FÁBIO VINÍCIUS BINDER

CONCEITOS E FERRAMENTAS PARA APOIAR O ENSINO DE LÓGICA DE PROGRAMAÇÃO IMPERATIVA

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Curso de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Alexandre I. Direne

CURITIBA
1999

Ministério da Educação
UNIVERSIDADE FEDERAL DO PARANÁ
SETOR DE CIÊNCIAS EXATAS

PARECER

Nós, abaixo assinados, membros da Comissão Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Fábio Vinícius Binder, avaliamos o trabalho intitulado **"Conceitos e Ferramentas para Apoiar o Ensino de Lógica de Programação Imperativa"**, cuja defesa foi realizada no dia 14 de abril de 1999. Após a Avaliação, decidimos pela Aprovação do Candidato.

Curitiba, 14 de abril de 1999.

Prof. Dr. Alexandre Ibrahim Direne
Presidente

Profª. Drª. Maria das Graças Volpe Nunes

Profª. Drª. Aurora Trinidad Ramirez Pozo

AGRADECIMENTOS

Ao meu orientador Alexandre, pela seriedade, dedicação e companheirismo.

Aos professores que colaboraram para a realização deste trabalho: Bruno, Patrícia e Francisco.

A toda a minha família pelo carinho, apoio e compreensão.

SUMÁRIO

LISTA DE ILUSTRAÇÕES

RESUMO

Este trabalho apresenta o sistema tutor *Asimov* que é composto de um conjunto de ferramentas utilizadas no apoio ao ensino de programação com linguagens imperativas. Estas ferramentas oferecem um rico ambiente de ensino que pode ser explorado livremente pelo aluno através da execução compassada de sua solução com visualização de variáveis. O sistema também fornece, caso solicitado, apoio tutorial sempre que for cometido algum tipo de erro de lógica. O sistema possui diversas características flexíveis que possibilitam o seu uso em variadas situações de ensino/aprendizagem, destacando-se a escolha de diversos graus de granulação do *feedback* tutorial. Sua independência de domínio permite que um autor altere tanto a sintaxe da linguagem-alvo quanto os enunciados de exercícios propostos para os alunos. Através de um modelo que representa a integração de diversas soluções possíveis (grafo de alternativas e dependências - GRADE) o autor pode limitar as soluções aceitas como corretas e guiar o aluno durante a resolução de problemas dentro das variações desejadas.

ABSTRACT

This work presents the tutoring system *Asimov* that is composed by a set of tools used to support the learning of programming concepts. The tools offer a rich environment that allows the students to freely explore their own solutions. Alternatively, students can be guided by comparisons with the system's solution, whenever logical errors occur. The system is flexible in several manners, for instance, the teacher or the student can change the granularity of the tutorial feedback. Also, the author can change the target-language syntax being taught as well as insert new exercises with solutions to be proposed to the learner. Author can model solutions through a graph of alternatives and dependencies called GRADE. This technique allows the integration of several solutions in one with a reasonable degree of variation while preventing the student from making mistakes or taking erroneous paths.

1) INTRODUÇÃO

Este trabalho expressa a concepção, projeto e implementação do sistema tutor *Asimov* que é composto de um conjunto de ferramentas utilizadas no apoio ao ensino de programação com linguagens imperativas. Os principais objetivos deste sistema são: 1) facilitar a aquisição de princípios de programação, disponibilizando um ambiente de exploração livre com compassamento de execução associado à visualização de células de memória e um ambiente guiado; 2) facilitar a aquisição inicial de perícia em programação por meio da identificação e explicação de erros de lógica encontrados nos programas de alunos iniciantes e 3) através de recursos de autoria e da independência de domínio, possibilitar a alteração de diversas características de conteúdo (como gramática da linguagem-alvo e enunciados de problemas propostos) e o seu uso em variadas situações de aprendizagem.

1.1) O ensino de programação de computadores para iniciantes

Qualquer sistema de computador que tenha como objetivo principal, o apoio ao ensino de programação deve levar em conta que um aluno, para ser considerado apto e capaz de resolver problemas algorítmicos de maneira satisfatória, deve adquirir diversas habilidades (figura 1) que podem ser divididas em dois aspectos de programação: princípio e perícia (du BOULAY; SOTHCOTT, 1987). O princípio de programação representa o

conhecimento da sintaxe (como um comando deve ser escrito) e da semântica (como um comando funciona) individual dos comandos e da estrutura da linguagem que está sendo estudada. A perícia engloba as habilidades de decomposição de problemas e composição de soluções para estes problemas através do agrupamento dos comandos da linguagem em uma determinada ordem para formar planos de alto nível (algoritmos).

As habilidades que um aluno iniciante deve adquirir para poder ser considerado um programador perito estão identificadas na Figura 1 (PIMENTEL; DIRENE, 1998). Este conjunto de características nos mostra a grande quantidade de conhecimento que um programador deve possuir para poder programar bem. Portanto, não deve ser surpresa para ninguém, a dificuldade que alunos iniciantes encontram para assimilar esta aptidão.

Figura 1 – Características do estereótipo de um programador perito

Precisão sintática
Precisão semântica
Identificação de estruturas principais no programa fonte (busca por palavra-chave)
Simulação mental dos estados do computador durante a execução
Catálogo de erros
Mapear mentalmente as estruturas do programa
Checagem de pré-condições
Análise do problema
Integração dos subproblemas
Generalização da solução
Reutilização de soluções já conhecidas
Catálogo de soluções

Atualmente, o ensino de programação utilizando linguagens formais imperativas ainda é feito com pouco ou nenhum apoio do computador. Na maioria das vezes, quando

existe tal apoio, ele resume-se ao compilador da linguagem que está sendo estudada. Mesmo que este compilador esteja inserido em um ambiente de programação amigável ele não é capaz de guiar o aluno através de uma sessão dedicada à solução de um problema. Está claro que um sistema de computador não pode insidir sobre todos os aspectos inerentes ao ensino/aprendizagem de programação, mas ele poderia atenuar as dificuldades do processo de aprendizagem de programação se utilizado de maneiras variadas, de acordo com cada contexto de ensino.

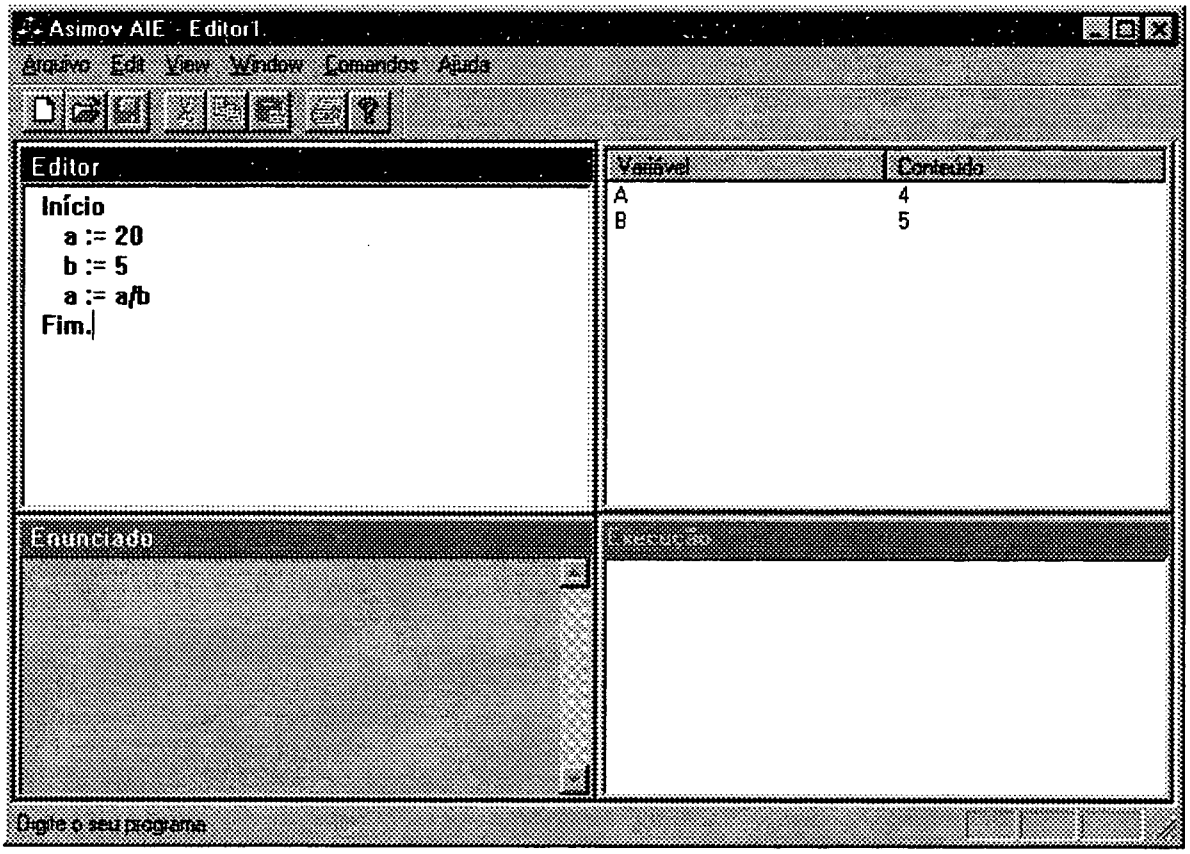
Como a programação não é uma maneira natural de expressão textual dos humanos, os alunos sentem enormes dificuldades em escrever, mesmo através de uma pseudo-linguagem, a solução para um problema de programação, ainda que já o tenham resolvido mentalmente. Erros em programas de alunos novatos podem ser ocasionados por vários motivos, entre eles, uso de condicionais ao invés de interação, deficiências totais parciais no entendimento de conceitos de entrada e saída, variável e processo de atribuição de valores, integração correta de comandos e finalmente, dificuldade em acompanhar o próprio fluxo de controle do programa (BONAR; SOLOWAY, 1985 & du BOULAY; SOTHCOTT, 1987).

1.2) Visão resumida do sistema Asimov

O sistema *Asimov* foi construído para apoiar o ensino de programação, ajudando na aquisição tanto de princípio quanto de perícia. Ele possui diversas características flexíveis para permitir o uso por diferentes perfis de aluno, de professor e também de currículo. A independência de domínio é a principal destas características e torna o sistema uma *shell*

para a montagem de sessões para exercitar problemas de programação. O sistema pode ser operado em dois modos: modo livre e modo guiado. No modo livre o aluno tem ao seu dispor um interpretador de programas com compassamento e um visualizador do estado das variáveis durante a execução do programa (Figura 2). No modo guiado, além das ferramentas acima, é ativado o diagnosticador automático, que procura identificar possíveis erros de lógica na solução do aluno.

Figura 2 – Visualização de células de memória no Sistema ASIMOV



A aquisição de princípio é auxiliada através da execução do algoritmo desenvolvido dentro do próprio sistema. Esta execução possibilita ao aluno a visualização das alterações das células de memória que representam as variáveis bem como o entendimento do fluxo de controle do programa através do compassamento da solução. Com relação à sintaxe da

linguagem, foram simplificadas diversas construções e incluídos alguns aspectos de facilidade (uso de sinônimos de comandos) para evitar que o aluno fosse desviado do objetivo principal do problema, que é a lógica do programa.

A perícia de programação é desenvolvida através do monitoramento variável da solução do aluno à medida em que ele vai digitando o seu programa no sistema. As mensagens de indicação de erros de lógica são apresentadas ao aluno de acordo com sua escolha (ou do professor). Esta flexibilidade é chamada de granulação do *feedback*, indicando o momento em que o sistema deve fornecer apoio tutorial. Este acompanhamento de solução (*model-answer*) é realizado através de um diagnosticador automático de erros que utiliza uma solução pré-cadastrada (solução de referência) no sistema pelo autor como padrão de comparação com a solução do aluno.

O sistema tem a característica de ser uma *shell*, pois permite a alteração de diversos aspectos, principalmente relacionados com o domínio que se pretende abranger. O professor pode alterar a linguagem que está sendo ensinada, através do desenvolvimento de uma nova gramática (**ver** Anexo I) para o sistema. Entretanto, o principal aspecto de variação de domínio é a possibilidade de fornecer ao aplicativo enunciados de exercícios diferentes e suas respectivas soluções para serem apresentados ao alunos que utilizarem o sistema. Maiores detalhes sobre a variação de domínio e representação de soluções serão apresentados nos capítulos 3 e 4.

1.3) Vantagens instrucionais

O ensino de programação é uma tarefa bastante complexa, exigindo um grande esforço tanto do professor quanto do aluno. São diversas habilidades a serem alcançadas e um quantidade significativa de conhecimento a ser adquirido. Em uma tentativa para amenizar estas dificuldades, muitos sistemas tutores inteligentes (STI) e micromundos de exploração tem sido desenvolvidos.

Os STI para apoio ao ensino de programação concentram-se, principalmente, na tarefa de identificar erros de lógica de um programa (**ver** cap. 2), através de diagnóstico automático. Estes sistemas procuram auxiliar o aluno na aquisição de perícia de programação. Entretanto, o uso de um STI isolado não resolve diversos problemas de aprendizagem e entre eles, a aquisição do princípio de programação. Mesmo que o aluno tenha recebido uma instrução sobre sintaxe e semântica de comandos antes de usar o sistema, ele necessita de outros meios didáticos para assimilar completamente esta instrução.

Os sistemas de micromundos de exploração ao contrário dos STI, permitem uma utilização livre do ambiente, ou seja, o aluno tem liberdade em explorar o seu programa, através do compassamento de soluções e da visualização de células de memória que indicam o estado das variáveis em determinado momento. O uso de micromundos de exploração pode oferecer ricas oportunidades de contexto de aprendizagem, principalmente com relação à maneira em que um programa funciona.

A ligação entre as fortes características guiadas de um tutor com a exploração livre em apenas um ambiente de descoberta guiada (WENGER, 1987) pode ser eficaz, já que reúne o auxílio a aquisição de princípio e perícia em um único sistema. O sistema *Asimov* possui tanto o modo de exploração livre, onde o aluno pode digitar a sua solução e utilizar um compassador de execução e um visualizador de variáveis quanto o modo de exploração guiado, onde mensagens tutoriais são fornecidas pelo diagnosticador automático caso o aprendiz cometa algum erro de lógica.

1.4) Esquema da dissertação

A dissertação foi dividida, contando com esta introdução, em 6 capítulos. No capítulo 2 são apresentados e analisados alguns sistemas tutores inteligentes para o apoio ao ensino de programação e sistemas de autoria. A primeira parte traz uma breve explicação sobre diagnóstico automático de programas de alunos e o capítulo segue, com o estudo de sistemas divididos em duas categorias: sistemas com análise pós-fixada ou posterior ao evento e com análise in-fixada, ou durante o evento. A última parte do capítulo traz uma análise de alguns sistemas de autoria e *shells* para a construção de sistemas tutores inteligentes.

O sistema *Asimov* é apresentado no capítulo 3. São explicadas as considerações que foram feitas em seu projeto: simplicidade da pseudo-linguagem de programação, uso de granulação de *feedbak* para contemplar diversos perfis de alunos e ensino de programação baseada em exemplos. O capítulo segue com a visão panorâmica do sistema *Asimov*, onde

são discutidos os seguintes tópicos: a interface utilizada pelo sistema, com sua divisão em quatro áreas de trabalho, a gramática da linguagem utilizada e uma explicação do funcionamento da programação guiada. A apresentação do sistema é finalizada com uma passada pelos cinco componentes de sua arquitetura: interface, compilador, seletor do modo de interação, interpretador e diagnosticador automático.

A representação da solução de referência e o funcionamento do diagnóstico automático do sistema são explanados no capítulo 4. Primeiro é explicado como a solução de referência é representada através de redes semânticas e como esta representação auxilia no processo de capturar variações em programas de alunos iniciantes. Em seguida, é revelado como os identificadores de erros funcionam e são apresentados alguns cenários de uso do sistema. Ao final do capítulo, são identificadas as principais limitações do sistema *Asimov* em relação ao diagnóstico automático.

No capítulo 5, é descrita uma avaliação empírica do sistema que foi realizada com dois especialistas na área de ensino de programação. São apresentados os enunciados dos exercícios propostos, a formação dos especialistas, os modelos de solução de referência criados pelos especialistas, a execução do sistema *Asimov* baseada nos modelos criados e, por fim, são discutidos os pontos principais desta avaliação.

