UNIVERSIDADE FEDERAL DA PARAÍBA - UFPb
CENTRO DE CIÊNCIAS E TECNOLOGIA - CCT
COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA - COPIN

# UMA FERRAMENTA PARA VERIFICAÇÃO INTERATIVA DE PROGRAMAS
# V I P

Francilene Procópio Garcia

Campina Grande, PB
Abril de 1994.

FRANCILENE PROCÓPIO GARCIA

# UMA FERRAMENTA PARA VERIFICAÇÃO INTERATIVA DE PROGRAMAS VIP

Dissertação apresentada ao curso de MESTRADO EM INFORMÁTICA da Universidade Federal da Paraíba como parte dos requisitos necessários à obtenção do grau de Mestre.

ÁREA DE CONCENTRAÇÃO: CIÊNCIA DA COMPUTAÇÃO

MARIA DE FÁTIMA CAMÊLO
Orientadora

Campina Grande, PB
Abril de 1994.

G216f Garcia, Francilene Procópio.
Uma ferramenta para verificação interativa de programas VIP / Francilene Procópio Garcia. - Campina Grande, 1994.
109 f.

Dissertação (Mestrado em Informática) - Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, 1994.
Referências.
"Orientação : Profa. M.Sc. Maria de Fátima Camêlo".

1. Programação. 2. Ambientes CAI. 3. Verificador Interativo de Programas - VIP. 4. Dissertação - Informática. I. Camêlo, Maria de Fátima. II. Universidade Federal da Paraíba - Campina Grande (PB). III. Título

CDU 004.43(043)

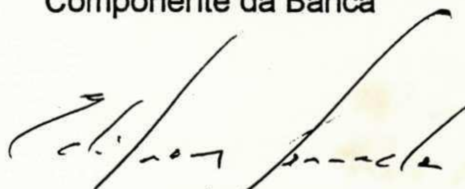
UMA FERRAMENTA PARA VERIFICAÇÃO INTERATIVA DE PROGRAMA.

FRANCILENE PROCÓPIO GARCIA

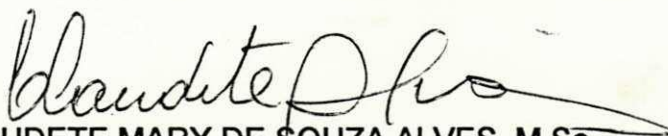
DISSERTAÇÃO APROVADA EM 20.04.1994

MARIA DE FÁTIMA CAMELO, M.Sc
Presidente

JOSÉ ANTÃO BELTRÃO MOURA, Ph.D
Componente da Banca



EDILSON FERNEDA, Dr.
Componente da Banca



CLAUDETE MARY DE SOUZA ALVES, M.Sc
Componente da Banca

Campina Grande, 20 de abril de 1994

GARCIA, FRANCILENE P.

Uma Ferramenta para Verificação Interativa de Programas - VIP (Campina Grande), 1994.

xi, 113 p. 27,9 cm (COPIN/UFPb, Mestrado em Informática. 1994).

Dissertação - Universidade Federal da Paraíba, COPIN

1. Verificador Interativo de Programas Pascal
2. Detecção de Erros Lógicos
3. Exploração de Erros Cometidos por Iniciantes
4. Representação de Conhecimento sobre Programação
5. Ensino de Programação Auxiliado por Computador

I. COPIN/UFPb II. TÍTULO (Série)

"... Dentro de mim quase existo
Quase tudo
Quase aquilo
Quase isso ..."
Caetano Veloso

Aos meus pais,
pela perseverança na crença de que a **educação** é,
em todos os sentidos,
a mola mestra da sociedade.

À Telmo.
pelo apoio e imenso carinho.
Este é mais um dos momentos que
compartilhamos juntos.

## AGRADECIMENTOS

Fátima Camêlo, amiga, orientadora presente.

Colegas do DSC, pacientes.

Aninha, atenciosa.

Maísa, Érica, Alessandra,
e toda a equipe de desenvolvimento
do DOCET, dedicados.

Colegas do PaqTc-Pb e
amigos, torcedores.

A todos, minha gratidão.

Resumo da Dissertação Apresentada ao Curso de Mestrado em Informática como parte dos requesitos necessários para a obtenção do grau de Mestre.

## UMA FERRAMENTA PARA VERIFICAÇÃO INTERATIVA DE PROGRAMAS - VIP

Francilene Procópio Garcia

Abril, 1994

O objetivo central deste trabalho é o desenvolvimento de uma ferramenta para análise de programas criados por iniciantes em programação. A verificação dos programas encontra-se voltada à identificação e correção de erros lógicos, cujas causas são oriundas de desvios, parcial ou integral, da especificação do problema.

A ferramenta proposta é parte do desenvolvimento de um ambiente para o ensino de programação - DOCET. Neste contexto, foi considerada a problemática relacionada ao ensino de programação com ênfase à formação de programadores mais qualificados.

São relacionadas algumas das metodologias usadas no ensino de programação sob a orientação de ambientes CAI (Computer Aided Instruction), especialmente àquelas voltadas para estudantes iniciantes - como se propõe o DOCET.

A descrição das principais características do Verificador Interativo de Programas - VIP completa este trabalho.

Abstract of Dissertation presented to COPIN/UFPb as partial fulfillment of the requirements for the degree of Master.

Francilene Procópio Garcia
April, 1994

The principal aim of this work is the software implement for the analysis of beginner created computer programs. The program checks are oriented to the identification and correction of logic errors caused by partial or total misconception in problem specification.

The proposed implement is part of the development of an environment for programming apprentice - DOCET. In this context, problems concerning programming education were analyzed in order to find a solution for the graduation of more qualified programmers.

Some methodologies used in programming tutorials following the conception of Computer Aided Instruction (CAI) are presented, specially those oriented to beginners, as it is proposed for the DOCET.

The exposition of the main characteristics of the Interative Program Checker (VIP) completes this work.

# SUMÁRIO

CAPÍTULO I: APRESENTAÇÃO ..... 2

CAPÍTULO II: O ENSINO DE PROGRAMAÇÃO ..... 5
II.1 RESOLUÇÃO DE PROBLEMAS ..... 6
II.2 A FORMULAÇÃO DO ENSINO ..... 8
II.3 A PRODUTIVIDADE NO DESENVOLVIMENTO DE SOFTWARE ..... 10
II.4: O COMPUTADOR APOIANDO O DESENVOLVIMENTO DE PROGRAMAS ..... 12
II.5 O COMPUTADOR APOIANDO O ENSINO DE PROGRAMAÇÃO ..... 15
II.5.1 O AMBIENTE PROUST ..... 16
II.5.2 O AMBIENTE BRIDGE ..... 17

CAPÍTULO III: VERIFICADOR INTERATIVO DE PROGRAMAS - VIP ..... 22
III.1 VISÃO GERAL ..... 22
III.2 VIP NO CONTEXTO DO AMBIENTE DOCET ..... 23
III.3 REPRESENTAÇÃO DO CONHECIMENTO SOBRE PROGRAMAÇÃO ..... 26
III.3.1 CONHECIMENTO GENÉRICO SOBRE PROGRAMAÇÃO ..... 29
III.3.2 ESPAÇO DE REPRESENTAÇÃO DAS SOLUÇÕES DOS PROBLEMAS ..... 30
III.3.3 CONHECIMENTO ESPECÍFICO ÀS METAS ..... 37
III.4 SISTEMATIZAÇÃO DA ANÁLISE ..... 40
III.4.1 MONTAGEM DO FORMATO VIP ..... 42
III.4.2 ANÁLISE DO CONHECIMENTO GENÉRICO SOBRE PROGRAMAÇÃO ..... 50
III.4.3 ANÁLISE DO CONHECIMENTO ESPECÍFICO ÀS METAS ..... 55
III.4.4 TRATAMENTO DE ERROS ..... 57
III.4.5 INTERFACE COM O USUÁRIO ..... 60
III.5 METODOLOGIA DE DESENVOLVIMENTO ..... 62
III.5.1 AMBIENTE ESCOLHIDO PARA O DESENVOLVIMENTO ..... 62
III.5.2 DEPURAÇÃO E VALIDAÇÃO ..... 63

CAPÍTULO IV: CONSIDERAÇÕES FINAIS .....: 66

REFERÊNCIAS ..... 68

ANEXOS ..... 71
- ANEXO A ..... 72
- ANEXO B ..... 77
- ANEXO C ..... 87
- ANEXO D ..... 95
- ANEXO E ..... 104

## ÍNDICE DE FIGURAS

Figura 1: DOCET: Ambiente para Ensino e Treinamento de Programação ............ 74
Figura 2: Comunicação do VIP com as ferramentas do ambiente DOCET ............ 24
Figura 3: Estrutura organizacional das soluções no Banco de Problemas. ............ 31
Figura 4: Árvore de soluções para um problema ............ 36
Figura 5: Classes e Objetos encontrados num programa ............ 39
Figura 6: Arquitetura Funcional do VIP ............ 41
Figura 7: Sistematização da Análise ............ 53
Figura 8: Representação do Nível de Embutimento ............ 100
SUBCONJUNTO DA LINGUAGEM PASCAL TRATADO PELO VIP ............ 77
ARQUITETURA INTERNA DOS MÓDULOS DO VIP ............ 104

## ÍNDICE DE TABELAS

Tabela 1: Classificação das Estruturas de Comando por Tipo ............ 33
Tabela 2: Sinônimos Naturais tratados pelo VIP ............ 35
Tabela 3: Representação de Operadores ............ 45
Tabela 4: Representação de Variáveis ............ 46
Tabela 5: Representação de Constantes ............ 47
Tabela 6: Representação de Funções Embutidas ............ 48

# CAPÍTULO I

# APRESENTAÇÃO

# CAPÍTULO I: APRESENTAÇÃO

Atualmente fala-se muito na utilização do computador no processo de ensino-aprendizagem. Explora-se o recurso tecnológico que a máquina, descendente de Von Neumann, representa à disseminação de conceitos em diversas áreas do conhecimento. Particularmente, destacam-se a sua interatividade e a flexibilidade com a qual se pode acompanhar a evolução dos diferentes níveis de aprendizado dos treinandos.

Inserida num projeto de pesquisa mais amplo, desenvolvimento de um ambiente de ensino de programação - DOCET, o objetivo desta dissertação é explorar aspectos relacionados à verificação de programas, desenvolvidos por estudantes de programação, para identificar possíveis erros lógicos. Isto é, erros oriundos do não atendimento, parcial ou integral, dos requesitos presentes na especificação do problema.

Inicialmente é feita uma síntese da problemática relacionada com o ensino de programação, em diferentes contextos, onde a preocupação central é a formação de programadores qualificados. Explora-se um pouco mais as alternativas metodológicas para o ensino de programação, apresentadas sob a concepção de ambientes CAI (Computer Aided Instruction), particularmente àquelas orientadas à iniciantes de programação, como se propõe o DOCET. Em seguida, o VIP é apresentado com a descrição de suas características.

No Capítulo II a temática abordada é o ensino de programação e sua complexa relação com os métodos de resolução de problemas, um aspecto de extrema importância no processo de ensino-aprendizagem de programação. Padrões tradicionais de ensino mais disseminados em salas de aula e literaturas sobre o tema são analisados. Mecanismos que se utilizam dos recursos do computador no processo de ensino são investigados, com uma descrição do Projeto Aprendiz de Programador e de outras ferramentas desenvolvidas para atender ao ensino de iniciantes de programação. Ressaltam-se as características e formas de organizar o conhecimento sobre programação. Em suma, apresenta-se uma discussão sobre mecanismos que viabilizem melhorias à formação de programadores, visando alcançar patamares de produção de software em níveis mais elevados.

As reflexões sintetizadas no Capítulo II auxiliaram fortemente no dimensionamento dos objetivos do Verificador Interativo de Programas Pascal - VIP, diante da pouca disponibilidade de literatura sobre o tema abordado. A experiência obtida no ensino de programação, por docentes da UFPb, observando aspectos cruciais que retardam a maturidade dos estudantes, trouxe contribuições importantes para a especificação do VIP. Alguns dos experimentos realizados em sala de aula são discutidos em diversos momentos dessa dissertação.

No Capítulo III encontra-se uma descrição completa do projeto e implementação do VIP, com a exploração de alguns aspectos de extrema relevância para o seu desempenho, no âmbito do DOCET: sua comunicação com o DOCET, a representação do conhecimento sobre programação, os módulos funcionais, o produto desenvolvido e os resultados obtidos.

Finalizando, no Capítulo IV, são feitas algumas considerações acerca das contribuições trazidas pelo VIP para o ensino introdutório de programação com a formalização de um banco de programas e um outro de erros e, alguns trabalhos já iniciados decorrentes das discussões desenvolvidas no âmbito do VIP. São apresentadas também sugestões para trabalhos futuros.

# CAPÍTULO II

# O ENSINO DE PROGRAMAÇÃO

# CAPÍTULO II: O ENSINO DE PROGRAMAÇÃO

O processo ensino-aprendizagem de programação torna-se complexo por necessitar que sejam colocados em prática conceitos de modelagem e de resolução de problemas. Geralmente, cria-se um cenário, no qual o estudante utiliza o conhecimento relativo às suas experiências em outras atividades, para selecionar o "melhor" método para resolução de um problema. O conhecimento recém adquirido sobre uma determinada linguagem de programação é, então, associado para expressar a solução modelada em uma linguagem entendível pelo computador.

O cenário da construção do programa pode ser palco da atuação de múltiplos planos de resolução, a partir dos quais as soluções podem ser obtidas. Cada estudante de programação pode estabelecer seu próprio plano de resolução. Programar, portanto, é fundamentalmente uma atividade de criação.

Ao tentar resolver um problema, construindo programas, o estudante necessita colocar em prática conhecimentos acerca do domínio do problema e dos mecanismos para armazenamento e manipulação desses conhecimentos na memória de um computador (linguagem e técnicas de programação).

No processo de abstração, inerente à tarefa de criação de programas, observa-se um grau de dificuldade que depende de algumas variáveis, relacionadas com o conhecimento global adiquirido anteriormente e com o conhecimento de programação apreendido pelo estudante:

- a habilidade do estudante no processo de construção de um plano de resolução para o problema;
- o domínio do estudante no manuseio de recursos do computador;
- o enfoque dado no ensino de programação: codificação ou resolução de problemas ;
- o domínio do estudante sobre as regras sintáticas e semânticas da linguagem utilizada;

- o domínio do estudante sobre o uso de técnicas e métodos de programação.

O ensino de programação alcança melhores resultados quando são enfatizadas algumas das técnicas de resolução de problemas: elaboração de um plano de resolução, descrição e implementação de metas específicas, sequenciamento do plano, execução e avaliação do plano.

Aspectos relevantes à formação de programadores cada vez mais produtivos, tema que tem motivado o surgimento de pesquisas e reflexões em vários centros ao redor do mundo, são abordados no desenvolvimento desse capítulo.

## II.1 RESOLUÇÃO DE PROBLEMAS

"O cérebro humano opera por associação. O homem não pode esperar duplicar totalmente seu processo mental artificialmente, mas ele certamente deve ser capaz de aprender com ele. Não se pode esperar equiparar à velocidade e flexibilidade com a qual o cérebro segue o caminho da associatividade, mas seria possível ultrapassar o cérebro ao observar a permanente clareza dos elementos emanados do armazenamento interno." [BUSH 45].

Para resolver um problema, necessita-se de um conjunto de conhecimentos previamente adquiridos ou acessíveis em alguma mídia. Por exemplo, o engenheiro tem a seu dispor um acervo de conhecimentos altamente especializados, como a teoria científica sobre a Resistência de Materiais, a sua própria experiência e a grande massa de experiência profissional acumulada na literatura técnica especializada.

Na atividade de programação, os conhecimentos necessários e os conceitos utilizados são complexos e não muito bem definidos. A habilidade é obtida a partir da prática. O uso excessivo da intuição, associada a uma gama de conhecimentos acumulados ao longo do tempo, resulta num caminho aleatório para criação de programas, o que não é conveniente para uma boa qualificação dos programas. A associação de dicas e instruções sobre como

programar, fundamentadas em conceitos mais formais, tem sido importante na fixação de conhecimentos que são colocados em prática. Por exemplo, o entendimento sobre a funcionalidade de estruturas de seleção ou de repetição condicional.

A investigação da heurística embutida na tarefa de programação, tem auxiliado também no conhecimento de operações mentais aplicadas ao processo de resolução de um problema. Esse conhecimento, se bem usado, pode vir a influenciar de forma benéfica na dinâmica empregada no processo de ensino-aprendizagem de programação [POLY 75].

