símbolo deve casar com a expressão regular R;
- `isExpression()` : o meta-símbolo deve ser uma expressão bem-formada na linguagem "C".
-

Através dessa linguagem de padrão, um padrão de erro/acerto é definido por três partes (vide exemplo de padrão na Figura 5.1):

- Padrão de Entrada: esse padrão deverá ser casado no texto que representa um código fonte de um programa. As variáveis associadas a meta-símbolos do padrão de entrada podem ser usadas nos dois padrões descritos a seguir como forma de indicar que o texto casado deve ser substituídos no padrão em questão;
- Padrão de Saída: esse padrão representa um modelo de texto que deve ser usado para substituir o segmento de texto que for casado com o padrão de entrada. No caso do nosso sistema, uma substituição no segmento de texto pode representar uma substituição de um subgrafo do GFC que representa o programa;
- Padrão de Explicação: esse padrão representa um modelo de texto que será apresentado caso o sistema confirme a presença do padrão de entrada no programa fonte. Isso poderia indicar, por exemplo, que tipo de erro foi detectado.

5.2.2 Procedimentos e técnicas

O problema de casamento de padrão é abordado como um problema de satistação de restrição, onde o padrão como um todo é interpretado como um conjunto de restrições que devem ser satisfeitas para que o texto de entrada seja casado com o padrão. Sendo um problema de satisfação de restrições, antes de resolvê-lo, é necessário identificar os conceitos básicos e modelar o problema de casamento como tal. A modelagem recai em responder as seguintes perguntas:

1. Quais são as variáveis do problema? Qual a natureza do valor dessas variáveis?

   Dado um meta-símbolo, este pode casar com um ou mais segmentos de texto distintos. A questão é com qual segmento de texto o meta-símbolo vai casar. Essa questão se traduz nas variáveis do nosso problema, e o objetivo final é justamente determinar o valor dessas variáveis tal que as restrições sejam satisfeitas.

2. Qual o domínio das variáveis?

   O domínio de uma variável que representa o casamento de um meta-símbolo é o conjunto de possíveis casamentos que esse meta-símbolo é capaz de cobrir. Um casamento é representado pelo segmento com o qual casa através de dois números, o número da string na qual o segmento começa e o número daquela na qual o mesmo termina.

3. Quais são as restrições do problema?

   Cada meta-símbolo impõe determinadas restrições sobre o texto de entrada. Por exemplo, um meta-símbolo de casamento unitário impõe o seguinte: na posição em que ele se encontra dentro do padrão, deve ser possível casar o meta-símbolo com exatamente um ítem léxico.

A abordagem algorítmica para o problema inicia com a determinação do domínio das variáveis, ou seja, para cada meta-símbolo, quais são os segmentos que podem ser casados. O importante nesse passo é determinar o menor subconjunto possível de valores que podem ser atribuídos dentro de um custo computacional pouco elevado. No presente trabalho, o padrão como um todo é responsável por manter o conjunto de meta-símbolos e por mandar mensagens para que os meta-símbolos se inicializem dados alguns parâmetros básicos, tal como o texto de entrada e o contexto no qual o meta-símbolo se encontra dentro do padrão.

O passo seguinte consiste em determinar quais valores cada variável deve receber tal que todas as restrições sejam satisfeitas. A principal heurística é instanciar primeiro aquelas variáveis cujo domínio é menor, ou seja, as variáveis mais restritas. Cada vez que uma variável é instanciada, todas as variáveis não instanciadas têm seu domínio reduzido,

eliminando valores que a variável não pode assumir em função da instanciação atual. Os próprios meta-símbolos são responsáveis, como objetos, por reduzirem seu domínio de acordo com o novo valor. Se uma instanciação fizer com que alguma variável tenha domínio vazio, isso indica que a mesma viola alguma restrição e, consequentemente, a instanciação é desfeita e a próxima é considerada.

O procedimento termina ou quando todas as restrições forem satisfeitas (houve casamento) ou quando o algoritmo determinar que não existe um conjunto de valores que satisfaz todas as restrições ao mesmo tempo (o texto de entrada não casa com o padrão).

É interessante observar como o processo acima é genérico: sempre que for necessário adicionar uma nova restrição para um meta-símbolo, basta que os objetos que representam os meta-símbolos sejam capazes de considerar uma nova restrição, uma vez que são os componentes encarregados tanto de inicializar o domínio como de restringí-lo em função de novas instanciações.

CAPÍTULO 6

COMPILAÇÃO DE PROGRAMAS E PADRÕES

Para que o interpretador tutorial seja capaz de analisar o programa provido pelo aluno, tanto as soluções de referência como a solução apresentada pelo aluno, são ambas representadas através de um Grafo de Fluxo de Controle (GFC), semelhante à representação descrita por por Aho *et al*[2], a não ser por algumas peculiaridades, tais como descritas no Capítulo 4.

O uso de uma ferramenta intermediária entre o programa fonte e a representação interna permite que o interpretador seja capaz de trabalhar independente da linguagem de entrada desde que seja possível realizar a tradução. Observe que esta característica permite que o processo de diagnóstico seja independente da linguagem. Essa independência é uma evidência clara de que a ferramenta não é voltada para ensinar sintaxe ou semântica de instruções.

O processo de compilação para GFCs é usado de duas formas nesse trabalho. A primeira serve para transformar os programas de entrada, sejam eles do autor ou do aluno. A segunda serve para gerar a representação interna dos padrões da LAPIDAE, de modo que também estes não estejam amarrados a qualquer linguagem específica de programação.

O objetivo deste capítulo é descrever os elementos sintáticos reconhecidos pelos compiladores, delineando o subconjunto de "C"que compõe a linguagem utilizada. Ainda, também apresentamos o mecanismo que converte os GFCs para a linguagem intermediária textual, uma vez que também envolve procedimentos de compilação de GFC para tal linguagem e da mesma para GFC novamente.

6.1 Mapeador de Código Fonte para GFC

A ferramenta para tradução de um programa em seu respectivo GFC foi consolidada e presentemente está funcional. Essa ferramenta recebe como entrada um programa escrito em um dialeto da linguagem "C"e produz como resultado um GFC (no qual o grafo é descrito em *dot plain*[1]). Um exemplo de compilação já foi visto no Capítulo 4, apresentando o programa e o GFC gerado pelo programa. A gramática aceita pelo compilador implementado é exatamente a seguinte:

[1] O *dotty*[16] é um software escrito para diversas plataformas usado para representar e renderizar grafos.

programa	$\longrightarrow$	main ( ) bloco_de_comandos
bloco_de_comandos	$\longrightarrow$	{ decl_opcional seq_comandos_opcional }
decl_opcional	$\longrightarrow$	lista_decl
	|	ε
lista_decl	$\longrightarrow$	lista_decl decl
	|	decl
decl	$\longrightarrow$	INT ID ;
	|	CHAR ID ;
seq_comandos_opc	$\longrightarrow$	seq_comandos
	|	ε
seq_comandos	$\longrightarrow$	seq_comandos comando
	|	comando
comando	$\longrightarrow$	expr ;
	|	IF (expr) comando ;
	|	IF (expr) comando ELSE comando ;
	|	WHILE (expr) comando ;
	|	FOR (lista_de_expr ; lista_de_expr ; lista_de_expr) comando ;
	|	imprime ;
lista_de_expr	$\longrightarrow$	lista_de_expr , expr
	|	expr
imprime	$\longrightarrow$	WRITE (lista_impr)
	|	WRITELN (lista_impr)
lista_impr	$\longrightarrow$	lista_impr , item_de_impressao
	|	item_de_impressao
item_de_impressao	$\longrightarrow$	expr
	|	CHARCONST
	|	CHARSTRING
expr	$\longrightarrow$	ID = expr
	|	expr ”==”expr
	|	expr ”!=”expr
	|	expr ’<’ expr
	|	expr ’>’ expr
	|	expr ”||”expr
	|	expr ”&&”expr

Observe que o compilador não aceita a declaração de vetores. De fato, essa limitação é apenas do compilador, uma vez que os procedimentos de comparação de padrões não estão limitados neste aspecto. A introdução de vetores é essencial para aumentar o poder

```
do {
     ??corpo: isStatementList ();
} while(??cond: isExpression ())
```

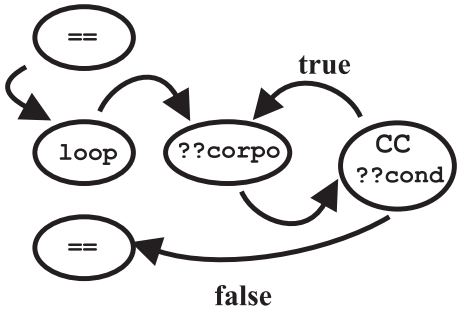


Figura 6.1: Padrão e Grafo Correspondente

expressivo da linguagem e certamente é um trabalho em vista no horizonte, mas esse tratamento não é essencial para discutir e trabalhar os conceitos mais importantes deste trabalho.

6.2 Mapeador de Padrão de Texto para Padrão de Grafo

Como foi observado no Capítulo 5, um dos elementos singulares desse trabalho é o uso de padrões de erros e acertos, o que permite a detecção de erros complexos e é o elemento chave no processo de enriquecimento de feedback. Ainda, foi observada a importância de manter o interpretador independente da linguagem utilizada. Da forma como foi descrito, um padrão é representado, na maior parte, através da própria sintaxe da linguagem objeto e portanto não é independente da mesma. Dessa forma, é necessário também que exista um compilador que transforme o padrão de entrada em uma representação independente.

Nesse trabalho, uma vez que a representação interna é baseada em GFCs, o interpretador deverá ser capaz de utilizar o padrão para detectar segmentos no grafo que correspondam ao erro. Uma vez que a linguagem objeto do padrão, a não ser pelos meta-símbolos, é a mesma utilizada pelos programas, do ponto de vista de parsing o compilador é quase o mesmo descrito na seção anterior. O único aspecto que distingue os compiladores é que os programas na linguagem de padrões possuem os meta-símbolos como elemento sintático extra. Para cada meta-símbolo é criado um vértice cujo tipo é marcado para identificar essa condição. Na Figura 6.1 é apresentado um exemplo de padrão e o grafo gerado. Observe os vértices marcados como `==`. Estes vértices são sempre incluídos para indicar que o padrão de entrada na verdade representa um trecho do programa apenas, portanto desobrigando o programa de ter a morfologia inteira casada segundo o padrão.

6.3 Convertendo Grafos para a Representação Intermediária

Como foi observado no Capítulo 4, os GFCs, sejam representando programas ou padrões, não podem ser processados diretamente por alguns dos mecanismos internos. Para tanto, é necessário que os GFCs sejam convertidos em texto, representados pela linguagem descrita

na Seção 4.2.3. Esta seção tem como objetivo descrever o mecanismo que realiza essa conversão.

O procedimento de conversão do GFC para a representação intermediária é bastante simples e pode ser realizado pelo seguinte algoritmo na Figura 6.2. É importante ter em mente que o código representado deve ser reversível para grafo e portanto deve haver um compilador capaz de fazer isso. Observe que o compilador é ainda mais simples do que aquele apresentado na primeira seção deste capítulo, uma vez que existem menos elementos sintáticos essenciais.

```
converteGFCintermediaria(GFC G, raiz r)
iterador ← r;
enquanto(nao(ultimo_vertice(iterador)))
    se (iteracao(iterador))
        se (condicao_inicio(iterador))
            emite("while(".  iterador->condicao") {");
            converteGFCintermediaria(G, iterador->raizescopo);
            emite("}");

        senao
            emite("do {");
            converteGFCintermediaria(G, iterador->raizescopo);
            emite("} while(".  iterador->condicao .  ");");


    senao
    se (condicional(iterador))
        emite("switch(".  iterador->condicao .  "){");
        Para cada resultado cd previsto no condicional de iterador:
            emite("case ".  cd .  "{");
            converteGFCintermediaria(G, cd->raizescopo);
            se (NOT(EhDoTipoCASE(cd))) então
                emite("break;");

            emite("}");

        emite("}");

    senao
        write(iterador->instrucao);

    iterador ← iterador->proximo;
```

Figura 6.2: Algoritmo para conversão do GFC para representação intermediária

CAPÍTULO 7

INTERPRETADOR TUTORIAL

Um interpretador tutorial é um sistema capaz de interpretar o material produzido pela autoria e conduzir o aluno através do processo de ensino. O que diferencia um interpretador de um STI efetivamente é o fato de que o intepretador não necessariamente abrange todos os aspectos de ensino de um STI. No caso desse trabalho, o interpretador é responsável por apresentar os problemas de programação construídos pelo autor de curso, receber a solução do aprendiz e realizar um diagnóstico da mesma, tendo como resposta ao aluno uma detalhada avaliação dos problemas encontrados na solução do aluno, no caso de existir algum. A ordem de apresentação dos problemas é arbitrária e não há nenhum mecanismo ou critério apresentado para lidar com essa questão.

Esse capítulo enfoca a descrição dos procedimentos e elementos envolvidos no processo de diagnóstico, inclusive debatendo as hipóteses simplificadoras, as heurísticas e algumas das limitações inerentes à natureza do problema.

7.1 Processamento Geral

Uma vez que o objetivo do processo de diagnóstico consiste em explicar ao aluno quais erros foram encontrados e porque estes, de fato, constituem um erro, duas questões foram levadas em consideração para maximizar o valor cognitivo do interpretador:

- O sistema deve modificar o mínimo possível a solução do aluno;
- É importante que as explicações estejam contextualizadas pela solução original do aluno.

Dessa forma, em particular, para satisfazer a segunda condição, o sistema mantém a versão original do aluno e realiza uma cópia, sobre a qual deverá efetuar uma série de transformações durante o processo de diagnóstico. Para que a relação entre os vértices da representação transformada e os da original não seja perdida, cada vértice da cópia possui um atributo que informa a qual vértice este equivale na original, tal como descrito no Capítulo 4. A partir do momento que a cópia é construída, o sistema desenvolve a comparação através das seguintes etapas:

- Normalização das Representações;
- Comparação

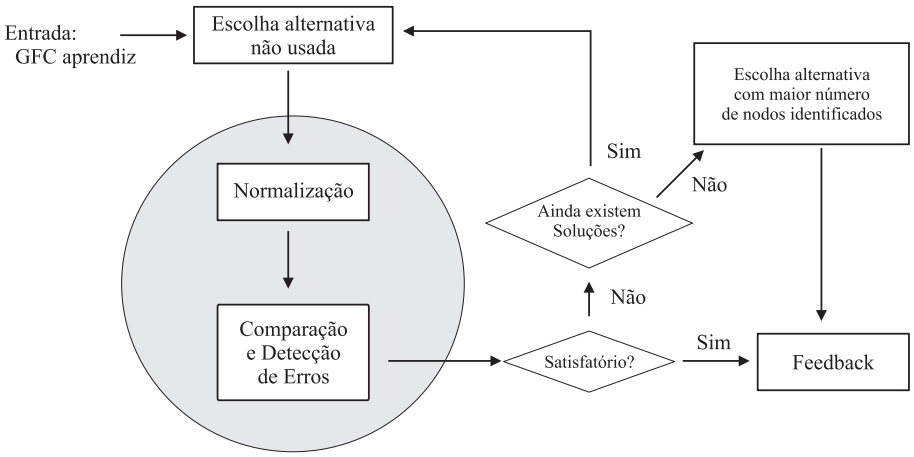


Figura 7.1: Fluxo de execução do Interpretador

- Identificação de vértices através de manipulação heurística;
- Identificação de vértices através de casamento de padrões de equivalência;
- Busca de erros através de manipulação heurística;
- Busca de erros através de casamento de padrões de erro.

A Figura 7.1 apresenta um esquema com o fluxo de execução do interpretador. O sistema de diagnóstico recebe duas entradas, um conjunto de soluções de referências supostamente corretas e a solução do aprendiz a ser avaliada. Em cada iteração uma solução correta é selecionada para ser a referência da vez. As soluções são comparadas e os vértices do GFC do aluno são anotados. A quantidade de vértices identificados no programa do aluno é usada como métrica básica para avaliar o desempenho do sistema no momento em que a análise é completada. Se o resultado for considerado satisfatório, ou seja, se um percentual mínimo de vértices for identificado, o sistema termina e apresenta os resultados da análise. Em caso de desempenho pobre, o sistema escolhe uma das soluções corretas que não foi utilizada e recomeça o ciclo. A partir do momento que forem esgotadas as soluções alternativas, aquela que obteve melhor desempenho será usada para feedback. Os procedimentos ilustrados na figura serão descritos em detalhes nas próximas seções deste capítulo. A proporção de vértices identificados não foi definida neste trabalho, mas provavelmente algo em torno de 70% (ou superior a isso) de identificação poderia ser considerado razoável.

Um conceito bastante usado nessa seção e, portanto, que vale a pena conhecer a priori é o de caminhar de um nodo ao próximo. Durante a discussão sobre representação, ficou claro que um vértice pode representar um sub-grafo inteiro correspondente a um bloco de comandos na fonte original. Dessa forma, o conceito de caminhar de um vértice ao próximo usado nesse capítulo significa caminhar através desses vértices representativos, de um escopo ao outro. Adentrar um escopo é o termo usado para descrever o processo

de visitar os vértices dentro de um determinado bloco e a entrada sempre inicia no vértice raiz deste escopo.

7.2 Exemplos e suas Notações

Nesta seção definimos um exemplo que será utilizado no decorrer do capítulo, integralmente ou parcialmente, para ilustrar os processos de transformação e comparação. O problema exemplo é desenvolver uma solução para calcular *Fibonacci*. *Fibonacci* é uma função recorrente tal que:

- Fibonacci(0) = 1
- Fibonacci(1) = 1
- Fibonacci(X) = Fibonacci(X-1) + Fibonacci(X-2)

O programa da Figura 7.4 é a solução de referência provida pelo professor. Na Figura 7.5 podemos observar o GFC desse programa. Observe que existem muitos outros detalhes envolvidos em cada vértice, mas que são omitidos para fins de melhor visualização. Os vértices do GFC usam a seguinte notação:

- *init* indica o vértice inicial e a raiz do programa;
- *halt* indica o vértice no qual a computação pára;
- *loop* é um vértice raiz de uma iteração;
- $[x]$ indica a declaração da variável x;
- `read`(X) representa a leitura de um valor para a variável X;
- *expr* indica que o vértice representa uma expressão. Esse tipo de vértice é acompanhado da árvore de derivação da expressão. No restante do capítulo, essa notação pode ser simplificada e ao invés da árvore, poderá ser apresentado somente o texto da expressão.
- *CC* é um vértice que representa a condição de contorno da iteração que o contêm.