Finalmente, no capítulo 6, as principais contribuições do trabalho são enfatizadas e são apontadas algumas direções para possíveis trabalhos futuros na área de apoio ao ensino de programação.

2) SISTEMAS DE APOIO AO ENSINO DE PROGRAMAÇÃO PARA INICIANTES

2.1) Diagnóstico automático de programas de alunos.

Um dos principais objetivos de um sistema tutorial inteligente para programação é apoiar o aluno durante a resolução de exercícios. Para atingir tal objetivo o sistema deve ter a capacidade de interagir com o aluno, analisando o programa, identificando erros e sugerindo novos rumos. Esta tarefa é realizada através de um diagnóstico automático do programa do aluno, que é uma forma de imitar o comportamento de um tutor humano que analisa o programa e sugere as correções ao aprendiz. O diagnóstico automático pode ser classificado, quanto ao momento de atuação, em dois tipos: ativo e passivo. Um sistema que possua um diagnóstico ativo está constantemente monitorando o comportamento do aluno, fornecendo *feedback* quando necessário. O diagnóstico passivo indica que o sistema deve receber um comando do aluno para iniciar o processo de análise. Como veremos a seguir, a forma que tem sido mais utilizada nos sistemas tutores recentes é o diagnóstico ativo, pois este evita que o aluno se desvie demasiadamente de alguma solução correta.

A construção de um sistema de diagnóstico automático não é tarefa simples, pois o processo de diagnóstico envolve diversos fatores, tais como: os objetivos do problema a ser resolvido, características internas da linguagem de programação usada, erros mais comuns encontrados, uso de comandos ou estruturas de dados semelhantes e soluções corretas mas totalmente diferentes da solução tomada como referência. Mesmo levando-se em

consideração todos estes fatores, os sistemas atuais não são totalmente precisos, ainda que sejam forçadas algumas reduções do escopo da linguagem, das classes de erros ou do conjunto de problemas propostos. Como qualquer área que envolva a representação do conhecimento humano no computador e sua posterior utilização, a modelagem de um sistema deste tipo requer métodos avançados e, muitas vezes, complexos. Existem, atualmente, diversas maneiras de se realizar o diagnóstico automático de programas, tais como: comparação com uma ou mais soluções de referência, aplicação de regras de construção corretas e incorretas e pela comparação das entradas e saídas do programa com os resultados esperados.

2.2) Resumo de abordagens de métodos de diagnósticos

As abordagens utilizadas no diagnóstico automático de programas em sistemas tutores inteligentes foram quase sempre baseadas em métodos de ensino bem específicos. Como existem diversas maneiras de se ensinar a programar, os sistemas tutores já desenvolvidos refletiram esta variedade em sua implementação. Apesar da grande variedade de métodos, podemos classificar os sistemas tutores para apoio a programação com diagnóstico automático em dois grupos: sistemas com análise pós-fixada ou posterior ao evento e com análise in-fixada, ou durante o evento.

2.3) Sistemas de análise *pós-fixada*

Esta categoria apresenta sistemas cujo diagnóstico automático é ativado após o evento, ou seja, após o aluno terminar de especificar a sua solução para o problema proposto. Os tipos de análise utilizados nesta classificação podem ser divididos em: *model-answer* ou análise por comparação com solução de referência, *specification-analysis* ou análise por especificação e *i/o- analysis* ou análise de entrada/saída.

2.3.1) Análise por "model-answer"

O diagnóstico é realizado através da comparação da solução apresentada pelo aluno com a solução do sistema, também chamada de solução de referência. Um erro é encontrado quando as soluções forem diferentes após um determinado número de transformações. As dificuldades com este tipo de diagnóstico são a falta de conhecimento dos objetivos do exercício por parte do sistema e a sua incapacidade em indicar onde os erros estão exatamente e porque eles foram cometidos.

Um dos principais representantes desta categoria é o sistema *Laura* (ADAM, A.; LAWRENT, J. 1980), utilizado no apoio ao ensino da linguagem *Fortran*. Inicialmente o sistema transforma as duas soluções em grafos, o que facilita a comparação. Em seguida, os grafos são reduzidos através da eliminação de variáveis redundantes e da separação de comandos independentes existentes no interior de uma repetição em várias repetições separadas. O passo final é a comparação dos elementos dos dois grafos gerados. Se os

grafos não puderem ser comparados, então existem um ou mais erros na solução do aluno. Estes erros são informados com base em um conjunto de erros pré-identificados. Se isto não for possível o sistema simplesmente informa a parte do grafo da solução do aluno que falhou na comparação e a parte correspondente na solução de referência. Apesar deste sistema falhar se a solução do aluno estiver correta mas diferente da solução de referência, ele apresenta uma vantagem significativa: a solução de referência é representada na própria linguagem-alvo, o que facilita o uso do sistema pelo professor.

2.3.2) Análise por especificação

Sistemas cujo diagnóstico atua através da análise por especificação são aqueles que possuem uma descrição de alto nível dos objetivos do problema, sendo capazes de emitir mensagens de erro mais significativas do que os outros sistemas, bem como relacionar os erros encontrados na solução do aluno com trechos do enunciado do exercício. O sistema faz uma checagem, verificando se a solução do aluno está compatível com a descrição dos objetivos. Os melhores exemplos de sistemas que utilizam este tipo de análise são: *Mycroft* (GOLDSTEIN, I. P., 1975), *Pudsy* (LUKEY, F. J., 1980), *Aurac* (HASEMER, T, 1983) e *Proust* (JOHNSON, W. L. ; SOLOWAY, E., 1987).

O sistema *Mycroft* pode detectar e corrigir erros em programas *Logo*. Devido à natureza da linguagem e ao fato dos problemas de programação estarem associados a desenho de linhas, é possível representar a base de conhecimentos como uma série de propriedades geométricas. A base de conhecimentos é construída com uma linguagem específica denominada de *assertion language* (linguagem de asserção). Esta linguagem

permite a criação de formas complexas baseadas em linhas. Para que o sistema funcione corretamente o aluno deve dividir o seu programa em procedimentos que estejam relacionados com as partes da figura a serem desenhadas. Na fase de diagnóstico são levados em consideração para a modelagem dos planos do programa apenas os comandos que alterem a posição da tartaruga (o cursor gráfico do ambiente *Logo*). Se os planos da solução do aluno não estiverem corretos, o sistema sugere modificações para que o desenho seja impresso corretamente.

Eliminar uma limitada classe de erros de pequenos programas codificados em *Pascal* é o principal objetivo do sistema *Pudsy*. O sistema divide a solução do aluno em sub-tarefas, tendo como base um conjunto de sub-tarefas pré cadastradas relacionadas com a descrição do problema e, em seguida, identifica como a informação flui de um trecho de código para o outro. Neste primeiro passo, podem ser identificados erros de redundância de comandos e uso de nomes de variáveis inadequadas. No segundo passo são comparados os valores de variáveis ao final das sub-tarefas com a especificação do problema. Se o resultado da comparação for negativo o sistema emite uma mensagem de erro e sugere uma possível correção através de um método de gerar-e-testar. Alguns problemas deste sistema, identificados pelo próprio autor são: falta de flexibilidade e incapacidade em usar o comportamento a partir da execução do programa.

O objetivo do sistema *Aurac* é depurar programas codificados na linguagem imperativa/procedimental *Solo* para manipulação de bases de dados de relações e utilizada para ensinar modelagem computacional para programadores novatos. O sistema funciona em três fases: na primeira são utilizadas 12 regras de produção de erros para a identificação de

erros sintáticos de alto-nível; na segunda etapa, segmentos do programa são comparados com uma biblioteca de clichês (trechos de código comuns em programas de iniciantes); na fase final é feita uma análise do fluxo de dados que detecta uma limitada classe de algoritmos e erros, tais como uma variável definida mas nunca usada. Como a linguagem utilizada é bastante simples e limitada, o sistema não necessita de um detalhado conhecimento dos objetivos do programa como os outros dois sistemas desta seção.

Segundo seus autores, o sistema *PROUST* foi projetado para encontrar todo erro de lógica existente em programas já compilados de programadores iniciantes. Ele é capaz de realizar esta tarefa, pois tem conhecimento da essência do problema, ou seja, ele sabe o que o programa deverá fazer quando for executado e mais: sabe também como resolver o problema proposto de várias maneiras. Esta abordagem de diagnóstico permite que o sistema avalie melhor a solução do aluno do que, por exemplo, uma análise de entradas e saídas, fluxo de dados e comparação com uma biblioteca de erros mais comuns.

O sistema possui um conjunto de descrições de problemas, sendo que estas descrições são relacionadas com os requisitos ou objetivos que devem ser satisfeitos. Quando um programa está sendo analisado, o sistema usa a descrição dos objetivos e seu conhecimento de programação para inferir determinadas hipóteses sobre como o problema pode ser resolvido. Se nenhuma hipótese combinar com a solução do aluno, uma base de dados contendo os erros mais comuns é acionada para tentar explicar as discrepâncias.

Este sistema é acionado após a compilação com sucesso de um programa em um compilador *Pascal*, ou seja, ele não encontra erros sintáticos nem é capaz de avaliar soluções parciais dos alunos. Da mesma maneira, o sistema não executa o programa nem

mostra o estado de células de memória durante esta execução. A incapacidade em avaliar soluções parciais permite que os alunos possam se desviar completamente de uma solução possível para o problema, embora permitam uma livre exploração do programa pelo estudante. A falta de um ambiente de visualização do funcionamento de um programa torna obscuro o modelo mental que um aprendiz constrói sobre como as coisas realmente acontecem quando um programa é executado.

2.3.3) Análise de entrada/saída

Uma maneira menos trabalhosa de se implementar o diagnóstico automático de programas é tentar imitar o papel de uma pessoa que não sabe como o sistema foi construído mas conhece bem a área em que o sistema atua: a análise de entrada/saída. Esta análise é feita através da execução da solução do aluno com determinados valores de entrada e a comparação dos valores de saída gerados com os resultados esperados de um programa correto. Os problemas com este tipo de análise são facilmente identificáveis: dificuldade em se definir quais conjuntos de valores de entrada poderão gerar erros, impossibilidade em se afirmar que um programa que apresenta os resultados corretos está realmente correto e falta de conhecimento do próprio código da solução do aluno.

A dificuldade encontrada na definição dos conjuntos de valores de entrada ocorre porque existem determinados valores que podem gerar erros em situações bastante específicas da execução do programa. Somente um teste exaustivo com um conjunto significativo de valores pode levar a conclusão de que determinada solução está realmente correta, já que muitas vezes um programa tem o comportamento esperado com certos

valores e com outros não. Além disso, a falta de conhecimento do próprio código do aluno, impossibilita tanto a determinação exata do erro do aluno, quanto a certeza de que o aprendiz programou a solução de acordo com o esperado. Por exemplo, para calcular a soma dos n primeiros números inteiros pode-se utilizar um laço com contador e acúmulo dos valores ou então lançar mão da fórmula de progressão aritmética. A primeira solução demonstra uma compreensão de diversos aspectos básicos de programação enquanto a segunda nos mostra que o aluno tem uma boa memória e talvez um bom conhecimento de matemática.

O representante mais significativo desta categoria é o sistema *Bip* (BARR, A.; BEARD, M.; ATKINSON, R. C., 1976), cuja linguagem-alvo é a *Basic*. Junto com a análise de entradas/saídas também possui um interpretador com diversas facilidades gráficas, conjunto de problemas de programação e um mecanismo de seleção de problemas que baseia-se no desempenho do aluno, no conjunto de problemas e no conjunto de habilidades que o aprendiz pretende adquirir. Para minimizar os problemas encontrados na análise de entradas/saídas o sistema possui um módulo rudimentar e bastante limitado para a identificação de erros de lógica.

2.4) Sistemas de análise *in-fixada*

Os sistemas de análise durante o evento, são aqueles que interagem com o aprendiz durante a resolução do problema, fornecendo *feedback* em caso de erro e, em alguns sistemas, no caso de acerto também. O diagnóstico pode ocorrer a pedido do aluno, sendo

chamado neste caso de diagnóstico passivo ou durante a resolução do problema, também chamado de diagnóstico ativo ou reativo. Independentemente da forma como o diagnóstico é realizado, estes sistemas podem ser classificados em dois grandes grupos: *model-tracing* e *model-answer*.

Na primeira categoria (*model-tracing*) enquadram-se sistemas que possuem como principal característica, o diagnóstico de programas através da utilização de regras de produção. Estes sistemas tem a capacidade de, efetivamente, *entender* o enunciado proposto e *apresentar* uma solução correta para o problema. Entre os sistemas tutores desta categoria, temos: *Greaterp* (du BOULAY, B.; SOTHCOTT, C., 1987, p. 348-350), um sistema tutorial para o ensino da linguagem LISP e *Gil* (REISER, B.; KIMBERG, D.; RANNEY M., 1988), que acrescentou uma interface gráfica e a capacidade de visualização ao sistema *Greaterp*.

Na segunda categoria (*model-answer*) temos os sistemas cujo diagnóstico é realizado com base em uma solução de referência inserida no sistema antes do início da sessão de tutoramento. Estes sistemas não entendem o enunciado do problema e também não são capazes de apresentar qualquer tipo de solução, correta ou errada. Os sistemas mais significativos que se encaixam nesta categoria são: *BRIDGE* (BONAR; CUNNINGHAM, 1985, cap. 15) para o ensino de programação Pascal através do método SSNLP, *SPADE* (du BOULAY, B.; SOTHCOTT, C., 1987, p. 359-361) utilizado para o ensino de programação através da linguagem Logo e *DISCOVER* (RAMADHAN; du BOULAY, 1993, p. 125-134) que auxilia o ensino de programação através de uma linguagem própria, semelhante ao pseudo-código.

2.4.1) GREATERP/GIL

Estes sistemas guiam os aprendizes na resolução de problemas de programação dos primeiros capítulos de um curso de LISP. O sistema *GREATERP* (du BOULAY, B.; SOTHCOTT, C., 1987, p. 348-350) segue a teoria de que um erro detectado em um programa de um aluno deve ser corrigido no mesmo momento. Ele cumpre esta tarefa através do uso de regras de produção para expressar habilidades de especialistas e de novatos. Devido a esta capacidade o sistema pode apresentar suas próprias soluções para os problemas e também fornecer comentários mais detalhados sobre um erro encontrado em um programa de um aluno.

O sistema *GREATERP* pode também gerar problemas para o aluno resolver. Ao mesmo tempo em que o aluno digita sua solução, o sistema tenta verificar se cada novo símbolo inserido está correto ou errado para o problema proposto. Um símbolo está correto se casa com uma regra de produção de especialista e está errado se casa com uma regra de produção de novato. Se não for encontrada nenhuma regra compatível, o sistema intervém, apresentando alternativas ao aluno e obrigando-o a escolher uma delas. As regras para novatos representam os erros mais comuns encontrados em programas de alunos iniciantes.

Embora o sistema *Greaterp* seja realmente poderoso, ele carece de um ambiente mais adequado ao entendimento de como funciona um programa quando executado dentro do computador. Esta ausência foi em parte suprida pelo sistema *Gil* (REISER, B.; KIMBERG, D.; RANNEY M., 1988), que adicionou a capacidade de visualização de células de memória durante a execução do programa. A capacidade avançada de diagnóstico de

programas apresentada, deve-se principalmente a dois fatores: relativa simplicidade dos exercícios iniciais de um curso de *Lisp* e a própria estrutura da linguagem. Outro ponto a ser destacado é que a base de exercícios apresentada é fixa, não havendo nenhuma indicação de que haja alguma maneira simples de se adicionar outros problemas e exemplos dentro do sistema.

2.4.2) BRIDGE

Este STI apresenta-se como um sistema tutorial completo para o ensino de programação a alunos novatos (BONAR; CUNNINGHAM, 1985, cap. 15). O seu nome indica que ele pretende ser uma *ponte* entre o conhecimento de programação e o aprendiz. É um sistema bastante rígido, tanto em sua concepção quanto em sua interface, obrigando o aluno a solucionar o problema seguindo três etapas bem distintas:

- Identificação e refinamento dos planos para a solução do problema apresentado, através do uso de frases em linguagem natural.
- Associação dos planos descritos em linguagem natural com planos descritos em uma linguagem mais organizada e próxima do computador.
- Tradução da fase anterior para um programa em Pascal.