A experiência acumulada no ensino de programação, em geral, tem mostrado que sugestões e questionamentos, feitos ao longo do processo de ensino-aprendizagem, se usados de modo adequado, ajudam o estudante a identificar situações oportunas para o uso de estruturas, técnicas e métodos de programação, vistos anteriormente. Essas formas de interação têm em comum duas características: bom senso e generalidade. Como se originam no senso comum, muitas vezes surgem naturalmente, refletindo o aprendizado evolutivo do estudante. Por serem genéricas, auxiliam discretamente, indicando a direção geral, deixando quase tudo para o estudante desenvolver.

Quando o estudante consegue resolver um problema que lhe é proposto, alguma melhoria é acrescentada à sua capacidade de resolver problemas. Se, associado ao sucesso obtido, forem-lhe feitos questionamentos genéricos, aplicáveis à muitos outros casos, em número e frequência adequados à cada estudante, dificilmente este deixará de notá-los, e, possivelmente passará a formular, ele próprio, esse questionamento em situações semelhantes.

A capacidade de resolver bem problemas pode ser uma das componentes mais importantes na criação de software. De fato, a maior parte do pensamento consciente relaciona-se com problemas a serem resolvidos. A não ser quando entregues a devaneios ou fantasias, os pensamentos se dirigem para um fim, procurando meios para resolver um problema. Entretanto, necessitamos organizar essa habilidade em torno de princípios menos intuitivos e mais oportunos.

## II.2 A FORMULAÇÃO DO ENSINO

O ensino de programação, muitas vezes, detém-se apenas aos aspectos sintáticos dos comandos de uma linguagem de programação, enfatizando a representação do produto final do processo de programação, afastando-se do processo em si. O estudante, assim, é levado a organizar seu conhecimento em função da sintaxe da linguagem de programação. Como consequência, o plano de resolução obtido é fragmentado pelo uso de conhecimentos sobre comandos da linguagem, permitindo o aparecimento de erros diversos no programa.

A ênfase sobre a sintaxe das linguagens de programação também é observada na maior parte dos livros textos disponíveis, que estruturam seus capítulos a partir da apresentação de novos conceitos sintáticos. A investigação de situações que reaparecem em segmentos de programas diferentes é esquecida. Os programas são apresentados já prontos, usando estruturas padrões, sem reflexões sobre os caminhos que levaram ao projeto e desenvolvimento do programa. A qualidade das decisões tomadas na criação do código é também pouco ou nada discutida.

Outros autores procuram descrever os conceitos de programação usando caixas ou uma forma gráfica qualquer que chame a atenção do estudante. Mas, esquecem de abordar exemplos significativos onde tais conceitos são reutilizados.

Por outro lado, também são utilizadas técnicas que permitem o estabelecimento de planos intermediários isentos das regras sintáticas presentes nas linguagens de programação, como o uso de algoritmos e fluxogramas. Todavia, não se observa uma coerência com as representações mentais do estudante, originadas no momento da resolução do problema.

Ao utilizar fluxogramas, diagramas de decisão e outras técnicas existentes, o instrutor auxilia o estudante ao reduzir, numa etapa inicial, a necessidade de informação envolvida no processo. O estudante, posteriormente, pode vir a representar seu plano de resolução diretamente numa linguagem de alto nível. No entanto, tais técnicas não ensinam a programar. Experimentos realizados demonstram que os estudantes são mais produtivos, em geral, quando não fazem uso de tais métodos [LINN 92].

Alguns instrutores de programação observam que o estudante, ao colocar em prática suas habilidades para resolver problemas, pode descobrir intuitivamente os caminhos mais indicados para a tarefa de programar. Desta forma, o estudante aprende a programar através de descobertas intuitivas (tentativa e erro), desprovidas de heurísticas fundamentadas em orientações de peritos no assunto.

Em alguns cursos de programação, os estudantes investigam programas-exemplos em busca de maturidade em programação. No entanto, um programa pronto nem sempre revela de forma clara o processo de tomada de decisão que está por traz dele. Programadores competentes são capazes de inferir tal conhecimento porque possuem uma base de conhecimento compatível com o nível do programa [WEIS 87].

Planos alternativos de resolução, o uso do código específico, a discussão sobre o funcionamento de cada parte do programa, a descrição da escolha dos dados para teste, são aspectos importantes da programação pouco abordados no ensino tradicional.

A equivocada ênfase no produto em detrimento do processo é reforçada quando se observa que as avaliações de aprendizado, em geral, atribuem nota mais naturalmente ao acompanhamento da execução de um código apresentado. O entendimento real do processo que está por trás da construção daquele código, isto é, a qualidade das decisões tomadas no projeto do programa é relegada a um segundo plano, talvez por ser mais difícil de se avaliar.

Os estudantes necessitam aprender como aplicar conhecimentos genéricos em domínios específicos. A permanência da abordagem centrada na linguagem dificulta a capacitação de bons programadores.

Alguns pesquisadores dedicam-se ao desenvolvimento de representações alternativas de conhecimento para o projeto e implementação de programas. A introdução do computador como ferramenta auxiliar no processo de ensino e aprendizagem apresenta caminhos alternativos, onde parte das tarefas pode ser automatizada e mecanismos mais dinâmicos e interativos podem assessorar o estudante no uso dessas representações [COHE 92, FETZ 88, LINN 92, REPS 87].

No entanto, a abordagem utilizada para introduzir o conhecimento básico sobre a prática

de programação é fundamental para o aprendizado do estudante.

## II.3 A PRODUTIVIDADE NO DESENVOLVIMENTO DE SOFTWARE

A demanda de software tem crescido em intervalos de tempo bem pequenos nos últimos anos. Por outro lado, os níveis de produção de software apresentam quantitativos de código por homem/hora abaixo das expectativas. A menos que esses níveis mudem, uma porcentagem significativa da procura por software não será satisfeita no futuro próximo. Isto não significa que os programadores estão ficando menos produtivos, mas que existe um deficit de bons programadores, que ofereçam alternativas eficientes na produção de software a custos mais baixos.

O crescimento da demanda de produtos de software reflete a busca do aumento de produtividade através do uso da automação e de informática. Cada vez mais, observa-se a busca de capacitação em informática por profissionais de outras áreas.

Pesquisas em andamento [BALZ 85, BARS 85, LUCE 87] apontam para a automação total do processo de construção de software através do uso de ambientes que propiciem, a partir de um conjunto de especificações, a geração automática de programas. No entanto, existem sérias dificuldades a serem superadas, dentre elas, o conceito da especificação do domínio (especialização em excesso) e a programação automática independente do domínio (generalização em excesso).

Outros grupos de pesquisa se dedicam à busca de métodos que permitam a reutilização de código, evitando o "fazer novamente". A utilização de bibliotecas ou de métodos de prototipação orientado ao projeto, apresentam resultados importantes na busca de patamares mais avançados na produção do software [COWL 87, HEND 87].

A construção de ambientes de treinamento que forneçam condições ao programador, mesmo que iniciante, de executar melhor a "tarefa de criação de código" é uma meta importante na busca de um melhor desempenho na produção de software.

Pesquisas em andamento em outros centros [WATE 85] apontam para a construção dos chamados "assistentes de programação". Baseados em conhecimento, eles buscam uma melhor visualização dos requerimentos do software e do projeto em especificação. Alguns desses ambientes auxiliam o programador durante o processo de criação do código, apontando os erros e fornecendo orientações para superação dos mesmos.

Os assistentes de programação baseados em conhecimento, de propósito geral ou voltados para áreas de aplicação, deparam-se com a dificuldade de obter a automação total das funções portadoras de conhecimento intensivo, cujo conhecimento muito especializado depende da intervenção de um especialista. Nestes casos, técnicas de Engenharia de Software são combinadas com técnicas de Inteligência Artificial para prover assistência aos programadores na execução de tarefas complexas. A redução das dificuldades na construção e depuração de um programa é a motivação principal dessa linha de pesquisa. Os iniciantes levariam menos tempo para aprenderem a elaborar, depurar e verificar programas.

Mecanismos poderosos, agregados aos ambientes de programação, melhoram a produtividade dos programadores. Como por exemplo, os recursos de depuração que controlam as estruturas passo-a-passo e, geralmente, ajudam a reduzir os erros que os programadores cometem nas fases de projeto e codificação.

Os paradigmas de programação, modelos de "como programar", auxiliam os programadores no projeto e estruturação de sistemas de software. Os paradigmas definem estilos de pensamento e programação que documentam a distância entre o problema a resolver e o programa - produto final. Diferentes domínios de programação frequentemente demandam diferentes paradigmas. A programação orientada a objetos e a funcional são exemplos de paradigmas poderosos atualmente em destaque.

Alguns estudos citados em [LUCE 87] apontam para definições cautelosas de processos de desenvolvimento de software, que auxiliam a melhorar o produto de software resultante. Modelos como a prototipação são realmente úteis. As atividades realizadas no projeto devem ser representadas e arquivadas, para serem posteriormente recuperadas e manipuladas de forma que se possa raciocinar a posteriori sobre elas.

A compreensão do processo de desenvolvimento de software e a consequente agregação dos resultados de tais pesquisas na criação de novos modelos estão entre os maiores desafios apresentados hoje à comunidade científica de engenharia de software. Beladay referenciado em [LUCE 87] escreveu sobre os impactos que a engenharia de software sofrerá com a mudança na natureza das aplicações de computação associada a um aumento nos tamanhos de códigos produzidos.

A utilização de ambientes automatizados para auxiliar no processo de ensino acerca do desenvolvimento de software, um dos aspectos considerados por Beladay, é o que será enfocado com mais detalhes a seguir.

## II.4: O COMPUTADOR APOIANDO O DESENVOLVIMENTO DE PROGRAMAS

A utilização de computadores no apoio ao desenvolvimento de software tem sido justificado por três razões fundamentais. Primeira, parte dos esforços envolvidos no desenvolvimento de software consiste em tarefas que podem vir a ser automatizadas.

Segunda, um dos problemas mais frustantes no desenvolvimento de software é a dificuldade de assegurar que o código esteja de acordo com os requerimentos do problema em análise. Ambientes automatizados procuram de várias formas auxiliar no processo de validação, apontando, em alguns casos, alternativas de implementação.

Terceira, a interatividade viável nesses ambientes, enriquecida pelos diversos meios audio-visuais possíveis para troca de informação com os usuários, mostra-se de extrema utilidade na avaliação e acompanhamento das diferentes fases do processo de elaboração de software.

Algumas barreiras ainda se apresentam, exigindo um esforço de pesquisa considerável. Da mesma forma, como não há consenso sobre uma linguagem de programação adequada à todos os tipos de tarefas, não tem sido possível construir um simples ambiente que seja adequado à todos os domínios. Técnicas de projeto e apoio à depuração frequentemente

variam de domínio a domínio. Conhecimento adicional sobre o domínio pode sempre ser usado para melhorar e suportar o ambiente criado e colocado à disposição do usuário.

Por outro lado, a programação tem menos de cinquenta anos de existência. O código fonte produzido ao longo desses anos é parte significativa da cultura de programação existente. Esse acervo codificado tem desmistificado a programação e viabilizado um mercado de serviços cada vez mais competitivo.

Algumas pesquisas, dirigidas às características desse código fonte produzido, procuram explorar os aspectos relacionados ao mundo dos programadores e às práticas de programação em uso. Muitos ambientes de programação, portanto, vêm encorajando o estudante a estudar o código fonte correto [WEIS 87]. Acredita-se que o estudante, ao estudar os códigos corretos produzidos para determinados problemas, adquira maturidade sobre programação. No ambiente DOCET, como será abordado mais a frente, considera-se a possibilidade de permitir ao estudante a visualização de um código fonte correto, fornecido pelo instrutor.

O enfoque deste trabalho é o desenvolvimento de ferramentas voltadas para estudantes iniciantes de programação, que demandam tratamento diferente daquele oferecido pelos ambientes voltados à programadores experientes.

Um desses ambientes, proposto pelo Projeto Aprendiz de Programador [WATE 85, LUCE 87] desenvolve uma teoria sobre programação, isto é, uma compreensão sobre como os programadores experientes entendem, desenvolvem, implementam, testam, verificam, modificam e documentam programas. A meta final é automatizar o processo de compreensão por inteiro.

O Projeto Aprendiz de Programador se propõe a atuar como um parceiro crítico, guardando a trilha dos passos dados pelo estudante e dando assistência em diferentes instantes da geração do código, sem, no entanto, inibir a sua capacidade de criação.

No ambiente projetado, usando técnicas de Inteligência Artificial, são desenvolvidas ferramentas que buscam influenciar convenientemente nas decisões de programação. Associa-se a isso, a representação de uma grande quantidade de conhecimento sobre

programação, em geral para utilização no entendimento de programas particulares. Essa postura, como em outras áreas onde a Inteligência Artificial vem sendo aplicada, abre as portas para o comportamento inteligente.

Uma das ferramentas propostas, o Editor Baseado em Conhecimento - EBC [WATE 85], concentra-se na tarefa de construção do programa. No entanto, a profundidade do seu entendimento sobre o programa é bem limitada. Ele procura tornar possível a operação direta sobre a estrutura algorítmica do programa, o que permite a construção de programas a partir de fragmentos algorítmicos em níveis de abstração mais próximos do programador. Esta ferramenta tem passado por refinamentos e expansões que buscam incrementar a cadeia de informações sobre programação e sobre os programas processados no ambiente. Os avanços mais recentes apontam para o desenvolvimento de Métodos de Dedução de Propósito Geral Automatizados, apropriados para o uso no domínio dos planos de programação.

Os resultados obtidos acerca da interação entre o conhecimento sobre os fundamentos e o conhecimento sobre a prática têm sido significativos. Vários dos melhoramentos oriundos das estruturas dos programas têm sido incorporados aos sistemas em demonstração. Mas, ainda é grande o intervalo de tempo entre o desenvolvimento desses novos conceitos e a incorporação deles ao processo de construção de programas.

Três elementos conceituais chaves: o método de assistência, os clichês e os planos, encontram-se presentes no Projeto como um todo, e no EBC em particular. Esses elementos conceituais, descritos em [WATE 85],definem a metodologia usada e são a base para definição da funcionalidade do ambiente.

No âmbito do DOCET, particularmente na especificação do VIP, alguns desses conceitos são utilizados, buscando uma associação entre o conhecimento previamente armazenado e os diferentes níveis de atuação dos estudantes de programação ao criarem seus programas.

## II.5 O COMPUTADOR APOIANDO O ENSINO DE PROGRAMAÇÃO

Numa perspectiva de trabalho semelhante ao Projeto Aprendiz de Programador, porém voltada à assistência de estudantes iniciantes de programação, estão três outros tipos de ferramentas: sistemas tutoriais, achadores de *bugs* e ambientes para iniciantes de programação.

Os sistemas tutoriais no contexto de ambientes de programação assistem à criação dos programas dos estudantes e emitem conselhos sobre possíveis mudanças ou limitam algumas das ações do programador. Exemplos desta categoria é o SPADE [ELSO 88], o TUTORLISP [ELSO 88] e o BRIDGE [BONA 88]. Esses ambientes oferecem ao estudante uma sequência predeterminada de exemplos e problemas com níveis crescentes de dificuldade. Os estudantes, por sua vez, podem formular questionamentos, estabelecendo um diálogo com o sistema.

Os achadores de *bugs* identificam e algumas vezes corrigem os erros encontrados nos programas dos estudantes. Alguns deles como o PROUST [SOLO 83, JOHN 87], analisam apenas a existência de erros não sintáticos.

Os ambientes de programação reunem mecanismos, geralmente com o uso de técnicas de inteligência artificial, que os distinguem dos outros ambientes pelo grau de controle permitido sobre os programas criados e pela flexibilidade das representações de conhecimento suportadas. São exemplos: o INTERLISP [ELSO 88] e PROPLOG [ELSO 88].

O principal problema encontrado nos sistemas tutoriais e achadores de *bugs* é sua capacidade de decidir se o estudante atendeu precisamente a todos os requerimentos do problema.

Entre as ferramentas citadas na categoria de achadores de *bugs*, observam-se posturas diferentes diante do problema. Algumas dessas posturas são complexas ou muito limitadas. Por exemplo, existem ferramentas extremamente detalhadas para um conjunto muito pequeno de problemas.

A seguir, são descritas algumas das particularidades de duas das ferramentas citadas na categoria de achadores de *bugs*: PROUST e BRIDGE.

## II.5.1 O AMBIENTE PROUST

O sistema PROUST atua como consultor para o estudante de programação, examinando potencialmente os *bugs* do código e reconstruindo o raciocínio do estudante, cujas decisões podem ter levado aos erros.