Ainda, na Figura 7.5, as arestas em preto indicam a seqüência do GFC realmente, enquanto a aresta pontilhada indica a seqüência quando considerado que o vértice de iteração é o representante na seqüência do programa.

```
int X_2, X_1, X;
int i, t;

read(X);

X_2 = 0;
X_1 = 1;

for (i=1; i < X; i++) {
    t = X_2 + X_1;
    X_2 =   X_1;
    X_1 = t;
}

write(X_1 + X_2);
```

Figura 7.2: Solução de referência

```
int Mb, Ma, X;
int F_X, k, tmp;

read(X);

Mb = 0;
Ma = 1;

k=1;

do
{
    if (k <= X) {
        Mb = Ma;
        tmp = Mb + Ma;
        k = k+1;
    }
}while(k <= X);
Ma = tmp;
F_X = Ma + Mb;

write(F_X);
```

Figura 7.3: Solução do aluno

Figura 7.4: Soluções de Exemplo, calculando Fibonacci de X

7.3 Normalização de Representações

O fato de que não existe uma forma canônica para representar um programa é bastante conhecido, mas ainda assim é possível realizar transformações no programa que reduzem as diferenças sintáticas, tornando o processo de comparação mais efetivo.

O processo de normalização tem como objetivo aplicar uma série de transformações em ambos os grafos de forma a reduzir a distância sintática entre eles, principalmente aquela advinda da diferença de granulação dos programas que estes grafos representam. Ou seja, ainda que dois programas A e B façam a mesma coisa e com base no mesmo algoritmo, pode haver uma diferença no número de instruções empregadas e a normalização busca minimizar essa diferença. O processo apresentado é capaz de lidar em maior ou menor escala com as seguintes diferenças relacionadas à granulação:

- diferença na composição de expressões, ou seja, uma expressão pode ser decomposta em várias expressões menores;
- reutilização de variáveis, ou seja, em um programa uma variável pode ser reutilizada, enquanto em outro pode ser declarada uma variável distinta que seja usada para atingir o mesmo propósito;

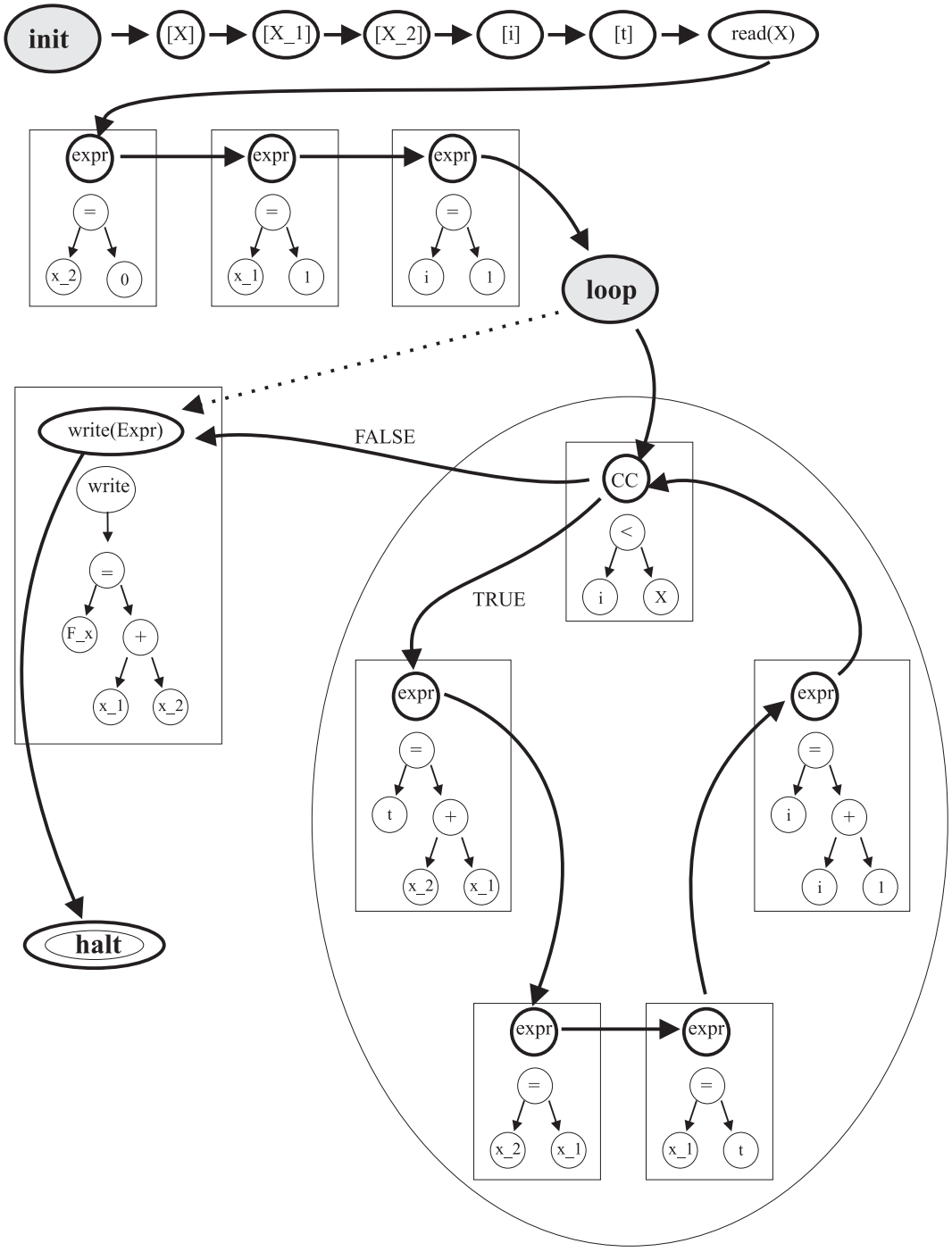


Figura 7.5: GFC da Solução de Fibonacci

- iterações compostas, ou seja, em um programa uma iteração pode executar a tarefa equivalente a mais de uma iteração em um outro, sem perda de semântica.

A questão de composição/decomposição do escopo de comandos das iterações não é detalhada nesse trabalho, mas vale ser mencionada como exemplo claro do que é possível em um processo de normalização. Além das diferenças por conta da granulação, ainda são consideradas as seguintes:

- instruções independentes do corpo de uma iteração, podem estar tanto dentro da mesma como fora;
- questão de ordem de instruções ou de termos comutáveis em expressões.

Uma vez que muitas das transformações envolvidas no processo de normalização descrito neste trabalho utilizam intensamente as relações de dependência, tanto de dados como de controle, um dos passos primordiais é a análise dessas dependências. Essa análise é um subproduto de grande importância e, a partir da mesma, são derivados outros resultados, os quais serão discutidos na Seção 7.3.1.

O processo de normalização pode ser descrito através dos seguintes procedimentos:

- Análise das dependências de dados;
- Análise das dependências de controle;
- Separação de variáveis em definições independentes;
- Otimização de iterações;
- Composição de expressões;
- Reordenação e paralelização de instruções.

7.3.1 Análise de dependências de dados e controle

A análise de dependência é o processo pelo qual determina-se, para cada vértice do grafo ou cada instrução do programa, quais vértices influenciam o resultado da instrução em questão. A importância desse procedimento é o relacionamento semântico, e não sequencial, entre as instruções do programa e essa informação é extremamente valiosa tanto para os outros procedimentos da etapa de normalização, como para o próprio mecanismo de comparação concebido.

Resumindo a funcinalidade do módulo de análise de dependências, o procedimento recebe como entrada um GFC e o resultado é um outro grafo direcionado, no qual cada vértice possui arestas que apontam para os vértices dos quais depende. O restante dessa seção descreve detalhadamente como o processo é realizado.

O algoritmo geral percorre cada vértice, para o qual as tarefas principais consistem em gerar uma indicação que resuma quais variáveis ele define no presente momento. Além disso, o algoritmo determina de que definições e controles o referido vértice depende, sendo estes representados pelos respectivos vértices que os contêm.

Antes de entrar nos detalhes do algoritmo, é importante estabelecer algumas definições e premissas:

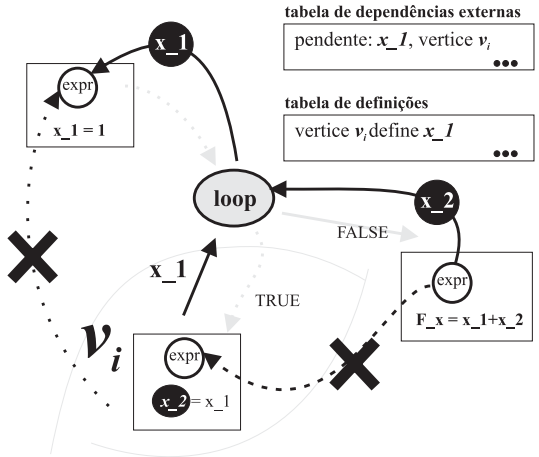


Figura 7.6: A Raiz e suas tabelas

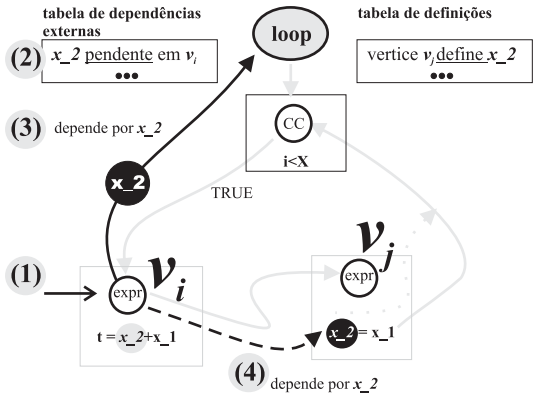


Figura 7.7: Efeitos da Análise de Dependência denttro de uma iteração

- A entrada é precisamente um GFC e um vértice, denominado *raiz*, no qual deve iniciar a análise;
- A raiz possui uma tabela, denominada *tabela de definições*, na qual são armazenadas informações que relacionam em qual vértice determinada variável foi definida mais recentemente. Esta tabela é preenchida durante o processamento e, ao final da análise, representará as últimas definições efetuadas dentro do escopo dessa raiz;
- Uma raiz r possui ainda uma outra tabela, denominada tabela de dependências externas. Essa tabela deverá possuir cada referência cuja definição correspondente não esteja presente no escopo de r e dizemos que essa referência está pendente; Na Figura 7.6 estão ilustradas as tabelas de definição e de dependências externas. Nessa figura, as arestas cinza claro representam a seqüência do programa, enquanto as pretas representam uma relação de dependência com variável responsável pela mesma sobre a aresta. Observe na figura que ao invés de haver uma aresta de v_i para a a definição externa, é criada uma pendência e v_i é dependente do vértice raiz. Por sua vez, o vértice raiz é que depende da definição. Da mesma forma, as definições realizadas dentro do escopo da iteração causam dependência para com a raiz e não os vértices realmente responsáveis. Dessa forma, a raiz representa todas as relações que envolvem dependência e esta pode ser manipulada levando em conta corretamente todos os vértices em seu escopo.
- É possível que uma variável seja definida por mais de um vértice.

Para proporcionar uma apresentação mais clara, as ações tomadas para criação de dependência de dados e de controle serão separadas, embora do ponto de vista algorítmico não exista problema de ambos ocorrem ao mesmo tempo. A Figura 7.7 apresenta o subgrafo correspondente a uma iteração do programa da Figura 7.4, sobre a qual algumas

das operações a seguir estão ilustradas. Nesta figura, as arestas pretas novamente representam dependência de dados. As ações executadas para determinação de dependências de dados são as seguintes:

- Se o vértice v_i realiza uma atribuição ou leitura a uma variável u, é criada uma entrada na tabela indicando que u é definida em v_i.
- Se v_i é uma operação de leitura, v_i recebe uma dependência para si mesmo;
- Se v_i é uma escrita (ou impressão), v_i recebe uma dependência para si mesmo;
- Se uma determinada variável u é utilizada na instrução do vértice atual, então verifique se u foi ou não definida por algum vértice no escopo atual:
  - Se u já foi definida, então indique que este vértice possui uma dependência de dados para com todos os vértices que definem essa variável;
  - Se u não foi definida, indique que o vértice tem uma dependência de dados com a raiz, na qual deverá estar indicado que a definição dessa variável está pendente. Observe na Figura 7.7, que no passo (1) a variável x_1 não tinha sido definida, portanto em (2) a tabela de dependências é atualizada com essa pendência e a dependência (3) é criada do vértice para a raiz.
- Se o vértice v_i é raiz de um ou mais escopos, deve-se realizar uma análise de dependência recursivamente nos escopos deste vértice. É possível que o vértice seja raiz de mais de um escopo, por exemplo no caso de um condicional.
- Se o vértice termina o presente escopo S, é realizado o tratamento do escopo no qual o vértice se encontra. Basicamente, o sistema trata duas questões importantes, o condicional e a iteração. Se for um condicional, as definições de uma mesma variável em diferentes escopos são representadas para representar a possibilidade das próximas referências dependerem de mais de uma definição ao mesmo tempo. Se for uma iteração, as expressões podem criar dependências por meio do ciclo. Os procedimentos que realizam estes tratamentos serão detalhados na Seção 7.3.1.1.

As dependências de controle indicam se a instrução do vértice depende do ponto de controle do escopo, representado pela raiz do mesmo. Dessa forma, toda dependência de controle é de um vértice para com a raiz do escopo vigente, a não ser que especificado o contrário. A Figura 7.8 apresenta parte do programa da Figura 7.4, mais especificamente a iteração e algumas das dependências de controle determinadas. O vértice v_6 não faz parte do programa original e foi introduzido para ilustrar a análise de dependência de controle. Nesta figura surge uma aresta nova, a aresta de dependência de controle, representada por uma linha tracejada e com o rótulo *CTRL* sobre a mesma. As outras arestas mantém

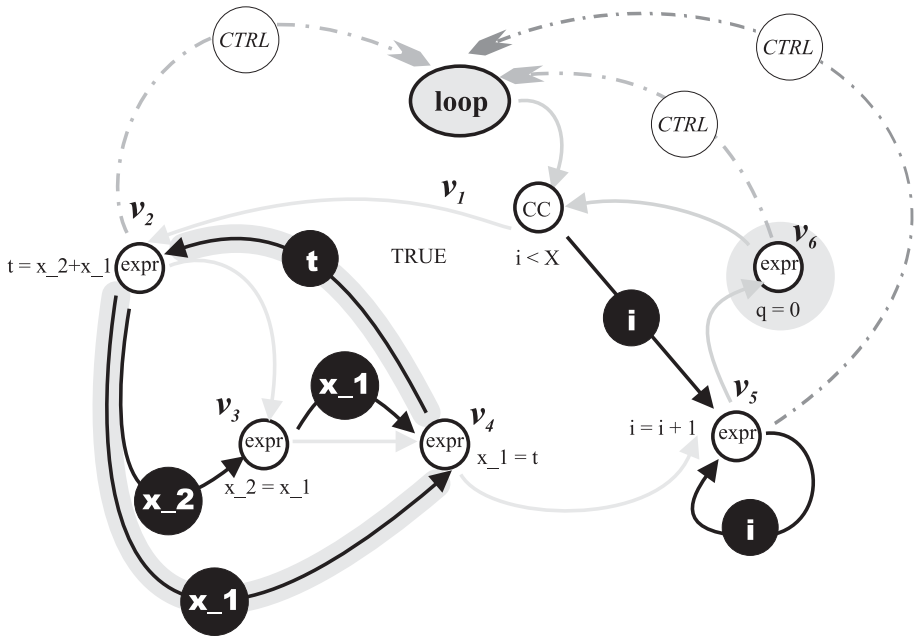


Figura 7.8: Adição de arestas de controle. Nem todos os vértice que devem ter arestas estão ilustrados, tal como v_3 e v_4.

o significado usual, para dados e sequência. Uma dependência de controle é estabelecida se alguma das seguintes situações for detectada:

- Se o vértice em questão representa a condição de controle do escopo, uma vez que o controle depende da condição.

- Se existe fechamento transitivo entre este vértice e o vértice da condição ou vice-versa. Se a condição influencia o controle, todos os vértices da qual ela depende também a influenciam. Na Figura 7.8, o vértice v_5 causa uma dependência na condição ao definir a variável i.

- Se existe um fechamento transitivo que relacione este vértice consigo mesmo. Essa condição se justifica porque se um vértice depende de si mesmo, então a cada vez que o ciclo da iteração visitá-lo estará usando o seu valor do ciclo anterior, de modo que o valor da variável ao final depende de quantas vezes o controle determinou que o ciclo deveria ser executado.

  Na Figura 7.8, os vértices v_2, v_3 e v_4 estão todos nessa situação. Observe que v_2 depende de v_4 pela variável x_2 e que v_4 depende de v_2 ela variável t, portanto criando um fecho transitivo. As arestas de dados envolvidas no fecho estão marcadas em cinza ao seu redor.

- Se existe fechamento transitivo que relacione o vértice com qualquer vértice que possua dependência de controle. Se um vértice v depende de um vértice u, tal que u

depende do controle, então v é influenciado pela definição de u a cada ciclo, portanto também dependendo do controle.

- A condição de controle está antes desse vértice, ou seja, a instrução no vértice pode ser executada zero vezes. Se esta for a única razão para existência da dependência de controle, o vértice recebe uma indicação dessa situação porque essa situação pode ser interessante para o passo de otimização. O vértice v_6 depende do controle somente porque a condição pode evitar que a instrução seja executada.

7.3.1.1 Tratamento de Escopos na Análise de Dependência

Quando um escopo é analisado, alguns procedimentos mais elaborados são necessários para garantir a coerência das dependências de dados. Basicamente, existem escopos para dois tipos de instruções: condicionais e iterações, cada qual com suas peculiaridades.

Os escopos de condicionais podem ser constituídos de mais de um bloco que pode ser executado em função do resultado da condição avaliada. Uma vez que não é possível saber a priori qual bloco será escolhido, todas as definições efetuadas em todos os blocos são válidas ao mesmo tempo. Ou seja, se uma variável v foi definida tanto no bloco para condição verdadeira como no bloco da falsa, qualquer instrução a seguir depende de ambas as definições ao mesmo tempo. Desse modo, é necessário que a tabela de definições do condicional seja atualizada para que cada variável contemple as definições de todas as alternativas do condicional.

O escopo de uma iteração provê a possibilidade de dependências não resolvidas serem relacionadas a definições no condicional que estão posicionadas após a referência na sequência do programa. O procedimento para tanto consiste em verificar se as variáveis anotadas como pendentes na raiz do condicional podem ser resolvidas usando a própria tabela de definições dessa raiz. Possivelmente, o aspecto mais traiçoeiro é que dentro do escopo podem haver outros escopos aninhados. A preocupação é com os escopos referentes a condicionais, pois é possível que exista uma referência pendente que é resolvida fora do escopo, através da iteração que envolve o condicional. Este problema é tratado logo após o escopo do condicional ter sido analisado por uma chamada recursiva. É realizada uma análise das tabelas de dependências externas e definições. Se houver uma definição que resolve alguma das pendências, é criada uma dependência do vértice para si mesmo e os vértices internos são resolvidos ou criando a dependência com o vértice assinalado na definição ou reaplicando recursivamente esse procedimento se algum destes vértices pendentes for a raiz de um escopo aninhado no condicional.