Estas etapas levam em consideração que o conhecimento de programação não é natural e, portanto, o aluno deve ter a possibilidade de iniciar a construção de seu programa com frases em inglês (ou em seu idioma). A turma de alunos que foi estudada para o

desenvolvimento deste sistema já possuía um prévio conhecimento da descrição de problemas do cotidiano usando uma abordagem passo-a-passo conhecida como *SSNLP(step-by-step informal language procedures)*. Esta abordagem sugere que os programas devem ser desenvolvidos de acordo com as etapas identificadas anteriormente.

Em cada uma das etapas, o aluno pode solicitar uma análise de sua solução, mesmo que ela não esteja completa. Esta avaliação verifica se a solução do aluno está correta e indica discrepâncias em relação aos resultados esperados. As respostas do sistema são fornecidas a partir da identificação do estado atual de aprendizado do aluno, ou seja, apenas mensagens que tenham importância para o conhecimento corrente do aprendiz são apresentadas. A tarefa de verificar uma solução parcial não é difícil para o sistema Bridge por dois motivos principais: a) o aluno é constantemente guiado e sempre deve escolher uma frase ou um comando de um menu de opções e posicioná-lo no local correto e b) o aluno apenas converte um determinado item da representação de uma etapa para um ou mais itens na representação da etapa seguinte.

Portanto, deve-se destacar que, embora este seja um dos sistemas tutores mais completos já desenvolvidos, existem aspectos deste aplicativo que podem inviabilizar o seu uso efetivo no ensino de programação. O primeiro aspecto diz respeito ao método utilizado para ensinar programação: os estudantes devem ter familiaridade com a identificação de objetivos e planos em um enunciado proposto. O sistema não permite que qualquer programador novato usufrua de seus ensinamentos, apenas aqueles que compreendam determinado método de trabalho (SSNLP) podem utilizá-lo. Outro aspecto a ser considerado é o ambiente de programação guiado, inadequado para uma exploração livre.

Esta limitação restringe o aluno a uma identificação dos planos pré-definidos existentes em cada etapa, inibindo a criação de soluções diferentes. Uma restrição menos significativa é que a única linguagem disponível para a solução final é a *Pascal*, embora os próprios autores indiquem que um dos melhoramentos seja o de se disponibilizar outras linguagens semelhantes.

2.4.3) DISCOVER

O *DISCOVER* (RAMADHAN; du BOULAY, 1993, p. 125-134) é um sistema tutor cujo objetivo é apoiar o ensino de programação básica, através de uma linguagem própria semelhante ao pseudo-código. Foi implementado como a união de algumas técnicas computacionais: visualização de domínios, micro-mundos, diagnóstico automático ativo e sistemas tutores inteligentes. Um aspecto diferencial deste sistema é que ele integra tanto a exploração livre de um programa, quanto o ensino guiado. Esta integração permite que o aluno beneficie-se das vantagens de ambos os métodos de ensino. Estes métodos são representados em dois módulos ou fases: fase livre, na qual o aluno explora o formato de execução de um programa através da visualização de células de memória, execução compassada e explicação de conceitos básicos, como: criação de variáveis e leitura e impressão de valores; fase guiada, na qual o aluno é convidado a resolver um determinado problema de programação, tendo como apoio, além dos componentes da fase livre, *feedback* em caso de sucesso ou erro e exemplos resolvidos de problemas semelhantes.

A fase livre é utilizada, basicamente, para facilitar o entendimento de como um programa funciona quando executado no computador. Através da exploração deste módulo,

o aluno é capaz de construir um modelo mental claro e robusto sobre o comportamento de conceitos de programação, funcionamento do programa como um todo e características inerentes à máquina. O aprendiz escolhe conceitos de um conjunto e completa-os com nomes de variáveis e expressões. No momento em que um conceito é finalizado o sistema executa-o, indicando dinamicamente o comportamento do conceito escolhido. Deve-se ressaltar que nesta fase o sistema não fornece qualquer *feedback* tutorial sobre a lógica do programa, apenas para o seu funcionamento.

A fase guiada tem como objetivo, "...ensinar o novato a compor e ajustar conceitos de programação e comandos para resolver determinados problemas de programação..." (RAMADHAN; du BOULAY, 1993, p. 127). Nesta fase o sistema escolhe um problema seguindo uma ordem pré-determinada e apresenta-o ao aluno. À medida que o estudante vai compondo os conceitos o sistema compara-o com uma solução pré-determinada e indica se a composição está certa ou errada. Através deste *feedback* o aluno é guiado e não se afasta demasiadamente da solução final.

Segundo experiências empíricas realizadas pelos autores, o *DISCOVER* obteve resultados bastante significativos em relação a dois aspectos do ensino de programação: quantidade de erros produzidos pelos alunos e tempo necessário para a conclusão de exercícios básicos. Nestas experiências foi observado que os alunos que iniciaram os estudos pela fase livre, apresentaram rendimento superior ao dos alunos que iniciaram o estudo pela fase guiada. A explicação pode residir no fato de que a fase livre é propícia para o entendimento de como um programa funciona dentro do computador.

Observando-se os sistemas até aqui analisados, percebe-se que, em todos eles, a principal preocupação foi com o modelo de tutoramento: como guiar o aluno, em que momento apresentar *feedback*, qual método de ensino utilizar e qual a representação interna das soluções. Nenhum deles apresenta de forma concreta uma generalização de seu funcionamento de forma a permitir ao professor, a inclusão de novos exemplos, enunciados de problemas e respectivas soluções.

2.5) Sistemas de autoria e *shells* para ensino/aprendizagem.

Os principais problemas que fazem com que os STI, de maneira geral, não sejam amplamente divulgados e utilizados em escolas são a dificuldade do seu desenvolvimento e, por conseguinte, o alto custo de sua produção (NICOLSON, R. I.; SCOTT, P. J., 1986; MURRAY, T., 1997; MAJOR, N. *et al.*, 1997). Além da demora usual de seu projeto e programação, temos ainda a grande complexidade da representação do conhecimento de um especialista em alguma forma que possa ser implementada em um sistema de computação. Estes argumentos valem para qualquer tipo de STI, incluindo os STI de apoio ao ensino de programação. A abordagem utilizada para amenizar estes problemas é o desenvolvimento de sistemas com independência de domínio conhecidos como sistemas de autoria e *shells*. Eles possibilitam a um usuário comum, respectivamente, desenvolver seus próprios sistemas tutores inteligentes ou alterar de maneira significativa o modelo de conhecimento e o modelo de tutoramento de um sistema pronto. Nenhum dos sistemas estudados neste capítulo possuem estas características, já que não foram projetados para este fim, ou seja, ainda não

foi desenvolvido nenhum sistema de autoria ou *shell* para apoiar o ensino de programação de computadores.

Os sistemas de autoria podem contribuir para a divulgação dos STIs de diversas maneiras (MURRAY, T., 1997): a elicitação de conhecimento é facilitada, pois praticamente não se exige programação e é também eficiente, já que a representação do conhecimento é modular e pode ser reutilizada em outros sistemas; o autor pode verificar a validade do sistema construído através de testes imediatos; o sistema pode incluir conhecimento que o especialista talvez não possua, como por exemplo, sistemas de autoria que baseiam-se em princípios educacionais; além disso, o sistema pode gerar conhecimento além do que é esperado de um especialista na área.

O problema da produção de software emerge também no desenvolvimento de um STI (NICOLSON, R. I.; SCOTT, P. J, 1986) e nos apresenta a questão: quantidade ou qualidade? Esta questão foi originada a partir da observação das falhas dos modelos tradicionais de desenvolvimento de STI: por autor individual ou por equipe de autores. Apesar do desenvolvimento individual produzir sistemas rapidamente e em quantidade, ele geralmente possui problemas de qualidade, além de dificultar a manutenção. Em contrapartida, com o desenvolvimento por equipe, composto de vários componentes, onde cada um é responsável por uma parcela substancial do sistema, consegue-se produzir sistemas de qualidade mas lentamente e com um alto custo. Devido aos obstáculos originados do uso destas duas abordagens, algumas estratégias foram criadas: ambientes híbridos de autoria, onde programadores profissionais utilizaram linguagens de autoria para o desenvolvimento de STIs; uso de *toolboxes*, módulos úteis para o desenvolvimento de

STIs, que oferecem robustez e reutilização de código facilitada; construção de *shells* de sistemas especialistas, com as quais pode-se utilizar um mesmo programa com domínios de conhecimento diferenciados para se obter especialistas em diferentes áreas; finalmente, como uma situação ideal, o professor deve ter o controle de uma parte significativa do processo de desenvolvimento através de uma ferramenta de prototipação e o apoio de um analista/projetista de sistemas, especializado no desenvolvimento de STIs..

Os sistemas de autoria podem possuir graus de generalidade de domínios diferentes: alguns podem ser usados para construir sistemas bem específicos, como o sistema RUI (DIRENE, A., I., 1997) enquanto outros são mais genéricos, podendo gerar STIs para diversas áreas, como é o caso do sistema COCA (MAJOR, N., 1997). Além dos diferentes graus de generalidade, estes dois sistemas apresentam outra característica que os diferencia significativamente: a facilidade de construção dos STIs. Quando o escopo tem limites bem definidos é possível implementar, nos sistemas de autoria, diversas facilidades com relação ao tipo de conhecimento e de estratégia de ensino que serão usados pelos professores para a construção do STI.

O sistema RUI é utilizado especificamente para o ensino de conceitos visuais para identificação de anomalias existentes em radiografias. No caso do sistema RUI, a principal tarefa do autor é descrever a anatomia e as anomalias encontradas em imagens através de uma ferramenta de autoria com interface gráfica e sem a necessidade de se programar. O sistema COCA, sendo mais genérico, necessita de um esforço bem maior do autor para construir seu STI. A estratégia de montagem de um STI no sistema COCA baseia-se em três tipos de conhecimento: conhecimento do domínio (descrição do assunto a ser ensinado),

estratégia de ensino (a maneira como o conhecimento do domínio será apresentado) e conhecimento meta-estratégico (condições nas quais certas estratégias de ensino devem ser aplicadas). Na construção do STI, o autor deve especificar todos os três tipos de conhecimento, através da entrada de objetos de domínio e da criação de regras de estratégia de ensino e de regras meta-estratégicas. Portanto, podemos concluir que as tarefas que um autor tem que realizar em sistemas de autoria com alto grau de generalidade são bem mais complexas e trabalhosas se comparadas com as tarefas de um sistema de autoria com baixo grau de generalidade.

3) O SISTEMA *ASIMOV*

O objetivo principal do sistema *Asimov* é apoiar o ensino de lógica de programação de linguagens imperativas/procedimentais. Para atingir este objetivo, foram levados em consideração no projeto, diversos aspectos positivos encontrados nos sistemas estudados no capítulo anterior. Dentre estes aspectos temos: exploração livre do ambiente de programação, diagnóstico automático de programas, visualização de células de memória, execução compassada, possibilidade de alteração de alguns parâmetros de tutoramento pelo próprio aluno, facilidade na inclusão de novos problemas (**ver** Anexo I) e mudança (desde que obedecidas determinadas restrições) da linguagem alvo do sistema.

3.1) Principais considerações de projeto

Para facilitar o uso do sistema por parte do aluno, procurou-se incluir no projeto apenas características necessárias ao acompanhamento da lógica do programa, retirando-se quaisquer itens que pudessem confundir, desconcentrar ou desviar o aprendiz de seu objetivo. A linguagem de programação foi projetada para atender a estes requisitos: existe apenas o tipo de dado numérico, a declaração de variáveis é opcional, os comandos podem ser substituídos por diversos sinônimos e podem ser usadas máscaras (*templates)* de programação que completam os comandos mais complexos como a condição e a repetição. Em relação à periodicidade em que o sistema deve fornecer apoio ao aluno, sabe-se que existem diversos níveis de aprendizagem que exigem mais ou menos resposta do sistema. Portanto, foi incluída a capacidade de se alterar a granulação do *feedback* a qualquer momento em que o aprendiz ou o professor desejarem. Outro aspecto relevante a ser considerado é a abordagem que deve ser usada para ensinar programação. A abordagem escolhida foi o ensino através de exemplos, que permite a aprendizagem através da repetição, induzindo o aluno a encontrar a solução correta gradativamente.

3.1.1) Simplicidade da pseudo-linguagem

Embora possam ser realizadas alterações na linguagem-alvo, foi escolhida uma linguagem mais próxima do pseudo-código para a implementação inicial do sistema (**ver** Seção 3.2.2). A definição da linguagem levou em consideração o objetivo principal do

sistema (apoio a lógica) e, portanto, procurou atenuar as dificuldades de aprendizagem inerentes aos aspectos sintáticos da programação. A rigidez sintática de uma linguagem de programação insere mais um obstáculo na dura tarefa de se aprender a programar. Ao invés de se concentrar na solução do problema propriamente dito, o aluno deve assimilar diversas regras absolutas para poder começar a escrever um programa. Como a parte mais importante e mais difícil de se aprender a programar é a capacidade de entender o funcionamento individual dos comandos e de seu comportamento quando inseridos em determinado contexto, a pseudo-linguagem foi simplificada em quatro pontos principais: podem ser utilizados sinônimos para expressar ações, a declaração de variáveis é opcional, existe somente o tipo de dados numérico e o aluno é guiado através de máscaras (*templates*).

Em qualquer classe de ensino de programação podem ser identificados diversos alunos cuja dificuldade em se aprender a programar está ligada ao não entendimento da sintaxe da linguagem. A flexibilidade sintática foi incluída no sistema *Asimov* para evitar o impacto negativo inicial que um erro sintático causa em um aluno novato. Foram implementadas duas características com o intuito de se atingir esta flexibilidade: é possível utilizar-se diversos sinônimos de comandos e não é necessário finalizar um comando com o ponto-e-vírgula. Quando um aluno inicia a resolução de um problema de programação ele deve procurar identificar quais ações levarão à construção de uma solução correta. Como o que realmente importa para a solução final é a escolha correta tanto das ações quanto da maneira como elas se relacionam, é desnecessário e prejudicial para o aprendizado que apenas uma palavra possa ser escolhida para expressar determinada ação ou que devam ser

utilizados determinados símbolos especiais dentro do programa. Além destas duas características que o sistema *Asimov* explora, a própria gramática da linguagem é bastante flexível, possibilitando a utilização de diversos elementos de forma opcional, como por exemplo, algumas palavras reservadas desnecessárias (faça e então) e parênteses de funções da linguagem (leia e escreva).

Outro tópico que causa diversos problemas no aprendizado de programação é a maneira como o computador armazena valores. Qualquer valor deve ser armazenado em memória e deve ser classificado como pertencente a algum tipo de dado disponível na linguagem. Este processo causa grande confusão em alunos novatos por se tratar de assunto totalmente anti-natural, diferente do processo de raciocínio que uma pessoa utiliza para resolução de problemas do cotidiano. No sistema *Asimov*, o aluno não é obrigado a declarar suas variáveis, e também não precisa saber que existem diversas classificações diferentes ou tipos de dados, pois só está disponível o tipo numérico. Pode-se argumentar que estas atitudes não ajudam o aluno a programar bem, de maneira correta, com todas as suas variáveis declaradas e bem definidas. Entretanto, deve-se lembrar que o nosso objetivo é ajudar o aluno a adquirir a perícia inicial da montagem de algoritmos e que os aspectos abordados anteriormente podem onerar de maneira significativa esta aprendizagem.

Outras características que, se implementadas, trariam um ganho significativo para a flexibilidade sintática seriam: um aproximador ortográfico, a coloração diferenciada dos *tokens* digitados e a inclusão automática de máscaras (*templates*). O aproximador ortográfico seria de grande valia, pois mesmo que o aluno iniciante esteja pensando em uma solução correta, um erro de digitação pode causar dúvida e como conseqüência um desvio

em relação ao que seria uma resposta certa para o problema. Além disso, um editor dirigido pela sintaxe facilitaria ainda mais a relação entre o aluno e a parte sintática da linguagem. Este editor poderia colorir de forma diferenciada os *tokens* da linguagem, facilitando a identificação dos diversos elementos do programa e também poderia inserir automaticamente *templates* para os comandos mais complexos. Por exemplo, quando o aluno digitasse "Se" seguido de espaço em branco, o editor completaria o comando colocando "<condição> Então <comandos> FimSe". Desta forma o aluno poderia se concentrar apenas na solução do problema, evitando os empecilhos decorrentes da rigidez sintática da linguagem.