O PROUST conta com uma base de conhecimento onde são armazenados os "planos de programação". Esses planos de programação são métodos estereotipados para satisfazerem determinadas metas de programação. O PROUST combina esses planos buscando chegar a uma possível implementação de cada meta, e então tenta casar os planos obtidos com o programa desejado. Se o código do estudante casar com um dos conjuntos de planos sugeridos, o PROUST conclui que as intenções do estudante casaram corretamente com a implementação dos planos.

Algumas vezes os planos sugeridos casam apenas parcialmente com a meta. O PROUST, nestes casos, tenta encontrar as diferenças entre os planos e o código do estudante. Isto é, recorre-se a um conjunto de regras de produção, chamado de "regras para diferenças entre planos", que tratam as diferenças entre os planos desejados e o código em análise. Essas regras são capazes de sugerir os "*bugs*" e os conceitos mal entendidos que julgam serem os causadores do não casamento.

Os planos são representados na forma de combinações de submetas a serem satisfeitas e padrões que casam com os comandos no programa. Faz-se necessário quebrar as metas em conjuntos de submetas, combinar metas numa meta mais abrangente e adicionar metas que não estão explicitamente expostas no programa. Quando os programas não são triviais, é comum o surgimento de um grande número de possíveis decomposições de metas.

Nestes casos, o PROUST gera várias decomposições de meta para cada meta existente

e para as diferenças de planos. Supondo que seria possível reconhecer muitas variações de programas nas decomposições das metas, o número de programas que o PROUST poderia analisar seria imenso.

De forma idêntica, o PROUST poderia interpretar um programa em vários caminhos alternativos possíveis. Nestes casos, ele procura escolher, de forma heurística, a interpretação que deve ser analisada de acordo com o número e a natureza dos *bugs* que estão previstos em cada interpretação. Isto implica numa capacidade de análise limitada quando se trata de programas mais complexos. Pois, o PROUST necessariamente teria de inferir as decomposições das metas do código.

Uma das propostas apresentadas como solução para este problema, é passar a permitir ao estudante a descrição de suas intenções diretamente e que o tratamento de tais descrições se dê numa forma entendível pela máquina. O computador, então, passaria a auxiliar o estudante no processo de desenvolvimento do raciocínio sobre o programa. Os erros seriam mais facilmente identificados. Todavia, essa proposta encontra-se ainda em fase de desenvolvimento.

O PROUST deve passar a atuar baseado na análise das intenções, manuseando um conjunto de descrições de possíveis soluções, armazenadas em planos, e regras para diferenças entre planos. No seu estágio atual, ele ainda é limitado pelo seu domínio, pois não permite a construção da descrição das intenções de cada solução particular.

### II.5.2 O AMBIENTE BRIDGE

Um dos aspectos mais criticados no PROUST é a falta de uma interação com o estudante.

O BRIDGE, que dispõe de uma rica interface, é um protótipo de um ambiente tutorial para programadores iniciantes. O seu objetivo é permitir que o estudante relate seu projeto e tarefas parciais, levando em consideração o processo cognitivo envolvido na atividade de

criação do programa.

As características da ferramenta são resultado de observações realizadas com iniciantes em programação, que possuem uma experiência ampla em procedimentos informais delineados passo-a-passo. São atividades simples do dia-a-dia como ir à casa de um amigo ou escolher um presente para o pai.

Essas investigações sugeriram que deve existir uma regularidade na forma como os não programadores descrevem estas atividades numa linguagem informal. Por exemplo, observou-se que diferentes iniciantes usaram as mesmas frases para descrever estruturas de laço ou outras ações padrões na programação.

É muito comum o estudante confundir frases da linguagem natural com as palavras chaves existentes em linguagens de programação. Ele escreve comandos, no PASCAL por exemplo, como se tivessem a semântica de uma frase na linguagem informal correspondente. Por exemplo, o estudante escreve o laço "while" no PASCAL esperando a semântica do "while" em inglês.

No conhecimento mais relevante que um estudante traz sobre programação, as pesquisas mostraram que os procedimentos passo-a-passo escritos numa linguagem natural podem ser o elo de ligação entre as idéias do estudante e o programa final. A maioria dos erros dos estudantes se originaram nas suas experências com a linguagem natural. O BRIDGE, ao permitir que o estudante use linguagem natural, procura inferir mais diretamente os erros e oferecer um modelo mais simples de ser usado.

Como já dito anteriormente, a maioria dos textos de programação não ensina quase nada sobre a prática de programação. O que existe são técnicas padronizadas para implementações de tarefas comuns, nas quais o estudante se baseia para construir seus raciocínios. Essas técnicas padronizadas são conhecidas por planos de programação, como as encontradas no PROUST.

As investigações procuram mostrar que o entendimento desses planos, com a devida orientação, são cruciais para o sucesso do estudante de programação. Desta forma, os criadores do BRIDGE propuseram a inclusão dos planos de programação como o segundo

passo a ser construído pelo tutor de programação.

Esta estratégia pode ser justificada pela atitude do estudante em adotar o casamento sintático com os exemplos dos textos de programação para resolver seus problemas. No entanto, sabe-se que os peritos em programação, ao abordarem inicialmente um problema, esboçam um plano de resolução em um nível de abstração maior que o permitido pela sintaxe da linguagem a ser usada.

Com a inclusão dos planos de programação, como um nível intermediário entre a linguagem informal e o código final da programação, os criadores do BRIDGE quebraram o processo de programação em três passos:

1. especificação da solução em uma linguagem informal;
2. tradução da linguagem informal na especificação do plano;
3. tradução da especificação do plano no código da linguagem de programação - a solução final.

No instante em que o estudante especifica sua solução numa linguagem informal, usando a língua inglesa, o BRIDGE procura entender essas especificações e prover uma ajuda inicial sobre as mesmas. Essas especificações são baseadas na experiência de pré-programação dos estudantes iniciantes e baseia-se nos erros que possam vir desta experiência.

Os planos são colocados em alto nível, permitindo o estudante chegar aos resultados finais combinando planos. Erros resultantes dessas combinações são críticos no universo de erros gerados por estudantes iniciantes. Os criadores do BRIDGE acreditam que com a representação explícita dos planos, o estudante pode refletir sobre os planos e suas interações, sem se importar em como os planos agrupados serão transformados em código.

Qualificando-se como tutor, o BRIDGE oferece a possibilidade de supervisionar as ações do estudante quando o instrutor estiver ausente.

Na interação com o BRIDGE, o estudante só é interrompido se existir algum erro relativo ao modelo que o tutor julgou estar sendo utilizado. Quando o erro não é muito relevante ao nível atual de entendimento, o estudante não é interrompido. A dificuldade é encontrar a medida exata para classificar o nível de relevância de um erro.

O estudante pode chegar a uma solução por sucessivas aproximações. A versão final para o problema, neste caso, é muito rica, pois associa muitas respostas parcialmente corretas. O estudante trafega pelo ambiente traduzindo um componente de uma representação para outro equivalente numa outra representação. Isto facilita ao BRIDGE o entendimento das soluções parciais do estudante.

A característica marcante do BRIDGE é a apresentação visual do problema em resolução. Muitos dos passos da solução são apresentados no vídeo.

Atualmente os planos de programação são traduzidos para construções da linguagem PASCAL.

# CAPÍTULO III

# VERIFICADOR INTERATIVO DE PROGRAMAS
# VIP

# CAPÍTULO III: VERIFICADOR INTERATIVO DE PROGRAMAS - VIP

Em geral, os ambientes disponíveis para iniciantes em programação costumam acompanhar o estudante apenas no processo de concepção e depuração do programa. Faz-se necessário ferramentas que explorem a capacidade de comunicação e interatividade do computador nas atividades de ensino e aprendizagem de fundamentos sobre programação.

O ambiente DOCET, proposto inicialmente em [CAME 91], foi projetado para apoiar o ensino e treinamento de estudantes de cursos introdutórios de programação nos moldes dos sistemas tutoriais, trazendo características dos achadores de bugs. Uma descrição sucinta do DOCET é feita no Anexo A.

O Verificador Interativo de Programas - VIP atua como o achador de *bug* do ambiente DOCET. A sua atuação extrapola a dos achadores de *bugs* por se encontrar inserido no contexto de um ambiente de ensino, que acompanha o estudante desde a fase de exploração e aquisição do conhecimento sobre resolução de problemas e programação, e não apenas durante os processos de avaliação.

Percebe-se como relevante os aspectos didáticos e de orientação trabalhados na fase de aprendizagem do estudante, na medida em que tais posturas de comportamento oferecem fundamentação teórica aos diferentes níveis de assistência proporcionados pelo VIP, posteriormente.

## III.1 VISÃO GERAL

O estudante, ao analisar um problema oferecido pelo ambiente, deve estabelecer um plano de resolução e codificá-lo na linguagem Pascal. O VIP, então, procura auxiliar o estudante na tarefa de validação dessa solução codificada.

O VIP utiliza um conjunto de regras pragmáticas direcionadas à prática de programação, mapeando o conhecimento sobre os conceitos básicos de programação envolvidos. Estas regras auxiliam na detecção e correção de erros decorrentes de deformações no aprendizado desses conceitos. É comum, por exemplo, o estudante construir um laço sem fim, omitindo a atualização das variáveis da condição no corpo do laço. Ele estaria supondo que esta atualização se dê de forma automática.

Na detecção dos erros oriundos do mau entendimento das metas do problema, o VIP utiliza planos de programação contendo o conhecimento particularizado sobre o conjunto de metas e submetas inerentes à cada problema cadastrado. Os planos de programação e os caminhos alternativos para construção de diferentes soluções, válidas para um mesmo problema, encontram-se armazenados no Banco de Problemas do ambiente.

A linguagem Pascal é muito usada como primeira linguagem no treinamento de programação, pois possui uma semântica muito próxima daquela usada pelos estudantes ao especificarem suas soluções em linguagem natural (particularmente, em inglês). Além disso, o fato de PASCAL ser extensamente utilizada em disciplinas introdutórias de programação na Universidade Federal da Paraíba - UFPB e em outras universidades do país, reforçou a sua escolha como primeira linguagem a ser adotada no ambiente.

De fato, como o VIP é orientado para dar assistência à estudantes iniciantes de programação, foi considerada também a complexidade dos problemas mais usados nestes casos. Consolidando-se assim, como domínio a ser usado pelo VIP, um subconjunto da linguagem PASCAL, apresentado no Anexo B. Em se tratando de iniciantes de programação, os recursos deste subconjunto mostrou-se adequado.

## III.2 VIP NO CONTEXTO DO AMBIENTE DOCET

Os programas a serem analisados pelo VIP necessitam estar sintaticamente corretos. Isto será assegurado pelo Gerente de Eventos do DOCET, que se encarrega de submeter o

código ao Analisador Sintático, de forma transparente ao estudante, quando este solicitar a análise pelo VIP, sem que o programa tenha obtido a condição de sintaticamente correto no ambiente. Não é necessário, portanto, o estudante seguir o formalismo do fluxo normal de edição, análise léxica e sintática, e, então, VIP (verificação de erros lógicos).

A integração do VIP com as ferramentas do DOCET é visualizada na figura 2.

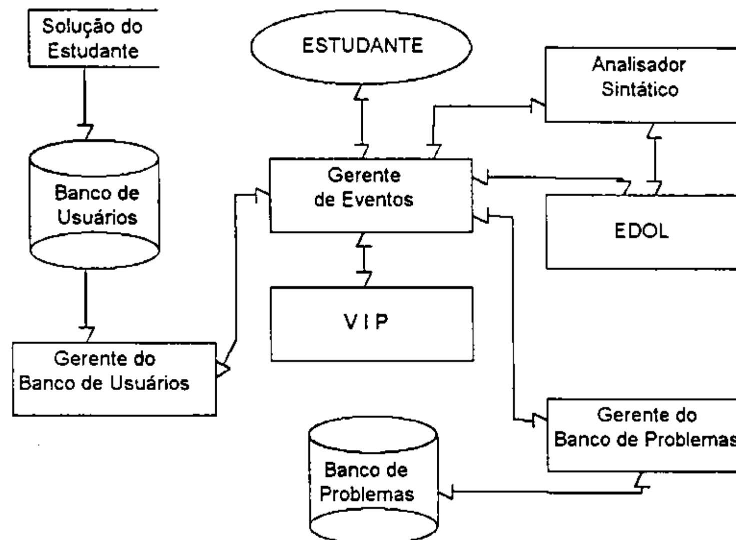



Figura 2 - Comunicação do VIP com as ferramentas do ambiente DOCET

Cada ferramenta do ambiente possui funções bem definidas no processo de interação com o VIP:

- O Editor Orientado à Linguagem - EDOL: apoia a edição do programa. Dispõe de facilidades adicionais para análise dos padrões léxicos e de algumas regras sintáticas básicas da linguagem PASCAL durante o processo de digitação, orientando na correção dos mesmos.
  O VIP envia para o EDOL, através do Gerente de Eventos, um arquivo contendo os erros lógicos encontrados no programa para serem apresentados ao estudante. Nesse arquivo são indicadas as linhas onde foram detectados erros ou advertências com o código respectivo da mensagem a ser apresentada. O código é decodificado pelo EDOL a partir de uma tabela de mensagens armazenada pelo instrutor.

- O Analisador Sintático - AS: ocupa-se da análise léxica, sintática e de algumas regras semânticas (estáticas) de programas. Estes programas podem ter sido editados no EDOL ou em qualquer outro editor externo ao ambiente.
  Os erros encontrados são apresentados ao estudante para correção através do EDOL.
  A ativação do Analisador Sintático pode passar despercebida ao estudante nos casos da inexistência de erros.
  Para o VIP é essencial uma sinalização positiva desta ferramenta, enviada ao Gerente de Eventos, para que se inicie a análise lógica do programa.

- O Gerenciador do Banco de Usuários - GBU: mantém um banco de informações sobre os instrutores e estudantes para uso de algumas das ferramentas do DOCET. Para uso do VIP, um conjunto de informações sobre os problemas vinculados a cada estudante é transmitido através do Gerente de Eventos:

  - nível corrente de dificuldade dos problemas a serem resolvidos;
  - lista de problemas resolvidos com o último "status" da solução (correta, parcialmente correta, errada);
  - metalinguagem e massa de dados para testes do problema a ser analisado.

- O Gerente do Banco de Problemas - GBP: mantém informações sobre os problemas cadastrados no DOCET, coloca à disposição do VIP um conjunto de informações referentes a cada problema:

  - o enunciado a ser apresentado ao estudante;
  - as informações relativas aos planos de programação das metas e submetas que compõem as múltiplas soluções do problema;
  - a massa de dados para simulação da execução da solução do estudante e/ou da solução armazenada pelo instrutor.
  - uma solução completa documentada, que pode, em alguns casos, ser apresentada ao estudante.

- O Gerente de Eventos: possibilita o atendimento das solicitações do estudante e mantém informações sobre as condições de funcionamento de cada uma das

ferramentas presentes no DOCET. É responsável pela ativação adequada de cada ferramenta, inclusive pela comunicação do VIP com outras ferramentas do DOCET.

## III.3 REPRESENTAÇÃO DO CONHECIMENTO SOBRE PROGRAMAÇÃO

Os peritos em programação, ao resolverem problemas, costumam organizar seu conhecimento em estruturas conceituais abstratas e não de acordo com a sintaxe das linguagens de programação. As estruturas de conhecimento produzidas são complexas e envolvem algoritmos de resolução de problemas associados ao domínio da informação trabalhada.

Algumas hipóteses sobre a forma dos peritos organizarem seu conhecimento têm sido formuladas por pesquisadores, já apontadas anteriormente. Investigações voltadas para programadores iniciantes também são realizadas, mas com menor intensidade.

Soloway [SOLO 83], um dos criadores do PROUST, acredita que os peritos utilizam "planos" que indicam os passos necessários para alcançar uma determinada meta, como por exemplo - o quadrado de um inteiro. Ele propõe que a utilização desses planos possa auxiliar estudantes a construirem programas. Os estudantes começariam a implementar planos abordando problemas mais simples.

Anderson e outros pesquisadores na Carnegie-Mellon University, criadores do TUTORLISP, motivam os estudantes ao uso de sequências de código [ELSO 88]. Os problemas são inicialmente descritos em termos de metas abstratas que posteriormente dão origem a código específico. Eles têm obtido sucesso no ensino de estudantes com a utilização de problemas relativamente pequenos, mas complexos.