Na Figura 7.7, observamos pendência (2) e no vértice v_j há uma definição da variável pendente. Portanto, é criada a dependência (4), representada pela aresta pontilhada, do vértice que estava pendente para o vértice que define x_2, refletindo o fato das variáveis estarem em uma iteração.

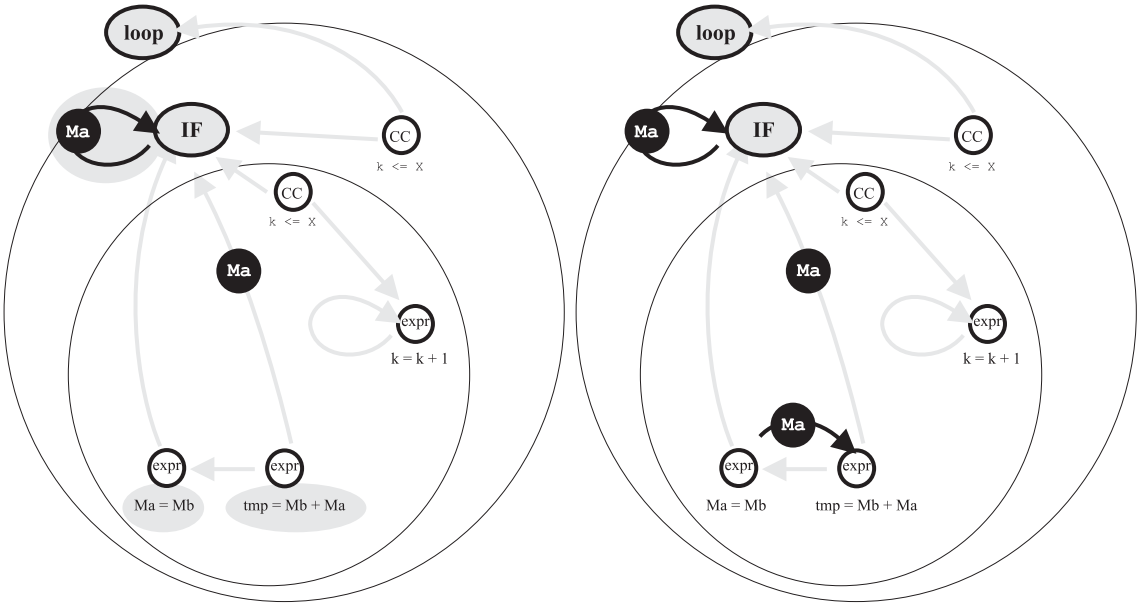


Figura 7.9:

Para exemplificar o processo de resolver dependências cíclias em escopos aninhados, observe a Figura 7.9. Observe, à direita, que no momento que a iteração for processad, será verificado que o vértice do condiciona tem uma dependência externa de Ma, ao mesmo tempo que define Ma. Essa situação indica que o condicional depende de si mesmo, e portanto é necessário verificar exatamente qual o elemento interno está pendente (o vértice que define *tmp*) e qual realiza a definição apresentada na tabela do escopo do IF. Na realidade, o processo de resolução das arestas dependentes dentro do condiocional ocorre exatamente como se o mesmo fosse uma iteração, e portanto qualquer elemento aninhado no escopo do condicional também será corretamente processado.

Uma questão importante a ser considerada quando se trata das arestas criadas durante a análise do escopo da iteração é que todas as arestas de dependência criadas nesse momento devem criar uma aresta que indica a ordem real de elementos dependentes. Muitas vezes, se já houver uma aresta de dependência que fora criada antes da análise do escopo, basta que esta seja marcada como a determinante. Somente quando não existe tal aresta é que se torna necessário criar uma nova.

7.3.2 Separação de Variáveis em Definições Independentes

Dados dois programas, uma característica comum que pode diferenciá-los é a reutilização de variáveis em definições distintas e independentes, enquanto outro programa opta por usar variáveis distintas, obtendo exatamente o mesmo resultado.

Dessa forma, vale a pena assumir como padrão que duas definições independentes que reutilizem uma váriavel devem usar variáveis distintas também. O procedimento consiste simplesmente em detectar essa situação e renomear as variáveis, bem como recondicionar

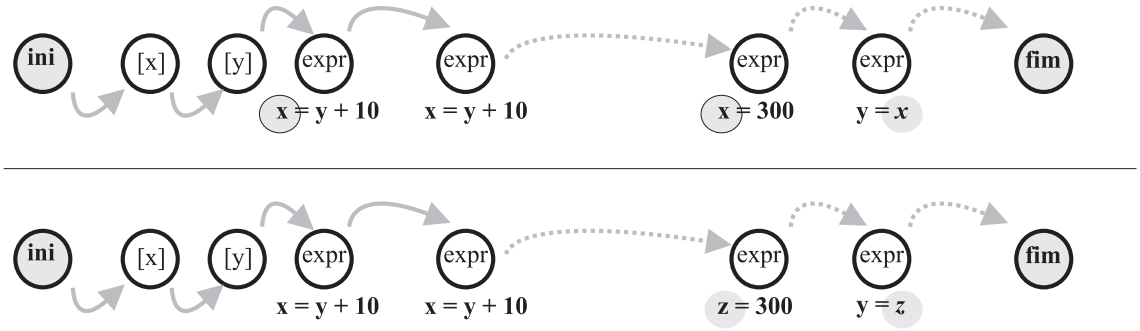


Figura 7.10: Exemplo de Separação de Variáveis

as instruções que dependam dessa definição. A Figura 7.10 demonstra o processo de separação de variáveis. Observe na parte superior que a variável x é definida duas vezes e pode ser separada. Uma variável denominada z é criada e a segunda definição, bem como seus dependentes, tem o nome da variável modificada atualizado para manter a coerência. Apesar da variável ter seu nome modificado, deve ser mantida uma referência interna indicando qual a identidade original da mesma. Este procedimento é necessário porque em alguns casos uma variável pode ser separada por erro no programa e deve ser possível reunificá-la durante o processo de diagnóstico.

7.3.3 Otimização de Iterações

Dada uma iteração e seu respectivo escopo é possível que existam instruções no escopo que não dependem e não causam qualquer dependência na iteração, ou seja, do ponto de vista semântico é indiferente se a instrução está dentro ou fora desse escopo. Se a semântica é a mesma, então qualquer escolha é correta. Uma forma de tornar um par de programas mais semelhantes, portanto, é escolher uma convenção igual para ambos e o que estiver fora da convenção deve ser transformado para satisfazê-la.

É nesse contexto que descrevemos a otimização de iteração. Se do ponto de vista semântico não há diferença, do ponto de vista do desempenho, uma iteração com menos instruções leva vantagem. O processo de otimização consiste em visitar cada vértice contido no escopo das iterações do programa e retirar a instrução desse escopo caso determinado que não há mudança semântica.

Dada uma instrução qualquer dentro do escopo de uma iteração, ela não afeta o curso do programa se estiver de acordo com as seguintes premissas:

- O vértice não possui dependências de controle;
- Se o vértice possui dependência de controle, mas respeitar todas as seguintes condições:
  - A dependência de controle é única e exclusivamente porque a condição de controle o precede na seqüência de execução;

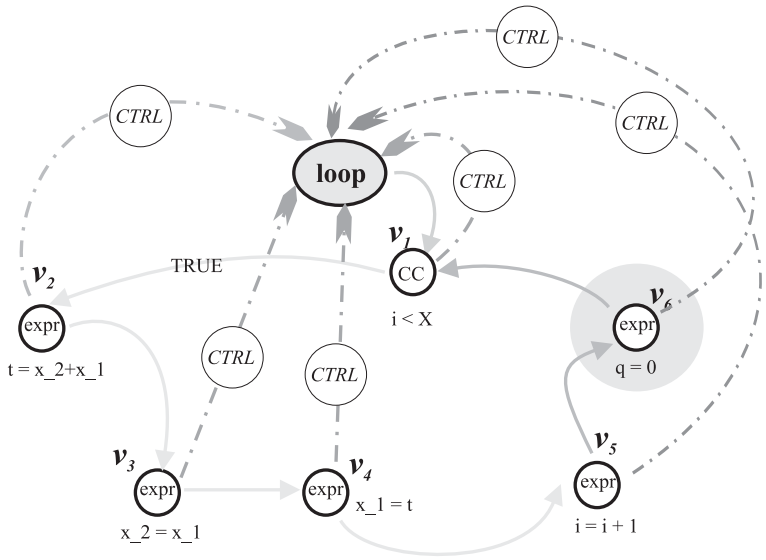


Figura 7.11: Otimização de Iteração

- A instrução é uma atribuição ou não é uma definição;
- Não existe nenhum vértice posterior ao escopo vigente que dependa desse vértice;

Essa situação ilustra instruções que não afetam e não são afetadas por nenhum elemento relacionado ao controle. Se a instrução é uma definição, a única razão que poderia mantê-la dentro do escopo seria o aspecto de condicional do escopo sobre esta instrução. Se nenhuma instrução posterior depende desta, então a retirada não deve afetar a semântica do programa. De fato, se as condições acima fossem respeitadas, esse tipo de alteração poderia ser realizada também em condicionais.

Ou seja, se o vértice analisado satisfizer tais premissas, sua presença é facultativa, e por convenção o vértice é retirado e posicionado antes da iteração. Na Figura 7.11 podemos observar uma iteração, na qual todos os vértices possuem dependência de controle. Mas observe que o vértice v_6 só depende do controle porque está depois da condição e é uma atribuição. Nesse contexto ainda não é possível averiguar se existe alguma instrução que dependa dessa atribuição, mas em caso positivo esta instrução poderia ser retirada sem prejuízo à semântica do programa.

7.3.4 Reordenação e Paralelização de Instruções

A diferença na ordem de instruções independentes é um dos primeiros fatores complicadores que vêm à mente quando se pensa no problema de comparar programas. E de fato, mesmo programas cujo algoritmo é o mesmo, com as mesmas estruturas de dados, podem ser muito diferentes. A estratégia de comparação escolhida em nossa abordagem, a ser

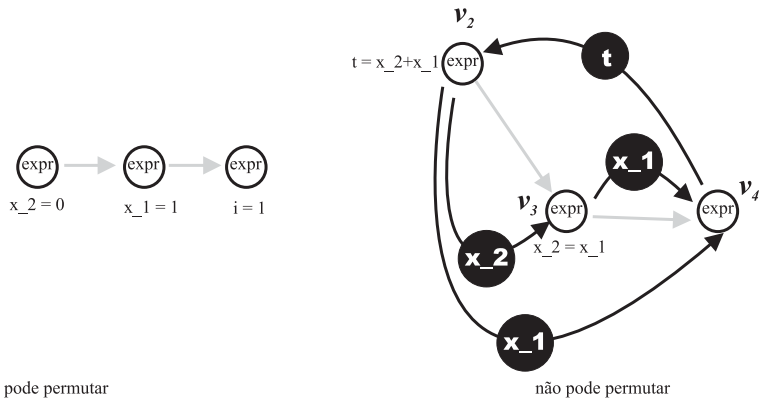


Figura 7.12: Exemplos de Instruções que podem (à esquerda) e que não podem permutar.

apresentada mais adiante, torna o problema da ordem praticamente irrelevante. O objetivo desta seção não é apresentar como resolver esse problema, mas revisar e reformular a questão e estudar as propriedades interessantes envolvidas.

Dados dois programas que fazem as mesmas coisas, usando as mesmas estruturas de dados de modo que podem diferir apenas na ordem, a questão central é identificar quais elementos devem sempre seguir a mesma ordem, e quais podem permutar sem prejuízo à lógica.

O fator que determina a ordem no programa de fato são as dependências de controle e dados. A grosso modo, dadas duas instruções distintas quaisquer elas podem ser permutadas se o fecho transitivo das dependências de dados de ambas não levar uma a depender da outra. A Figura 7.12 ilustra duas situações, uma na qual é possivel permutar e outra na qual não é. Observe que o critério é justamente a existência ou não das arestas de dependência.

Em casos nos quais a ordem de determinadas instruções não importa, é o mesmo que dizer que estas podem estar sendo executadas ao mesmo tempo. Dessa forma, uma análise cuidadosa pode determinar vários conjuntos de instruções que poderiam ser executadas em paralelo e esses conjuntos são ordenados de tal forma que cada conjunto estará na sequência obrigatória. Na Figura 7.13 o programa apresentado no início do capítulo é reorganizado de modo a agrupar as instruções em conjuntos, os quais podem por sua vez conter subconjuntos. O conjunto que envolve todos os conjuntos é o programa. Instruções em um mesmo subconjunto podem ser executadas ao mesmo tempo. A ordem de execução é dada pelas arestas entre os conjuntos. Assim, o fluxo inicia no vértice *init* e deloca-se para o primeiro conjunto.

O aspecto interessante é que determinados esses conjuntos, basta compará-los na ordem na seqüência em que estão, ou seja o primeiro de cada programa, depois o segundo. Se os conjuntos forem considerados um a um equivalentes, temos uma comparação bem sucedida. Imediatamente reduzimos o problema, mas ainda resta comparar os conjuntos.

Nesse contexto, podemos imaginar várias soluções para representar esse aspecto paralelo, mas certamente um dos mais interessantes é o uso de hipergrafos, tal como no trabalho de Guedes[19].

A comparação dos conjuntos pode ser a mais simples, para cada elemento do primeiro, determinar se este se encontra no segundo. Essa comparação pode ser aperfeiçoada comparando-se apenas vértices de mesma natureza. Uma outra forma de aperfeiçoar é arbitrar, como fizemos anteriormente através de uma convenção, uma ordem para os elementos nos conjuntos.

Antes de entrar nos detalhes de como a determinação e ordenação dessas instruções poderia ser realizada, algumas definições importantes:

- Um conjunto de instruções é uma abreviação para conjunto de vértices que representam instruções;
- Sendo C um conjunto de instruções, `ordem`(C) é a ordem na qual as instruções contidas podem executar na sequência do programa;
- Seja v um vértice, `ordem`(v) é a ordem na qual a instrução de v deveria ser executada. Se v já pertence a algum conjunto Q de instruções, `ordem`(v)=`ordem`(Q).

Um algoritmo capaz de realizar a tarefa de criar reordenar os passos deveria seguir, aproximadamente, os seguintes passos:

- Considere que todos os vértices que não dependem de nenhum outro vértice, são dependentes da raiz r do GFC e `ordem`(r)=0;
- Considere também que, com exceção da raiz, todos os vértices tem ordem indefinida;
- Crie uma fila com r;
- Enquanto houver vértices na fila faça:
  - retire v da fila;
  - se v ainda não pertence a nenhum conjunto de instruções:
    - faça com que todos os dependentes de v tenham ordem igual a `ordem`(v)+1 e que estejam no mesmo escopo de v;
    - se não existir um conjunto S de instruções tal que `ordem`(S)=`ordem`(v), crie esse conjunto;
    - se v é a raiz de um escopo, aplique recursivamente esse algoritmo tendo v como raiz. O conjunto resultado é associado a v;
    - insira v em S;

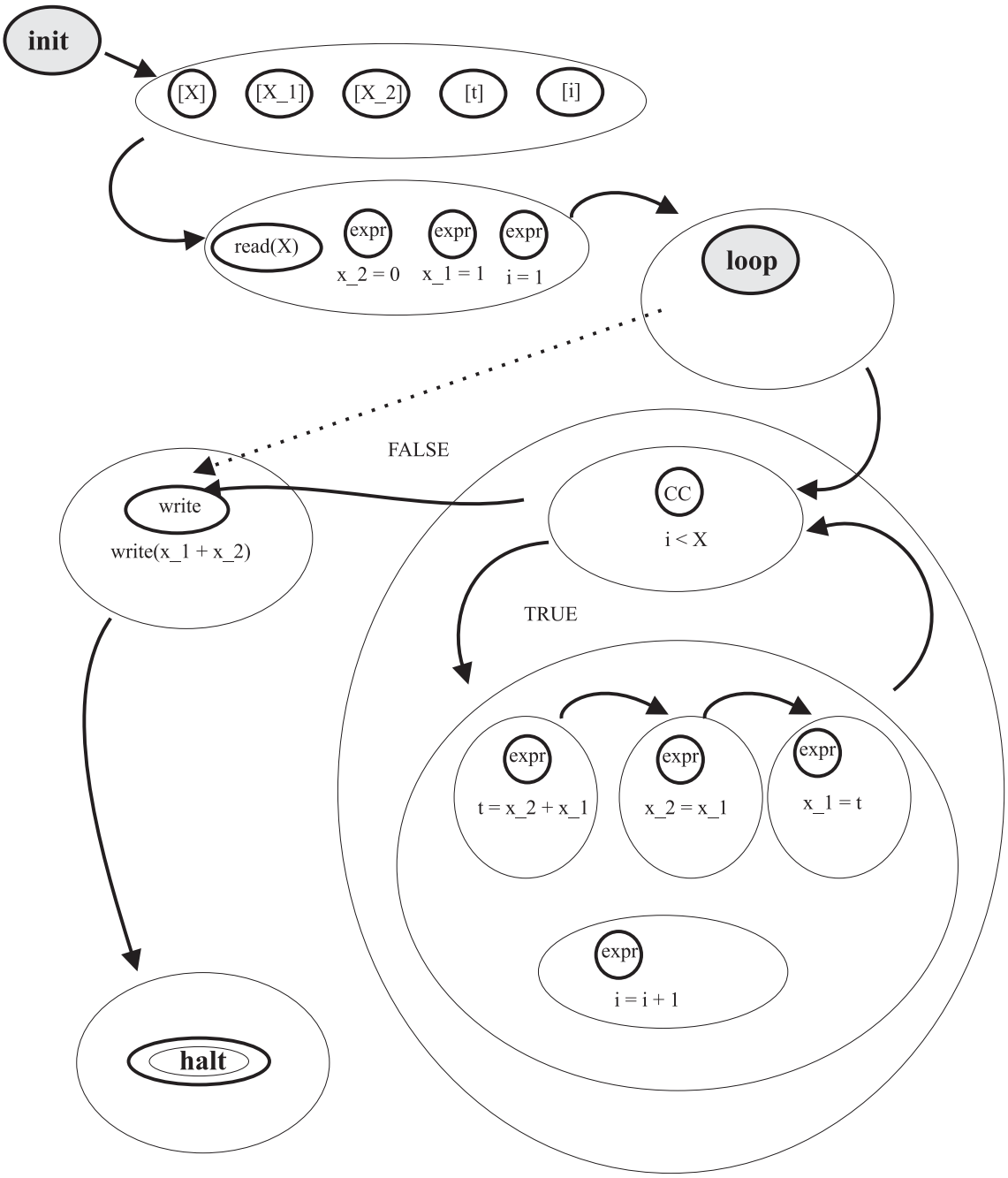


Figura 7.13: Instruções Paralelizadas

- Retorne a lista de conjuntos ordenados de instruções;

Observe que o algoritmo de ordenação nada mais é do que um caminhamento em largura. O resultado final será uma lista ordenada de conjuntos de instruções, cada qual pode ser um bloco inteiro com seu próprio conjunto. Uma vez que a abordagem escolhida para busca e comparação dos grafos utiliza GDDs e portanto não depende da ordem no GFC, este algoritmo não é necessário no processo de normalização. Ao invés disso, este é utilizado como parte do processo de ordenação de vértices para geração da representação

intermediária de programas e padrões, tal como será estudado na Secão 7.6.