3.1.2) Granulação do *feedback*

Os sistemas estudados no capítulo 2 demonstraram que sempre houve uma preocupação dos autores com o momento em que um STI deve interferir na execução do programa e exibir uma mensagem de erro ou de acerto para o aluno. Enquanto alguns sistemas iniciam o seu diagnóstico apenas depois do término do programa (análise pós-fixada), outros atuam durante a interação com o aluno (análise in-fixada). Embora, atualmente, exista uma tendência em se perseguir a segunda alternativa, ainda existe muita controvérsia em relação à periodicidade da interferência do sistema. Devido a esta indefinição, o sistema *Asimov* foi projetado para permitir que o professor, ou até mesmo o aluno, alterem esta periodicidade, também chamada de granulação do *feedback*.

Esta alteração dinâmica permite que o sistema seja utilizado por alunos com diferentes níveis de entendimento de programação, desde o mais novato, que não consegue

seguir adiante sem a certeza de que o comando digitado está correto, até o mais adiantado que prefere digitar toda a sua solução para só então receber as críticas do sistema. Desta maneira, dois aspectos do ensino de programação estão contemplados: a exploração livre do ambiente, sem nenhuma interferência do sistema e a programação guiada, na qual o sistema entra em ação sempre que houver algum desvio em relação à solução correta.

A flexibilidade do sistema *Asimov* em relação à granulação do *feedback* é alcançada através da alteração de determinados parâmetros que permitem a realização do diagnóstico em momentos diferentes. Os tipos de granulação permitidos são: *token*, expressão, comando, bloco, programa e nenhuma. Respectivamente, o sistema atua após a conclusão da digitação de um dos seguintes elementos: um *token*; uma expressão aritmética ou relacional; um comando que é finalizado por um ponto-e-vírgula ou pelo início de um outro comando; um bloco que é finalizado por um *token* específico de fechamento de blocos, de acordo com cada linguagem; do programa como um todo e, ainda, nenhuma interferência se o aluno não quiser *feedback* por parte do sistema.

3.1.3) Aprendizagem baseada em exemplos

Para considerarmos que um aluno esteja apto a programar, devemos lembrar de dois objetivos a serem atingidos durante o estudo: aquisição de princípio e de perícia. A aquisição do princípio de programação leva em conta a aprendizagem da sintaxe e da semântica individual de comandos e da estrutura da linguagem-alvo. A aquisição da perícia de programação vem logo após o domínio dos itens básicos (princípio) e engloba o entendimento do comportamento dos comandos quando inseridos em um determinado

contexto e a capacidade de integrar estes comandos para resolver problemas de programação.

Embora estes dois aspectos devam ser aprendidos como um todo, eles são ensinados separadamente e de formas diferentes. Enquanto os tópicos básicos podem ser ensinados através da explanação simples e pura, o comportamento e a integração dos comandos são muito complexos para serem aprendidos através de uma atitude passiva. Através da explicação de um determinado algoritmo, um aluno novato até consegue entender o seu funcionamento, mas dificilmente conseguirá resolver um problema semelhante sem o auxílio de alguém que já tenha adquirido a perícia em programação.

A maneira mais difundida para se ensinar a programação é através do estudo de exemplos prontos e da aplicação do algoritmo destes exemplos em problemas semelhantes. Ao estudar como um novo algoritmo funciona, o aluno descobre novas formas de resolver uma certa faixa de problemas. Com a aplicação deste mesmo algoritmo em um ou mais problemas semelhantes o aluno adquire efetivamente o conhecimento embutido no conjunto de comandos que formam este algoritmo. O aluno novato é induzido a resolver certo conjunto de problemas de uma determinada maneira e assim adquire a perícia ou a capacidade de agrupar comandos para formar soluções através da repetição do uso de um conjunto de comandos que foram apresentados através de um exemplo pronto.

O sistema *Asimov* contribui para esta abordagem no sentido de que permite ao professor a fácil inclusão de novos problemas para serem propostos ao aluno. Com esta característica o aluno pode treinar determinado formato de algoritmo em diversos problemas semelhantes ao mesmo tempo em que é guiado enquanto digita sua solução. Como é o

próprio professor que inclui a resposta ao problema (**ver** Seção 4.1), ele pode limitar a solução para conter apenas determinado algoritmo que deve ser assimilado pelo aluno naquele momento de seu aprendizado.

3.2)Visão panorâmica do sistema *Asimov*

O sistema *Asimov* possui uma interface contendo quatro áreas bem distintas: edição de programas, visualizador, entrada e saída de dados do programa e enunciado do exercício proposto. A linguagem escolhida como linguagem nativa do sistema está bem próxima do pseudo-código, embora possa ser adaptada pelo professor desde que algumas regras sejam seguidas. O professor pode também criar seu próprio conjunto de exercícios que servirão de base para a abordagem de programação guiada, na qual o aluno pode ser constantemente avaliado e induzido a escrever uma solução correta.

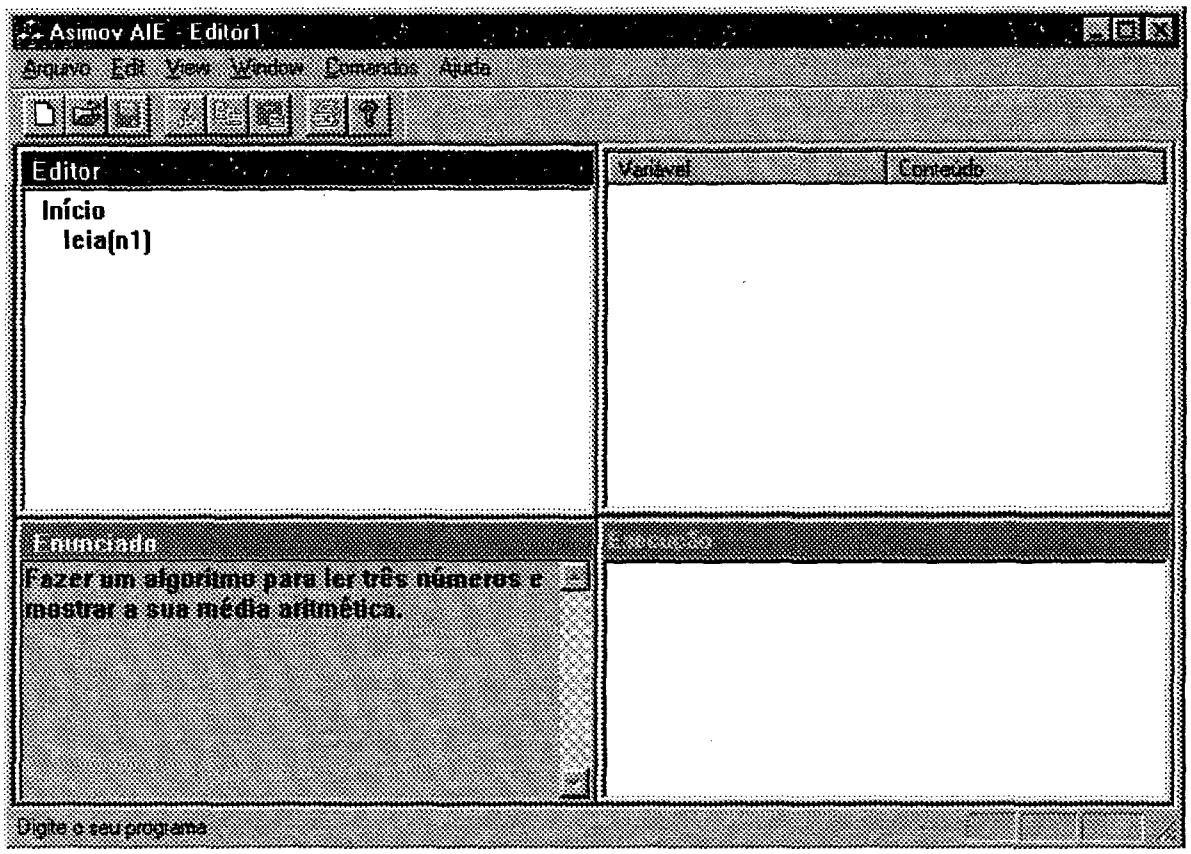
3.2.1) Interface do usuário

A interface do sistema (Figura 3) é composta de um menu com as opções de uso e de quatro janelas secundárias que estão permanentemente visíveis: área de edição de programas, onde o aluno digita sua solução; visualizador de células de memória, na qual o estado das variáveis é mostrado; janela de entrada/saída, utilizada para apresentar os dados decorrentes da execução do programa do aluno e janela de exibição de enunciados, que exibe o enunciado atual ou também pode estar vazia se nenhum enunciado for escolhido. A

divisão da interface do aplicativo em janelas sempre visíveis permite uma melhor identificação dos elementos visuais, facilitando a sua utilização.

O aluno tem duas opções para iniciar o uso do sistema: começar a digitar um programa qualquer ou escolher um exercício da base de problemas. No primeiro caso, o sistema funcionará como um compilador e um interpretador, permitindo uma exploração livre do ambiente, execução compassada e visualização das células de memória. Se o aluno escolher um exercício ele terá, além dos itens apresentados no caso anterior, apoio sempre que cometer algum erro, ou seja, será guiado até alcançar uma solução correta.

Figura 3 – Interface do Sistema ASIMOV



A janela de enunciados poderá conter o enunciado de um problema escolhido ou estar vazia, dependendo do modo de interação escolhido. Desta maneira, o aluno poderá

voltar ao enunciado sempre que precisar, enquanto procura identificar a solução correta para o problema. O conteúdo desta janela está vinculado aos critérios de ensino do professor, podendo conter, além do enunciado da questão, dicas para resolução, lembretes sobre determinados tópicos obscuros do assunto e, até mesmo, soluções utilizadas em problemas semelhantes que poderão servir como base para o aprendiz.

A solução a ser avaliada pelo sistema deve ser digitada diretamente no editor de programas, da mesma maneira em que um texto é digitado em um editor comum. Dependendo do tipo de granulação escolhido e da existência de um exercício a ser resolvido, o sistema fornecerá diagnóstico à medida em que o programa fonte é apresentado ao sistema. Além de receber este apoio, com relação à lógica, o aluno pode solicitar a execução do programa (total ou compassada) a qualquer momento para verificar como ele está se comportando. Nesta fase, chamada de interpretação, poderão ser visualizadas as alterações das variáveis utilizadas de acordo com a execução do programa na janela de visualização de células de memória. Esta janela apresenta todas as variáveis contidas na solução do aluno e seus respectivos valores correntes. Durante esta fase, os dados de saída que o programa gerar e os dados de entrada necessários ao funcionamento do programa são exibidos na janela de entrada/saída.

3.2.2) A linguagem de programação

A linguagem de programação escolhida para a implementação inicial do sistema é procedural/imperativa estando bem próxima de uma pseudo-linguagem de programação

amplamente utilizada no ensino de lógica de programação. Entretanto, esta definição pode ser alterada pelo professor através da inclusão de uma nova gramática no sistema (**ver** Anexo I). Esta gramática deve ser uma *ESLL(1)* (SETZER, V. W.; MELO, I. S. H., 1983) e toda palavra reservada pode ter diversos sinônimos associados. A gramática da linguagem alvo está descrita em BNF a seguir, com as palavras reservadas e símbolos da linguagem em negrito, elementos especiais em caracteres maiúsculos e não-terminais em caracteres normais.

programa ::= [**início**] declaração comando **fim**

declaração ::= [**declare** {IDENT || ‘,’}$^{+}$ ‘**:**’ **número** [‘;’]]

comando ::=

{

(

IDENT ‘**:=**’ expressão |

escreva ‘**(**‘ {expressão || ‘,’}$^{+}$ ‘**)**’ |

leia ‘**(**‘{IDENT || ‘,’}$^{+}$ ‘**)**’ |

se condição **então** comando [**senão** comando] **fimse** |

enquanto condição [**faça**] comando **fimenquanto** |

repita (NUM **vezes** comando **fimrepita** | comando **até** condição) |

para IDENT **de** expressão **até** expressão [**passo** expressão] [**faça**] comando **fimpara**

) [‘;’]

}*

expressão ::= ['-'] {termo || '+' | '-' | **ou**}$^{+}$

termo ::= {fator || '*' | '/' | **e**}$^{+}$

fator ::= NUM | '(' expressão ')' | IDENT | **não** fator | function

condição ::= expressão [('=' | '<' | '<=' | '>' | '>=' | '#')expressão]

function ::= **raizquadrada** '(' expressão ')' |

quadrado '(' expressão ')' |

resto '(' expressão ',' expressão ')'

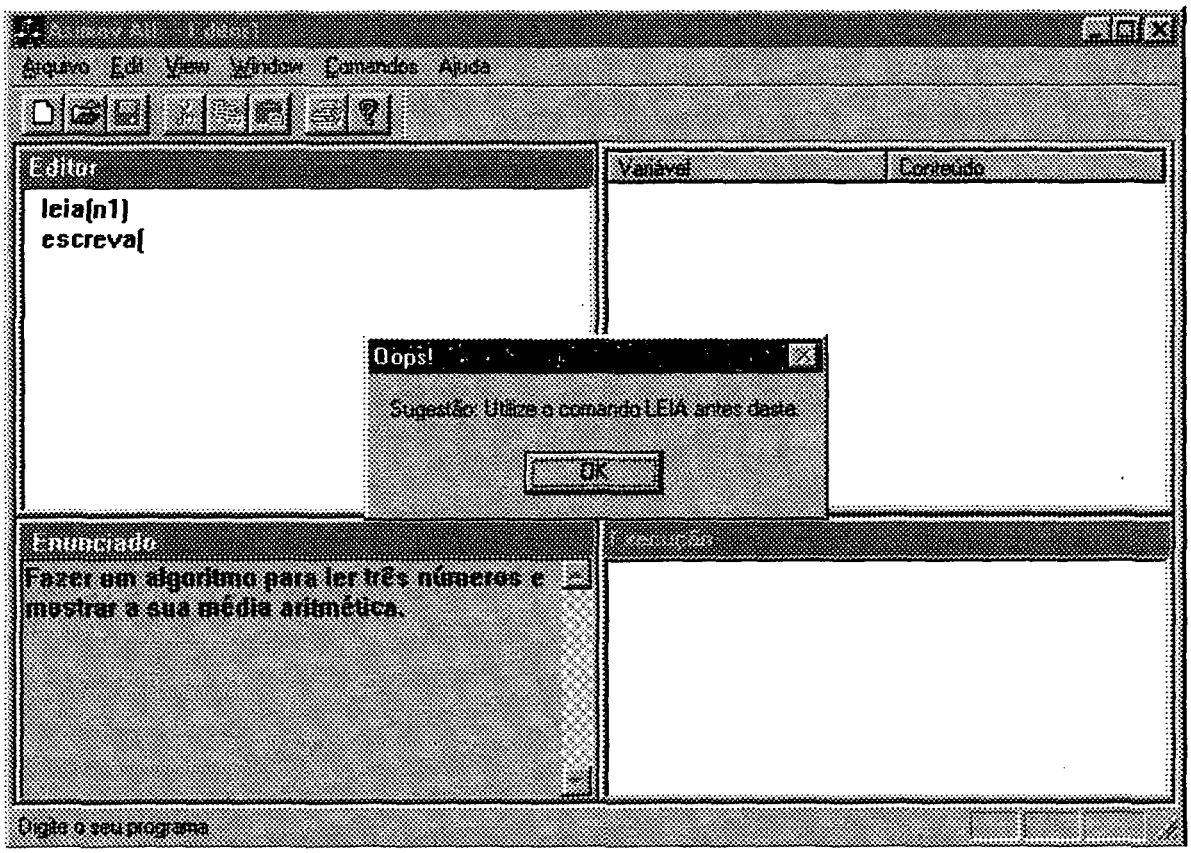
3.2.3) Programação guiada

No sistema *Asimov*, o aluno pode receber auxílio tutorial com relação à lógica enquanto desenvolve o seu algoritmo. Esta modalidade de programação com acompanhamento e diagnóstico da lógica é chamada de programação guiada e procura evitar que o aprendiz desvie-se demasiadamente da solução correta. Como esta abordagem não é ideal para todos os perfis de alunos encontrados, permite-se a alteração da granulação do *feedback*, já explanada no item 3.1.2.