Vários estudos de caso sobre programação [LINN 92, BOUL 86] dedicam-se à observação dos planos de programação; ao estudo da descrição do processo usado por peritos em programação para codificação dos planos; à exploração das listagens de código produzidas por iniciantes em programação; ao entendimento das questões oriundas da prática

de projeto, resolução de problemas e análise de programas; e ao entendimento das questões mais comuns no processo de aprendizado do estudante. Tais investigações resultam em representações de conhecimento alternativas para o projeto e desenvolvimento de programas.

Os estudos, aqui desenvolvidos, abordaram sempre as múltiplas visões possíveis encontradas em programas de computadores tendo como referência principal os estudantes iniciantes de programação. Foram discutidas inúmeras situações observadas na "zona próxima ao desenvolvimento", assim chamada por Vygotsky [VYGO 62], onde os estudantes se deparam com dificuldades acerca da continuidade do desenvolvimento de programas a partir de um ponto inicial. Observou-se que os estudantes constroem novos conhecimentos por refinarem e ampliarem o que já foi apreendido.

As dificuldades observadas podem ser classificadas nas seguintes áreas:

1. Problemas no entendimento das características da máquina. Como se dá o comportamento físico da máquina a partir do emprego de alguma notação.

2. Problemas com o entendimento de estruturas padrões. Planos de resolução genéricos que auxiliam na obtenção de metas, tais como o somatório com o uso de um laço.

3. Problemas com a notação das linguagens de programação. Os estudantes conseguem dominar a sintaxe, mas entendem pouco a semântica.

4. Problemas de orientação específica. Como atacar metas existentes nos problemas com o uso de programação e quais os esforços necessários.

Procurou-se uma representação de conhecimento robusta, que modelasse as soluções de programação combinadas às situações onde os conceitos ensinados, se aplicados, possam resultar em padrões alternativos de código. Além disso, foram considerados como relevantes os seguintes aspectos:

- o ensino de estratégias para reduzir a complexidade da solução do problema. A apresentação da solução ótima do instrutor é uma das estratégias usadas;

- o uso de comentários elaborados pelo instrutor sobre padrões usados pelo estudante na criação da solução;
- o engajamento do estudante em torno da reflexão crítica sobre a sua própria solução e, algumas vezes, sobre a solução do instrutor.

O VIP possui dois conjuntos de informações sobre o conhecimento de programação em níveis que levam em consideração o conhecimento trazido pelo estudante (CGP) e o conhecimento por ele adquirido no ambiente (CEM):

1. O Conhecimento Genérico sobre Programação - CGP. Leva em consideração o entendimento do estudante sobre alternativas padrões usadas em programação que, muitas vezes, é influenciado por seus modos de "pensar" e "resolver problemas".
   É utilizado na verificação do nível de qualificação do uso do conhecimento pragmático sobre programação na solução proposta pelo estudante. Este conhecimento é identificado a partir de Regras Pragmáticas armazenadas, cuja ativação se dá de acordo com o contexto dos comandos utilizados na solução.
   Estas regras descrevem subconjuntos de conhecimento sobre programação, que interligam a informação abstraída da especificação do problema ao componente do programa.
   O que se deseja é poder oferecer assistência aos estudantes, de modo a explicitar o emprego de tais abstrações.

2. O Conhecimento Específico sobre Metas - CEM. Considera os aspectos relativos ao entendimento do estudante quanto à funcionalidade dos comandos da linguagem e às metas do problema.
   É utilizado na verificação do nível de atendimento da solução proposta, pelo estudante, aos requerimentos específicos do problema.
   Este conhecimento é representado através de planos de programação, cadastrados pelo instrutor e organizados na estrutura definida no Espaço de Representação das Soluções dos Problemas, que descrevem as alternativas possíveis para implementação de cada meta envolvida no problema.
   Considera-se, ainda, como elemento de diferenciação importante dois tipos de

metas: metas principais e metas secundárias. A meta principal é condição *sine qua non* para atendimento das especificidades do problema. A meta secundária contribui para o alcance da meta principal do problema.

As duas formas de representar o conhecimento sobre programação definem o conjunto de informações que são armazenadas para cada problema cadastrado pelo instrutor. Os problemas cadastrados passam a integrar o Banco de Problemas, cuja organização possui uma classificação própria, como será apresentado posteriormente.

### III.3.1 CONHECIMENTO GENÉRICO SOBRE PROGRAMAÇÃO

O conhecimento genérico sobre programação explicita as exigências mínimas ou pré-condições de funcionamento satisfatório dos planos de programação usados na construção do programa. Independe da linguagem de programação usada e das metas específicas do problema.

Em geral, os experimentos realizados com estudantes de programação mostraram que essas exigências ou pré-condições relacionam-se:

- às restrições das estruturas de programação. Por exemplo, uma estrutura de repetição condicional, para encerrar o processo de execução, necessita que a condição controladora passe por mudanças de estado a cada nova interação;

- ao contexto necessário para o uso das estruturas de programação. Por exemplo, uma estrutura de decisão só funciona com a análise de condições que resultem em estado falso ou verdadeiro (lógica booleana);

- à operacionalização das estruturas de programação. Por exemplo, dificuldades podem surgir ao se tentar atribuir o resultado de uma expressão aritmética à um elemento definido como não numérico. Ou quando elementos presentes na expressão aritmética não foram inicializados com algum conteúdo;

- aos níveis de ligações possíveis entre estruturas de programação. Por exemplo, é necessária a atribuição de um valor inicial à variável de controle no contexto de uma repetição automática.

As regras pragmática definidas consideram tais relações nos níveis de dificuldade apresentados pelos problemas analisados pelo VIP. As regras encontram-se classificadas em dois grupos: regras genéricas e regras orientadas às estruturas de programação.

Todas as regras são definidas em forma de procedimentos que são ativados de acordo com o contexto da análise. Esses procedimentos descrevem, para cada situação, as pré-condições e os modos de funcionamento da estrutura de programação.

### III.3.2 ESPAÇO DE REPRESENTAÇÃO DAS SOLUÇÕES DOS PROBLEMAS

O espaço de representação das soluções dos problemas cadastrados no Banco de Problemas é organizado numa estrutura hierárquica, mostrada na figura 3, que associa informações gerais e específicas a grupos de problemas, definidos em função:

1. da forma de resolução : CLASSES;

2. da estrutura de controle do fluxo : TIPOS; e

3. do grau de dificuldade : NÍVEIS.

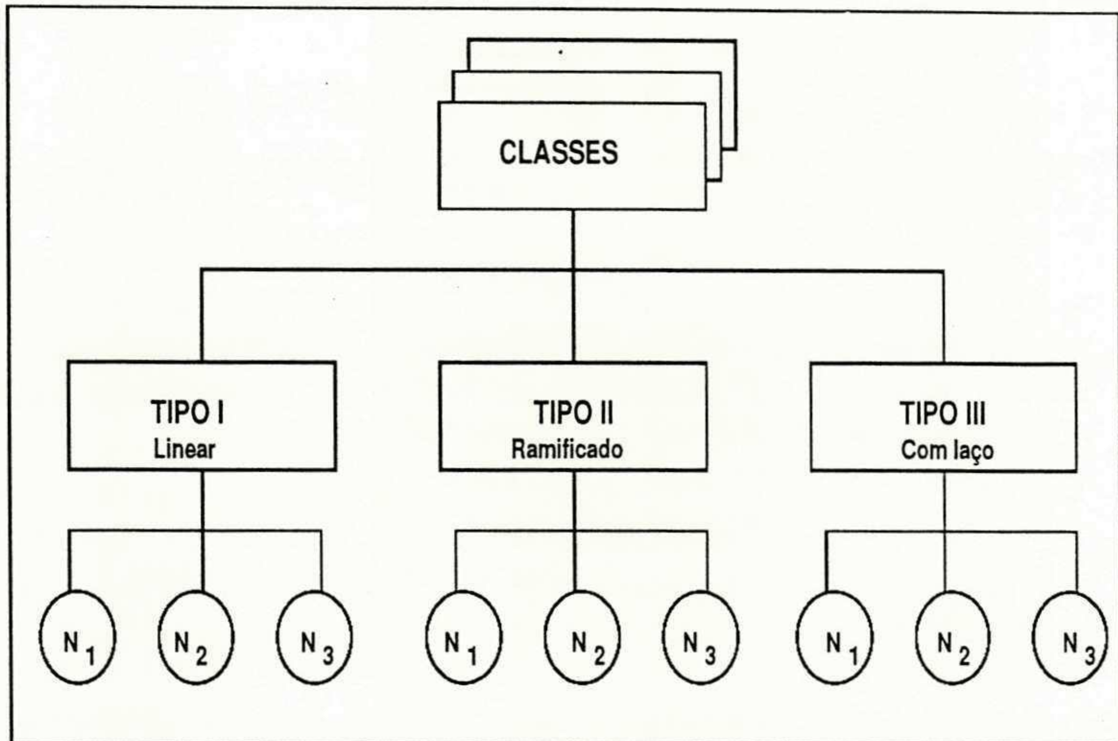



Figura 3: Estrutura organizacional das soluções no Banco de Problemas.

Cada classe de problemas possui características próprias de resolução, ou seja, metas específicas com funções semelhantes. Dependendo do nível de complexidade inerente à classe de problemas, as metas podem ser decompostas em submetas. De tal forma que, em alguns casos, uma determinada classe pode conter as metas/submetas de uma outra classe. Como, por exemplo, nas classes de problemas indicadas abaixo:

CLASSE :: [Cálculo de Expressão Aritmética]

- ⇨ Meta 1 :: [Obtenção de Dados]
- ⇨ Meta 2 :: [Cálculo da Expressão Aritmética]
- ⇨ Meta 3 :: [Impressão do Resultado];

CLASSE :: [Cálculo de Somatório]

- ⇨ Meta 1 :: [Obtenção de Dados]
- ⇨ Meta 2 :: [Cálculo do Somatório]
    - ⇨ Submeta 2.1 :: [Processo de Repetição]
        - ⇨ Submeta 2.1.1 :: [Obtenção de Dados]

⇨ Submeta 2.1.2 :: [Cálculo da Expressão Aritmética]

⇨ Meta 3 :: [Impresão do Resultado]; e

CLASSE :: [Cálculo de Média Aritmética]

⇨ Meta 1 :: [Obtenção dos Dados]

⇨ Meta 2 :: [Cálculo da Média Aritmética]

⇨ Submeta 2.1 :: [Cálculo do Somatório]

⇨ Submeta 2.1.1 :: [Processo de Repetição]

⇨ Submeta 2.1.1.1 :: [Obtenção de Dados]

⇨ Submeta 2.1.1.2 :: [Cálculo da Expres-são Aritmética]

⇨ Submeta 2.2 :: [Obtenção da Média Aritmética]

⇨ Meta 3 :: [Impresão do Resultado].

Considerando-se um conjunto de problemas normalmente usado em cursos introdutórios de programação, a partir de um levantamento realizado junto à instrutores de programação, foram definidas, para preenchimento da estrutura descrita, nove (9) classes de problemas:

① Impressão de Mensagens;
② Cálculo de Expressões Aritméticas;
③ Cálculo de Médias;
④ Cálculo de Produtórios;
⑤ Geração de Séries;
⑥ Operações de Busca em Vetores;
⑦ Operações Aritméticas com Vetores;
⑧ Operações Aritméticas com Matrizes; e
⑨ Ordenação de Conjuntos.

O cadastramento no Banco de novas classes de problemas, cujas características funcionais sejam próximas às dos problemas das classes relacionadas acima, poderá ser feita sem mudanças na estrutura definida.

As classes relacionadas podem conter especificações de problemas com diferentes níveis de exigências, e portanto, estruturas de programas que diferem entre si quanto ao

subconjunto de comandos e o fluxo de controle utilizados. Nesses casos, os problemas são classificados internamente em até três tipos de estruturas distintas:

**TIPO I**: Estrutura Linear;

**TIPO II**: Estrutura Ramificada; e

**TIPO III**: Estrutura com Laço.

As estruturas, definidas por subconjuntos de comandos PASCAL, associadas a cada tipo, podem ser visualizadas na Tabela 1. Elas procuram se identificar com as crescentes necessidades de uso de comandos mais complexos nos problemas sugeridos ao longo das sessões de treinamento.

Cada um dos três tipos de problemas podem ser ainda subdivididos em até três níveis, que indicam o grau de dificuldade intrínseco ao problema. Esse grau de dificuldade, é medido pelo instrutor a partir do número, diversidade e alternativas de composições dos comandos usados na solução do problema.

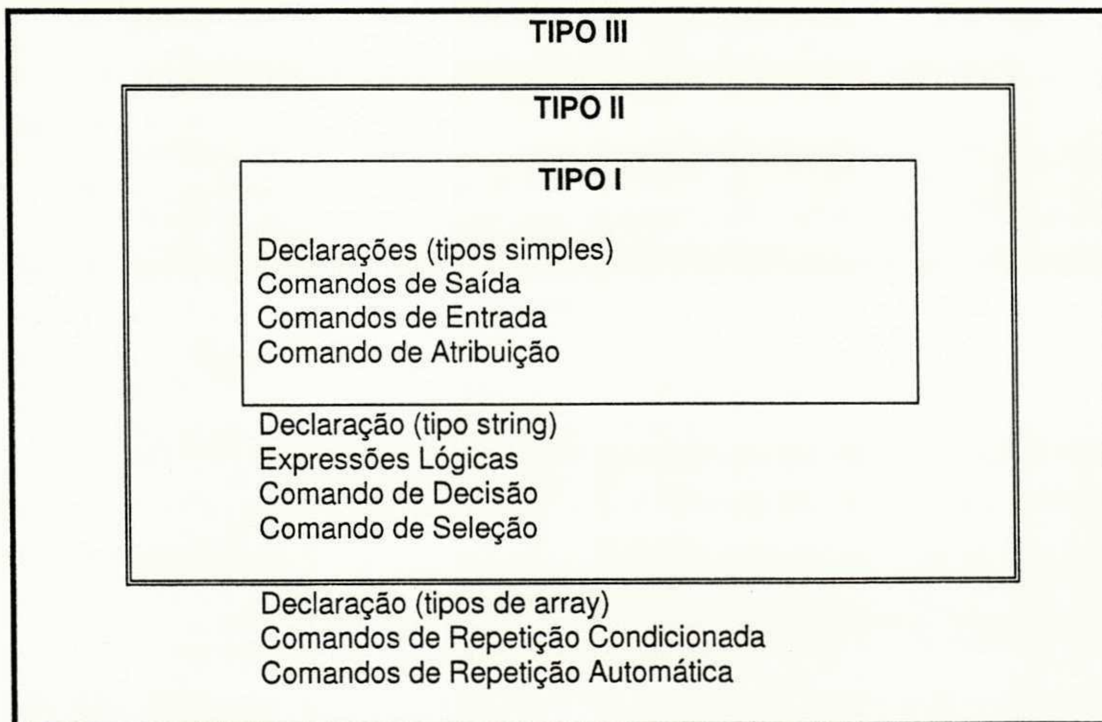



Tabela 1: Classificação das Estruturas de Comando por Tipo

Consideram-se também, as alternativas de soluções para os problemas previstos

naquele tipo e as investigações sobre o comportamento mais comum dos estudantes na resolução desses problemas, classificando-os em:

**NÍVEL I**: Baixo grau de dificuldade;

**NÍVEL II**: Médio grau de dificuldade; e

**NÍVEL III**: Alto grau de dificuldade.

As metas ou submetas que modelam as soluções dos problemas cadastrados no Banco de Problemas podem apresentar variações no formato de implementação do código. Isto ocorre com a utilização de diferentes construções da linguagem que são semanticamente equivalentes.

Quando construções semanticamente equivalentes encontram-se presentes no mesmo grupo de um tipo de estrutura, são associados aos planos básicos de implementação - os chamados planos "sinônimos".

Os planos sinônimos permitem, durante o processo de avaliação da solução do estudante, a geração dinâmica de versões distintas baseadas na solução ótima (plano básico) cadastrada pelo instrutor.

As diferentes versões geradas, a partir da substituição dos planos sinônimos dão origem a novas soluções completas -"soluções alternativas". Os comandos substituintes asseguram a mesma semântica, mas possuem sintaxe distinta dos originais. Esses comandos são chamamos de "sinônimos naturais".

Um exemplo de sinônimo natural pode ser a substituição de um comando FOR por um comando WHILE, adicionando-se as informações necessárias: inicialização, alteração e teste da variável de controle do laço.

Os sinônimos naturais são gerados automaticamente pelo VIP, a partir de informações solicitadas ao instrutor e armazenadas durante o processo de cadastramento dos problemas.