7.3.5 Erros Semânticos Detectáveis sem Comparação

A partir do momento que os passos de normalização foram aplicados, é possível detectar alguns erros e código inócuo apenas estudando as dependências. A Figura 7.14 será usada para ilustrar os erros descritos a seguir. Nesta figura, as arestas pontilhadas são arestas de sequência com um número indefinido de operações que podem ocorrer entre o ponto de partida e o destino. As outras mantêm o significado usual.

- Uso de variável indefinida: é o caso de uma expressão que usa uma variável cuja definição não existe em momento anterior. Essa situação é detectada analisando a tabela de dependências externas da raiz do GFC. Na Figura 7.14 no GFC (1), a expressão $x = y + 10$ utiliza a variável y que nunca foi definida.

- Variável não utilizada: embora não seja um erro, é uma situação que vale a pena comentar porque pode ser um indício de que havia intento de usar uma determinada variável, mas não ocorreu porque seu nome foi digitado incorretamente no decorrer do programa. Essa situação é detectada quando a variável não faz parte de nenhuma definição ou referência. No GFC (2) da Figura 7.14 a variável y é declarada, mas nunca utilizada. O sistema remove a declaração.

- Definição inócua: situação na qual uma definição é realizada, mas o valor atribuído nunca é utilizado. Novamente, não é necessariamente um erro. Essa situação é detectada quando uma definição não tem nenhum dependente. No GFC (2) da Figura 7.14 o vértice da definição $x = 10$ nao possui nenhum dependente, portanto consituindo na exceção de definição inócua. O sistema remove a definição.

- Iteração potencialmente interminável: este é um dos erros mais interessantes que pode ser detectado. Uma iteração é potencialmente infinita se **não** existe um vértice v, tal que exista um fecho transitivo entre v e condição de contorno e tal que v satisfaça as seguintes condições:

  - v está dentro do escopo da iteração;
  - existe um fechamento transitivo tal que v depende de si mesmo ou v depende de u que depende de si mesmo por um fecho.

  A idéia por trás da mesma é que se a condição de contorno for suficiente para que o fluxo entre na mesma e nenhuma das variáveis dessa condição de contorno tiver seu valor modificado, então a condição de contorno nunca será falsa.

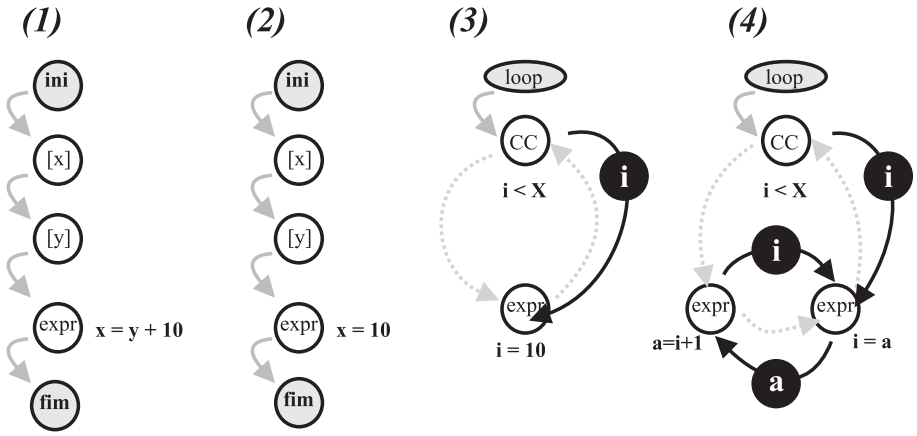


Figura 7.14: Erros detectáveis sem comparação

O GFC (3) da Figura 7.14 demonstra uma interação na qual a condição depende de uma definição que é invariável, pois não é função do número de ciclos executados. Essa é uma iteração potencialmente infinita. Em contraste, na iteração ilustrada em (4) a condição depende de um vértice que pode ter seu valor alterado no decorrer das iterações.

Estas são apenas algumas das condições possíveis de serem detectadas. Com análises mais complexas, seria possível determinar se blocos inteiros de código são inócuos, se determinadas expressões são invariantes, entre outros. Muitas dessas técnicas são utilizadas para otimização de programas, caso em que códigos inócuos sequer fazem parte do produto final[2], mas não é objetivo desse trabalho fazer análises extensivas e muito menos otimizar programas.

7.3.6 Composição de Expressões

Uma característica que pode facilmente variar de um programador para outro é a opção de trabalhar com expressões maiores contra o uso de várias expressões menores que atingem o mesmo objetivo. De maneira semelhante ao caso da otimização, a semântica pode ser a mesma, embora o programa seja diferente. A composição é o processo de normalização nesse caso que opta por expressões maiores, em particular porque o aumento no número de características na expressão também torna mais confiável o processo de identificação das mesmas.

O processo de composição é bastante simples e consiste em substituir as variáveis de uma expressão pelas expressões que a definem. O único aspecto a ser dada uma especial atenção é que uma variável será substituída pela expressão que a define somente se esta possuir ao menos uma variável. Essa restrição tem como objetivo evitar que as variáveis desapareçam das expressões, caso em tornaria mais difícil identificar variáveis.

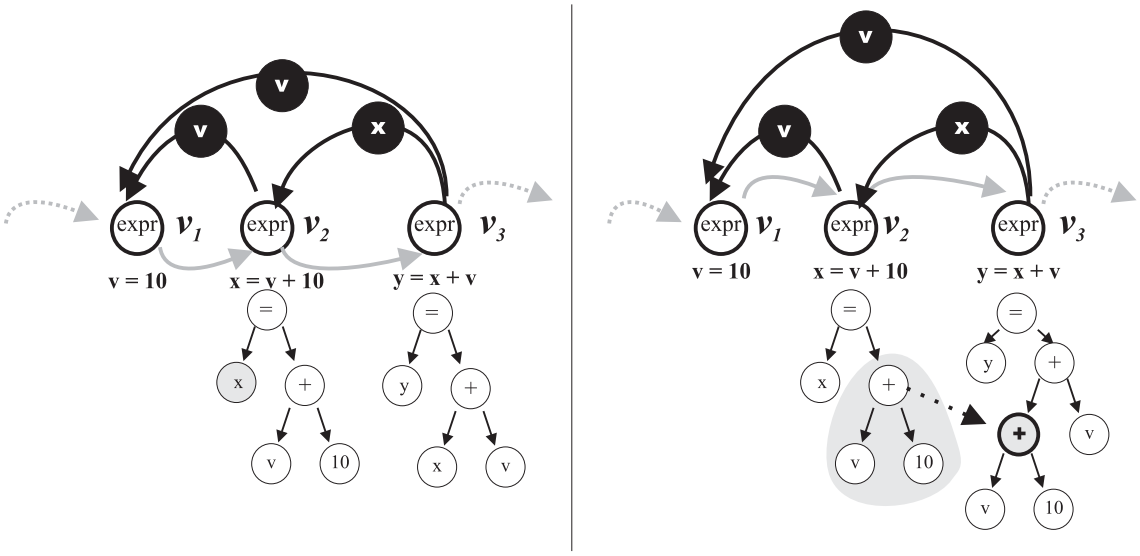


Figura 7.15: Processo de Composição de Expressões

Um fato a ser notado é que a estratégia de comparação que será posteriomente apresentada é capaz de trabalhar mesmo sem realizar a composição a priori, ou seja, a composição pode ser realizada na medida do necessário. A Figura 7.15 demonstra o processo de composição em dois passos. Primeiramente, à esquerda, a aresta de dependência de v_3 para v_2 indica que v_3 representa uma expressão que deve referenciar a variável x em algum momento. Uma vez que a expressão em v_2 não consiste apenas de literais, o sistema promove a substituição de x em v_3 pela expressão de v_2. O resultado final, sendo apresentado à direita na mesma figura.

7.3.7 Cenário

Nada melhor do que observar todas as técnicas apresentadas, aplicadas em conjunto em um exemplo integrado. Para isso, vale a pena recordar o exemplo descrito na Seção 7.2 e aplicar as técnicas estudadas. Uma vez que as técnicas observadas não envolvem comparação, vamos detalhar a aplicação das técnicas apenas sobre o grafo do aprendiz, o qual é mais interessante. Ao final da seção serão apresentados ambos os grafos normalizados para que o leitor possa comparar os resultados.

Primeiramente, são realizados os procedimentos de análise de dependências. Na Figura 7.16 é possível observar o grafo do programa apenas com as arestas de dependência. Em seguida são aplicados os procedimentos de separação de variáveis, otimização e composição de expressões. A Figura 7.17 apresenta estes procedimentos. Também nessa figura é possível observar o processo de detecção de erros sem comparação, no qual uma variável e uma definição inócuas são encontradas e portanto serão removidas. Finalmente, observe a figura 7.18 com o grafo normalizado de ambas as soluções.

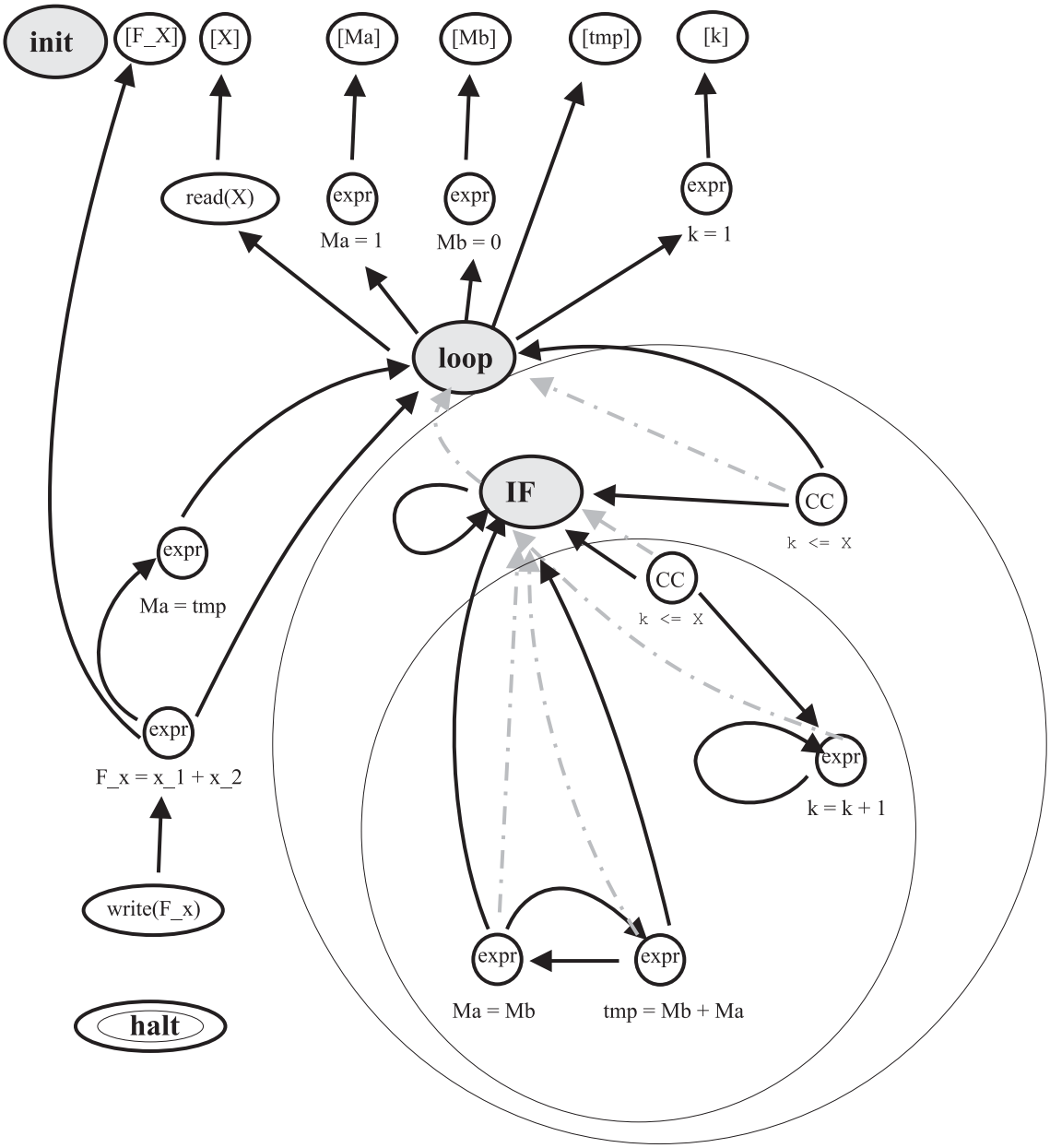


Figura 7.16: Grafo de Dependências

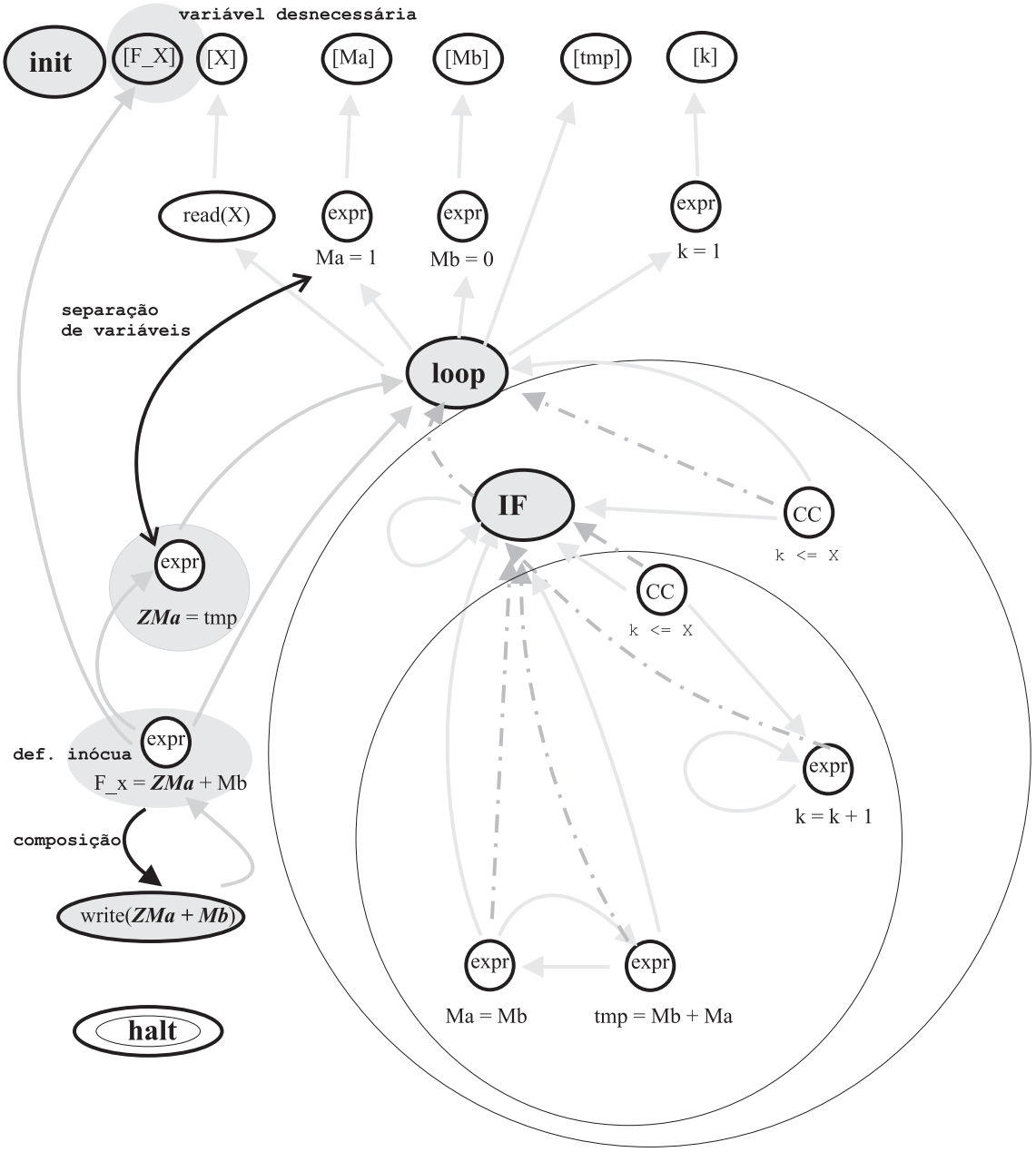
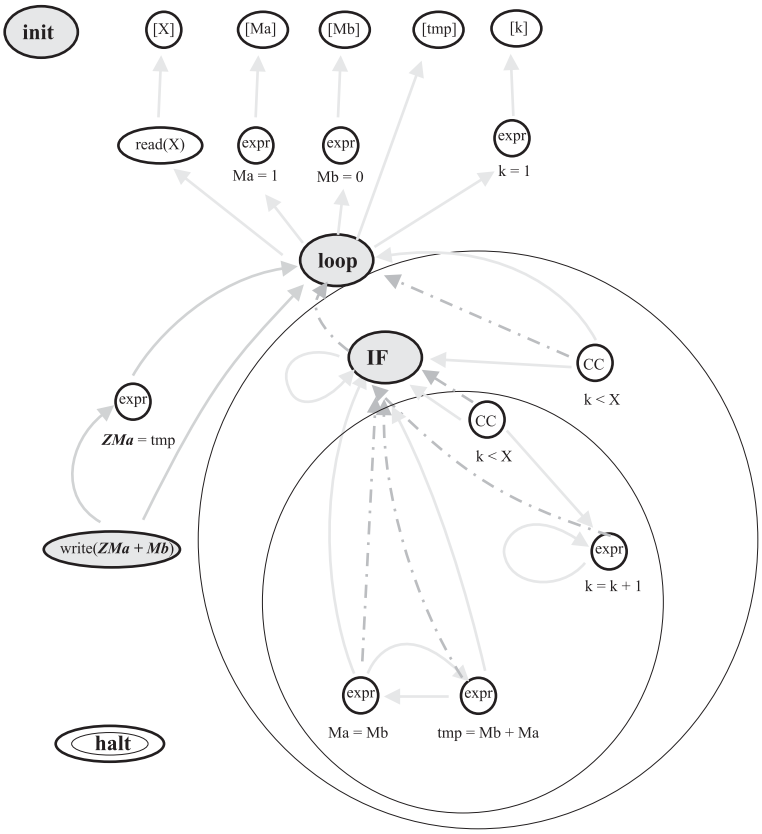
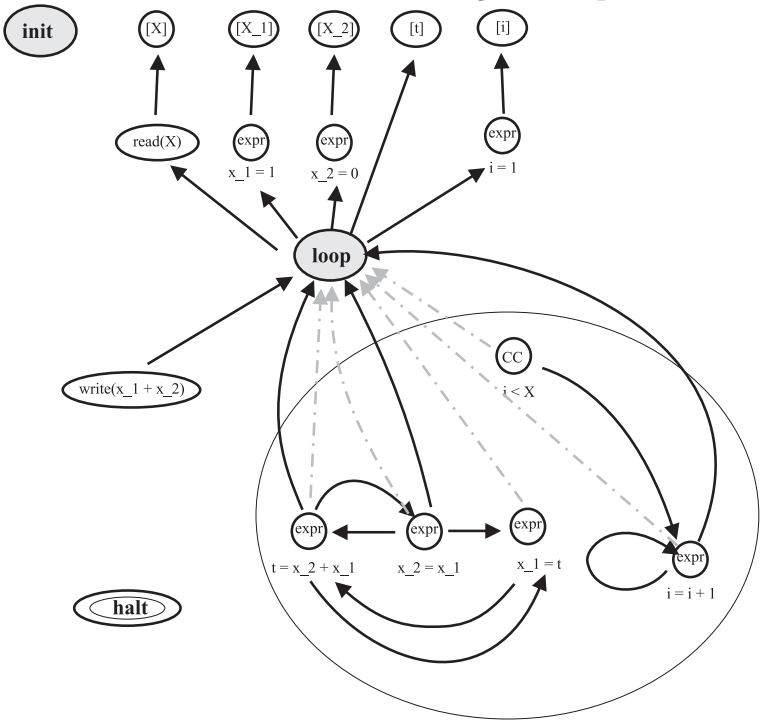


Figura 7.17: Normalização do Programa



Resultado Final da Normalização do Aprendiz



Resultado Final da Normalização da Referência

Figura 7.18: Comparação dos Resultados das Normalizações

7.4 Comparação de Grafos

Quando procuramos verificar se um par de grafos é o mesmo, estamos verificando se são isomorfos. O problema de verificar se algum subgrafo de outro grafo G é isomorfo a um determinado grafo H é NP- Completo[17, 31], mas de um grafo com outro grafo inteiro têm resistido às tentativas de classificação[31]. Se aceitamos que podemos aplicar transformações para tornar os grafos isomorfos, entramos no mérito do homeomorfismo que é um problema ainda pior[17].