Os erros de lógica mais comuns encontrados em programas de alunos iniciantes são bastante semelhantes e podem ser classificados em erros **intra-comandos** e **inter-comandos** (du BOULAY, B.; SOTHCOTT, C., 1987), embora existam situações de sobreposição entre os dois tipos de erros(**ver** seção 4.2.1). A Figura 4 mostra um erro inter-comandos detectado pelo sistema *Asimov* em uma solução de aluno e a mensagem tutorial correspondente. Os erros intra-comandos são aqueles cuja correção ocorre apenas no escopo do comando onde ocorreu o erro, como, por exemplo, uso incorreto de expressões

relacionais e aritméticas. Os erros inter-comandos são aqueles que influenciam ou são influenciados por outros comandos do programa (a correção de tais erros pode afetar um ou mais comandos), como: colocação errada de comandos em relação a uma repetição ou condição e inversão de comandos. Estes erros são sinalizados para o aprendiz de acordo com o tipo de granulação de *feedback* escolhido.

Figura 4 – Mensagem de erro emitida pelo sistema ASIMOV

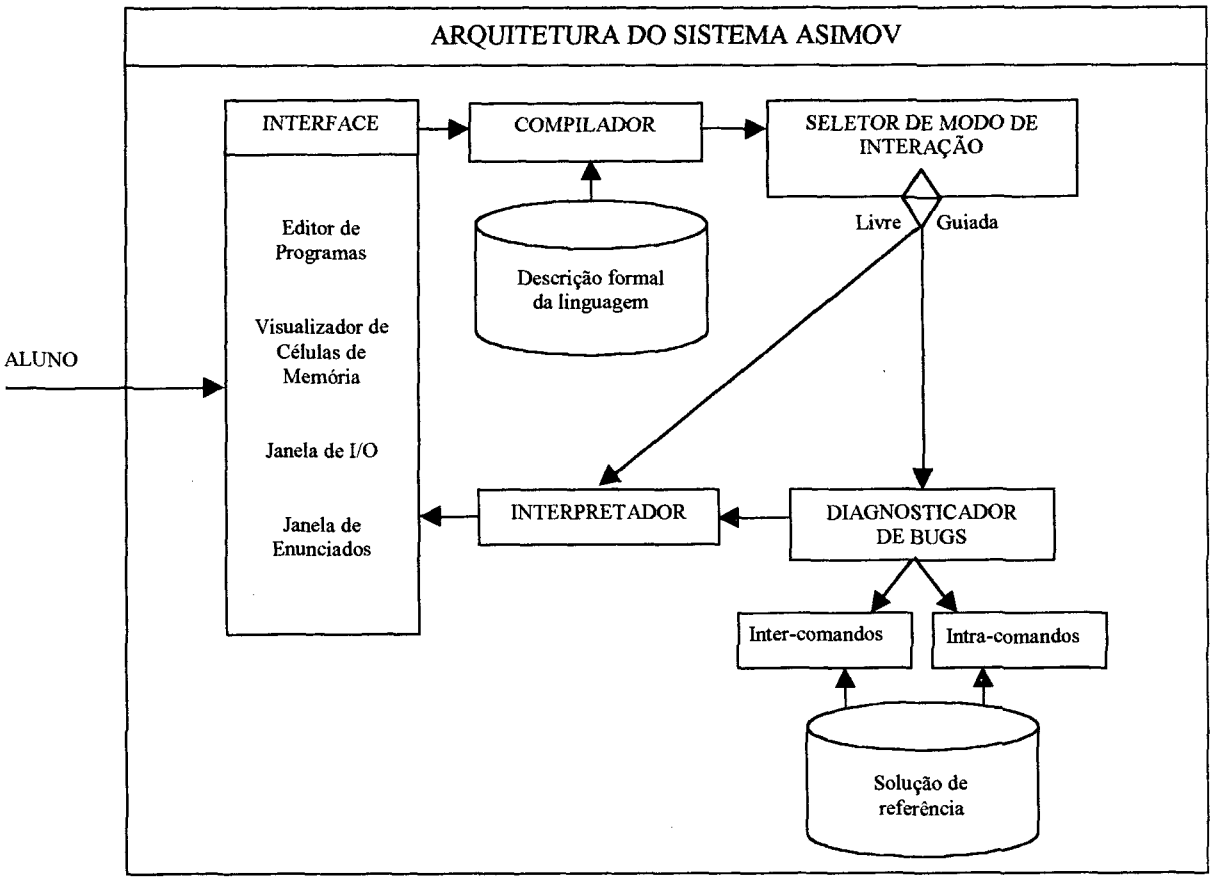


3.3) Arquitetura funcional

O sistema *Asimov* é capaz de realizar as tarefas de: edição de textos, compilação do texto digitado de acordo com uma gramática flexível, diagnóstico automático da solução do

aluno e interpretação do programa, com execução compassada e visualização do conteúdo das variáveis deste programa em células de memória. A implementação destes processos exigiu o uso de diversas técnicas computacionais tais como: construção de compiladores e interpretadores, redes semânticas e casamento de padrões. Todas estas características estão distribuídas em diversos módulos que formam a arquitetura do sistema (Figura 5): interface, compilador, seletor de modo de interação, diagnosticador de bugs e interpretador.

Figura 5 – Arquitetura do Sistema ASIMOV



3.3.1) Interface

Através da interface, o aluno pode ativar as diversas características do sistema, bem como receber informações tutoriais. A entrada de dados no sistema resume-se a: alteração da granulação do *feedback*, digitação da solução proposta para o problema, execução compassada da solução e entrada de dados requisitadas pela execução. As informações fornecidas pela interface são: *feedback* tutorial em relação à lógica da solução do aluno, exibição das mensagens geradas por esta solução e visualização das células de memória que representam as variáveis do programa do estudante.

A alteração da granulação de *feedback* interfere na periodicidade em que o diagnosticador automático fornece as mensagens tutoriais. A digitação da solução proposta para o problema será validada sintaticamente e traduzida pelo compilador, avaliada logicamente pelo diagnosticador e executada pelo interpretador. A execução do solução proposta é tarefa do interpretador e pode ser realizada através de compassamento, de acordo com o comando do aluno. Esta execução pode fornecer mensagens geradas pela própria solução e também requisitar entradas de dados numéricos do teclado. Todo o processamento realizado por esta solução poderá ser visualizado através das células de memória que indicam, a cada momento, o conteúdo das variáveis. Na interface também ocorre a exibição das mensagens correspondentes aos erros identificados pelo diagnosticador automático.

3.3.2) Compilador

O compilador inicia o seu processamento com a leitura de um arquivo externo da gramática da linguagem que deverá ser traduzida e com a correspondente montagem das estruturas de dados necessárias, tais como tabelas de símbolos e grafo sintático. O processo de tradução é realizado à medida em que o aluno vai digitando a sua solução, independente do tipo de granulação escolhido. Cada caracter é validado pelo analisador léxico, cada *token* é validado e classificado pelo analisador sintático e cada comando tem seu código gerado pelo gerador de código durante a digitação da solução. Este formato possibilita tanto a criação de um editor dirigido pela sintaxe, quanto a execução da solução em qualquer momento, mesmo que esta ainda não tenha sido completada.

O compilador, através do processo de validação e tradução, fornece dados para outros dois componentes da arquitetura do sistema: diagnosticador e interpretador. Os elementos da solução identificados e classificados são usados pelo diagnosticador automático para a comparação com a solução de referência, enquanto o código gerado com informações de depuração (número das linhas do código fonte e nomes das variáveis) é utilizado pelo interpretador para a execução do programa, compassamento desta execução e visualização das variáveis através das células de memória. Nesta versão inicial do sistema, na qual a performance do código gerado é irrelevante, não foram implementadas as rotinas que fazem a otimização do programa objeto.

3.3.3) Seletor de modo de interação

O seletor de modo de interação é o componente responsável pela verificação do estado corrente do sistema e pela conseqüente ativação ou desativação do diagnosticador automático. O diagnosticador pode ser desativado de duas maneiras: pela opção "nenhuma" da granulação do *feedback*, ou quando não for escolhido nenhum exercício para ser solucionado. Este modo de uso foi denominado de **modo de exploração livre**, já que o aluno não recebe qualquer auxílio tutorial do diagnosticador, apesar de ainda poder utilizar os recursos de compassamento e visualização do interpretador. Quando o aluno escolher um exercício da base de dados e uma das opções da granulação do *feedback* (*token*, comando, expressão, bloco, programa) estiver selecionada, o diagnosticador é ativado e o sistema passa a fornecer auxílio tutorial. Este modo foi denominado de **modo guiado**, pois, através da comparação com a solução de referência, guia o aluno durante o desenvolvimento de sua solução, evitando que ocorram desvios em relação ao modelo correto.

3.3.4) Diagnosticador

O diagnosticador automático do sistema *Asimov* é responsável pela identificação de algumas classes de erros lógicos comuns encontrados na solução proposta pelo aluno. Ele é composto por módulos altamente especializados, divididos em dois grandes grupos de elementos de conhecimento dinâmico: identificadores de erros inter-comandos e intra-comandos. Estes identificadores encontram erros através da comparação da solução do

aluno com uma solução de referência cadastrada no sistema, geralmente montada pelo professor.

O diagnosticador depende do tipo de granulação de *feedback* escolhida para identificar o momento de exibir as mensagens de erro, mas, independentemente da granulação, ele realiza o seu diagnóstico constantemente, durante o processo de entrada de dados no sistema. Ele se restringe a chamadas aos identificadores de erro para encontrar e classificar os possíveis erros da solução do aluno e não é influenciado pelo funcionamento do interpretador. Veremos os detalhes das representações e do funcionamento detalhado dos identificadores e do processo de comparação das soluções (de referência e do aluno) no capítulo 4.

3.3.5) Interpretador

O objetivo do interpretador é o de fornecer um mecanismo para o *Asimov* permitir a condução de tarefas de visualização da execução de programas através da apresentação das alterações de células de memória. Sendo assim, o interpretador é tipicamente ativado no modo de exploração livre.

O interpretador é o componente que realiza a execução simbólica dos programas de aprendizes quando o *Asimov* encontra-se em seu estado passivo. Esta execução é realizada com base no código gerado pelo compilador, que além das informações de quais instruções devem ser executadas, também contém alguns dados de depuração.

O módulo do interpretador é geralmente utilizado após o término da digitação, embora possa realizar sua tarefa em qualquer ponto do desenvolvimento da solução, ou seja, ele é capaz de executar soluções parciais, quando solicitado pelo aluno. Esta característica permite que o aluno explore o ambiente e verifique se o seu programa está tendo o comportamento esperado a qualquer momento. Ainda durante a execução, o interpretador envia e recebe dados da interface, de acordo com as entradas e saídas do código gerado.

4) REPRESENTAÇÃO DE CONHECIMENTO E DIAGNÓSTICO AUTOMÁTICO

4.1) A solução de referência do Asimov (Model Answer)

O sistema *Asimov* identifica erros de lógica em programas de alunos iniciantes através da comparação da solução do aluno para um determinado problema com uma solução pré-definida pelo professor, denominada de solução de referência. As soluções de referência tradicionais são bastante rígidas, aceitando como soluções corretas apenas aquelas cuja comparação resulte em total igualdade com a solução do sistema. A solução de referência do *Asimov* é mais flexível, pois permite dois tipos de variações em relação à solução correta: dentro de uma mesma solução, através da definição de dependências de comandos e entre soluções diferentes, pelo uso de caminhos alternativos de comandos. O modelo de representação da solução de referência expressa o conhecimento estático do sistema e o processo de diagnóstico, o conhecimento dinâmico.

4.1.1) Descrição da "solução de referência"

A maior dificuldade na escolha de uma representação para a solução de um problema de programação é a complexa interdependência entre os diversos comandos existentes em um programa. Mesmo que a representação seja a mais completa possível ainda nos resta o problema de como identificar desvios e variações que não tornam a solução incorreta e que são bastante comuns nos programas de alunos iniciantes. A partir de uma representação de **rede semântica**, estrutura onde o significado de determinado item está associado com a maneira como ele está conectado a outros itens da rede (QUILLIAN, R., 1968), desenvolvemos um formalismo de modelagem da lógica de solução de programas denominado de GRADE (grafo de alternativas e dependências). O objetivo deste modelo é facilitar tanto a criação de novos exercícios quanto a comparação de soluções corretas com a solução de alunos.

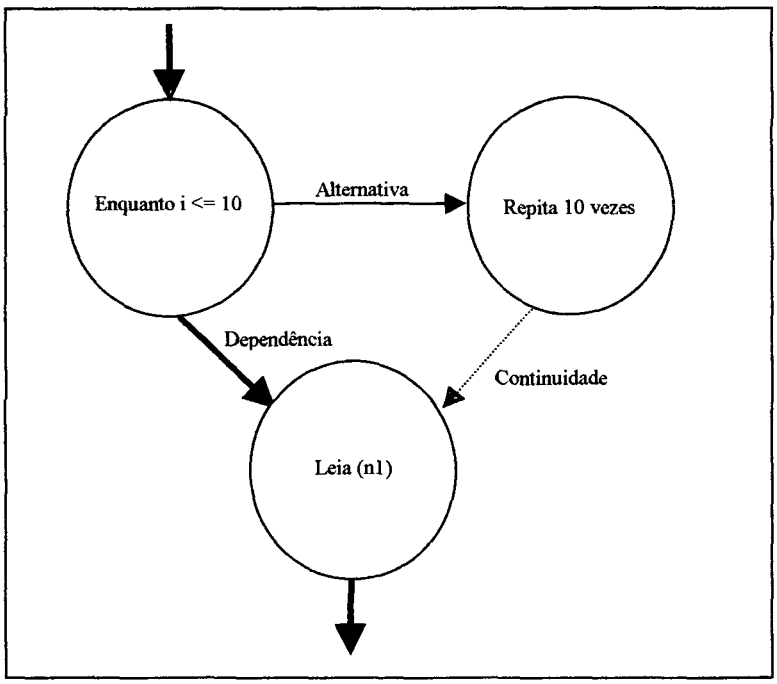
A rede semântica identifica os mapas conceituais da solução de referência de um enunciado. Esta solução contém a representação do conhecimento de uma faixa de soluções expressas pelo professor, descritas através de conjuntos de comandos e de relações entre eles. O sistema utiliza esta solução como base para a comparação com a solução do aluno, tentando encontrar semelhanças e/ou divergências entre elas. O processo de comparação e a solução de referência complementam-se formando o conhecimento necessário para a realização do diagnóstico. Enquanto o conhecimento embutido na solução de referência é declarativo e de natureza estática, já que por si só não identifica os erros de programas de alunos, o conhecimento do processo de comparação possui natureza dinâmica, pois

consegue efetuar a correspondência entre a solução do aluno e os parâmetros indicados na solução de referência, verificando assim, se existem erros de lógica.

4.1.2)Capacidade de capturar variações

Cada exercício proposto pelo sistema é composto da área de texto livre, onde o professor pode inserir o enunciado, exemplos, sugestões e dicas e da solução de referência, conjunto de comandos e parâmetros representada pelo GRADE. O GRADE (figura 6) é um grafo que representa a solução de referência de determinado exercício levando em consideração as dependências obrigatórias e opcionais que existem entre comandos e o uso de algoritmos diferentes para resolver um mesmo problema.

Figura 6 – Grafo de Alternativas e Dependências (GRADE)



O GRADE é composto de quatro elementos básicos: nós, arcos de dependência, arcos de alternativa e arcos de continuidade. Os nós possuem os atributos que identificam uma determinada linha do programa fonte: identificador, comando, parâmetros do comando e mensagem de erro mostrada no caso de ausência do comando. Estes nós são ligados a outros através de três tipos de arcos: **arco de dependência** que indica que o nó destino é dependente do nó de origem, ou seja, o comando descrito no nó destino só será válido se vier depois do comando que está no nó de origem; **arco de alternativa**, indicando que o nó destino é uma alternativa ao nó origem, ou seja, se a comparação do comando do nó origem falhar, o comando do nó destino é ativado; **arco de continuidade**, usado para representar a ordem de continuação para alternativas. Esta representação possibilita a capacidade de capturar variações, através das relações de ordem entre os comandos da solução de referência que são a dependência e a alternatividade.