A análise da relação existente entre os sinônimos naturais previstos pelo VIP e as restrições da solução, inerentes à especificação do problema, resultou em algumas decisões de implementação resumidas na tabela 2.

| Comando Original | Sinônimo Natural | Restrições de Uso | Observações |
|---|---|---|---|
| IF | CASE | Dependentes da especificação. O instrutor decide sobre sua existência. | Caso exista, são armazenadas as expressões para a condição seletora e para as opções. |
| CASE | IF | Não existem. O armazenamento é automático. | São armazenadas as condições de cada IF presente na estrutura definida. |
| FOR | WHILE | Não existem. O armazenamento é automático. | São armazenadas as atribuições inseridas antes (opcional quando o limite inicial do FOR é uma variável lida) e internamente ao WHILE e sua condição. |
| | REPEAT | Não existem. O armazenamento é automático. | São armazenadas as atribuições inseridas antes (opcional quando o limite inicial do FOR é lido) e internamente ao REPEAT e sua condição. |
| WHILE | REPEAT | Dependentes da especificação. O instrutor decide sobre sua existência. | Caso exista, é armazenada a condição do REPEAT. |
| | FOR | Não são consideradas. | Este sinônimo não existe. (*) |
| REPEAT | WHILE | Não existem. O armazenamento é automático. | É armazenada a condição do WHILE. |
| | FOR | Não são consideradas. | Este sinônimo não existe. (*) |

Tabela 2: Sinônimos Naturais tratados pelo VIP

(*) As relações WHILE/FOR e REPEAT/FOR não são consideradas pelo VIP no cadastramento da solução. Assume-se que, quando a especificação do problema indica o uso de repetição automática, o instrutor sempre usará o comando FOR na solução ótima. Por outro lado, se o estudante esboçar o seu plano de implementação com a utilização de um "repeat" ou "while", é, então, analisada a estrutura sinônima do "for".

Não são previstos sinônimos naturais para declarações de dados e nem para qualquer outra estrutura não citada acima.

A figura 4 apresenta a árvore de soluções para um problema armazenado no Banco de Problemas. O plano básico ou a Solução Ótima (SO), as Soluções Alternativas (SA) e os sinônimos naturais (indicados por um asterisco (*)), permitem a construção de várias possibilidades de implementação para o problema.

A solução ótima sempre será a primeira cadastrada. As soluções alternativas dependem da análise do instrutor, que decidirá sobre a sua existência. Como será abordado mais a frente, espera-se que o ambiente possa vir a cadastrar as soluções corretas construídas por estudantes, sem similar no Banco, como mais uma solução alternativa.

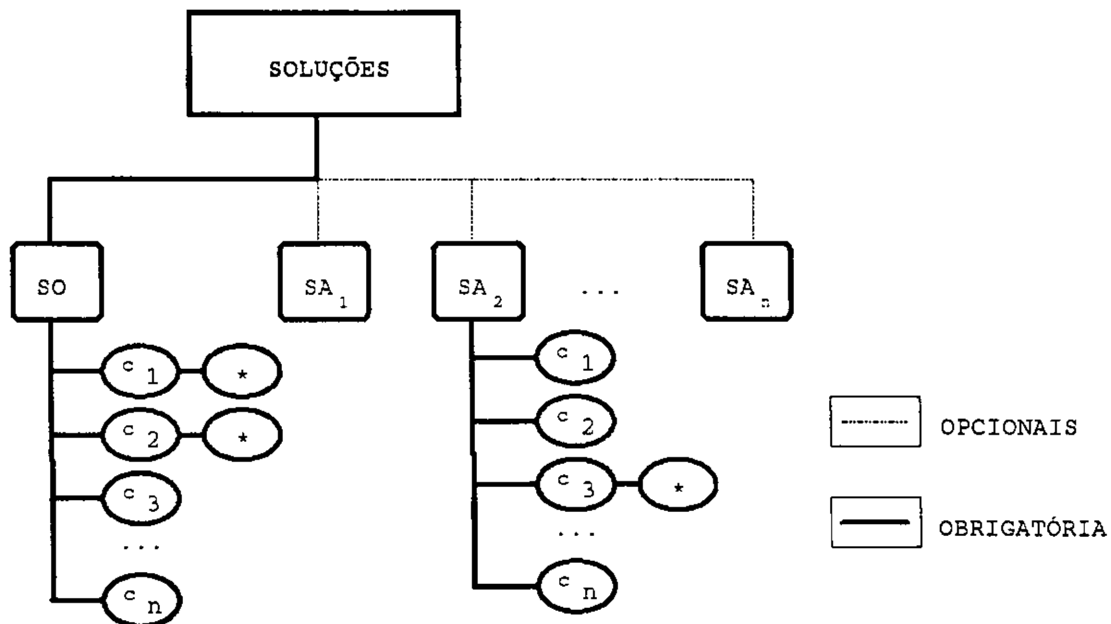



Figura 4: Árvore de soluções para um problema

No Banco de Problemas também foi considerada a existência de problemas que apresentam diferenças nas suas metas específicas, porém apresentam planos de resolução muito próximos. Isto ocorre devido a proximidade observada entre as metas principais dos problemas.

Essas ocorrências, muito comuns no ensino introdutório de programação, originaram

uma categorização de problemas: "problemas primos". Foram criados elos de ligação na estrutura de dados indicando os grupos de problemas primos existentes numa mesma classe. Essa informação, poderá ser utilizada pelo instrutor no processo de avaliação do estudante.

Por exemplo, considere-se um problema cuja especificação requer a leitura dos coeficientes de uma equação quadrática e o cálculo de suas raízes reais. A meta principal, que influencia a construção da solução, é o cálculo das raízes. Mas, pequenas variações na especificação podem enquadrar o problema em qualquer um dos três tipos de estruturas possíveis:

1. solução linear- a especificação solicita o cálculo das raízes de um só conjunto de coficientes, supondo o discriminante ($b^2$-4ac) maior ou igual a 0 (zero). Enquadra-se no TIPO I;

2. solução onde ocorrem ramificações- a especificação solicita a verificação da existência de raízes reais (($b^2$-4ac) >= 0) antes do cálculo das raízes de uma equação. Enquadra-se no TIPO II; e

3. solução onde ocorre repetição- a especificação é a mesma do item anterior, porém para um conjunto maior de coeficientes. Enquadra-se no TIPO III.

### III.3.3 CONHECIMENTO ESPECÍFICO ÀS METAS

O Conhecimento Específico às Metas de programação - CEM, representa um conjunto de informações voltadas à implementação da solução de um dado problema. Para isto, são consideradas as condicionantes impostas pelo domínio da linguagem de programação usada - PASCAL.

As informações estão associadas às metas e submetas que devem ser alcançadas pela solução do problema, apresentadas na forma de planos de programação. O planos detalham o uso de estruturas de PASCAL, relacionando:

- a quantidade e o tipo das variáveis e constantes definidas;
- as formas de entrada e saída de dados;
- as expressões aritméticas e lógicas definidas no contexto das diversas estruturas de comandos;
- as estruturas obrigatórias (e/ou indicações de sinônimos naturais com os dados adicionais para substituição);
- o sequenciamento das estruturas de controle conforme o modelo de resolução usado;
- a indicação de erros previsíveis no contexto de cada estrutura; e

Para armazenamento dos planos de programação elaborados pelo instrutor, foram criadas estruturas de dados com níveis de abstração que refletem características da programação orientada a objeto. O conhecimento sobre o domínio de programação usado é encapsulado em diferentes níveis com definição clara de suas permissões e formas de comunicação.

A partir da definição hierárquica da estrutura de comandos a ser utilizada em um programa, considerando as características dos elementos mais internos à estrutura, foram modeladas as <u>classes</u> - mais abrangentes e, os <u>objetos</u> - mais especializados. Os dados específicos a cada META, encontrada num programa em particular, são instâncias de classes e objetos cujas definições acolhem as características explicitadas pelo problema.

O conhecimento básico usado nos planos de programação para implementação das metas é agrupado numa super classe - chamada de <u>PROGRAMA</u>, composta dos seguintes objetos e classe:

- objeto <u>IDENTIFICAÇÃO</u>;
- objeto <u>DECLARAÇÃO</u>; e

- classe BLOCO, composta dos objetos:
    - objeto WRITE;
    - objeto READ;
    - objeto ASSIGN;
    - objeto IF;
    - objeto CASE;
    - objeto FOR;
    - objeto WHILE; e
    - objeto REPEAT.

A figura 5 ilustra a representação de um programa. As características de cada objeto serão tratadas no ANEXO D.

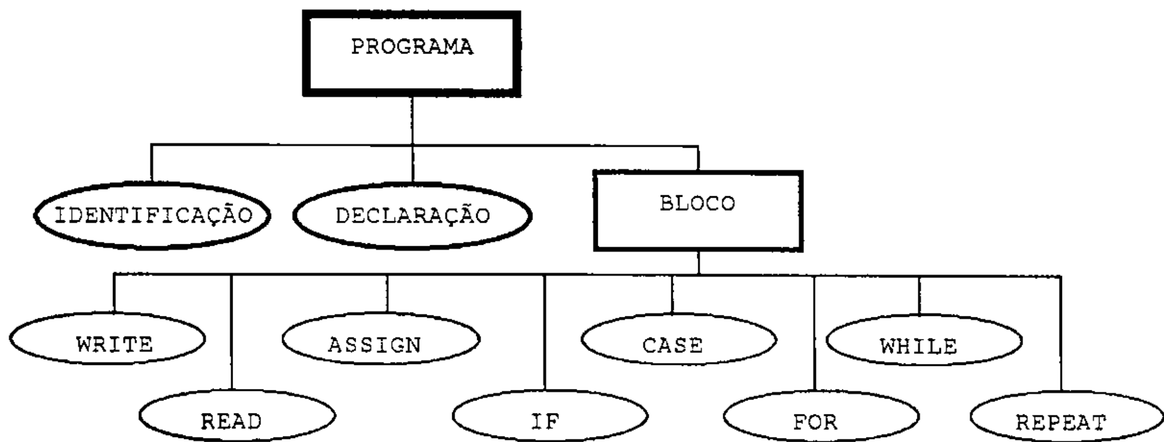



Figura 5: Classes e Objetos encontrados num programa

## III.4 SISTEMATIZAÇÃO DA ANÁLISE

Os principais módulos do VIP podem ser vistos no diagrama da arquitetura funcional apresentado na figura 6. A comunicação do VIP com as demais ferramentas do DOCET é gerenciada pelo Gerente de Eventos, como visualizado na figura 2.

O VIP é chamado pelo Gerente de Eventos a partir da solicitação do estudante. Ele recebe o arquivo "Solução do Estudante", que contém o programa fonte editado, já validado léxica e sintaticamente pelo Analisador Sintático do ambiente.

Inicialmente é realizada a conversão do código fonte, armazenado no formato ASCII, para um formato entendível pelo VIP, onde as estruturas do programa são recuperadas e combinadas de acordo com as necessidades das análises posteriores.

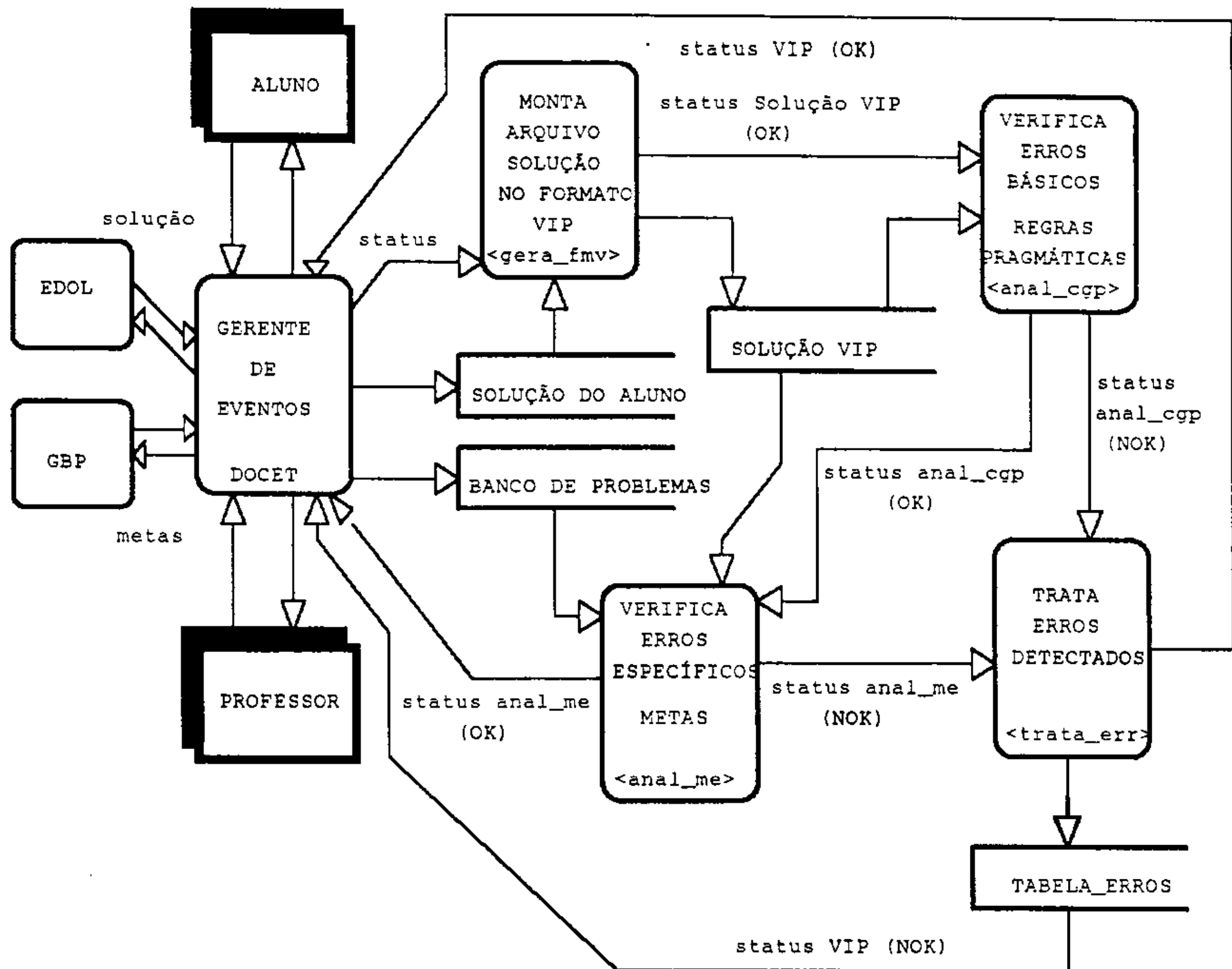



Figura 6: Arquitetura Funcional do VIP

Na primeira etapa da análise, o VIP verifica a adequação do programa do estudante à conceitos pragmáticos de "boa programação". Numa segunda etapa, após o programa ter obtido sucesso na análise inicial, o VIP analisa a adequação do programa às metas específicas do problema. Em ambos os casos, os erros e as advertências são armazenados numa tabela que é encaminhada ao EDOL para apresentação ao estudante. O processo de sistematização da análise é apresentado na figura 7.