Por outro lado, verificar se dois programas fazem a mesma coisa é um problema simplesmente indecidível. Uma prova informal segue do fato que o mínimo que podemos perguntar sobre a igualdade de dois algoritmos é se ambos param, ou se apenas um pára. Portanto, reduzimos imediatamente ao problema da parada, o qual segundo Turing[38] é indecidível.

O que este trabalho se propõe é verificar se dois programas, representados por grafos, fazem a mesma coisa. Em caso negativo é necessário explicar quais as discrepâncias encontradas e mostrar que correções seriam necessárias para que façam a mesma coisa. Ou seja, temos envolvidos isomorfismo, homeomorfismo, verificação de equivalência de soluções e, como se fosse pouco, ainda é necessário explicar porque não produzem o mesmo resultado, se for o caso. Com esse prospecto, é induvidável que a ferramenta será bastante limitada. Se os limites são tão grandes, é natural questionar a utilidade de tal ferramenta.

O que garante que mesmo uma ferramenta limitada pode ter resultados práticos interessantes é o público alvo: alunos iniciantes de programação. Em um curso elementar, a solução para os problemas propostos são normalmente curtas, os alunos têm a tendência de desenvolverem soluções ingênuas e semelhantes, os alunos incorrem em erros semelhantes e, principalmente, o professor que seria um hipotético autor de material, no geral tem uma boa avaliação do que se espera em termos de soluções dos alunos e é capaz de prover um pequeno conjunto de soluções que abranjam a grande maioria daquelas que os alunos deverão imaginar. Outra evidência é o relevante sucesso obtido por várias ferramentas de diagnóstico construídas no passado, tal como descrevemos no Capítulo 2.

O módulo de comparação é composto de quatro partes, tal como pode ser observado na Figura 7.19. O arcabouço principal do comparador é o identificador de vértices, procedimento que tem como objetivo determinar para cada vértice da solução de referência qual ou quais vértices da solução do aluno são responsáveis pela mesma tarefa. O identificador dispõe dos seguintes recursos, a serem detalhados na sequência, que podem ser utilizados para cumprir o objetivo de comparação:

- Determinação de trechos, por meio de padrão, que executam a mesma tarefa que outro trecho, culminando em substituição do trecho;

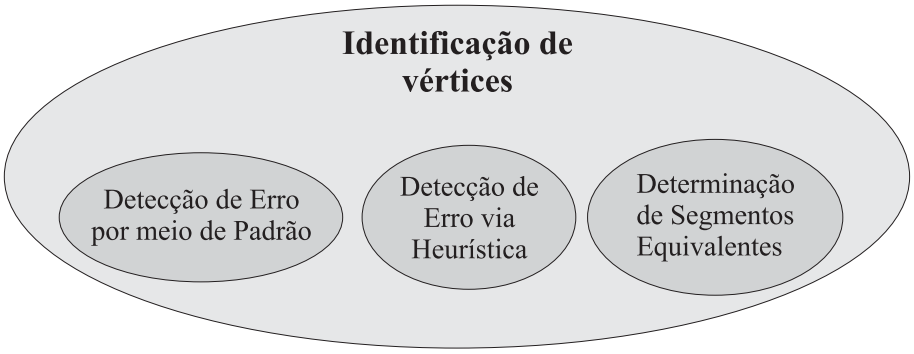


Figura 7.19: Organização do Módulo de Comparação

- Detecção e correção de erros por meio de heurística;
- Detecção e correção de erros por meio de padrão.

7.4.1 Direcionamento por Acertos

Uma característica importante do sistema concebido é que este assumi, a priori, que a solução do aluno é correta e realiza um esforço maior procurando identificar os aspectos corretos do programa do aluno tendo o grafo do professor como referência. A partir do momento que são exauridas as alternativas de identificação, ou quando foi dispendido muito tempo nessa empreitada, o sistema busca erros seja usando heurísticas ou padrões de erros construídos pelo autor.

7.4.2 Identificação de Vértices

O processo de identificação de vértices de dados consiste em determinar, para cada vértice de uma solução de referência, qual vértice ou conjunto de vértices efetuam a mesma operação. Dessa forma, este processo é o núcleo do procedimento de comparação, uma vez somente neste procedimento é possível que um par de vértices seja identificado.

Para evitar ambiguidades, desde já são convencionados dois grafos, R e A, respectivamente representando as soluções de referência e do aluno. Ainda, quando não especificado o contrário, o grafo padrão de trabalho é R.

A estratégia geral para solucionar o problema utiliza as dependências de dados e controle como elemento chave no processo heurístico de identificação. Parte-se do princípio que se ambos os programas representam o mesmo algoritmo e estrutura de dados, então as variáveis, expressões e definições deverão estar relacionadas de forma praticamente idêntica, e portanto respeitam as mesmas dependências. Dessa forma, como já foi mencionado, o algoritmo proposto não compara grafos de fluxo de controle, mas sim Grafos de Dependências de Dados (GDD). Um GDD é uma representação que tem como foco os

dados e as operações realizadas sobres os mesmos, análogo ao GFD. Dessa forma, a ordem imposta pelas arestas de um GDD é a mandatória e elementos que podem ser realizados em qualquer ordem entre si compartilham o mesmo nível de profundidade no GDD.

A abordagem escolhida consiste em modelar o casamento de vértices como um problema de satisfação de restrições, mas no qual é aceito satisfação parcial das restrições, refletindo principalmente o fato de que não é obrigatório que todos os vértices estejam casados ao final de uma execução do algoritmo. Embora a idéia de satisfazer apenas parte das restrições seja estranha a princípio, já houve outros trabalhos que abordaram essa questão, tal como Voudouris *et al*[11]. Os elementos comparados são os GDDs de referência e do aprendiz, G_R e G_A respectivamente, e o resultado esperado é que cada vértice de G_R esteja relacionado a algum vértice de G_A. Antes de descrevermos a modelagem, algumas notações importantes:

- `dependentes`(v) é o conjunto de vértices que dependem de v;
- `precedentes`(v) é o conjunto de vértices dos quais v depende, ou ainda, que *precedem* v;
- $v \in G_R$ e $v' \in G_A$, $v \varphi v'$ é uma relação verdadeira se v' é o vértice equivalente de v;
- `identifica`(G, G') determina se os grafos G e G' podem ser casados pelo algoritmo a ser apresentado;
- `escopo`(v) é o subgrafo que representa o escopo de v se este for a raiz de algum bloco de instruções;
- `exprequiv`(v, v') se e somente se v e v' representam expressões e estas são equivalentes segundo um algoritmo que será apresentado mais adiante;
- o `tipo`(v) é o tipo da instrução em v, o qual está relacionado ao que a instrução faz. Alguns exemplos:
  - expressão : a instrução é uma expressão aritmética, lógica, etc.
  - leitura: a instrução é uma leitura de dados;
  - controle de fluxo: a instrução efetua, ou pode efetuar, desvio do fluxo normal de execução.

A modelagem do problema consiste no seguinte:

- Variáveis do problema: dado um $v \in G_R$ desejamos determinar $v' \in G_A$ tal que $v \varphi v'$. Portanto, temos um conjunto de variáveis formado pelas icógnitas v' para cada $v \in G_R$.

- Domínio das variáveis: a princípio, dado $v \in G_R$, o domínio D(v) é o conjunto de vértices de G_A. Esse domínio certamente pode ser reduzido usando algumas heurísticas bem simples que veremos a seguir.

- Restrições: dados $v \in G_R$ e $v' \in G_A$,

  - $v \varphi v'$ somente se:

    $|\texttt{precedentes}(v)| = |\texttt{precedentes}(v')|$

    $\forall m \in \texttt{precedentes}(v), \exists m' \in \texttt{precedentes}(v') | m \varphi m'$

    $m, n \in G_R, m \neq n, m \varphi m', n \varphi n' \rightarrow m' \neq n'$

    Apesar de parecer complexa, esta restrição simplesmente diz que um vértice só pode estar relacionado a outro se todos os precedentes de ambos forem equivalentes entre si.

  - Se v é raiz de um escopo, $v \varphi v'$ somente se `identifica`(`escopo`(v), `escopo`(v'));

  - Se $v \in G_R$ e $v' \in G_A$ representam expressões, $v \varphi v'$ somente se `expequiv`(v, v');

  - `tipo`(v) = `tipo`(v'). Essa restrição pode ser usada para reduzir o domínio das variáveis ao invés de ser testada durante o processamento.

  - $v \in G_R$ somente pode ser identificada com um $v' \in G_A$ se estes estiverem no mesmo escopo e nível. Essa restrição é automaticamente imposta pela forma como os grafos são processados.

7.4.2.1 Procedimentos de Comparação

O procedimento de comparação, denominado `identifica`, recebe um par de grafos (G, G') e um conjunto H de hipóteses de identificação que representam quais vértices foram identificados e o nível de certeza dessa identificação. Um vértice é considerado completamente identificado se houver um vértice equivalente no outro conjunto e toda a árvore de dependentes desse mesmo vértice for casada também. O procedimento pode ser descrito da seguinte forma:

```
identifica(G, G', H)
Seja C_R o conjunto de vértices independentes de G;
Seja C_A o conjunto de vértices independentes de G';
retorne idConjuntos(C_R, C_A, H)/|G|
```

O procedimento de comparação não está limitado a comparar somente os grafos inteiros, mas também é capaz de comparar subgrafos. Uma vez que os blocos de comandos são tratados como apenas um vértice do ponto de vista do comparador, será necessário eventualmente comparar esses os subgrafos que representam esses blocos, e `identifica` pode ser usado para esse mesmo fim.

```
idConjuntos(C_R, C_A);
Sejam C_R e C_A, conjuntos de vértices de R e A, respectivamente;
Seja NV o número de vértices identificados, NV ← 0;
Para cada v ∈ C_R, determine o domínio D(v), tal que:
    ∀v' ∈ D(v), tipo(v') = tipo(v)

Ordene C_R, de modo que vértices com mais dependentes ficam na frente;
Para cada v ∈ C_R, na ordem acima faça:
    Instancie v com v' ∈ D(v);
    Verifique se equivalentes(v, v');
    Se a resposta for verdadeira então
    incremente NV;

retorne NV + idConjuntos(dependentes(v), dependentes(v));
```

Figura 7.20: Algoritmo do procedimento `idConjuntos`

De qualquer forma, os procedimentos nos quais o trabalho todo realmente é efetuado são `idConjuntos` e `equivalentes`. O procedimento `idConjuntos`, apresentado na Figura 7.20, tem como objetivo comparar um par de conjuntos e estabelecer relações entre os vértices de cada. Por sua vez, `equivalentes`, apresentado na Figura 7.21, é o procedimento responsável por comparar um par de vértice apenas e determinar se podem ser identificados.

Observe na Figura 7.21 uma referência a um procedimento denominado `exprequiv`. Tal procedimento poderia em seu próprio direito agregar uma série de heurísticas e transformações algébricas para determinar se as expressões são algebricamente equivalentes. Nesse trabalho, definimos de forma muito mais simples: duas expressões são equivalentes se os operadores são exatamente os mesmos e se as variáveis corresponderem. Uma variável corresponde a outra ou no caso de já estarem identificadas entre si, ou no caso de nenhum estar identificada, caso no qual é uma é identificada com a outra.

Nos algoritmos apresentados, foram omitidas estruturas de dados e procedimentos necessários para tratar de *backtracking*, com fim de se concentrar nos aspectos essenciais. Em cada momento no qual existe uma instanciação, o sistema deve ser capaz de retornar a este ponto e tentar uma nova instanciação no caso de falha.

7.4.3 Cenário

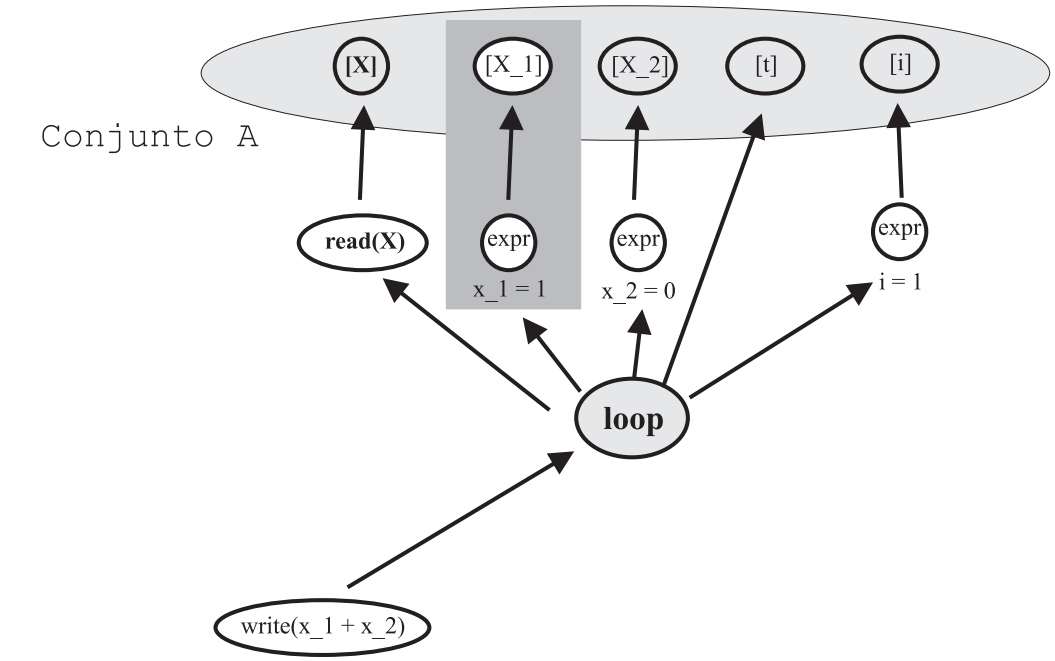
Supomos que o processo de identificação parte do grafo do aluno apresentado na Figura 7.32. Neste momento, nenhuma tentativa de identificação foi executada, ou seja, não existe nenhuma hipótese de identificação. Os vértices que correspondem às declarações de variáveis são os primeiros candidatos, uma vez que não possuem nenhuma dependência. Os vértices de início e fim de programa são automaticamente identificados, uma vez que

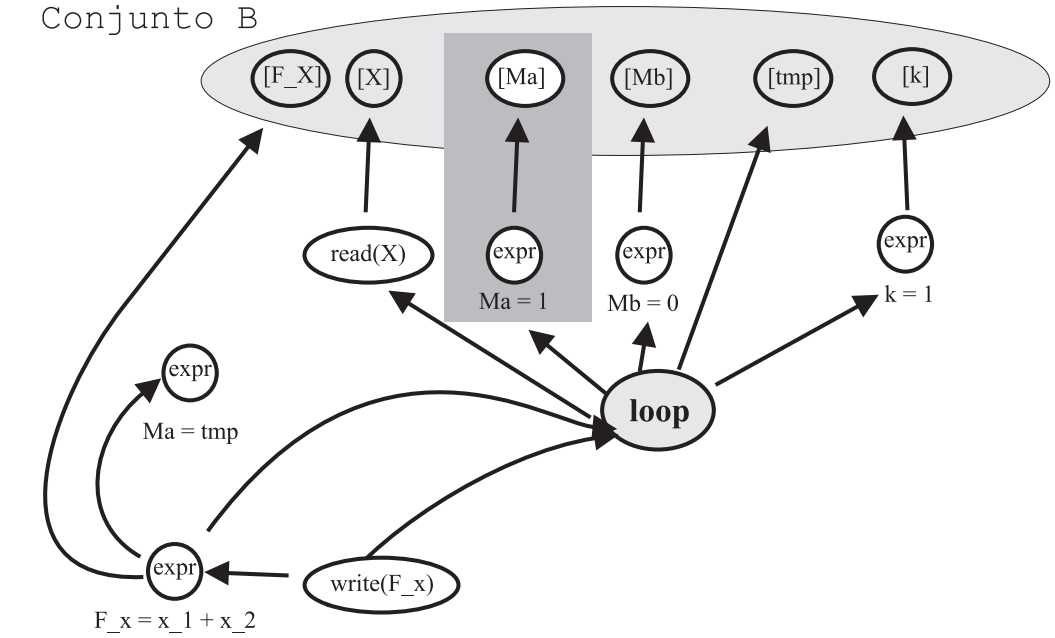
```
equivalentes(v, v')
se v é a raiz de um escopo:
    verifique se identifica(escopo(v), escopo(v'));
    se o retorno for falso, retorne falso.

senão, se v é uma expressão:
    verifique se exprequiv(v, v');
    se o retorno for falso, retorne falso.

senão:
    a instrução de v e v' é exatamente a mesma?
    Se não, retorne falso.
    se houver variáveis envolvidas na instrução:
        se a variável não foi identificada então
            instancie um valor para a mesma;
            senão conseguir instanciar, retorne falso;

        caso contrário
            a variável em v' deve ser identificada com v.

    se não houve sucesso nas comparações acima, retorne falso.

retorne verdadeiro.
```