4.2) Diagnóstico automático

O conhecimento dinâmico do sistema é originado da natureza procedimental do diagnosticador o qual é guiado por conhecimento declarativo dos identificadores de classes de erro. A comparação entre a solução de referência e a solução do aluno é realizada através de um algoritmo de casamento de padrões ou *pattern matching* (BURTON, M.; SHADBOLT, N., 1987, p.53-60; RICH, E., 1993, p.211-219). Embora existam diversas linguagens de IA tradicionais que possuem rotinas de casamento de padrões nativas (Pop-11, Miranda), foram implementadas funções próprias no sistema *Asimov*.

Com base nas relações de dependência e alternatividade citadas na seção 4.1.2, o diagnosticador procura encaixar a solução do aluno dentro da solução de referência e, se este encaixe não for possível, um erro inter-comando é identificado. Embora possam existir vários erros no programa do aluno, dependendo do tipo de granulação escolhida, o sistema apresenta mensagem apenas para o primeiro, sugerindo ao aprendiz qual a ação a ser tomada para corrigir o erro.

A forma geral de representação das ações do diagnosticador é semelhante a de regras de produção, podendo ser resumida ao seguinte comportamento: se padrão encontrado na solução do aluno casa com parâmetros de qualquer identificador de erro então emitir mensagem. Assim, se o diagnosticador encontrar algum erro no programa do aluno, ele altera o estado do ambiente de programação, emitindo mensagens de erro. Estas mensagens são originadas a partir de classes de erros mais comuns encontradas em programas de alunos iniciantes. Estes erros foram identificados em 66 provas (cuja ênfase era o uso de comandos de condição e repetição) de alunos iniciantes de uma turma com 8 semanas de estudo de programação básica (**ver** Anexo II). Os identificadores das classes de erros podem ser de dois tipos, de acordo com a explicação da seção 3.2.3: inter-comandos e intra-comandos. Apesar da importância de ambos os identificadores para um diagnóstico correto, o sistema *Asimov* implementa apenas os identificadores de erros inter-comandos em sua versão inicial. O diagnosticador, assim como o restante do sistema, foram implementados em C++ como parte integrante deste trabalho.

4.2.1) Representação de Identificadores de Erros Intra-Comandos

Os erros de lógica intra-comandos são aqueles cuja correção é feita apenas no escopo do comando onde o erro ocorre, não afetando outros comandos do programa. Os identificadores de erros intra-comandos necessários para o diagnóstico de programas de alunos iniciantes são:

- Expressão relacional ou aritmética incorreta. Este tipo de erro é difícil de ser identificado de maneira precisa por um diagnosticador automático, havendo a necessidade de se gerar gramáticas específicas para cada expressão a ser avaliada.
- Inversão de operadores aritméticos, relacionais ou lógicos. Outro erro difícil de ser identificado, também exigindo a geração de gramáticas específicas. Ex.: **Enquanto (i >10) E (i < 0)** no lugar de **Enquanto (i >10) OU (i < 0)**.
- Erro no uso de parênteses. Mais um erro de complexa identificação através da geração de gramáticas. Ex: **valor ←(a+b/2)** no lugar de **valor ←(a+b)/2**.
- Atribuição inútil. Este erro acontece nos estágios iniciais de aprendizagem, quando o aluno ainda não tem um modelo claro do funcionamento de um programa e do armazenamento de variáveis em memória. A identificação deste erro é bem simples, pois basta verificar se ambos os lados da atribuição contém a mesma variável. Ex: **cont ←cont**.

4.2.2) Representação de Identificadores de Erros Inter-Comandos

Os erros inter-comandos tem uma relação estreita com a perícia que um aluno deve adquirir, pois são aqueles erros originados da falta de entendimento do comportamento dos comandos quando agrupados para formar um algoritmo. Os principais identificadores de erros inter-comandos encontrados em programas de alunos são:

- Posicionamento incorreto de comandos. Este é um dos erros mais freqüentes e também é ocasionado pela falta de entendimento do fluxo de controle de um programa. O identificador desta classe é ativado quando todos os comandos de um certo trecho da solução de referência existem na solução do aluno, porém não estão na ordem correta. Ex: um comando de contagem que deveria estar dentro de uma repetição mas está fora dela.
- Erro de ajuste de valor inicial. Quando uma variável é inicializada com um valor incorreto em relação ao problema que deve ser resolvido. O identificador reconhece este erro através da comparação do comando do aluno com as alternativas da solução de referência.
- Uso de variável do lado direito de uma expressão sem inicialização ou leitura. Outro erro cuja identificação é simples: basta verificar se a variável contém um valor de inicialização.
- Uso de variável incorreta em comandos. Este erro é ocasionado pelo uso de uma variável incorreta em um comando qualquer. Sua identificação é feita através do uso de meta-símbolos para representar as variáveis da solução de referência com relação às variáveis da solução do aluno. Ex: **cont ←soma + 1** ao invés de **cont ←cont +1**".
- Leitura de variável que deveria ser calculada. Outro erro comum que é identificado da mesma maneira que o erro anterior. Ex: **leia (soma)**

- Ausência de um comando dentro de uma seqüência. A identificação é feita através da comparação com a solução de referência onde todos os comandos estão corretos, faltando apenas um.
- Não entendimento da estrutura de funcionamento de um programa. Este identificador é acionado quando a comparação da solução do aluno com a solução de referência tem como resultado uma impossibilidade de encaixe de qualquer tipo. Neste ponto deve-se ressaltar que uma quantidade bastante pequena de soluções pode estar correta, apesar de ser totalmente diferente de uma resposta padrão.

Dentre as classes acima citadas, o sistema *Asimov* identifica em sua versão inicial os seguintes erros de lógica inter-comandos (em ordem de complexidade de reconhecimento):

- Uso de variável sem inicialização ou leitura;
- Falta de qualquer comando dentro de uma seqüência;
- Uso de variável incorreta;
- Inversão da posição de comandos e
- Colocação errada de comandos em relação a uma repetição ou condição.

4.2.3) Exemplos de Cenário de Uso

Apresentamos a seguir dois exemplos de exercícios que podem ser propostos pelo sistema *Asimov* e quais as ações que o sistema executa quando o aluno comete um erro.

Neste primeiro exemplo vamos considerar que o aluno escolheu a granulação de *feedback* do tipo *token*, ou seja, o sistema fornecerá apoio tutorial sempre que for cometido algum erro de lógica.

Enunciado: Fazer um programa que leia dois números inteiros quaisquer e calcule a sua média aritmética.

Primeira solução possível:

Leia (n1)

Leia (n2)

Valor := (n1+n2)/2

Escreva (Valor)

Segunda solução possível:

Leia (n1)

Valor := n1

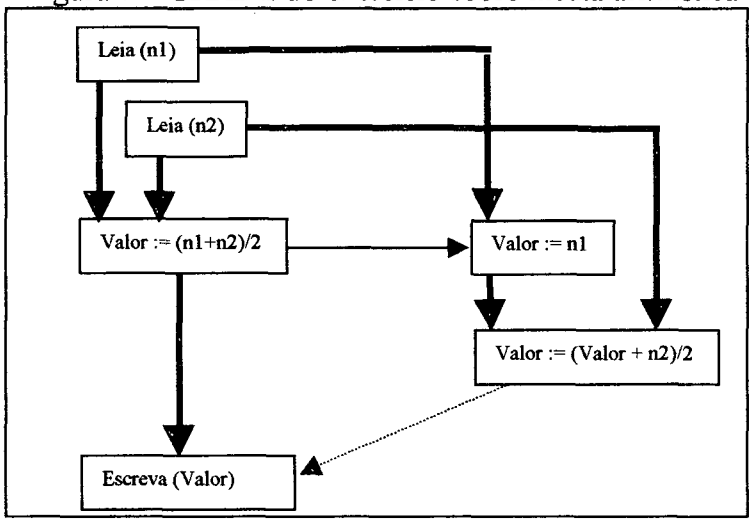
Leia (n2)

Valor := (Valor + n2)/2

Escreva (Valor)

A figura 7 mostra a representação destas duas soluções através do GRADE. As flechas simples indicam alternativas, as flechas largas indicam dependência e as flechas pontilhadas indicam continuidade.

Figura 7 – GRADE do exercício sobre média aritmética



Como a granulação escolhida foi a mais restritiva de todas, a cada item que o aluno digitar, o diagnosticador fará a validação do programa. Se, por exemplo, o aluno começar o seu programa com um comando diferente de leia, o sistema emitirá uma mensagem sugerindo que o aluno utilize este comando. As duas relações de ordem estão aqui representadas: 1) dependência, pela qual o cálculo só pode existir depois da leitura das variáveis e 2) alternatividade, onde o mesmo cálculo pode ser dividido em dois comandos. Através destas duas variações possíveis e da solução de referência para este exercício, o sistema *Asimov* consideraria a solução abaixo como correta, não emitindo nenhuma mensagem de alerta:

```
Leia (B)
Leia (A)
Média := A
Média := (Média + B)/2
Escreva (Média)
```

Para o segundo exercício, vamos considerar que a granulação escolhida foi a do tipo programa, ou seja, o aluno irá receber apoio tutorial apenas após ter completado a digitação de seu programa.

Enunciado: Construir um programa que leia n números inteiros quaisquer e calcule e mostre a sua soma.

Primeira solução possível:

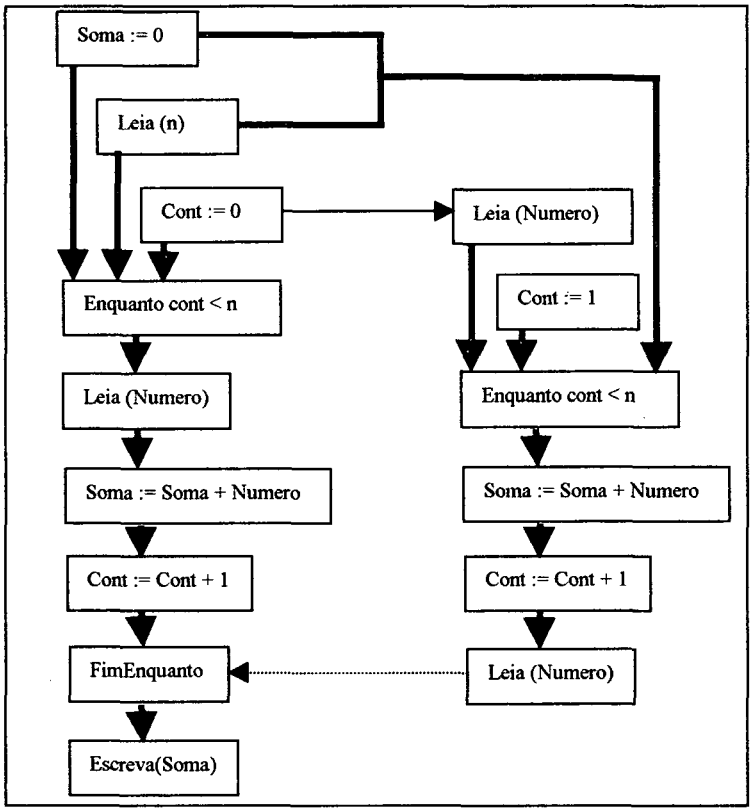
```
Soma := 0
Leia (n)
Cont := 0
Enquanto cont < n faça
        Leia (Numero)
        Soma := Soma + Numero
        Cont := Cont + 1
FimEnquanto
Escreva (Soma)
```

Segunda solução possível:

```
Soma := 0
Leia (n)
Cont := 1
Leia (Numero)
Enquanto cont < n faça
        Soma := Soma + Numero
        Cont := Cont + 1
        Leia (Numero)
FimEnquanto
Escreva (Soma)
```

A figura 8 mostra a representação destas duas soluções através do GRADE.

Figura 8 – GRADE do exercício sobre soma



Neste cenário, com a granulação por programa ativada, o diagnosticador pode identificar erros que não seriam possíveis no modo de granulação por *token*, pois ele consegue analisar a solução do aluno como um todo. Assim, erros de posicionamento de comandos podem ser reconhecidos. Desta maneira, supondo que o aluno coloque a inicialização das variáveis dentro da repetição, o sistema *Asimov* consegue identificar que o erro não é simplesmente a falta de inicialização de variável, mas sim, a posição das inicializações esta errada com relação ao escopo do comando enquanto.

4.3) Limitações do Sistema *Asimov*

Além de algumas características adicionais desejáveis e de determinadas restrições apresentadas no decorrer deste documento, existem mais duas limitações significativas no sistema *Asimov*: tanto a verificação de expressões aritméticas e relacionais quanto o reconhecimento de uma solução correta estão condicionados às dependências e alternativas identificados pelo professor na solução de referência.

O problema da verificação de expressões de um modo geral é a dificuldade em se identificar uma expressão correta. Esta dificuldade surge pois existem infinitas combinações possíveis de símbolos que geram o resultado esperado para todas as faixas de valores, assim como existem aquelas combinações que estão corretas para algumas faixas de valores e incorretas para outras. Por exemplo, para calcular a média aritmética entre dois elementos pode ser usada a forma natural (valor←(a+b)/2), uma outra forma correta qualquer (valor←a; valor←(valor+b)/2) e ainda, uma solução que está correta apenas para alguns casos (valor←a+b/2). As soluções viáveis para este problema são bastante complexas, já que tratam da manipulação de conjuntos de variações de magnitude muito grande, e mesmo sendo genéricas não conseguem tratar de todos os casos. Uma das soluções é a geração de uma gramática para cada expressão da solução de referência a ser testada, seguida de uma verificação que identifica se a expressão montada pelo aluno pertence ou não à gramática.

Existem diversas soluções corretas para um problema de programação e estas soluções podem ser muito diferentes umas das outras. O sistema *Asimov* só consegue identificar como corretas, as soluções de alunos que tenham poucas variações em relação à

solução de referência. Soluções corretas com diferenças estruturais significativas e que não estejam descritas na solução de referência serão consideradas incorretas. Existem diversas maneiras de se amenizar esta situação e entre elas destacamos: a inclusão de um módulo de inferência, capaz de associar a solução do aluno com os principais pontos do enunciado; a ativação de um processo de análise de entrada e saída de valores caso o diagnosticador automático considere a solução incorreta.

5) A AVALIAÇÃO DO ASIMOV

Neste capítulo apresentaremos uma avaliação empírica do potencial e das limitações do método de modelagem GRADE e do sistema *Asimov*. Os objetivos principais da avaliação do sistema são: 1) verificar se o método de modelagem GRADE é uma forma adequada de se representar soluções de referência e 2) validar a interface e as mensagens tutoriais emitidas. Para realizar tal avaliação, foram consultados dois especialistas no ensino de algoritmos e programação que utilizaram o GRADE para modelar soluções para dois enunciados. Estas soluções de referência foram aplicadas no sistema para a validação de quatro situações possíveis referentes a soluções corretas e incorretas de alunos.

5.1) Enunciados da Avaliação

Os enunciados de exercícios escolhidos para a avaliação do sistema fazem parte do início de um currículo de disciplinas de ensino de programação. Eles englobam os conceitos de armazenamento de valores em variáveis, seqüências de comandos, blocos de comandos,

entrada e saída de dados, utilização de variáveis em expressões aritméticas e repetição. Ambos os exercícios(figura 9) são bastante significativos dentro de seus tópicos e, apesar de suas resoluções serem razoavelmente simples, abrangem uma grande variação de possíveis soluções corretas.

Figura 9 – Enunciados propostos aos especialistas

Exercícios de Programação:

1) Construir um programa que leia dois números quaisquer, calcule e mostre a sua soma e multiplicação.
2) Desenvolver um programa que mostre os números de 1 a 100 utilizando comandos de repetição.