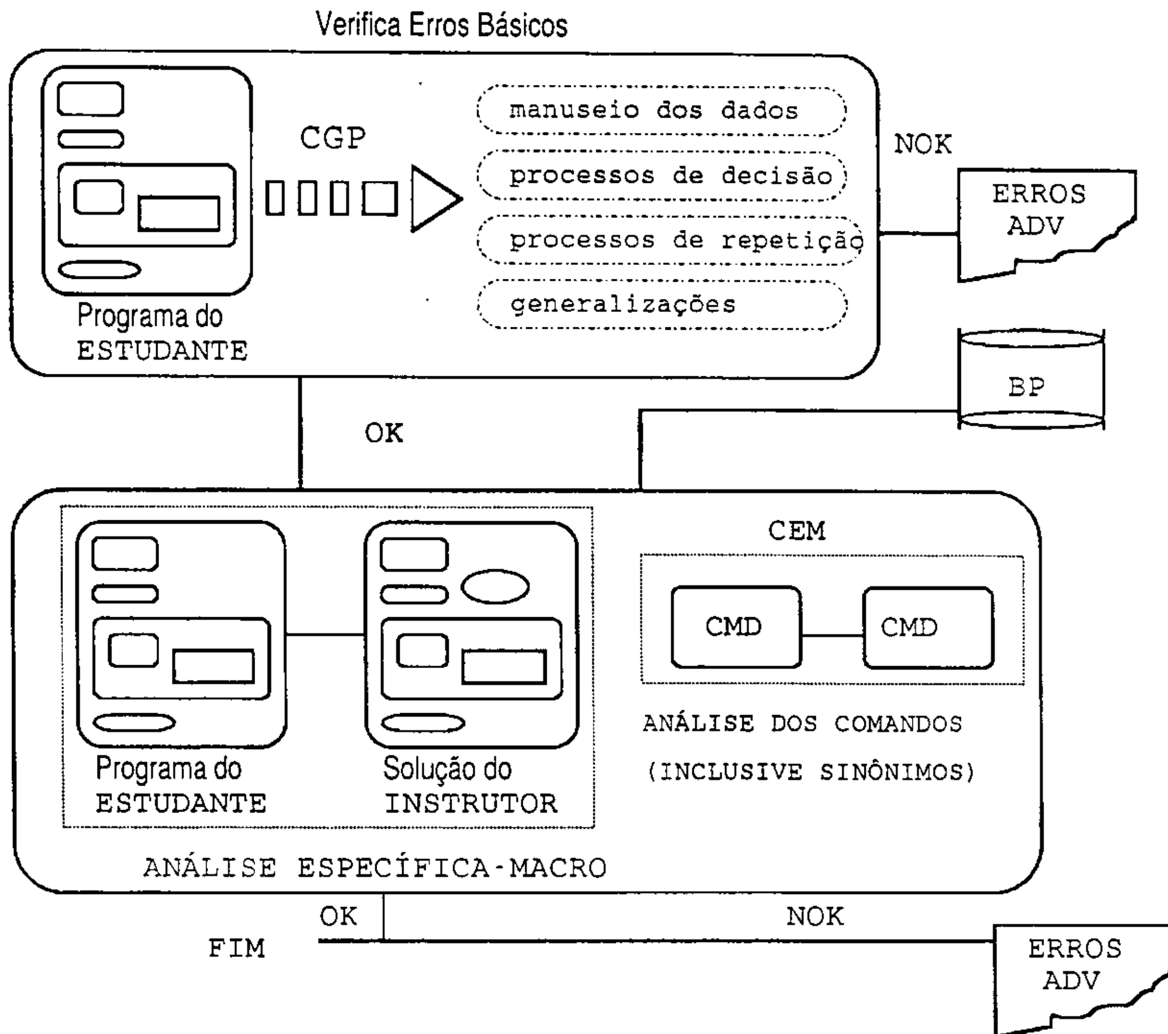



Figura 7: Sistematização da Análise

## III.4.1 MONTAGEM DO FORMATO VIP

Este módulo faz a conversão do programa do estudante do formato ASCII para o formato VIP, compatível com a forma de armazenamento das metas da solução correta (e/ou alternativas) fornecidas pelo instrutor. Este formato é chamado de "Solução_VIP", e encontra-se descrito no ANEXO C.

Conforme discutido anteriormente, a solução no formato VIP baseia-se na representação

dos planos de programação sob a forma de objetos, cujas características definem cada contexto encontrado no programa do estudante.

Após a criação do formato VIP, é iniciada pelo módulo responsável, a etapa da análise de erros genéricos de programação.

A estrutura de dados para representação da solução do estudante, apresentada no ANEXO C, demanda extensivamente de armazenamento de expressões em diferentes contextos. Considerando que a sua descrição é de suma importância para o processo de análise, buscou-se uma forma de representação mais elaborada que o simples armazenamento de conjuntos de caracteres. A forma de armazenamento buscada deveria também ser compacta para ocupar o mínimo de memória.

Explorando-se o processo de avaliação de expressões, comumente encontradas em programas para iniciantes, criou-se uma linguagem específica para manuseio e recuperação dos campos dessas expressões. Na realidade, parte significativa da análise do conhecimento específico das metas de um programa encontra-se relacionado à especificação de expressões.

Na linguagem criada, chamada EXPRESSA, estabeleceu-se, inicialmente, quatro **categorias** (IDC) em função dos elementos válidos nas expressões (aritméticas ou lógicas) construídas na linguagem Pascal, quais sejam:

- "variáveis";
- "constantes";
- "operadores"; e
- "funções embutidas".

Para cada uma das categorias relacionadas, procurou-se identificar as informações adicionais necessárias ao armazenamento das características funcionais da respectiva categoria no contexto da expressão. Nesta direção, foram levantados três grupos de informação para cada categoria: o **tipo do elemento** (IDT), a **identificação individual** (IDI) do elemento e um **status** (STU) contextual relacionado à particularidades de cada uma das categorias.

No processo de codificação da linguagem EXPRESSA, foram levantados alguns aspectos importantes:

1. Forma de armazenamento. A quantidade de expressões envolvidas é, em geral, parte significativa do código produzido;
2. Forma de recuperação. O processo de avaliação não deve ser demorado;
3. Forma de implementação. A flexibilidade e portabilidade dos recursos existentes no ambiente de desenvolvimento usado na implementação do VIP; e
4. Aplicação. A linguagem deve ser representativa e permitir evoluções. Melhorias no processo de avaliação de expressões são previstas.

As tabelas de 3 a 6 mostram a codificação binária adotada na representação da linguagem EXPRESSA.

Tabela 3: Representação de Operadores

| OPERADOR | IDC (2 bits) | IDT (*) (3 bits) | IDI (4 bits) | STU (2 bits) |
|---|---|---|---|---|
| NOT | 00 | 010 | 0000 | NU |
| * | 00 | 000 | 0001 | NU |
| / | 00 | 000 | 0010 | NU |
| DIV | 00 | 000 | 0011 | NU |
| MOD | 00 | 000 | 0100 | NU |
| AND | 00 | 010 | 0101 | NU |
| + | 00 | 000 | 0110 | NU |
| - | 00 | 000 | 0111 | NU |
| OR | 00 | 010 | 1000 | NU |
| = | 00 | 001 | 1001 | NU |
| <> | 00 | 001 | 1010 | NU |
| < | 00 | 001 | 1011 | NU |
| <= | 00 | 001 | 1100 | NU |
| >= | 00 | 001 | 1101 | NU |
| > | 00 | 001 | 1110 | NU |
| ( ou ) | 00 | 011 | 1111 | NU |
| [ ou ] | 00 | 100 | 1111 | NU |
| ( ou ) | 00 | 101 | 1111 | NU |
| , | 00 | 110 | 1111 | NU |
| NU | 00 | 111 | NU | NU |

NU=Não Usado

(*): 000 - operador aritmético

001 - operador relacional

010 - operador lógico

011 - ( ou ) usados no contexto de expressões

100 - [ ou ] usados em array nas expressões

101 - ( ou ) usados em funções nas expressões

110 - , usada em arrays-bi nas expressões

111 - não usado

Tabela 4: Representação de Variáveis

| VARIÁVEL | IDC (2 bits) | IDT (3 bits) | IDI (4 bits) | STU (2 bits) |
|---|---|---|---|---|
| Inteira sem Valor | 01 | 000 | 0000 a 1111 | 00 |
| Inteira Lida | 01 | 000 | 0000 a 1111 | 01 |
| Inteira Atribuída | 01 | 000 | 0000 a 1111 | 10 |
| Inteira Atribuída/FOR | 01 | 000 | 0000 a 1111 | 11 |
| Real sem Valor | 01 | 001 | 0000 a 1111 | 00 |
| Real Lida | 01 | 001 | 0000 a 1111 | 01 |
| RealAtribuída | 01 | 001 | 0000 a 1111 | 10 |
| Real Atribuída/FOR | 01 | 001 | 0000 a 1111 | 11 |
| Booleana sem Valor | 01 | 010 | 0000 a 1111 | 00 |
| Booleana Lida | 01 | 010 | 0000 a 1111 | 01 |
| Booleana Atribuída | 01 | 010 | 0000 a 1111 | 10 |
| Booleana Atribuída/FOR | 01 | 010 | 0000 a 1111 | 11 |
| Char sem Valor | 01 | 011 | 0000 a 1111 | 00 |
| Char Lida | 01 | 011 | 0000 a 1111 | 01 |
| Cadeia sem Valor | 01 | 100 | 0000 a 1111 | 00 |
| Cadeia Lida | 01 | 100 | 0000 a 1111 | 01 |
| Cadeia Atribuída | 01 | 100 | 0000 a 1111 | 10 |
| Cadeia Atribuída/FOR | 01 | 100 | 0000 a 1111 | 11 |
| Array-Uni sem Valor | 01 | 101 | 0000 a 1111 | 00 |
| Array-Uni Lido | 01 | 101 | 0000 a 1111 | 01 |
| Array-Uni Atribuído | 01 | 101 | 0000 a 1111 | 10 |
| Array-Uni Atribuído/FOR | 01 | 101 | 0000 a 1111 | 11 |
| Array-Bi sem Valor | 01 | 110 | 0000 a 1111 | 00 |
| Array-Bi Lido | 01 | 110 | 0000 a 1111 | 01 |
| Array-Bi Atribuído | 01 | 110 | 0000 a 1111 | 10 |
| Array-Bi Atribuído/FOR | 01 | 110 | 0000 a 1111 | 11 |
| NU | 01 | 111 | NU | NU |

NU - Não Usado

Tabela 5: Representação de Constantes

| CONSTANTE | IDC (2 bits) | IDT (3 bits) | IDI (4 bits) | STU (2 bits) |
|---|---|---|---|---|
| Inteira Declarada | 10 | 000 | 0000 a 1111 | 00 |
| Inteira Não Declarada | 10 | 000 | 0000 a 1111 | 01 |
| Real Declarada | 10 | 001 | 0000 a 1111 | 00 |
| Real Não Declarada. | 10 | 001 | 0000 a 1111 | 01 |
| Booleana Declarada | 10 | 010 | 0000 a 1111 | 00 |
| Booleana Não Declarada | 10 | 010 | 0000 a 1111 | 01 |
| Char Declarada | 10 | 011 | 0000 a 1111 | 00 |
| Char Não Declarada | 10 | 011 | 0000 a 1111 | 01 |
| Cadeia Declarada | 10 | 100 | 0000 a 1111 | 00 |
| Cadeia Não Declarada. | 10 | 100 | 0000 a 1111 | 01 |
| NU | 10 | 101 a 111 | NU | NU |

NU - Não Usado

Tabela 6: Representação de Funções Embutidas

| FUNÇÃO EMBUTIDA | IDC (2 bits) | IDT(1) (3 bits) | IDI (4 bits) | STU(2) (2 bits) |
|---|---|---|---|---|
| SQR | 11 | 000 | 0000 | 01 |
| SQRT | 11 | 000 | 0001 | 01 |
| LN | 11 | 000 | 0010 | 00 |
| EXP | 11 | 000 | 0011 | 00 |
| ABS | 11 | 000 | 0100 | 01 |
| TRUNC | 11 | 000 | 0101 | 00 |
| ROUND | 11 | 000 | 0110 | 00 |
| SIN | 11 | 000 | 0111 | 00 |
| COS | 11 | 000 | 1000 | 00 |
| ARCTAN | 11 | 000 | 1001 | 00 |
| CHR | 11 | 001 | 1010 | 00 |
| ORD | 11 | 011 | 1011 | 00 |
| SUCC | 11 | 100 | 1100 | 01 |
| PRED | 11 | 100 | 1101 | 01 |
| ODD | 11 | 010 | 1110 | 10 |
| NU | NU | NU | 1111 | NU |

NU - Não Usado

(1): tipo da função ARGUMENTO⇒RESULTADO

000 - função NUMÉRICO⇒NUMÉRICO

001 - função NUMÉRICO⇒LITERAL

010 - função NUMÉRICO⇒LÓGICO

011 - função LITERAL⇒NUMÉRICO

100 - função QUALQUER⇒QUALQUER

101, 110 e 111 -NU

(2): se a função possui operações substitutas

00 - função sem substituto (sin())

01 - função com substituto simples (sqr(x) por x*x)

10 - função com substituto complexo ( odd(x) por if (x MOD 2) <> 0 ...)

11 - NU

Algumas considerações sobre EXPRESSA:.

☑ Não existe necessidade de representar as aspas ('). Pois quando uma CONSTANTE NÃO DECLARADA aparece numa expressão como, por exemplo, em nome:='Fulano de Tal', uma nova entrada é criada na tabela de constantes para a cadeia declarada. Na expressão, associada ao comando de atribuição, a cadeia passa a ser representada pelo seu código (idc:10, idt:100, idi:0000 e stu:01).

☑ A estrutura interna de uma expressão corresponde a um conjunto de inteiros sem sinal (binário), que representam os campos como, por exemplo:

em $b^2$-4ac,
onde a,b e c foram lidas,
**e**
assumindo que:
idi(a)=0000,
idi(b)=0001,
idi(c)=0010 e
idi(4)=0000, temos:

| | IDC IDT IDI STU |
|---|---|
| **b⇒** | 01-000-0001-01 |
| ***⇒** | 00-000-0001-!! |
| **b⇒** | 01-000-0001-01 |
| **-⇒** | 00-000-0111-!! |
| **4⇒** | 10-000-0000-01 |

| | IDC IDT IDI STU |
|---|---|
| ***⇒** | 00-000-0001-!! |
| **a⇒** | 01-000-0000-01 |
| ***⇒** | 00-000-0001-!! |
| c⇒ | 01-000-0010-01 |

!!- não é usado.

### III.4.2 ANÁLISE DO CONHECIMENTO GENÉRICO SOBRE PROGRAMAÇÃO

A solução do estudante, após ser convertida para o formato VIP, é encaminhada para este módulo. Sua função é analisar a correticidade dos planos de programação quanto ao uso do conhecimento básico descrito nas regras pragmáticas.

A análise é feita através de um conjunto de regras, escolhido de acordo com o TIPO do problema. São ativados procedimentos que analisam os planos de programação de acordo com dois grupos de regras: genéricas e orientadas às estruturas de comandos.

Durante o processo de análise, as ocorrências de erros são armazenadas numa tabela para apresentação posterior ao estudante. Ocorrendo erros nesta fase, a análise é concluída e uma mensagem "NOK" é encaminhada ao módulo de tratamento de erros.

O programa deve ser modificado pelo estudante até que as ocorrências de erros desapareçam. Quando o programa obtém sucesso no processo de análise, uma mensagem de "OK" é encaminhada ao módulo que se encarrega da segunda parte da análise, relacionada com as metas do problema.

As regras genéricas consistem de procedimentos construídos para verificação da ocorrência das seguintes inconsistências:

- [Estruturas Incompatíveis com o Tipo de Problema]
  O plano de resolução utiliza estruturas incompatíveis com o TIPO do problema. O estudante deve estar tornando sua solução mais complexa do que o problema requer. A verificação realizada por este procedimento depende do conjunto de comandos definidos para cada TIPO. Por exemplo, uma estrutura de repetição não deve ser usada na resolução de problemas do TIPO I ou TIPO II.

- [Constantes Definidas e Não Usadas]
  O estudante cria um conjunto de constantes desnecessárias à sua solução. A verificação deste procedimento, indicado para todos os TIPOS de problemas, é

UFPb/BIBLIOTECA/PRAI

realizada ao final da verificação completa do código, quando se tem uma descrição de todos os usos dados às constantes criadas no programa.

- [Variáveis Definidas e Não Usadas]
  O estudante criou um conjunto de variáveis desnecessárias à sua solução. A verificação deste procedimento, indicado para todos os TIPOS de problemas, é realizada ao final da verificação completa do código, quando se tem uma descrição de todos os usos dados às variáveis criadas no programa.

As regras, orientadas aos comandos, são as seguintes:

---

⇔ Comandos de Entrada:

- [Variável Perde o Conteúdo Anterior Lido/Atribuído]
  Este procedimento, válido para todos os TIPOS de problemas, verifica se alguma variável lida foi anteriormente lida ou atribuída, sem a utilização do conteúdo fornecido pela leitura ou pela atribuição anterior. A sinalização desta ocorrência pode auxiliar o estudante a entender melhor o processo de armazenamento e recuperação de conteúdos na memória.

---

⇔ Comandos de Escrita:

- [Variável Enviada à Saída Sem Conteúdo]
  Este procedimento, válido para todos os TIPOS de problemas, verifica se alguma variável é utilizada sem conteúdo em um comando de saída. É comum o estudante achar que o processo de declaração é suficiente, não sendo necessária uma inicialização.

---

⇔ Comando de Atribuição:

- [Variável Perde o Conteúdo Anterior Lido/Atribuído]

Este procedimento, válido para todos os TIPOS de problemas, verifica se alguma variável que se encontra à esquerda de uma atribuição foi anteriormente lida ou atribuída, sem a utilização do conteúdo fornecido pela leitura ou pela atribuição.