Figura 7.21: Algoritmo do procedimento `equivalentes`

são únicos. Todos os vértices deste primeiro conjunto possuem apenas um dependente direto e portanto não importa a ordem de instanciação. Suponha que escolhemos a declaração de X. Observe na Figura 7.22 que os conjuntos de vértices comparados no primeiro passo estão marcados. A princípio, X pode ser identificado com qualquer vértice no conjunto B. Sendo assim, primeiramente será necessário identificar os vértices dependentes. Observando os dependentes de todos os vértices do conjunto B, somente dois vértices podem casar efetivamente, os vértices v_1 e v_3. Sem mais informações, é impossível dizer qual é a idenficação correta. Será necessário obter informações sobres os dependentes de v_4 e v_5. Observe que o vértice de iteração depende de ambos, mas a razão pelas dependências difere, ou seja, os vértices internos à iteração que causam as dependências são diferentes. Antes de tratar dessa questão, será necessário identificar os vértices de iteração. Na Figura 7.23 estão apresentados os grafos das iterações. O algoritmo não vai conseguir casar estes grafos, para começar porque um representa um *while* e o outro representa um *do ...while*. Será necessário fazer uma transformação no grafo do aprendiz, descrita na Seção 7.6.5, antes de tentar novamente. O sistema deverá continuar o processo de identificação até que não seja possível realizar nenhuma nova relação. Por exemplo, se o vértice u_2 fosse escolhido, pela análise das dependências imediatamente v_2 seria identificado. É importante perceber que o uso das relações de dependência dirige e

Solução de Referência



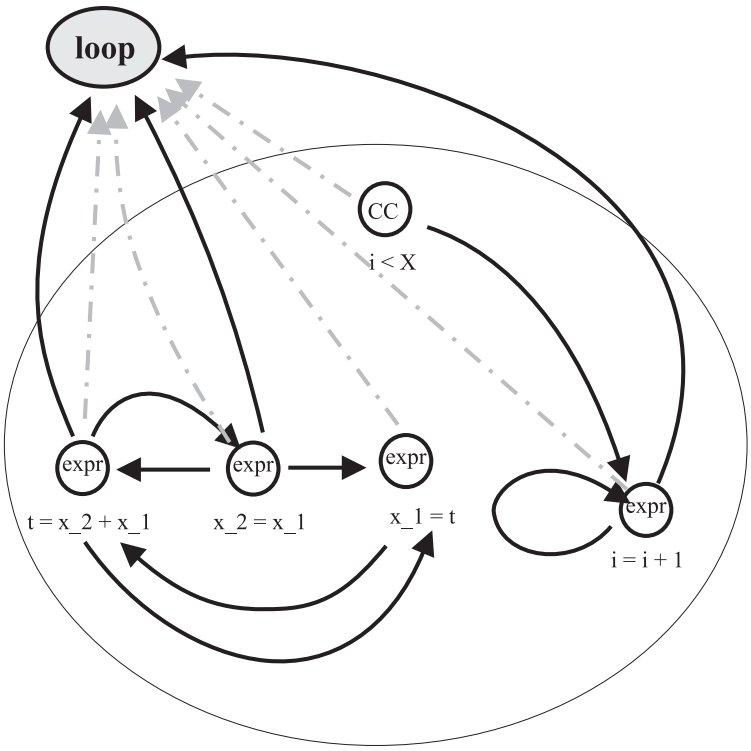


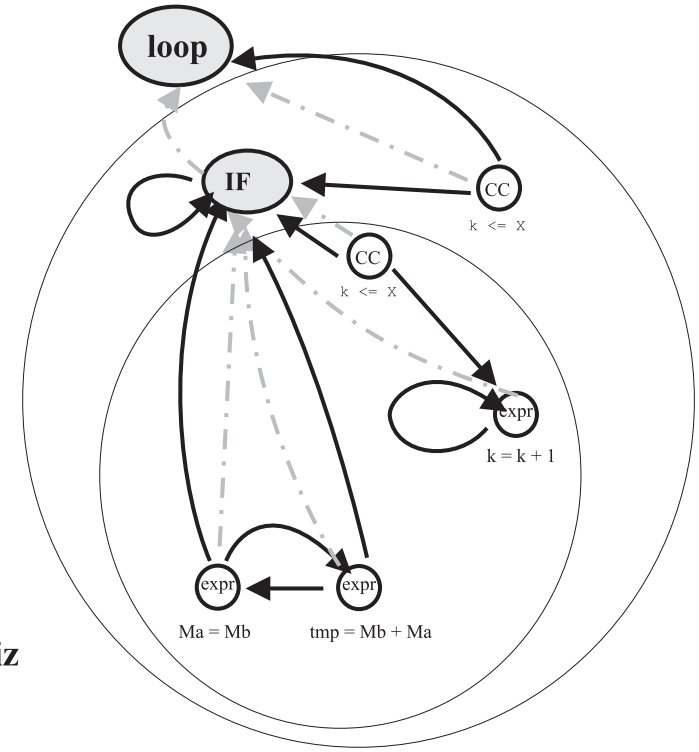
Solução do aprendiz

Figura 7.22: Ilustração da Identificação de Vértices

restringe bastante o processo de casamento, de modo que a convergência é relativamente rápida.

Solução de Referência





Solução do aprendiz

Figura 7.23: Ilustração da Identificação de Vértices

7.5 Detecção de Erros via Manipulação Heurística

A detecção de erros via manipulação heurística é entendida como o processo de determinar erros manipulando diretamente o grafo segundo determinadas heurísticas, sem uso de qualquer recurso provido pelo autor de curso. Essa detecção é um recurso ao qual se apela apenas quando o processo de comparação e identificação de instruções corretas não pode prosseguir, seja porque o tempo limite expirou ou porque não existem hipóteses válidas.

Esta seção se concentra em apresentar que tipos de erros são detectáveis por meio desse mecanismo, quais os procedimentos envolvidos e, por fim, quais as limitações. Ao invés de manter a abordagem usual de apresentar um cenário para cada processo ao final da seção, optamos por não fazê-lo nesta para ilustrar os procedimentos de detecção via manipulação heurística no mesmo que descreve o uso de padrões, na Seção 7.6.5. Esta escolha se deve ao fato de que a detecção de erros ocorre entre o procedimento de uso do padrão de acerto e do padrão de erro.

7.5.1 Objetivos e Procedimentos

O objetivo desse mecanismo é o de, usando apenas o grafo e alguns recursos internos, detectar erros que possam ser explicados de forma significativa. Os erros e procedimentos a serem descritos partem sempre do princípio que somente os elementos não identificados serão testados.

- Violação de dependência: situação na qual uma expressão viola uma dependência de dados em relação à solução de referência. Para detectar essa situação, a restrição de comparar apenas elementos com mesmas características de dependência é retirada. A Figura 7.24 apresenta uma situação na qual uma violação de dependência ocorre. No centro vertical da mesma, é possível observar a solução de referência, a qual está sendo comparada com duas soluções possíveis. A solução B possui um par de vértices que foi identificado que violam a dependência por x_2.

- Violação de escopo: instrução na solução do aluno se encontra em escopo diferente da equivalente na referência. Essa situação é detectada no momento em que é retirar a restrição de que somente elementos de mesmo escopo e nível podem ser comparados. Voltando à Figura 7.24, a solução A demonstra esse tipo violação.

- Instrução excedente, mas inócua: no momento em que todos os vértices do grafo foram identificados, é possível que alguma instrução a mais exista no grafo do aluno, embora não consista em prejuízo para o resultado final. O vértice que a representa pode ser removido.

Solução Correta	Equivalências	Solução do Aluno	Erro detectado
$X = Y + (M * 4)$	$X = Q, Y = K, M = L$	$Q = K + (L/4);$	operador incorreto (/ onde deveria ser $*$)
$X = Y + (M * 4)$	$X = Q, Y = K, M = L$	$Q = K + (L * 14);$	constante incorreta (14 onde deveria ser 4)
$X = Y + (M * 4)$	$X = Q, Y = K, M = C, C \neq L$	$Q = K + (L/4);$	variável incorreta (L onde deveria ser C

Tabela 7.1: Ilustração dos Erros de Expressão

- Instrução excedente causando violação de dependência: pode ser permitido remover um vértice para testar se alguma violação de dependência ocorreu. No geral, é evitado esse tipo de procedimento, a não se que todas as alternativas de identificação tenham exauridas, devido ao fato de que um vértice essencial pode ser removido.

- Instrução omitida: uma instrução do programa de referência pode migrar para o programa do aluno se, dado um vértice v, este já está identificado e todos os seus dependentes estão identificados, mas há uma (e apenas uma) dependência a menos em relação ao programa de referência, indicando que faltou alguma instrução no programa do aluno. Esse procedimento deve ser usado com cautela, uma vez que pode alterar demais a solução do aluno e porque não é provável que o sistema seja capaz de dar uma explicação muito significativa para o aluno sobre esta correção.

- Expressão incorreta: a expressão possui um, e somente um, dos seguintes erros em relação à referência:

  - Operador incorreto: um dos operadores está trocado;
  - Constante incorreta: uma das constantes está diferente da correta;
  - Variável incorreta: uma das variáveis está diferente da correta.

  Uma expressão incorreta é identificada quando as restrições respectivas, seja quanto a operador, variável ou constante, são deixadas de lado. A Tabela 7.1 ilustra cada um dos erros mencionados acima.

Observe que todas as detecções de erros consistem em desabilitar a imposição de alguma restrição. A abordagem apresentada é que cada tipo de erro deve ser testado em uma fase diferente, de modo que em momento algum mais de uma restrição é quebrada. Outro aspecto importante é que a partir do momento que um erro é detectado, esse procedimento relata o erro, corrige-o e retorna ao procedimento de identificação de elementos equivalentes. Seguindo esses princípios, o impacto de quebrar uma restrição é reduzido, uma vez que evita-se uma explosão ainda maior de combinações.

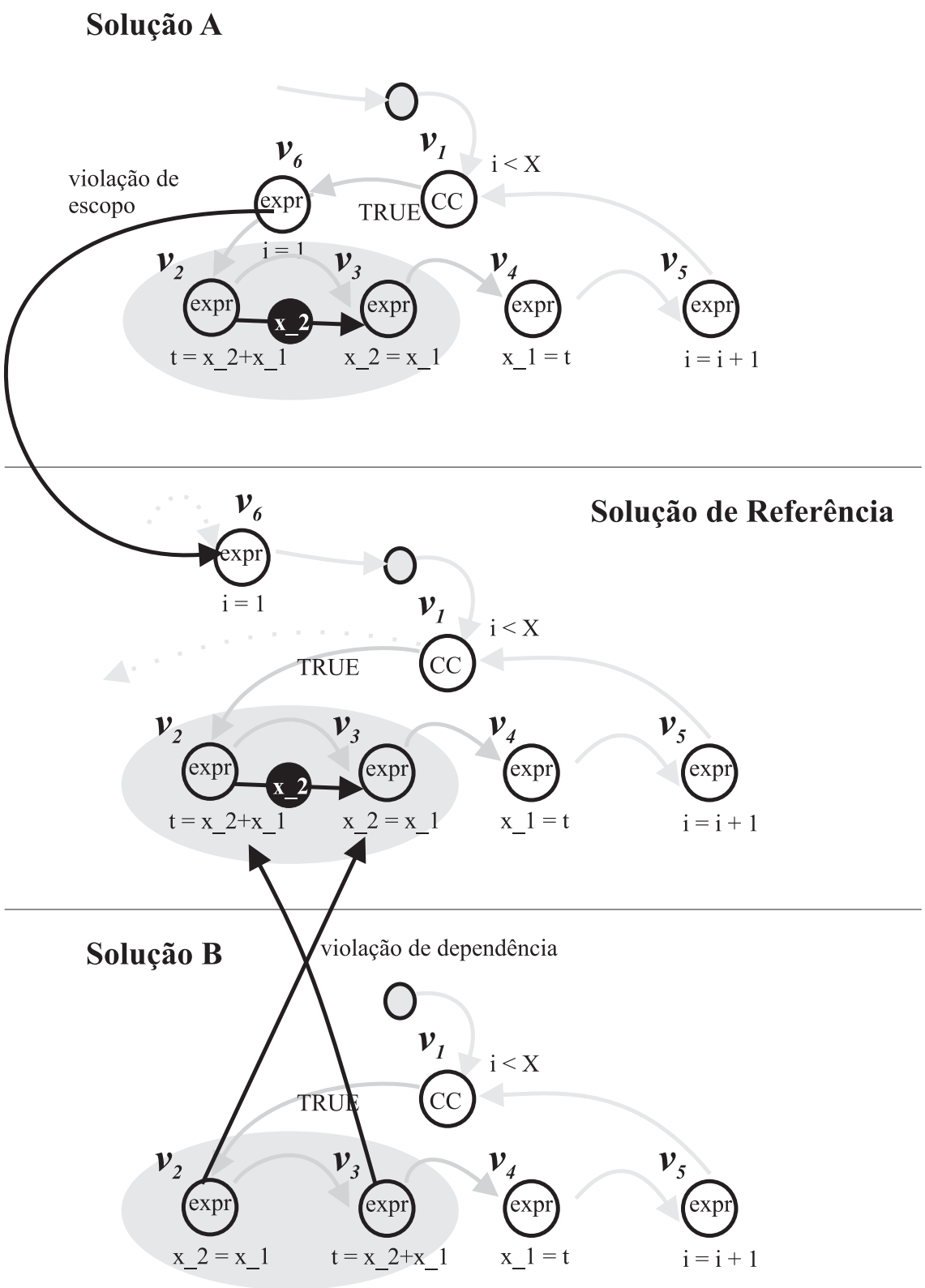


Figura 7.24: Demonstração dos erros de violação de dependência ou escopo

7.5.2 Limitações

Uma vez que os procedimentos de detecção de erros envolvem quebra de restrições, dessa forma permitindo e comparações entre elementos "proibidos", o número de hipóteses que podem ser aceitas é proibitivamente maior do que a comparação entre elementos teoricamente corretos. É por essa razão que tais mecanismos são usados com cautela e o porquê do processamento ser voltado para acerto ao invés de erro.

7.6 Padrões para Detecção de Erros e Acertos

O uso de padrões responde por um dos aspectos mais difíceis de gerenciar, mas ao mesmo tempo é o maior responsável pela detectacção de erros complexos, viabilização de feedback autorável e reconhecimentos de procedimentos equivalentes que de outra forma seriam quase impossíveis de processar.

Neste trabalho, o casador de padrão concebido e implementado, é capaz apenas de lidar com o casamento de textos. Dessa forma, é mandatório que o grafo seja convertido para um texto que possa ser usado com o mecanismo implementado. Essa representação já foi discutida na Seção 4.2.3.

A decisão de utilizar texto é fundamentada em dois alicerces: o fato de que casar texto exige menos estruturas complexas e é um assunto melhor dominado na comunidade em geral. Apesar de um padrão representado exclusivamente como grafo e processado dessa forma ser dotado de maior flexibilidade, o casamento de tais estruturas é um problema que envolve comparação de grafos, portanto justificando evitar tal solução.

7.6.1 Procedimento de casamento

Como foi observado, o sistema de casamento de padrão concebido e desenvolvido no âmbito desse trabalho é apenas capaz de lidar com textos. Para resolver esse problema, na Seção 6.3 foi apresentado um procedimento capaz de realizar a conversão de um GFC para uma representação textual sem prejuízo à semântica do programa. Apesar de todas as ferramentas necessárias para viabilizar o uso do casador de texto estarem disponíveis, converter os grafos do padrão de programa imediatamente é uma estratégia pouco efetiva e que pode prejudicar e até comprometer a qualidade dos resultados desse mecanismo.

O problema principal problema encontrado neste caso é a questão da ordem das instruções. No mecanismo de comparação de grafos, esse problema é irrelevante devido à estratégia de caminhamento por meio do grafo de dependências, mas neste caso em que os padrões e programa são textos não é possível dispor destes recursos.

A solução mais simples para este problema, consiste em realizar todas as permutações possíveis entre os passos, ou seja, um algoritmo de ordem fatorial. Ao invés de apelar para uma solução exaustiva, a abordagem escolhida envolve o uso de uma ordenação topológica

e a adoção de uma convenção homogênea na ordem dos passos permutáveis de modo a aumentar a chance de casamento sem que seja necessário tentar todas as alternativas.

Uma ordem parcial é uma relação na qual determinados elementos devem anteceder outros. O grafo de dependência de dados descreve uma ordem parcial, uma vez que determina que algumas intruções necessariamente devem ser executadas antes. Uma ordenação topológica é um processo de ordenação que utiliza uma ordem parcial como critério para determinar a posição dos elementos. O procedimento ordena a entrada de forma que componentes relacionados pela ordem deverão estar na mesma ordem na saída, enquanto aqueles isentos entre si podem estar em qualquer ordem arbitrária. Dessa forma, a ordem arbitrária pode usar qualquer convenção que seja conveniente. Uma ordenação topológica é usada para determinar a ordem que instruções de um grafo devem ter na representação intermediária. A chave para reduzir ao máximo o problema da ordem é determinar um critério de ordenação que seja capaz de discernir ao máximo instruções de mesma sequência. Os seguintes fatores podem ser utilizados para determinar qual a instrução que deve ser posicionada antes:

- vértices identificados;
- vértices que representam expressões;
- ordem lexicográfico do nome das funções e chamadas a procedimentos;
- valor da soma das constantes literais numéricas;
- ordem lexicográfica das constantes de string;
- mantendo apenas os operadores na ordem, a ordem lexicográfica da string que representam;
- número das dependências de uma definição (quanto menos, menor a ordem).
- se ainda houver empate, pode ser usada a ordem dos dependentes como fator de desempate, ou seja, se um par de vértices A e B empataram, aquele que possuir o dependente de menor ordem vêm antes.