5.2) Formação dos Especialistas

Para a avaliação do sistema foram consultados dois especialistas na área de ensino de programação imperativa de computadores. O especialista 1 é professor da disciplina de algoritmos e estruturas de dados, na qual o aluno aprende programação com uma pseudo-linguagem, sem qualquer auxílio do computador. O especialista 2 têm maior experiência no ensino de programação do ponto de vista prático, ou seja, em laboratório, com o uso de computadores e ferramentas de programação. Para realizar a avaliação, foram utilizados os seguintes passos, separadamente para cada especialista: 1) apresentação do sistema ao especialista; 2) explicação da modelagem de soluções de referência utilizando o GRADE; 3) proposição dos enunciados da seção 5.1; 4) modelagem de soluções de referência pelos

especialistas; 5) simulação da entrada das soluções imitando o comportamento do sistema *Asimov*.

5.3) Soluções de Referências dos Especialistas

Como os dois especialistas são da área de informática e estão acostumados com os aspectos de programação e de modelagem de sistemas, nenhum deles teve grande dificuldade em entender o GRADE e em modelar as soluções de referência para os enunciados propostos (**ver** Anexos III e IV). Com apenas uma explicação inicial, ambos não demoraram mais do que trinta minutos para elaborar as soluções de referência para os dois enunciados. Neste ponto foi identificado mais um tipo de solução possível para um problema de programação: a solução "típica". Esta solução representa o modelo mental de um instrutor humano, ou seja, a solução que ele desenvolveria se fosse confrontado com o problema. Os dois especialistas iniciaram a modelagem da resposta de cada exercício com a sua solução "típica" e com base nela, desenvolveram as demais variações.

5.4) Testes com Variações de Programas

Após a modelagem das soluções de referência, elas foram testadas no sistema *Asimov* com quatro diferentes variações de programas: duas variações corretas e duas incorretas. As variações corretas e incorretas são diferentes entre si no que diz respeito ao que se espera do sistema em relação a elas. A solução 1 está correta, não é a solução "típica" (existem diversas variações em comparação com a solução de referência), embora

siga a mesma estrutura da solução de referência e o sistema deve considerá-la correta. A solução 2 também está correta, sua estrutura difere em um ou mais pontos da estrutura da solução de referência e o sistema deve considerá-la incorreta. A solução 3, apesar de estar incorreta, está bem próxima de uma solução correta identificada na solução de referência e o sistema deve considerá-la incorreta. Finalmente, a solução 4 também está incorreta e sua estrutura difere fundamentalmente da estrutura da solução de referência e, portanto, o sistema deve considerá-la incorreta.

O sistema reagiu de maneira positiva para todas as variações esperadas, tanto para as soluções aceitas como para as não aceitas. Para exemplificar esta situação apresentamos na figura 10, as soluções esperadas para o GRADE do exercício 1 construída pelo especialista 1 e as respectivas mensagens tutoriais emitidas, considerando-se a granulação por programa. As demais soluções dos dois especialistas e as respectivas mensagens emitidas pelo sistema podem ser encontradas nos anexos III e IV.

Figura 10 – Soluções esperadas e mensagens emitidas

	1	2	3	4
Soluções	prod := 1 leia(n1) leia(n2) soma := n1 + n2 prod := prod * n1 prod := prod * n2 imprima(prod) imprima(soma)	leia(n1) leia(n2) n1 := n1 + n2 imprima(n1) n1 := (n1-n2) * n2 imprima(n1)	prod := 1 leia(n1) leia(n2) soma := 0 soma := soma + n1 prod := prod * n1 prod := prod * n2 imprima(prod) imprima(soma)	soma := n1 soma := n2 soma := soma + soma imprima(soma) prod := n1 prod := n2 prod := prod*prod imprima(prod)
Mensagens	Nenhuma. Solução aceita.	Erro na linha 3! Você deveria usar soma no lugar de n1: soma := n1 + n2!	Erro na linha 9! Faltou soma:=soma+n2 antes deste comando!	Erro na linha 1! Inicialização da variável soma está incorreta!

Quanto a soluções corretas (1 e 2 da Figura 10), temos as situações descritas a seguir. Na solução 1 temos diversas variações com relação a uma solução considerada como "típica". Portanto, a solução é aceita pelo sistema já que as dependências modeladas permitem este tipo de variação. A solução 2, apesar de estar correta, não é aceita pelo sistema, pois ela não existe dentro do GRADE deste enunciado. O sistema tenta realizar um encaixe e pára na terceira linha do programa, local onde o encaixe não é mais possível: foi encontrado o comando *n1 := n1 + n2* mas o sistema esperava *soma := n1 + n2*. Para o sistema *Asimov*, o comando está incorreto já que a variável *n1* está sendo usada incorretamente no lugar de *soma*.

Ambas as soluções incorretas (3 e 4 da Figura 10) foram consideradas como tal pelo sistema, indicando que, na comparação com a solução de referência, foram encontradas divergências entre as soluções. A solução de número 3 está praticamente correta, sendo que o único problema é a falta da variável *n2* somada com a variável *soma*. O sistema captura corretamente este erro, indicando, além da mensagem, uma possível localização para o comando que está ausente. Esta identificação é possível, pois o comando *imprima(soma)* tem dependência em relação ao comando *soma := soma + n2*. A última solução está completamente incorreta, não sendo possível fazer qualquer tipo de consideração a seu respeito. Na tentativa de encaixe, o sistema identifica que a variável soma está sendo inicializada com um valor incorreto de acordo com o que está representado no GRADE do enunciado proposto.

5.5) Pontos Principais da Avaliação

Embora o sistema *Asimov* não tenha sido avaliado com estudantes reais, os especialistas que participaram deste experimento consideraram as ferramentas e as mensagens tutoriais emitidas como sendo bastante úteis para o apoio ao aprendizado de programação. Os resultados obtidos da simulação de um aluno usando o sistema descrito na seção anterior, demonstrou que o sistema comporta-se apropriadamente como um guia que evita desvios em relação a soluções corretas. Aliado a estes resultados, o modelo utilizado para representar soluções de referência (GRADE) foi julgado simples e eficaz pelos especialistas. Portanto, o sistema pode ser admitido como uma ferramenta adequada para apoiar estudantes durante o desenvolvimento de suas soluções. Mesmo o fato do diagnosticador considerar incorretas algumas soluções válidas foi considerada uma característica positiva pelos especialistas, já que restringe o escopo de soluções e possibilita maior controle sobre o tipo de algoritmo que o aluno deve treinar.

Quanto às limitações do sistema, foi considerado pelos especialistas que determinados elementos tidos como absolutos, poderiam ser ligados e/ou desligados, tais como: uso de sinônimos e rigidez sintática da linguagem-alvo. Sugeriu-se também que o sistema pudesse restringir a solução do aluno, permitindo que fossem digitadas respostas contendo apenas os comandos já explicados pelo professor em aula. Para fortalecer a capacidade tutorial do sistema de guiar o aluno, foi proposta a apresentação, após a solução do aprendiz ter sido digitada, da solução "típica", mais indicada para o enunciado. Além

disso, foi identificada, por um dos especialistas, a ausência de um elemento importante no GRADE: a possibilidade de se indicar que um determinado comando é opcional. Esta capacidade tornaria possível a inserção de comandos que seriam ignorados caso existissem na solução do aluno, mas que são bastante utilizados, como inicializações de variáveis lidas ou calculadas e exibição de mensagens do próprio programa.

6) CONCLUSÃO E TRABALHOS FUTUROS

No desenvolvimento do sistema *Asimov* procurou-se reunir diversas características positivas de sistemas já desenvolvidos em um único ambiente com características flexíveis e com independência de domínio para auxiliar alunos iniciantes na aquisição de princípio e perícia de programação. Os pontos que merecem destaque em relação ao que foi desenvolvido e alcançado são: um modelo flexível de tutoramento, o diagnóstico automático de soluções, a independência de domínio e um ambiente de descoberta guiada.

O modelo flexível de tutoramento possibilita a variação da granulação do *feedback*, isto é, a periodicidade com que o sistema fornecerá apoio tutorial pode ser alterada. Esta flexibilidade permite que o sistema seja utilizado em variadas situações de ensino/aprendizagem, por alunos com diferentes graus de experiência em programação. Por exemplo, no início de um curso de programação pode ser escolhida a granulação mais restritiva (por *token*) para evitar que os alunos mais inexperientes desviem-se demasiadamente da solução correta e, com o desenrolar dos assuntos, liberar-se o sistema

gradativamente (granulação por comando, expressão e bloco) até que os alunos tenham conhecimento suficiente para precisar de apoio tutorial apenas ao final do desenvolvimento da solução (granulação por programa).

Um instrutor, ao ensinar programação, geralmente considera dois aspectos relevantes com relação às respostas para problemas que esperam dos alunos: não existe apenas uma solução correta e nem todas as soluções corretas podem ser aceitas como respostas para o problema. O sistema *Asimov* atende a ambas as expectativas através do diagnóstico automático de programas guiado por uma solução de referência que é representada pelo grafo de alternativas e dependências (GRADE). O GRADE possibilita a integração de diversas soluções em uma rede semântica que é utilizada pelo sistema para validar as respostas de alunos. Existem dois tipos de variações possíveis: dependência e alternativa. A dependência entre comandos permite que uma mesma solução possa ser escrita de várias maneiras diferentes e demonstra que em uma solução correta existem situações onde a ordem das linhas não interfere no objetivo final. Assim, se uma seqüência de comandos possuir o mesmo grau de dependência ela pode ser escrita em qualquer ordem sem prejudicar o resultado do programa. A capacidade do GRADE de manipular alternativas, permite que várias soluções sejam representadas dentro de um mesmo enunciado. Desta maneira, o professor pode criar enunciados cujas soluções pertençam a uma determinada faixa de algoritmos que devem ser avaliados naquele momento de aprendizado do aluno. Soluções que estejam corretas, mas que não estejam de acordo com o critério do professor não serão aceitas, simplesmente porque não foram representadas na solução de referência. Portanto, o GRADE acrescenta uma boa dimensão de variações a

esquemas de *model-answer* tradicionais, onde apenas uma faixa restrita de soluções são consideradas como corretas e não podem ser alteradas.

A independência de domínio permite que o professor altere o conteúdo do sistema, ou seja, a linguagem-alvo e os enunciados dos problemas. Esta característica de mudança de conteúdo torna o sistema uma *shell* para a montagem de sessões de tutoramento com domínios diferentes. Turmas com diferentes graus de experiência e até mesmo com diferentes linguagens de programação como alvo de estudo podem se beneficiar das características tutoriais do sistema, sem que exista a necessidade de reprogramação a cada nova turma ou a cada nova mudança.

O ambiente de descoberta guiada (STI + micromundos) oferece uma rica oportunidade para aprendizagem, pois engloba a exploração livre e a programação guiada. A ligação entre estas características ser eficaz para o aprendizado, já que reúne o auxílio a aquisição de princípio e perícia em um único sistema. O sistema *Asimov* possui o modo de exploração livre, onde o aluno pode digitar a sua solução e utilizar um compassador de execução para entender o controle de fluxo de um programa e um visualizador de células de memória para a percepção da mudança do estado das variáveis. Além disso, temos o modo de exploração guiado, onde somado às características do modo livre, mensagens tutoriais são fornecidas pelo diagnosticador automático de programas caso o aprendiz cometa algum erro de lógica.

O sistema *Asimov* pode ser melhorado em diversos aspectos de acordo com o que foi explicado em vários trechos do texto, porém, mudanças significativas serão alcançadas se dois pontos principais forem estudados: 1) o aumento da capacidade de diagnóstico e 2) a

extensão da *shell* para uma ferramenta de autoria. O primeiro item tem relação com o entendimento do enunciado por parte do sistema e o segundo com a possibilidade de tornar o sistema o mais amigável possível para os autores.

Na versão atual, o sistema *Asimov* não tem a capacidade de entender a relação existente entre a solução para um problema e o seu enunciado. Uma das implicações disto é que ele considera algumas soluções corretas como sendo incorretas, já que o seu único parâmetro de comparação é a solução de referência. Além disso, o fato do sistema considerar soluções corretas como incorretas não tem como causa única a falta de compreensão do problema. Assim, uma solução para este problema deveria ser melhor estudada e trabalhada para capturar todos os tipos de variações que o espectro humano consegue perceber.

Uma extensão bastante útil do sistema seria um estudo para expansão do modelo de descrição de enunciados (GRADE) no sentido de identificar o relacionamento existente entre determinado trecho da solução de referência com partes do enunciado proposto. Além de possibilitar a detecção de soluções corretas com maior precisão, esta expansão também permitiria que as mensagens tutoriais fossem montadas com uma maior precisão de significado para o aluno.

Apesar do sistema ser flexível e possuir independência em relação ao domínio estudado, possíveis mudanças da linguagem e dos enunciados são todas feitas através de alterações em arquivos texto(**ver** Anexo I), alterações estas, que não são nem muito simples nem amigáveis. A composição de novas ferramentas de autoria para serem associadas e acopladas ao sistema, que atualmente é uma *shell*, seria outra direção futura de pesquisa que

traria ganhos significativos para o processo de apoio ao ensino de programação. Dentre as ferramentas de autoria necessárias as mais úteis seriam: 1) a criação de um editor de soluções de referência para a montagem do GRADE com facilidades gráficas e um banco de dados com trechos de algoritmos (clichês) disponíveis para pronto uso e 2) a construção de um editor de gramáticas de linguagens imperativas que ofereceriam não só um ambiente amigável de trabalho, mas também poderiam guiar o professor na tarefa de desenvolvimento de gramáticas para as variações desejadas de linguagens imperativas.

ANEXO I – Formatos de arquivos do sistema *Asimov*

O sistema *Asimov* permite a alteração do seu domínio (linguagem e enunciados) em sua versão inicial, através da mudança de valores em arquivos do tipo texto: 3 para a mudança da linguagem e 1 para a inclusão de novos exercícios. Os três arquivos para a mudança da linguagem-alvo são baseados no trabalho de Setzer (SETZER, V. W.; MELO, I. S. H., 1983) e englobam as seguintes estruturas:

1) Tabela de símbolos,

   Código, descrição do símbolo, tipo (SMB para símbolo, PR para palavra reservada, FNC para função pré-definida, ID para identificador e NUM para constante numérica) e sinônimos.

2) Tabela de não terminais e

   Código, descrição do não terminal, início do não terminal no grafo sintático.

3) Grafo sintático da gramática.

   Código, 1 para terminal e 0 para não terminal, código do elemento na tabela correspondente, próximo elemento da gramática, alternativa a este elemento, código da função semântica a ser executada se este nó for reconhecido.

O arquivo de exercícios é a codificação da representação das soluções de referência utilizando o GRADE e é composto de duas partes: área de enunciado e área da solução de referência.

Formato do arquivo:

Área de enunciado@<conjunto de nós do GRADE>

Cada nó: Código, comando, lista de parâmetros, lista de depêndencias, alternativa, próximo, mensagem de erro.

Apenas os campos código e comando são obrigatórios.

Obs: O separador padrão para os elementos dos arquivos é a tabulação, a não ser que exista indicação em contrário.

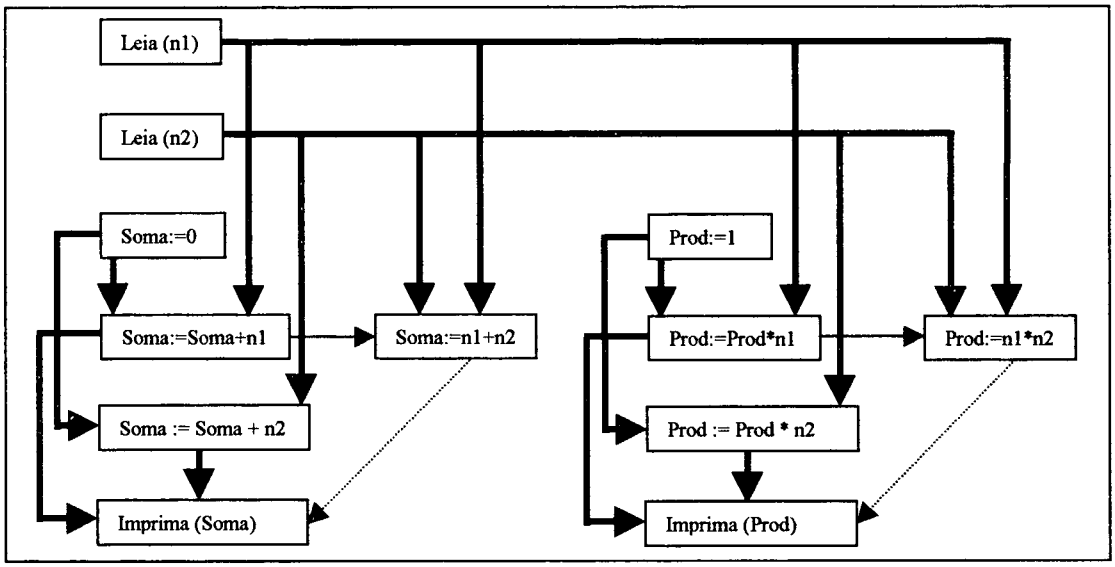
ANEXO II – Resultado do estudo de identificação de erros em provas de alunos iniciantes

Com o objetivo de identificar os principais erros de lógica cometidos por alunos iniciantes, foram analisadas 66 provas de 2 turmas de alunos de um curso de algoritmos. Estas provas versavam sobre o uso de comandos de condição e repetição. Os resultados obtidos encontram-se na tabela abaixo, com os tipos de erro na coluna da esquerda e a quantidade de provas com este tipo de erro na coluna da direita.