- [Variável Usada Sem Conteúdo]
  Este procedimento, válido para todos os TIPOS de problemas, verifica se alguma variável utilizada numa expressão de atribuição não foi inicializada previamente.

---

⇔ Comando de Decisão:

- [Variável na Condição Sem Conteúdo]
  Este procedimento, válido apenas para os TIPOS II e III de problemas, verifica se alguma variável sem inicialização é utilizada na expressão da condição.

- [Condição Sempre Verdadeira / Parte ELSE Jamais Será Executada]
  Este procedimento, válido apenas para os TIPOS II e III de problemas, verifica se a condição da estrutura de decisão é uma tautologia e existe a parte ELSE, que jamais será executada. O estudante cria um trecho de código desnecessário.

- [Condição Sempre Negativa / Parte THEN Jamais Será Executada]
  Este procedimento, válido apenas para os TIPOS II e III de problemas, verifica se a condição da estrutura de decisão é sempre falsa.

---

⇔ Comando de Seleção:

- [Constante Usada como Seletora]
  Este procedimento, válido para os TIPOS II e III, verifica se a expressão seletora é constante. A regra para comando de seleção trata esta situação como indesejável.

- [Variável Presente na Condição Seletora Sem Conteúdo]
  Este procedimento, válido para os TIPOS II e III de problemas, verifica se alguma

variável sem inicialização é utilizada na expressão seletora.

- [Variável Presente na Expressão da Opção Sem Conteúdo]
  Este procedimento, válido para os TIPOS II e III de problemas, verifica se alguma variável é utilizada sem conteúdo nas expressões da opção.

---

⇔ Comando de Repetição Automática:

- [Variável de Controle Lida/Atribuída Indevidamente - Perde-se o Controle do Laço]
  Este procedimento, válido para problemas do TIPO III, verifica se a variável de controle é lida ou atribuída no interior do laço. Esta prática é inadequada. O estudante pode desconhecer o funcionamento correto do comando.

- [Variável Presente na Expressão do Limite Inicial Lida/Atribuída Indevidamente - Não é uma boa prática]
  Este procedimento, válido para os problemas de TIPO III, verifica se alguma variável presente na expressão de limite inicial é lida ou atribuída no interior do laço. Esta prática é inadequada, embora alguns compiladores a permita. O estudante deve adquirir maturidade suficiente sobre o funcionamento do comando.

- [Variável Presente na Expressão do Limite Inicial Sem Conteúdo]
  Este procedimento, válido para o TIPO III de problemas, verifica se alguma variável é utilizada sem conteúdo na expressão do limite inicial.

- [Variável Presente na Expressão do Limite Final Lida/Atribuída Indevidamente - Não é uma boa prática]
  Este procedimento, válido para os problemas de TIPO III, verifica se alguma variável presente na expressão de limite final é lida ou atribuída no interior do laço. Esta prática é inadequada, embora alguns compiladores a permita. O estudante deve adquirir maturidade suficiente sobre o funcionamento do

comando.

- [Variável Não Inicializada Presente na Expressão do Limite Final]
  Este procedimento, válido para os problemas de TIPO III, verifica se alguma variável é utilizada sem conteúdo na expressão do limite final.

- [Repetição Automática Jamais Executada - Condições Impróprias nos Limites]
  Este procedimento, válido para os problemas TIPO III, verifica alguns casos onde a avaliação de expressões constantes presentes nos limites impossibilita a execução do laço. O código, portanto, é desnecessário nas circunstâncias de controle do laço.

- [Variável de Tipo Estruturado Usada como Var. de Controle]
  Este procedimento, válido para o TIPO III de problemas, verifica se a variável de controle usada é de um tipo estruturada. Esta não deve ser uma prática de programação incentivada a um estudante iniciante de programação.

---

⇔ Comando de Repetição Condicionada - Teste a Priori:

- [Variável Presente na Condição Sem Conteúdo]
  Este procedimento, válido para o TIPO III de problemas, verifica se alguma variável é utilizada sem conteúdo na expressão da condição da estrutura de repetição.

- [Repetição Condicionada - Teste a Priori - Jamais Executada - Condição Sempre Falsa]
  Este procedimento, válido para o TIPO III de problemas, verifica alguns casos onde a expressão da condição impossibilita a execução do comando. O código é desnecessário nas circunstâncias de controle do laço.

- [Variável da Condição Não é Alterada no Laço - Repetição Gera Loop]
  Este procedimento, válido para o TIPO III de problemas, verifica a condição fundamental ao bom funcionamento da repetição condicional - a alteração, no

interior do laço, de alguma das variáveis presentes na condição de controle. Este é um dos erros mais cometidos pelos estudantes iniciantes em programação.

- [Constante Usada na Condição - Repetição Gera Loop]
  Este procedimento, válido para o TIPO III de problemas, complementa o procedimento anterior, na verificação do bom funcionamento da repetição condicional. Uma expressão constante é inadequada, pois coloca o programa em loop.

---

⇔ Comando de Repetição Condicionada - Teste a Posteriori:

- [Variável Presente na Condição Sem Conteúdo]
  Este procedimento, válido para o TIPO III de problemas, verifica se alguma variável é utilizada sem conteúdo na expressão da condição do comando de repetição.

- [Variável da Condição Não é Alterada no Laço - Repetição Gera Loop]
  Este procedimento, válido para o TIPO III de problemas, verifica a condição fundamental ao bom funcionamento da repetição condicional - a alteração, no interior do laço, de alguma das variáveis presentes na condição de controle.
- [Constante Usada na Condição - Repetição Gera Loop]
  Este procedimento, válido para o TIPO III de problemas, complementa o procedimento anterior, na verificação do bom funcionamento da repetição condicional. Uma expressão constante é inadequada à estrutura.

### III.4.3 ANÁLISE DO CONHECIMENTO ESPECÍFICO ÀS METAS

Uma vez analisados os conceitos genéricos de programação, o programa do estudante passa a ser analisado quanto à sua adequação às metas específicas presentes na especificação do problema.

As metas específicas, armazenadas para cada um dos problemas cadastrados no Banco de Problemas, são analisadas a partir de processos que relacionam a estrutura da solução ótima (e/ou alternativas) armazenada e a estrutura da solução do estudante no formato VIP. O ANEXO D descreve a estrutura de dados definida para armazenamento das metas específicas dos problemas.

Durante o processo de análise, as ocorrências de erros são armazenadas na tabela de erros para serem tratadas posteriormente. Na inexistência de erros, é encaminhada uma mensagem "OK" ao gerenciador de eventos do ambiente. Havendo erros, uma mensagem "NOK" é encaminhada ao módulo de tratamento de erros. O programa deve ser modificado até que os erros desapareçam ou que o estudante interrompa o processo de análise.

A comparação das metas específicas se dá, inicialmente, a partir da comparação do esqueleto estrutural das duas soluções - a armazenada pelo instrutor e a proposta pelo estudante. Em seguida, a análise é feita comparando-se cada um dos planos de programação encontrados na solução ótima, com os planos equivalentes apresentados pelo estudante. Caso o casamento não seja obtido, sinônimos naturais de expressões e/ou comandos são recuperados e comparados na busca de equivalência.

Esgotadas as possibilidades de geração de versões da solução ótima, sem sucesso na comparação, as soluções alternativas, quando existentes, passam a ser comparadas, seguindo o mesmo procedimento adotado com a solução ótima.

Como se trata da análise de produtos da "criação humana", o VIP deverá se deparar com soluções de estudantes, que mesmo não sendo equivalentes às formas de resolução armazenadas ou àquelas das versões geradas durante o processo de comparação, estejam corretas. Nestes casos, o VIP nada poderá fazer. Outra ferramenta do ambiente DOCET, o Simulador de Execução de Programas - SEP, entra em cena, recebendo a massa de testes armazenadas e a solução do estudante para execução. Tal processo deverá ser acompanhado pelo estudante para visualização dos resultados de cada comando usado no programa até a obtenção do resultado final.

### III.4.4 TRATAMENTO DE ERROS

Nas duas etapas de análise, o VIP cria uma tabela de erros/advertências com os códigos individuais e as linhas do programa onde eles ocorreram.

Os códigos individuais são chaves de acesso às mensagens armazenadas pelo instrutor, cujos conteúdos são apresentados ao estudante pelo editor do ambiente, o EDOL.

Este módulo do VIP tem a função de gerar o arquivo com esses códigos e linhas, que é enviado ao EDOL ao final de cada etapa de análise do VIP.

Para a primeira etapa da análise, a verificação dos conceitos básicos de programação, foi definido um conjunto de mensagens que é válido para o subconjunto da linguagem PASCAL em uso. As mudanças futuras neste subconjunto poderá incidir em alterações nas mensagens. Abaixo, a relação de erros e/ou advertências cadastradas.

⇨ Erros de Origem Pragmática:

☑ Uso de variável não inicializada;
☑ Alteração da variável de controle no corpo do for;
☑ Loop Infinito. Não há alteração da condição;
☑ O uso da estrutura IF é incompatível com o problema;
☑ O uso da estrutura CASE é incompatível com o problema;
☑ O uso da estrutura FOR é incompatível com o problema;
☑ O uso da estrutura WHILE é incompatível com o problema;
☑ O uso da estrutura REPEAT é incompatível com o problema;
☑ Loop Infinito. A condição do WHILE só possui constantes; e
☑ Loop Infinito. A condição do REPEAT só possui constantes.

⇨ Advertências de Origem Pragmática:

☑ Constante declarada e não utilizada;

- ☑ Variável declarada e não utilizada;
- ☑ O conteúdo lido para a variável não foi usado;
- ☑ O conteúdo atribuído à variável não foi usado;
- ☑ O conteúdo lido anteriormente para a variável não foi usado;
- ☑ O conteúdo atrubuído anteriormente à variável não foi usado;
- ☑ Atenção! Variável inicializada por resíduo de comando FOR;
- ☑ A variável da expressão limite (inicial ou final) não deve ser alterada internamente;
- ☑ A condição do IF é sempre verdadeira. O else jamais será executado;
- ☑ Atenção! A condição do IF é constante. O then ou else jamais será executado;
- ☑ A condição do IF é sempre falsa. O then jamais será executado;
- ☑ Atenção! Foi utilizada uma expressão constante como seletora do CASE. Algumas opções jamais serão executadas.
- ☑ O limite inicial do FOR crescente é maior que o limite final. O FOR jamais será executado;
- ☑ O limite inicial do FOR decrescente é menor que o limite final. O FOR jamais será executado; e
- ☑ A condição do WHILE é sempre falsa, ele jamais será executado.

Para a segunda etapa da análise, a verificação dos erros ou advertências, relacionadas às metas específicas de cada um dos problemas, é contextual. Foi criada uma tabela de mensagens, acessível ao instrutor, de onde ele deve selecionar o código correspondente à mensagem que melhor se adeque à cada situação. Quando se fizer necessário, ele poderá acrescentar novas mensagens à tabela.

Como as mensagens de erros, relacionadas à segunda etapa da análise, relatam situações que serão inseridas pelo instrutor no Banco de Problemas, alguns exemplos, dados no exemplo abaixo, ilustram o formato desses erros.

Especificação do Problema:

**Calcular a média aritmética de dez números inteiros lidos e imprimir o resultado obtido.**

Solução proposta inicialmente pelo estudante:

```
program calcula_media;
var
  i, num, total: integer;
  media: real;
  begin
          read(media);
          for i:=1 to 10 do
          begin
                  writeln('Digite um numero inteiro');
                  read(num);
                  total:=total + i;
          end;
  media:= total / 10;
  writeln('A media obtida e:',media);
end.
```

Ocorrências de erros:

(Apresentadas ao estudante após a primeira etapa da análise)

linha 11: Uso de variável não inicializada.

linha 14: O conteúdo lido anteriormente para a variável não foi usado.

Solução proposta inicialmente modificada para a segunda etapa da análise:

```
program calcula_media;
var
  i, num, total: integer;
  media: real;
  begin
          total:= 0;
```

```
          for i:=1 to 10 do
          begin
                    writeln('Digite um numero inteiro');
                    read(num);
                    total:=total + i;
          end;
  media:= total / 10;
  writeln('A media obtida e:',media);
end.
```

Ocorrências de erros:

(Apresentadas ao estudante após a segunda etapa da análise)

linha 10: O conteúdo da variável lida não está sendo usado.
linha 11: Expressão incorreta.

## III.4.5 INTERFACE COM O USUÁRIO

O VIP não se comunica diretamente com o estudante. Ele utiliza a interface padronizada do DOCET, através do Editor, que apresenta o código fonte em uma janela e os erros enviados pelo VIP em outra. Esta decisão de projeto deve-se ao fato das ferramentas do ambiente terem sido projetadas de forma modular e ortogonal, evitando duplicação de funcionalidade.

Em seguida, alguns exemplos de telas apresentadas ao estudante pelo EDOL.

```
COL  21   LIN  9      Ins    ->       edol    FATIMA\DOCET\EDOL\media.pas
                                   EDOL
program media;
var
   numero:integer;                        ---- Identificadores ----
   media, total :integer;                 MEDIA             variavel
                                          NUMERO            variavel
begin                                     TOTAL             variavel
   writeln('Este programa acha a media
   writeln;
   for n:=1 to 10 do
      begin
         writeln( 'Digite o ', n, 'nume
         read( numero );

         total := to              Er
      end;            Variavel não decla ESC-sai I-Inclui E-Exclui A-Altera
   media := total /
   writeln('A media      ESC-ignora  A-ajuda  E-edita
end.

F1-Help F2-Grava F3-Identif. F4-Funções F5-Reservadas F8-Edol F9-AS F10-Menu
```

Tela 1: O EDOL assiste à digitação do programa.

```
Col 1     Lin 8       Ins   ->          edol          c:\usr\aluno\media.pas
                                 EDOL
 PROGRAM MEDIA_IDADES;
 VAR  IDADE, NUM_PESSOAS, SOMA_IDADES : INTEGER;
 BEGIN
      NUM_PESSOAS  := 0;
      SOMA_IDADES  := 0;
      IDADE        := 0;
      READ (IDADE);
      WHILE (IDADE <> 0) THEN BEGIN
           NUM_PESSOAS  := NUM_PESSOAS + 1;
           SOMA_IDADES  := SOMA_IDADES + IDADE;
      END;
      IF (NUM_PESSOAS < 0) THEN
         WRITELN ('NAO FOI LIDA IDADE VALIDA . ');
      ELSE WRITELN (' IDADE MEDIA = ', SOMA_IDADES DIV NUM_PESSOAS );
 END.
 F1-Ajuda  F2-Grava  F3- Identif.  F4-Funções  F5- Reservadas  F6- Alterna  ESC-Sai

 LIN.07 : VARIÁVEL ALTERADA - VALOR NÃO USADO
 LIN.08 : LOOP INFINITO - CONDIÇÃO INALTERADA

 <ENTER> - Seleciona
```

Tela 2: O EDOL apresenta os erros detectados pelo VIP na primeira etapa da análise

## III.5 METODOLOGIA DE DESENVOLVIMENTO

As decisões tomadas no processo de implementação do VIP, em parte, são vinculadas ao DOCET - o ambiente onde a ferramenta vai atuar. Isto é, a sua caracterização como um ambiente de ensino baseado no conhecimento, levou-nos a considerar alguns aspectos:

- a integração e coexistência com metodologias diversas, abordadas em outras ferramentas já existentes no DOCET;
- uma interface padronizada com o usuário (instrutor e/ou estudante);
- a possibilidade de incorporação gradativa de novas ferramentas;
- a representação interna do conhecimento, compartilhado com outras ferramentas do DOCET.

Ao longo do processo de especificação do VIP, os aspectos citados foram fundamentais para a delimitação do seu escopo de atuação. A prototipação dos módulos estruturais do VIP, associados a própria filosofia de representação de conhecimento adotada, determinaram a dinâmica da implementação.

Em cada etapa da implementação e depuração do VIP, foram feitas anotações, que integram a documentação da ferramenta para suporte às decisões de futuras alterações.

O processo de discussão sobre a proposta de desenvolvimento de uma ferramenta para detecção de erros lógicos, o estudo de ambientes ou propostas já existentes e a idéia de integração presente no DOCET, mais a especificação do VIP, demandaram um período corrido de 180 dias de trabalho.

### III.5.1 AMBIENTE ESCOLHIDO PARA O DESENVOLVIMENTO

A escolha da utilização da linguagem "C" para o desenvolvimento do VIP levou em

consideração a integração com as demais ferramentas do DOCET e a utilização de recursos de gerenciamento de memória disponíveis.