Estes são apenas alguns do fatores que podem ser utilizados para o critério e como parte de uma função que atribui um número de ordem. Quanto maior o número de critérios, maior a probabilidade de dois programas equivalentes distintos terem os elementos na mesma posição. Naturalmente, o abuso de critérios pode causar um excesso de processamento sem resultados muito significativos. Se após a aplicação de todos os critérios não houver desempate entre um par de instruções, pode-se gerar tantas representações quantas forem necessárias para cobrir todas as alternativas ou simplemente manter a ordem de aparecimento. Ao adotar esta última, é certo que corre-se o risco do processo falhar,

```
while(??cond: isExpression ()) {
    ??corpo: isStatementList ()
}
```

Figura 7.25: Padrão de Erro

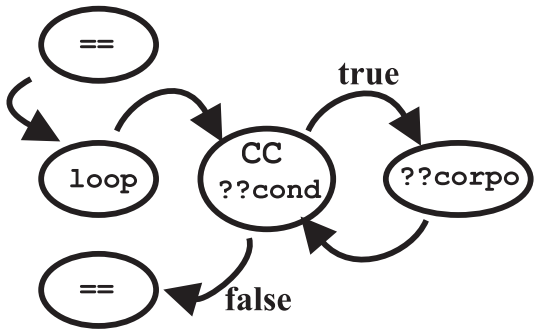


Figura 7.26: Grafo do Padrão de Erro

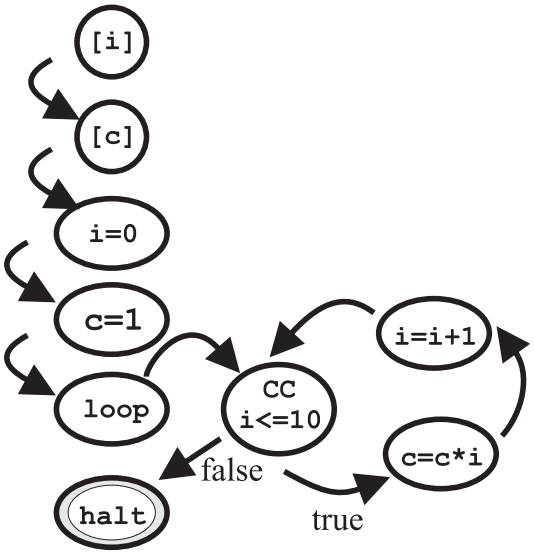


Figura 7.27: Programa na forma de GFC

```
[i];
[c];
i=0;
c=1;
while(i<=10){
    c=c*i;
    i=i+1;
}
```

Figura 7.28: Programa na representação intermediária. Em itálico, o trecho casado.

Resultados do casamento do padrão:
??cond: $i <= 10$

??corpo = {
$c = c * i$;
$i = i + 1$;
}

Figura 7.29: Exemplo de padrão convertido e casado

mas de qualquer forma o problema será bastante amenizado após a aplicação de todos os critérios. Na Figura 7.29 é apresentado um exemplo de casamento de padrão, desde o passo de conversão até o casamento efetivo.

```
do {                                              while (?? condicao : isExpression ()) {
    if (?? condicao : isExpression ()) {              ??corpoWhile:isStatementBody()
        ??corpoWhile: isStatementList ()          }
    }
} while(?? condicao : isExpression ());
```

Figura 7.30: Exemplo de Padrão de Acerto

7.6.2 Acertos e Padrões de Equivalência

Os padrões de acertos são padrões que indicam segmentos equivalentes de códigos. Ou seja, através de um padrão de acerto é possível que o autor especifique transformações arbitrárias[1]. Dessa forma, a capacidade de comparação do sistema pode ser melhorada sob demanda e de forma desacoplada dos mecanismos internos.

- um padrão que representa o trecho que se espera encontrar no programa do aluno;
- um padrão que representa a forma equivalente.

O idéia é bastante simples: a partir do momento que o processo puramente heurístico não produzir novas identificações ou simplesmente esgotar um limite de tempo, o sistema procura o padrão no código do aluno. Se o padrão for encontrado, como foi visto na Seção 7.6, os meta-símbolos do padrão estarão instanciados. Uma vez que o padrão que representa a forma equivalente deve usar os mesmos meta-símbolos, estes serão devidamente substituídos pelos valores instanciados e o trecho casado será substituído pelo trecho equivalente. Nesse caso de sucesso, após a substituição o processo retorna à comparação heurístico em busca de novas identificações. Em caso de fracasso, o caminho tomado é a busca de erros, primeiramente heurístico e depois usando padrões, tal como será visto na próxima seção.

Uma questão que vale ser levantada é quanto a possível intersecção desse recurso com o uso de múltiplas alternativas de solução. De uma certa forma, seria possível usar esse mecanismo para substituir o uso de alternativas, mas o uso de padrões para tal fim é menos intuitivo. O objetivo desse mecanismo é estar voltado para representar trechos mais curtos de código. Na figura 7.30 temos um exemplo desse tipo de padrão.

7.6.3 Padrões para detecção e Explicação de Erros

Assim como existem padrões para reconhecer e substituir trechos corretos, exitem padrões construídos especificamente para encontrar erros conhecidos. Um padrão de erro é formado de três partes:

[1]Observe que mesmo sendo potencialmente arbitrários, padrões excessivamente complexos podem causar impacto no desempenho.

```
read(?v2);                                  read(?v2);
==                                          ==
while(?v<\\=?v2:notEqual(v)) {              while(?v<?v2:notEqual(v)) {
    ??corpoWhile: isStatementList ()            ??corpoWhile: isStatementList ()
}                                           }
```

xxxx A iteração do programa deve ser usada para calcular o Fibonacci de **?v2-1** e ?v2-2. Isso signfica que o número máximo de passos é ?v2-1, mas no seu programa a variável de iteração ?v pode se tornar igual (?v¡=2). Para corrigir este problema, basta usar a relação de estritamente menor, ao invés de menor ou igual.

Figura 7.31: Exemplo de Padrão de Erro

- um padrão que representa um trecho incorreto;
- um padrão que representa a forma correta de redigir o trecho;
- um padrão que representa que explicação deve ser dada ao aluno caso este erro seja identificado.

O processamento é quase idêntico à busca de trechos equivalentes, ou seja, o sistema tenta encontrar o erro e, em caso positivo, substitui o trecho incorreto pelo correto. O que diferencia os processos é que o padrão de explicação também tem seus meta-símbolos instanciados e será armazenado no grafo que representa o código original que este erro foi detectado e quais vértices estavam envolvidos[2]. Na Figura 7.31 é apresentado um padrão que pode ser usado para identificar erros em um programa específico, no qual a iteração deveria ocorrer ao menos uma vez. É importante observar que os erros no geral dependem muito do contexto, pois existem casos no qual a situação inversa seria erro ou até mesmo ambas seriam corretas se incluídas algumas linhas, por exemplo replicando as instruções do corpo antes da iteração.

7.6.4 Considerações Sobre o Uso de Padrões

Como foi possível observar, o uso de padrões traz muitas vantagens, desde aumentar consideralvemente a capacidade do sistema em aceitar soluções equivalente até conferir um flexível mecanismo de identificação e explicação de erros. Mas certamente o sistema não é livre de sérias limitações.

A maior limitação que deve-se impor no processamento, seja referente a acerto e erro, é que um vértice deve estar envolvido em somente uma trasnformação que o envolva. Em primeiro, porque o uso de padrões pode causar processamento interminável, seja por culpa de padrões que causam reincidência, ou simplesmente porque o sistema nunca converge

[2]Vale lembrar que os vértices do grafo de trabalho sempre mantêm uma referência para os vértices equivalentes no original.

pela inexistência dos padrões necessários. Em segundo e não menos importante, é o fato de que o programa do aluno não deveria ser alterado demais porque estaria fugindo da intenção original do aprendiz e portanto desperdiçando o conhecimento do mesmo que pudesse estar correto. E por fim, o custo do procedimento de casamento de padrões, o qual por si só é um problema NP-Completo.

7.6.5 Cenário

Para finalizar o Capítulo, veremos neste cenário a aplicação dos padrões no programa, fazendo tanto correções como criando novas oportunidades de detectar segmentos equivalentes. Suponha que os padrões dos quais dispomos são aqueles descritos nas Figuras 7.30 (acerto) e 7.31(erro). Continuaremos aplicando transformações no programa do aluno no intuito de aproximar do GFC do professor.

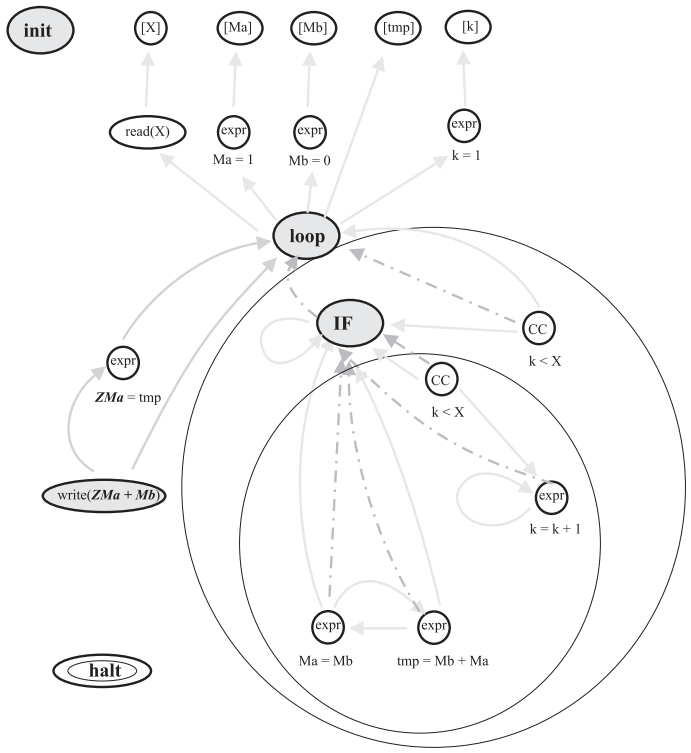
O primeiro passo a ser dado antes de testar o casamento é a conversão do programa alvo para representação intermediária. Na Figura 7.32 temos o grafo representando o programa do aprendiz e sua respectiva representação intermediária.

Como foi dito, o acerto deve ser testado antes de testar o erro, e isso inclui ser testando antes dos erros detectados via manipulação heurística, razão pela qual esse procedimento será ilustrado nesta mesma seção. Usando o padrão de acerto da Figura 7.30, o sistema é capaz de detectar o trecho ilustrado na Figura 7.33. Na parte superior da Figura 7.34 pode ser observado o grafo da solução após a substituição dos segmentos.

Uma vez que o padrão foi aplicado, o sistema tentará realizar novos casamentos. Suponha que o sistema consegue determinar que o escopo do *while* na programa do aprendiz provavelmente deve casar (heurístico). A condição para este casamento é corrigir a expressão. O mecanismo de detecção via manipulação heurística seria capaz de lidar com o erro da expressão, marcando o erro *operador incorreto*. Mas vamos supor que esse erro não é tratado para ilustrar a detecção de erros via padrão. Mas outros erros, mais elaborados, podem ser detectados: *violação de dependência* e *violação de escopo*. Ambos os erros estão ilustrados na Figura 7.34, a qual apresenta uma comparação entre os grafos de referência e do aluno. Nessa figura as arestas tracejadas representam as identificações ocorridas quando foi deixado de lado alguma restrição. As arestas claras representam dependências de dados e as arestas pretas representam a dependência que obriga a ordem entre dois pares de vértices, ou seja, são as dependências principais. A partir do momento que as correções são aplicadas e o problema de violação de escopo for resolvido, ZMa deverá retornar ao nome original, uma vez que no novo contexto esta não poderia ser separada. Essa é uma situação que ilustra a importância da variável, apesar de separada, manter uma relação com a original.

Finalmente, chegamos ao estágio de detecção de erro por meio de padrão. Na Figura 7.35 podemos observar o grafo novamente transformado para a representação in-

termediária e a aplicação do padrão de erro da Figura 7.31, corrigindo o erro. Observe na Figura 7.36 que, após sofrer as modificações corretivas e de equivalência, o grafo da solução do aluno fica praticamente idêntico ao grafo de referência.




```
[tmp];
[Mb];
[Ma];
[k];
[X];
Mb = 0;
Ma = 1;
k = 1;
read(X);
do {
    if (k <= X) {
        Ma = Mb;
        tmp = Mb + Ma;
        K = K + 1;
    }
} while(k <= X);

ZMa = tmp;
write(ZMa + Mb);
```

Figura 7.32: GFC para representação intermediária

```
[tmp];
[Mb];
[Ma];
[k];
[X];
Mb=0;
Ma=1;
k=1;
read(X);
do {

    if (k<=X) {
       Ma=Mb;
       tmp = Mb + Ma;
       K=K+1;
    }
} while(k<=X);
ZMa=tmp;
write(ZMa+Mb);
```

O trecho em Itálico representa o segmento casado.

```
[tmp];
[Mb];
[Ma];
[k];
[X];
Mb=0;
Ma=1;
k=1;
read(X);
while(k<=X) {
        Ma=Mb;
        tmp = Mb + Ma;
        K=K+1;
}
ZMa=tmp;
write(ZMa+Mb);
```

Programa após a substituição do segmento equivalente.

Figura 7.33: Usando o padrão de equivalência para obter um novo programa

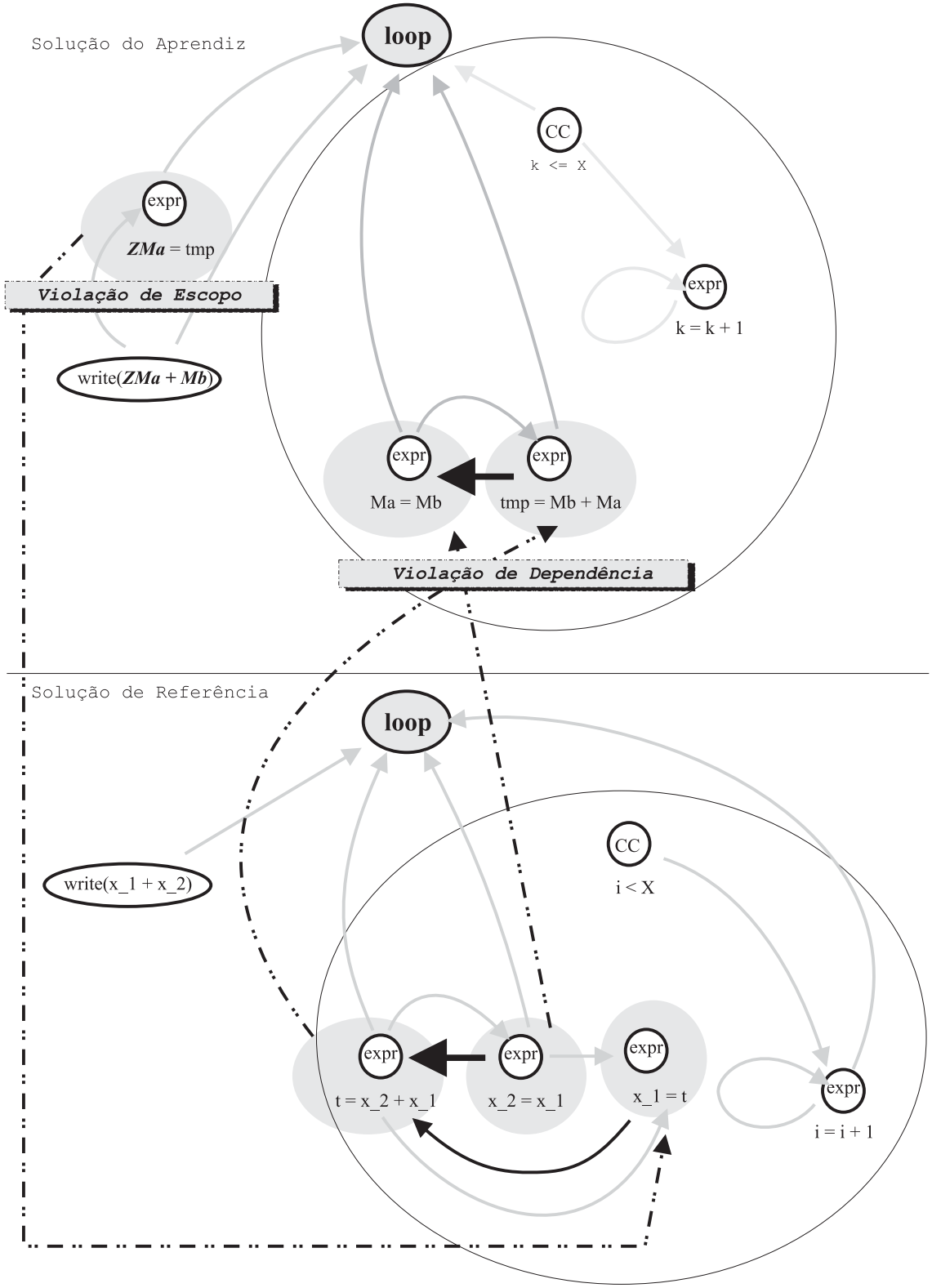


Figura 7.34: Processo de detecão de erros por meio de heurística

```
[tmp];
[Mb];
[Ma];
[k];
[X];
Mb=0;
Ma=1;
k=1;
read(X);
while(k<=X) {
        tmp = Mb + Ma;
        Ma=Mb;
        Ma=tmp;
        K=K+1;
}
write(ZMa+Mb);
```

O trecho em Itálico representa o segmento casado.

```
[tmp];
[Mb];
[Ma];
[k];
[X];
Mb=0;
Ma=1;
k=1;
read(X);
while(k<X) {
        tmp = Mb + Ma;
        Ma=Mb;
        Ma=tmp;
        K=K+1;
}
write(ZMa+Mb);
```

Programa após a substituição do segmento equivalente.

Figura 7.35: Usando o padrão de equivalência para corrigir o erro

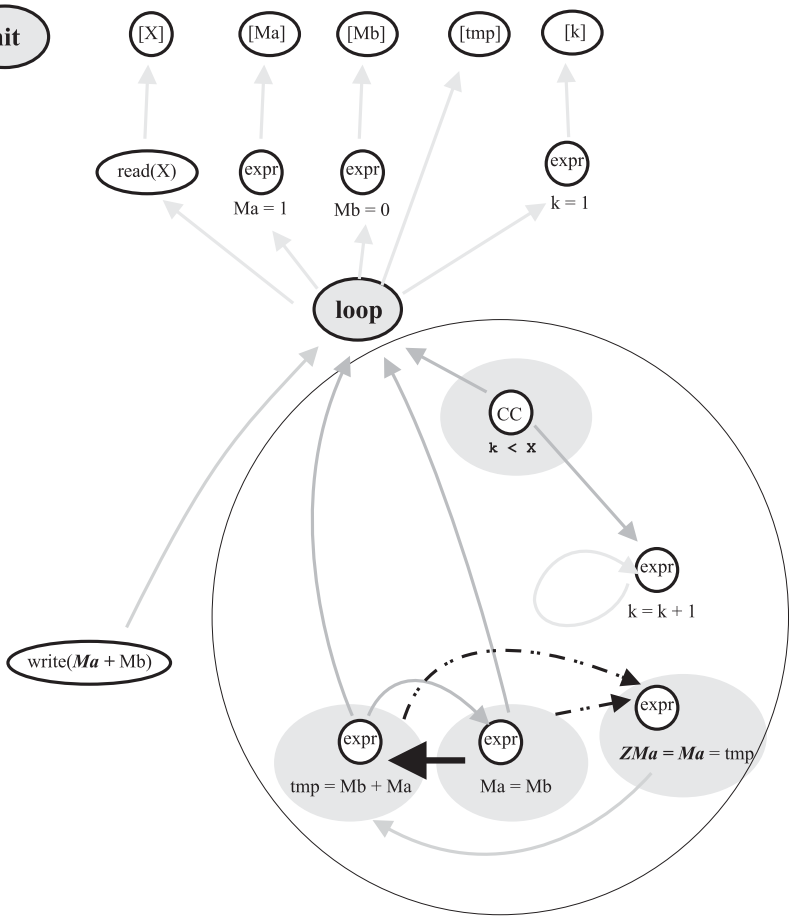


Figura 7.36: Grafo representando a solução do aprendiz após as correções

CAPÍTULO 8

RESULTADOS OBTIDOS E CONTRIBUIÇÕES

Este trabalho procurou estabelecer a arquitetura de duas ferramentas para o ensino de programação, um sistema de autoria e um interpretador tutorial dotado da capacidade de diagnosticar uma solução proposta pelo aluno.