Tipo de Erro	Quantidade de Provas
Erros em expressões aritméticas	26
Colocação de comandos em posições erradas relação a repetição ou condição	21
Erros na inicialização de variáveis	20
Uso de variável sem inicialização ou leitura	19
Erro na definição de limites de repetição	12
Uso incorreto de parênteses	11
Uso da variável incorreta em condição	10
Leitura de variável que deveria ser calculada	8
Troca de e por ou	6
Falta de leitura dentro de repetição	5
Uso de variável incorreta para armazenamento	3
Inversão da posição de comandos	3
Não entendimento do funcionamento de um programa (sem sentido)	3
Esquecimento de repetição	2
Operador relacional errado	2
Comando inútil (atribuição de variável para ela mesma)	2
Leitura de mais variáveis que o necessário	1
Troca de comandos (repetição por condição ou vice-versa)	1
Leitura de variável que não é usada	1
Inversão de atribuição	1
Erro em comando de armazenamento	1
Esquecimento do senão	1

ANEXO III – GRADE produzido pelo especialista 1

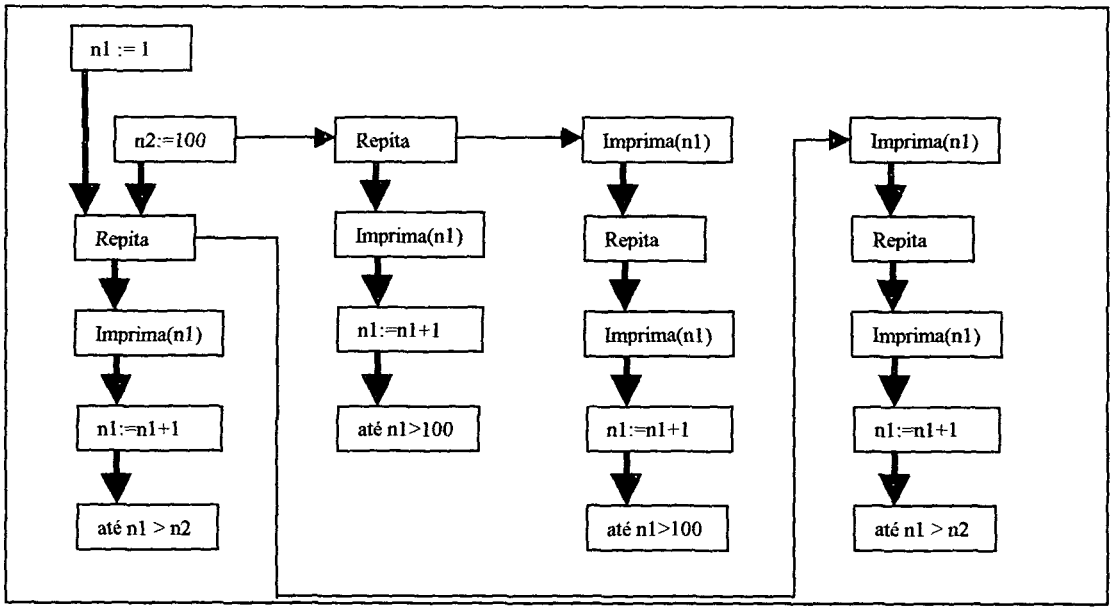
GRADE para o exercício 1:



Soluções esperadas e mensagens para o exercício 1:

	1	2	3	4
Soluções	prod := 1 leia(n1) leia(n2) soma := n1 + n2 prod := prod * n1 prod := prod * n2 imprima(prod) imprima(soma)	leia(n1) leia(n2) n1 := n1 + n2 imprima(n1) n1 := (n1-n2) * n2 imprima(n1)	prod := 1 leia(n1) leia(n2) soma := 0 soma := soma + n1 prod := prod * n1 prod := prod * n2 imprima(prod) imprima(soma)	soma := n1 soma := n2 soma := soma + soma imprima(soma) prod := n1 prod := n2 prod := prod*prod imprima(prod)
Mensagens	Nenhuma. Solução aceita.	Erro na linha 3! Você deveria usar soma no lugar de n1!	Erro na linha 9! Faltou soma:=soma+n2 antes deste comando!	Erro na linha 1! Inicialização da variável soma está incorreta!

GRADE para o exercício 2:

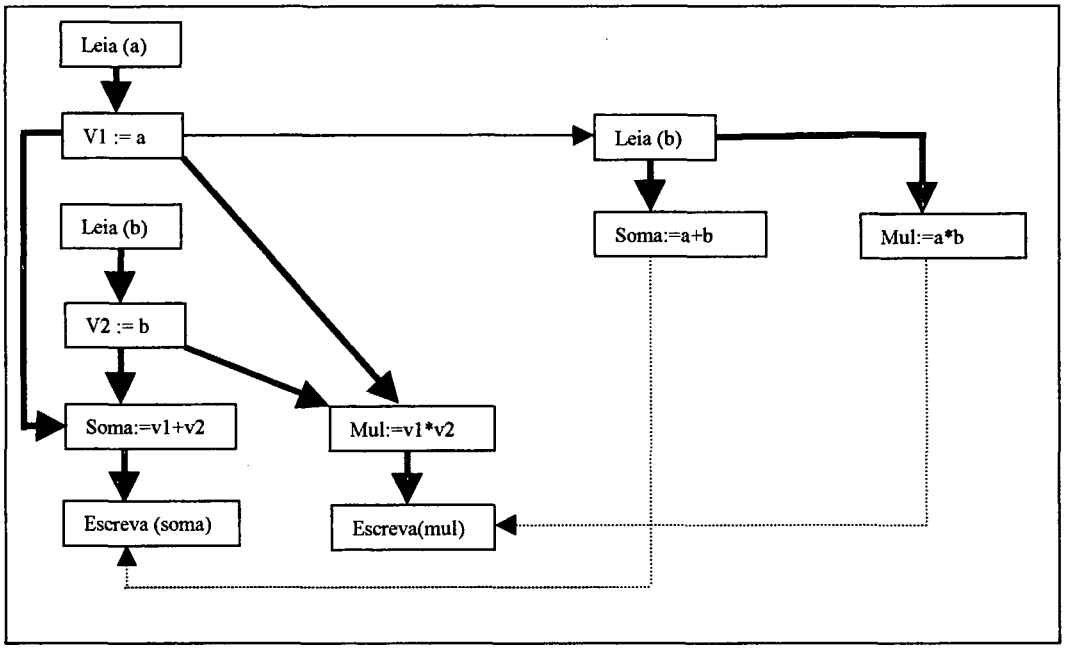


Soluções esperadas e mensagens para o exercício 2:

	1	2	3	4
Soluções	lim := 100 i := 1 repita imprima(i) i := i + 1 até i > lim	i := 0 repita i := i + 1 imprima(i) até i >= 100	lim := 100 i := 1 repita imprima(i) i := i - 1 até i > lim	i := 1 repita i := i + 1 imprima(i) até i <= 0
Mensagens	Nenhuma. Solução aceita.	Erro na linha 1! Inicialização da variável i está incorreta!	Erro na linha 5! Faltou i := i + 1 antes deste comando!	Erro na linha 3! Faltou imprima(i) antes deste comando!

ANEXO IV – GRADE produzido pelo especialista 2

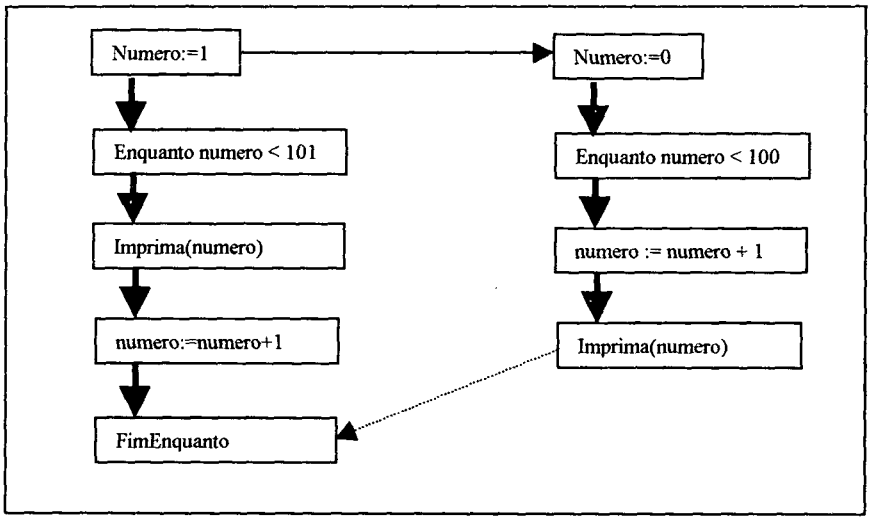
GRADE para o exercício 1:



Soluções esperadas e mensagens para o exercício 1:

	1	2	3	4
Soluções	leia(n1) leia(n2) soma := n1 + n2 prod := n1 * n2 imprima(prod) imprima(soma)	leia(n1) leia(n2) n1 := n1 + n2 imprima(n1) n1 := (n1-n2) * n2 imprima(n1)	leia(n1) leia(n2) soma := n1 + n2 imprima(prod) imprima(soma)	soma := n1 soma := n2 soma := soma + soma imprima(soma) prod := n1 prod := n2 prod := prod*prod imprima(prod)
Mensagens	Nenhuma. Solução aceita.	Erro na linha 3! Você deveria usar soma no lugar de n1!	Erro na linha 4! Faltou prod := n1 * n2 antes deste comando!	Erro na linha 1! Inicialização da variável soma está incorreta!

GRADE para o exercício 2:



Soluções esperadas e mensagens para o exercício 2:

	1	2	3	4
Soluções	i := 1 enquanto i < 101 imprima(i) i := i + 1 FimEnquanto	i := 1 enquanto i <= 100 imprima(i) i := i + 1 FimEnquanto	i := 1 enquanto i < 101 imprima(i) i := i - 1 FimEnquanto	i := 1 enquanto i < 101 i := i + 1 imprima(i) FimEnquanto
Mensagens	Nenhuma. Solução aceita.	Erro na linha 2! Use enquanto i < 101 no lugar deste comando!	Erro na linha 4! Faltou i := i + 1 antes deste comando!	Erro na linha 3! Faltou imprima(i) antes deste comando!

REFERÊNCIAS BIBLIOGRÁFICAS

ADAM, A.; LAWRENT, J. **A system to debug students programs.** In: **Journal of Artificial intelligence**. v. 15, 1980. p. 75-122.

ANDERSON, John R. **Acquisition of cognitive skill.** In: **Psychological review.** n. 89, 1982. p.369-406.

BARR, A.; BEARD, M.; ATKINSON, R. C. **The computer as a tutorial laboratory:** the Stanford BIP project. In: **International journal of man-machine studies**, v. 8, p. 567-595.

BONAR, Jeffrey R.; CUNNINGHAM, Robert. **Bridge** : tutoring the programming process. 1985. p. 409-434.

BONAR, J.; SOLOWAY, E.. **Pre-programming knowledge:** A major source of misconceptions in novice programmers. In: **Human-Computer Studies in Mathematics 20**. v.20, 1985. p. 293-316.

BURTON, Mike; SHADBOLT, Nigel. **Pop-11 programming for artificial intelligence.** Great Britain : Addison-Wesley, 1987.

DIRENE, Alexandre I. **Methodology and tools for designing concept tutoring systems.** In: **The Third White House Papers.** CSRP 172, School of Cognitive and Computing Sciences - University of Sussex, 1990.

DIRENE, Alexandre I. *et al.* **Sistemas tutoriais para assistir o treinamento de operação de centrais de comutação.** In: **Simpósio Brasileiro de Informática na Educação**. Belo Horizonte, 1996.

du BOULAY, Benedict; SOTHCOTT, Christopher. **Computers teaching programming: an introdutory survey of the field**. In: LAWLER, R. W.; YAZDANI, M. **Artificial intelligence and education: learning environment and tutoring systems.** v. 1, 1987. cap. 16, p. 345-372.

GOLDSTEIN, I. P. **Summary of Mycroft: a system for understanding simple picture programs.** In: **Artifical intelligence,** v. 6, 1975, p. 249-288.

HASEMER, T. **An empirically-based debugging system for novice programmers.** In: **Human cognition research laboratory.** Open University, 1983.

JOHNSON, W. Lewis ; SOLOWAY, Elliot. **PROUST: an automatic debugger for pascal programs.** In: KEARSLEY, Greg. **Artificial intelligence in education**: applications and methods. Addison Wesley, 1987. cap. 3, p. 49 - 67.

LUKEY, F. J. **Understanding and debugging programs.** In: **International journal of man-machine studies.** v. 12, 1980, p. 189-202.

MAJOR, Nigel. **Using COCA to build an intelligent tutoring system in simple algebra.** In: **Intelligent tutoring media.** v. 2, n. 3/4, p. 159-169, 1991.

MAJOR, Nigel; AINSWORTH, Shaaron; WOOD, David. **REDEEM: Exploiting symbiosis between psychology and authoring environments.** In: **International journal of artificial intelligence in education.** UK: International AIED Society, v. 8, n. 3/4, p. 317-340, 1997.

MURRAY, Tom. **Expanding the knowledge acquisition bottleneck for intelligent tutoring systems.** In: SELF, John. **International journal of artificial intelligence in education**: authoring systems for intelligent tutoring systems. UK: International AIED Society, v. 8, n. 3/4, p. 222-232, 1997.

NICOLSON, R. I.; SCOTT, P. J. **Computers and education: the software production problem.** In: **British journal of educational technology.** v. 17, n. 1, p. 26-35, january 1986.

PIMENTEL, Andrey Ricardo; DIRENE, Alexandre Ibrahim. **Medidas cognitivas no ensino de programação de computadores com sistemas tutores inteligentes.** In: **Anais do Simpósio Brasileiro de Informática na Educação.** 1998.

QUILLIAN, R. **Semantic memory.** In: **Semantic information processing.** Ed. M. Minsky, Cambridge, MA, MIT Press, 1968.

RAMADHAN, Haider; du BOULAY, Benedict. **Programming environments for novices.** In: LEMUT, Enrica. **Cognitive models and intelligent environments for learning programming.** Springer Verlag, p. 125-134.

REISER, B.; KIMBERG, D.; RANNEY M. **Knowledge representation and explanation in GIL, na intelligent tutor for programming.** In: **Cognitive science laboratory report.** NJ, USA: Princeton University, 1988.

RICH, Elaine; KNIGHT, Kevin. **Inteligência artificial.** 2. ed. São Paulo : Makron Books, 1993.

SETZER, Valdemar W.;MELO, Inês Homem de. **A construção de um compilador.** Rio de Janeiro : Campus, 1983.

WENGER, Etienne. **Artificial intelligence and tutoring systems:** Computational and cognitive approaches to the communication of knowledge. California : Morgan Kaufmann, 1987.

WOOLF, Beverly. **Representing, acquiring, and reasoning about tutoring knowledge.** In: BURNS, Hugh *et al.* **I.T.S.: evolutions in design.** Hillsdale, New Jersey : Lawrence Erlbaum Associates, Publishers, 1991.