De um ponto de vista geral, um dos resultados obtidos é a arquitetura que envolve as ferramentas de autoria e interpretação tutorial e os mecanismos envolvidos para integração das ferramentas. Entre estes mecanismos, figuram os compiladores de código fonte para GFC e dos padrões na linguagem LAPIDAE para padrões de grafo, ou melhor, para uma linguagem intermediária.

Do ponto de vista de autoria, foram determinadas primitivas de proposição de problemas, na linguagem LAPIDAE e no conjunto de primitivas para auxílio no desenvolvimento de material de curso. Alguns dos aspectos envolvidos e suas vantagens são os seguintes:

- O uso de soluções alternativas é um mecanismo intuitivo e permite que o autor aumente a capacidade de reconhecimento do sistema.
- O uso de uma solução de referência é um mecanismo que não exige nenhuma habilidade além daquelas que um autor de cursos de programação já possuiria, reduzindo a distância semântica no processo de autoria;
- O uso de padrões permite que o autor determine que aspectos considera relevantes explicar em vista de determinados erros. Dessa forma, o autor tem controle sobre os aspectos pedagógicos mais importantes envolvidos, usando um mecanismo que também não foge muito do domínio de conhecimento do autor.

Do ponto de vista de interpretação tutorial, o resultado é uma arquitetura integrando modelos que abrangem desde o pré-processamento dos programas de referência e do aluno, até a identificação de segmentos corretos e incorretos sem deixar de considerar o aspecto de feedback. Ou seja, o resultado consiste na reengenharia dos mecanismos anteriormente desenvolvidos[1] e extensão para incorporar mecanismos complexos de feedback que se integram com a ferramenta de autoria apresentada. As abordagens que surgiram como resultado desse trabalho são as seguintes:

- Normalização através do uso das dependências de dados e controle;
- Detecção de erros por meio de análise das dependências, de particular interesse incluindo a detecção de iterações potencialmente intermináveis;

- Mecanismos para comparação de GFCs, modelando o problema como satisfação de restrição;
- Ampliação da capacidade de comparação através de padrões de equivalência;
- Detecção de erros por meio de heurísticas;
- Detecção de erros e comentários por meio de padrões.

Observe que nem todos os resultados citados são contribuições originais. Em muitos casos, foram aproveitadas idéias de trabalhos anteriores, as quais foram revisadas e estendidas. As contribuições mais relevantes estão na arquitetura de autoria para garantir estensibilidade e flexibilidade, na abordagem para comparação de grafos e na forma de detecção de erros através de padrões.

8.1 Limitações

Como foi mencionado durante todo o trabalho, o problema o qual nos propomos a resolver é intratável no caso geral. Em particular, o sistema é incapaz de comparar soluções muito distintas daquelas incluídas no conjunto de alternativas propostas pelo autor, ainda que a do aluno esteja incorreta.

Mesmo dentro das comparações que o sistema é capaz de fazer, uma vez que o tempo de execução deve ser limitado, não existem garantias de que a análise do programa será completa. A complexidade do problema também limita o tamanho dos programas que podem ser analisados de forma minimamente satisfatória. Uma vez que este trabalho não envolveu experimentação ou simulação exaustiva dos algoritmos, o tamanho dos programas tratáveis ainda não foi determinado.

Outro aspecto limitante é a questão de padrões, tanto na autoria como no interpretador tutorial. Ainda que a linguagem usada para construir padrões seja simples, construir um padrão que seja realmente útil não é uma tarefa trivial e não foram concebidas muitas ferramentas para auxiliar o autor nesse caso. Do ponto de vista do interpretador, um padrão fica limitado à linguagem intermediária de representação de programas, desprovido das vantagens que teria se fosse realmente um padrão representado por um grafo, tal como o uso das várias técnicas utilizadas no processo de comparação.

Por fim, não foram implementados protótipos funcionais para todas as ferramentas e, portanto, a viabilidade das abordagens não foi experimentalmente comprovada.

CAPÍTULO 9

CONSIDERAÇÕES FINAIS

Durante este trabalho apresentamos uma breve revisão crítica dos trabalhos em STI e autoria dedicados ao ensino de programação. Em particular, foram enfocados os trabalhos relativos ao diagnóstico automático de programas, devido à relação com o problema tratado. Ainda, apontamos a inexistência de trabalhos em autoria para STIs de programação.

O objetivo do trabalho consistiu em abordar os problemas de autoria e diagnóstico automático de soluções providas pelo aprendiz. Foi claramente exposta a complexidade do problema de avaliação, em particular na questão de comparação de grafos, demonstrando que o problema no caso geral é simplesmente intratável. Dessa forma, a solução proposta, na forma de arquiteturas, modelos e algoritmos, certamente é severamente restrita em relação ao caso geral. A despeito das limitações dessa solução, está claro que mesmo tal ferramenta tem grande potencial de enriquecer o processo de ensino.

Seja como for, o trabalho atingiu os objetivos de definir um arcabouço para construção de sistemas de diagnóstico e a respectiva ferramenta de autoria, mas ainda está longe de concretizar um STI por completo. Muitos aspectos importantes ainda podem ser explorados e a ferramenta de diagnóstico pode funcionar como base para solução de problemas que vão além da avaliação, desde o modelo do aluno até o núcleo pedagógico do sistema, permitindo usar o feedback como forma de determinar as deficiências do aluno e orientar o ensino para suprí-las.

É importante observar que o resultado mais importante é o arcabouço provido e a reformulação do problema como um todo. Até o presente momento, nem toda a arquitetura foi concretizada na forma de uma implementação, mas as técnicas foram detalhadas e discutidas com profundidade e podem servir de base para novos trabalhos e como contribuição para a área de ensino de programação por meio de sistemas inteligentes.

9.1 Trabalhos Futuros

Para finalizar este trabalho, vale a pena apresentar os trabalhos que serão ou poderão ser desenvolvidos a partir das arquituras e modelos sugeridos. Naturalmente, existe uma gama enorme de caminhos que podem ser seguidos, mas vamos focalizar em três aspectos, implementação e experimentação, estensão da ferramenta de autoria e estensão da ferramenta de interpretação tutorial e módulo de feedback da mesma.

9.1.1 Implementação e Experimentação

O passo mais lógico é completar a construção de um protótipo funcional e avaliar o desempenho da ferramenta. Vale a pena colocar a ferramenta a prova, nas mãos de autores de curso para avaliar a produtividade proporcionada pela ferramenta de autoria. Por outro lado, vale a pena colocar o sistema em campo nas mãos de alunos de cursos iniciais de programação para avaliar não somente os aspectos técnicos da ferramenta, mas principalmente comparar o progresso de alunos apoiados pela ferramenta com outros alunos. Os aspectos da ferramenta que devem avaliados consistem dos seguintes:

- **Autoria**
  - Curva de aprendizado para o uso da ferramenta;
  - Tempo de desenvolvimento de um curso, nesse contexto consistindo de problemas e respectivas soluções e dos padrões de erros e acertos;
- **Interpretação: Aspectos técnicos**
  - Capacidade de reconhecimento de soluções corretas;
  - Corretude dos diagnósticos;
  - Uso dos recursos (equivalência, erro, etc);
  - Qualidade do feedback.
- **Interpretação: Desenvolvimento do Aluno**
  - Curva de aprendizao para uso da ferramenta;
  - Progresso do aluno.

Em muitos casos, a ferramenta certamente será incapaz de identificar o programa inteiro do aluno. Vale a pena introduzir um mecanismo que mantenha um histórico do desempenho da ferramenta, centralizando os resultados de todas as sessões em um servidor central. Nessa caso, uma ferramenta adicional poderia ser criada para que o autor possa analisar o feedback da ferramenta, estendendo o material para cobrir as exceções nas quais houve falha ou ainda fornecendo material para aperfeiçoameno da própria ferramenta.

9.1.2 Estendendo a Autoria

Um importante aspecto a ser desenvolvido na autoria para que esteja ainda mais próxima de uma ferramenta genérica é o desenvolvimento de uma meta-linguagem para especificação da linguagem-objeto a ser utilizada para ensino. Ou seja, a autoria pode ser estendida para permitir a entrada de parâmetros do próprio gerador de grafo, desvinculando a ferramenta de uma linguagem específica.

Em outra direção, a linguagem de autoria pode ser estendida para organizar os problemas em uma hierarquia de classes, arcabouço de grande valia para uma abordagem idático-pedagógica mais elaborada.

Finalmente, outra extensão a ser considerada é a introdução da capacidade de especializar classes de erros, inclusive permitindo que o autor possa associar uma classe de erros a uma classe de problemas ou até mesmo a um problema específico. Dessa forma, os erros podem ser organizados em uma hierarquia, cada qual pode ter associada informações importantes para construção de modelos de aluno, domínio e núcleo pedagógico.

9.1.3 Estendendo o Interpretador

No caso do interpretador existe muito espaço para aperfeiçoamento, desde melhoria nos mecanismos, modelos e heurísticas existentes, até orientar o desenvolvimento para um STI completo. Nesse caso, estaremos nos restringindo apenas nos mecanismos internos e integração com novos mecanismos eventualmente desenvolvidos na área de autoria.

Primeiramente, deve haver trabalho no interpretador como contraparte de enventuais trabalhos em cima da autoria, em particular na questão de hierarquização de problemas e erros. Um trabalho neste caso deve ser orientado de modo a ampliar a arquitetura do interpretador para abranger os novos aspectos e, no mínimo, abrir meios para que novas ferramentas possam ampliar a atuação da ferramenta atual de forma desacoplada.

Na normalização, foi mencionado na Seção 7.3 a manipulação da granulação de escopos. Tal processo seria capaz de reduzir consideravelmente a diferença entre programas e inclusive já foi estudado até um certo ponto, portanto consistindo de um passo lógico em um futuro trabalho. Do ponto de vista de determinação de erros via manipulação heurística, outros erros poderiam ser detectados através de heurísticas mais elaboradas.

O mecanismo de comparação abre muito espaço para trabalhos futuros, sejam teóricos ou mais aplicados. Uma vez que o sistema já envolve o uso de estruturas semelhantes a GFDs, vale a pena aprofundar estudos em GFDs tal como o desenvolvido por Beck *et al*[9]. Beck, em seu trabalho, apresenta técnicas para conversão de GFCs para GFDs, enquanto no presente trabalho apresentamos meios de determinar um grafo semelhante, o GDD.

Outro aspecto que pode ser explorado é o estudo das propriedades de reducibilidade, uma vez que um GFC é um grafo redutível, tal como mencionado no Capítulo 4. Propriedades de redutibilidade são vistas em trabalhos como o de Vernet e Markenzon[39] e Karp[25]. Ainda, quanto às propriedades dos grafos, já foi mencionado no Capítulo 7 como o problema pode ser formulado em parte usando hipergrafos. Dessa forma, um estudo da aplicação de hipergrafos certamente lançaria novas idéias de como aperfeiçoar a representação do problema.

Na questão de padrões de grafos, o sistema poderia ser modificado para utilizar padrões

representados como grafos. Sozinho, esse trabalho é de grande magnitude, não apenas na questão de determinar uma nova representação, mas principalmente porque envolve comparação de grafos também. Nesse caminho, os mecanismos apresentados neste trabalho podem ser bastante úteis para desenvolver tal comparador.

O presente trabalho supõe um feedback muito simples, no qual o sistema percorre os vértices em busca dos erros anotados e apresenta-os na tela, no máximo listando a solução de referência e os trechos incorretos do código do aluno. Em um sistema ideal, mesmo a mera apresentação dos resultados do diagnóstico merece mais atenção e um estudo mais profundo para determinar a forma ideal de apresentação, portanto consistindo em uma das direções que poderia ser tomada a partir dos resultados desse trabalho.

Por fim, outro assunto que poderia até mesmo ser o foco central de um trabalho é a comparação de expressões algébricas. Esse problema, que a primeira vista parece tão simples, é na verdade um grande desafio e não foi resolvido nos trabalhos estudos por este autor. Abordagens interessantes poderiam envolver derivações, manipulações algébricas complexas, decomposição de fatores, entre outros.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] A. Adam e J. Laurent. A system to debug student programs. *Artificial Intelligence*, (15):75–122, 1980.

[2] A. V. Aho, R. Sethi, e J.D.Ulman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[3] J. R. Anderson, editor. *The Architecture of Cognition.* Harvard University Press, 1983.

[4] J. R. Anderson. *Rules of the Mind.* Hillsdale, NJ: Lawrence Eribaum Associates, 1993.

[5] J. R. Anderson, C. F. Boyle, A. T. Corbett, e M. W. Lewis. Cognitive modeling and intelligent tutoring. *Artificial Intelligence*, 42(1):7–50, fevereiro de 1990.

[6] J. R. Anderson, R. Farrel, e R. Sauers. Learning to program in lisp. *Cognitive Science*, (8):87–129, 1984.

[7] J. R. Anderson e B. J. Reiser. The LISP tutor. *BYTE*, 10(4), 1985.

[8] R. Barrett, A. Ramsay, e A. Sloman, editors. *POP-11: a Practical Language for Artificial Intelligence.* Ellis Horwood, 1985.

[9] M. Beck, R. Johnson, e K. Pingali. From control flow to dataflow. Relatório Técnico TR 89-1050, 1989.

[10] S. B. Blessing. A programming by demonstration authoring tool for model-tracing tutors. *International Journal of Artificial Intelligence in Education*, 8:233–261, 1997.

[11] E. Tsang C. Voudouris. Partial constraint satisfaction problems and guided local search. *Proceedings of the Second International Conference on the Practical Application of Constraint Technology (PACT-96)*, páginas 337–356, London, April 24-26 de 1996.

[12] T. H. Cormen, C. E. Leiserson, e R. L. Rivest. *An Introduction to Algorithms.* The MIT Press, 1993.

[13] A. I. Direne. *Intelligent Training Shells for the Operation of Digital Telephony Stations*, páginas 71–78. IOS Press, 1997.

[14] B. du Boulay e C. Sothcott. Computer teaching programming: An introductory survey on the field. R.W Lawler e M. Yazdani, editors, *AI and Education: Learning Environments and Intelligent Tutoring Systems*. Ablex Publishing, 1987.

[15] B. du Boulay e C. Sothcott. Intelligent systems for teaching programming. P. Ercoli e R. Lewis, editors, *Artificial Intelligence Tools in Education*. Elsevier Science Publishers, 1988.

[16] E. R. Gansner e S. C. North. An open graph visualization system and its applications to software engineering. *Software Practice and Experience*, 30(11):1203–1233, Setembro de 2000.

[17] M. R. Garey e D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, New York, 1979.

[18] I. P. Goldstein. Summary of MYCROFT: a system for understanding simple picture programs. *Artificial Intelligence*, (6):249–288, 1975.

[19] A. L. P. Guedes. *Hipergrafos Direcionados*. Tese de Doutorado, Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brasil, 08 de 2001.

[20] T. Hasemer. An empirically-based debugging system for novice programmers. Relatório Técnico 6, Human Cognition Research Laboratory, 10, Open University, 1983.

[21] J. Hopcroft e J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1979.

[22] W. L. Johnson, editor. *Intention-based Diagnosis of Novice Programming Errors*. Research Notes in Artificial Intelligence, Morgan Kaufmann, 1986.

[23] W. L. Johnson e E. Soloway. PROUST: Knowledge-based program understanding. Relatório Técnico 285, Department of Computer Science, Yale University, New Haven, CONN, 1983.

[24] W. L. Johnson e E. Soloway. PROUST. *BYTE*, 10(4), 1985.

[25] R. M. Karp. Reducibility among combinatorial problems. R. E. Miller e J. W. Thatcher, editors, *Complexity of Computer Computations*, páginas 85–103, New York, NY, 1972. Plenum Press.

[26] E. B. Koffman e S. E. Blount. Artificial intelligence and automatic programming in CAI. *Artificial Intelligence*, (6):215–234, 1975.

[27] F. J. Lukey. Understanding and debugging programs. *International Journal Man-Machine Studies*, (12):189–202, 1980.

[28] A. Munro, M. C. Johnson, Q. A. Pizzini, David S. Surmon, D. M. Towne, e J. L. Wogulis. Authoring simulation-centered tutors with RIDES. *International Journal of Artificial Intelligence in Education*, 8:284–316, 1997.

[29] T. Murray. Authoring intelligent tutoring systems: An analysis of the state of the art. *International Journal of Artificial Intelligence in Education*, 10:98–129, 1999.

[30] H. S. Nwana. Intelligent tutoring systems: an overview. *Artificial Intelligence Review*, 4:251–277, 1990.

[31] C. H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[32] S. Papert. *Mindstorms: Childrens, Computers and Powerful Ideas*. Basic Books, New York, 1980.

[33] B. J. Reiser, J. R. Anderson, e R. G. Farrell. Dynamic student modeling in an intelligent tutor for LISP programming. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence Conference*, páginas 8–14, Los Angeles, 1985.

[34] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[35] E. M. Soloway e W. L. Johnson. Remembrance of blunders past: a retrospective on the development of PROUST. *Proceedings of the Sixth Cognitive Science Society Conference*, 1984.

[36] J. L. Szwarcfiter e L. Markenzon. *Estrutura de dados e seus algortimos*. Editora LTC - Livros Técnicos e Científicos, 1994.

[37] E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.

[38] A. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, páginas 230–265, 1936.

[39] O. Vernet e L. Markenzon. Hamiltonian problems for reducible graphs. *17th International Conference of the Chilean Computer Science Society (SCCC '97)*, páginas 264–267, 1997.

[40] E. Wenger. *Artificial Intelligence and Tutoring Systems: Computational and Cognitive Approaches to the Communication of Knowledge*. Morgan Kauffmann, 1987.

GABRIEL DOS SANTOS

AUTORIA E INTERPRETAÇÃO TUTORIAL DE SOLUÇÕES ALTERNATIVAS PARA PROMOVER O ENSINO DE PROGRAMAÇÃO DE COMPUTADORES

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Dr. Alexandre Ibrahim Direne
Co-Orientador: Prof. Dr. André Luiz Pires Guedes

CURITIBA

2003