A estrutura de dados especificada requer uma forte utilização de "ponteiros", pela complexa rede de informação que é construída. A manipulação desse tipo de dado, apresentou-se de forma bastante atrativa no ambiente usado.

Além disso, ao se criar uma linguagem para representação de expressões, observou-se a necessidade de se obter uma ocupação mínima de memória, concretizada com a utilização de estruturas baseadas na manipulação de "*bits*".

A implementação desta versão do VIP está disponível para plataformas baseadas em IBM PC compatíveis.

A implementação do VIP envolveu a criação de 6500 linhas de código. É importante ressaltar, que pela própria característica da ferramenta proposta, pouco explorada em pesquisas locais, melhorias na implementação foram feitas a partir de alguns experimentos realizados. Considerando o tempo total de implementação e validação da ferramenta, de forma corrida, totalizaram-se 5 (cinco) meses de trabalho.

## III.5.2 DEPURAÇÃO E VALIDAÇÃO

Os módulos estruturais do VIP foram depurados isoladamente, numa primeira etapa, para validação funcional de cada módulo e obtenção de resultados parciais a partir da utilização de um subconjunto de problemas selecionados do Banco de Problemas. Numa segunda etapa, a validação do VIP se deu a partir da análise de um conjunto mínimo de três problemas para cada uma das nove classes cadastradas no Banco de Problemas.

Nesta fase de depuração e validação, participaram dois estudantes de iniciação científica. As suas atividades compreenderam a exploração e depuração do VIP e cadastramento de problemas no Banco.

É importante ressaltar as contribuições de outros participantes do projeto DOCET, ao emitirem suas opiniões em reuniões coletivas realizadas pelo grupo coordenado pela Profa. Fátima Camêlo.

# CAPÍTULO IV

# CONSIDERAÇÕES FINAIS

# CAPÍTULO IV: CONSIDERAÇÕES FINAIS

As pesquisas até aqui desenvolvidas, tentam mostrar que a atividade de programação tem um efeito significativo sobre o desenvolvimento cognitivo e sobre a transferência de habilidades de um domínio de conhecimento para outro [LINN 92]. Embora as evidências ainda sejam insuficientes para tal afirmação, não se pode deixar de considerar que o processo de abstração resultante da interação homem/máquina é bastante original e rico, necessitando ser mais explorado.

A linguagem de programação pode representar um fator complicador para o estudante, se este não tiver associado o conhecimento sobre a linguagem ao conhecimento da matéria envolvida no problema a ser resolvido. O estudante não pode esquecer que para fazer um programa que verifique se um triângulo é isósceles, ele necessita dominar conceitos e saber "como fazê-lo".

Por outro lado, o processo de programação, em si, não pode ser associado a nenhuma "heurística geral". A atividade de programar pode ser produtiva, desde que o estudante compreenda o vínculo existente entre as várias etapas: iniciando com a compreensão do problema, passando pela formulação (planejamento) da solução, codificação e teste do programa obtido.

O projeto e desenvolvimento do VIP, foi um desafio, que gerou vários momentos de incerteza seguidos de uma satisfação estimulante. Ao se fazer a análise de tarefa do desenvolvimento de programas, tornaram-se explícitas as situações onde o "modo de pensar" do estudante iniciante levam à ocorrência de erros. O entendimento dessas situações foi fundamental para se tentar entender os processos de raciocínio e definir a forma de tratamento presente no VIP.

Foi oportuno alcançar um grau de maturidade sobre o assunto para a caracterização do produto final deste trabalho. As pesquisas nesta área carecem ainda de resultados significativos. Acredita-se que este trabalho contribui para acrescentar algo ao processo de

investigação.

Nesta última etapa, a experiência do trabalho integrado, com a participação de estudantes de iniciação científica (IC) e outros mestrandos, todos engajados no projeto do ambiente DOCET, trouxe uma nova dinâmica, possibilitando a visualização de pontos de continuidade.

Neste sentido, parte do trabalho de especificação do VIP, propiciou o desenvolvimento de um dos projetos de IC - o Analisador Sintático e, o pontapé inicial de outras duas teses de mestrado (o Gerenciador do Banco de Problemas e o Simulador de Execução de Programas), ambas em andamento.

Uma sugestão para implementação futura, é o desenvolvimento de uma ferramenta que possibilite a construção de programas através de icones, propiciando ao instrutor uma maior produtividade na autoria de programas. Uma outra sugestão encontra-se relacionada à possibilidade de se fazer a verificação automática das soluções corretas, construídas pelo estudante, que não casaram com as geradas no ambiente, a partir da massa de testes armazenada. Caso a solução se apresente correta, deveria ser automaticamente armazenada no Banco.

Nas expansões previstas para o DOCET, imagina-se o desenvolvimento de versões para "estações de trabalho". Considera-se que a utilização da linguagem "C", no desenvolvimento do VIP, é um facilitador importante para este objetivo a médio prazo.

Como sugestões finais para outros trabalhos, apontam-se o desenvolvimento de versões do VIP para outros domínios de linguagens de programação procedural e o desenvolvimento de um analisador inteligente de expressões. A análise inteligente de expressões, possivelmente com o uso de técnicas de Inteligência Artificial, prevê a prova de teoremas e refutação para se chegar a algo mais eficaz.

## REFERÊNCIAS

[BALZ 85] Robert Balzer - "A 15 Year Perspective on Automatic Programming", IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, pp 1257-1268, 1985.

[BARS 85] David R. Barstow - "Domain-Specific Automatic Programming", IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, pp 1321-1335, 1985.

[BOEH 87] Barry W. Boehm - "Improving Software Productivity", Computer, Setembro, pp 43-57, 1987.

[BONA 88] Jefrey Bonar & Robert Cunningham - "Bridge: An Intelligent Tutor for Thinking about Programming", Artificial Intelligence and Human Learning, Chapman and Hall, pp 391-409, 1988.

[BOUL 86] Benedict Du Boulay - "Some Difficulties of Learning to Program", Journal Educational Computing Research, Vol. 2(1), pp 57-73, 1986.

[BUSH 45] V. Bush - "As We May Think", Atlantic Monthly, pp 101-108, 1945.

[CAME 91] M. F. Camêlo & J. A. B. Moura - "Um Ambiente para Ensino/Aprendizagem de Programação", Anais do I Congresso IberoAmericano de Educacion Superior en Computacion, Outubro, pp 71-80, Rio de Janeiro, 1991.

[COHE 92] Jacques Cohen - "First Specialize. Then Generalize", Communications of the ACM, Vol. 35, No. 3, pp 34-39, 1992.

[COWL 87] M. F. Cowlishaw - "LEXX - A Programmable Structured Editor", IBM Journal Research Development, Vol. 31, No. 1, pp 73-80, 1987.

[ELSO 88] Mark Elson-Cook & Benedict du Boulay - "A Pascal Checker", Artificial

Intelligence and Human Learning, Chapman and Hall, pp 361-373, 1988.

[FETZ 88] James H. Fetzer - "Program Verification: The Very Idea", Communications of the ACM, Vol. 31, No. 9, pp 1048-1062, 1988.

[HEND 87] Peter B. Henderson & David Notkin - "Integrated Design and Programming Environments", Computer, Novembro, pp 12-16, 1987.

[JOHN 87] W. Lewis Johnson & Elliot Soloway - "PROUST: An Automatic Debugger for Pascal Programs", Artificial Inteligence in Instruction, Addison-Wesley, pp 49-67, 1987.

[LINN 92] Marcia C. Linn & Michel J. Clancy - "The Case for Case Studies of Programming Problems", Communications of the ACM, Vol. 35, No. 3, pp 121-132, 1992.

[LUCE 87] Carlos Lucena - Inteligência Artificial e Engenharia de Software, Jorge Zahar Editor, Livro, 1987.

[MIKI 92] Miki Hiroyuki - "Especificação de uma Ferramenta para criação de Módulo de Ajuda Baseada em Hipertexto - HiperHelp", Fevereiro, Dissertação de Mestrado, COPIN/UFPb, 1992.

[POLY 75] G. Polya, "A Arte de Resolver Problemas", Editora Interciência, 1975.

[REPS 87] Thomas Reps & Tim Teitelbaum - "The Synthesizer Generator", Computer, Novembro, pp 29-40, 1987.

[SOLO 83] Elliot Soloway - "Meno-II: An AI-Based Programming Tutor", Journal of Computer-Based Instruction, Vol. 10, Nos. 1&2, 20-34, 1983.

[SPOH 86] J. C. Spohrer & Elliot Soloway - "Novice mistakes: Are the folk wisdoms correct?", Communications of the ACM, Vol 29, pp 624-632, 1986.

[VYGO 62] L. Vygotsky - "Thought and Language", MIT Press, Cambridge, 1962.

[WATE 85] Richard C. Waters - "The Programmer's Apprentice: A Session with KBEmacs", IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, pp 1296-1320, 1985.

[WEIS 87] Mark Weiser, "Source Code", Computer, Setembro, pp 66-73, 1987.

# ANEXOS

# ANEXO A

DESCRIÇÃO SUCINTA DO AMBIENTE DOCET

O ambiente DOCET objetiva, de forma interativa, apoiar a aprendizagem de estudantes de programação, ressaltando aspectos da resolução de problemas, analisando os conhecimentos pragmáticos sobre programação, o uso da linguagem PASCAL e as soluções apresentadas para problemas específicos.

O ambiente, apresenta ainda, um método alternativo para a avaliação do conhecimento e das atitudes de estudantes de programação. O método proposto faz uso de métricas da engenharia de software para aferir as habilidades apreendidas pelos estudantes.

As tarefas desempenhadas pelo DOCET no processo de ensino-aprendizagem são obtidas por um conjunto de ferramentas:

- um Sistema de Autoria - COMPOSER;
- um Editor Orientado à Linguagem - EDOL;
- um Analisador Sintático - AS;
- um Gerenciador de Banco de Problemas - GPB;
- um Gerenciador de Banco de Usuários - GBU;
- um Verificador Interativo de Programas - VIP;
- um Simulador de Execução de Programas - SEP;
- um Analisador de Padrões de Qualidade de Programas - QUANTUM.

A figura 1 ilustra o esquema global do ambiente.

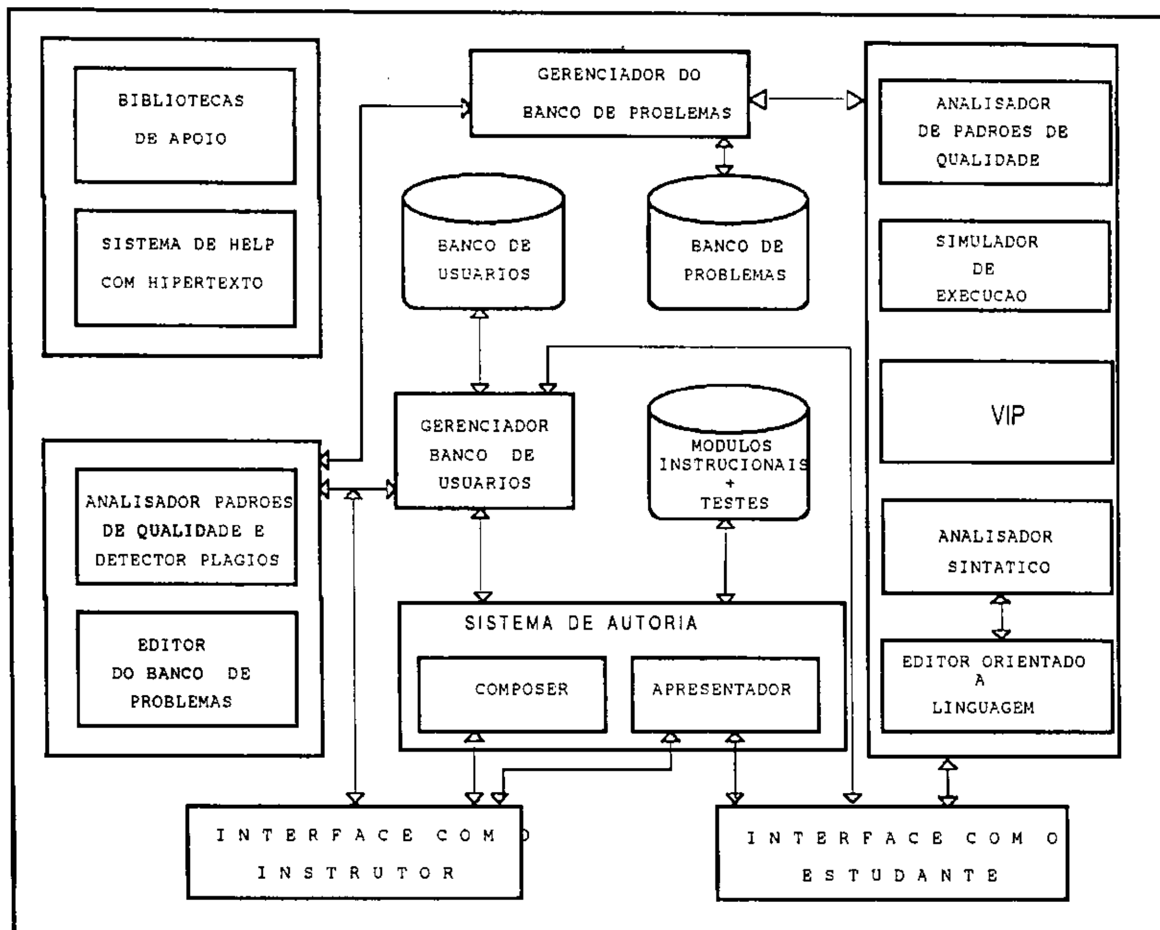



Figura 1: DOCET: Ambiente para Ensino e Treinamento de Programação

As ferramentas foram especificadas de forma modular e ortogonal, possibilitando que sejam feitas combinações de acordo com as necessidades da disciplina a ser ensinada. Um sistema independente de ajuda, baseado em hipertexto [MIKI 92], integra o ambiente oferecendo ajuda sensível ao contexto e personalizada.

O Sistema de Autoria é usado para elaborar e apresentar o material instrucional, testes, exercícios e documentações, oferecendo ao usuário, em ambiente amigável e simplificado:

- acesso padronizado aos recursos computacionais, tais como: editores, formatadores, gerenciador de banco de dados, entre outros;

- facilidades para a elaboração e apresentação de eventos instrucionais sobre

resolução de problemas e prática de programação;

- acompanhamento do estudante, permitindo-o direcionar o uso do ambiente de acordo com seus interesses, erros ou dúvidas.

O Editor Orientado à Linguagem oferece facilidades de edição totalmente integradas com o ambiente, dispondo de um nível de ajuda compatível com o conhecimento retido pelo estudante iniciante. O estudante pode construir o programa totalmente no editor ou importar um programa editado em outro ambiente para análise e possíveis melhoramentos. Após a edição, o programa é submetido ao analisador sintático, que se comunica com o editor para apresentação/correção dos erros detectados. Quando o programa está sintaticamente correto, o estudante pode submetê-lo ao VIP para análise lógica.

O Gerenciador de Banco de Problemas mantém um banco de informações sobre os problemas cadastrados no ambiente. Estas informações, detalhadas no capítulo IV, são introduzidas pelo instrutor que é responsável pela definição das metas e submetas das soluções cadastradas. Um diálogo se dá entre o ambiente e o instrutor para aquisição de informações adicionais acerca de permissões e restrições na implementação do código mínimo de uma solução e de suas alternativas.

O Gerenciador do Banco de Usuários mantém um cadastro de estudantes e instrutores, com informações sobre:

- tipo do usuário (instrutor/estudante);
- módulos instrucionais a serem estudados (para o estudante);
- lista de problemas cadastrados no Banco de Problemas (pelo instrutor);
- lista de estudantes cadastrados (para o instrutor);
- roteamento das sessões de estudo (sobre o estudante);
- lista de problemas do Banco de Problemas já resolvidos (para o estudante).

O Verificador Interativo de Programas - VIP, é usado para assistir os estudantes no reconhecimento, entendimento e correção de erros lógicos oriundos do mau uso das regras de programação ou no mau entendimento das metas do problema.

O Simulador de Execução de Programas recebe os programas, sintaticamente corretos, para simulação de sua execução. Opcionalmente, o estudante pode requisitar a execução de uma solução armazenada pelo instrutor no Banco de Problemas. Durante a simulação são apresentadas janelas contendo:

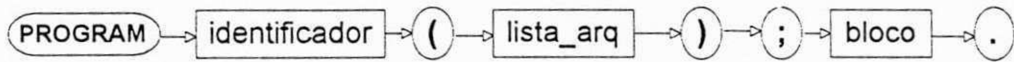
- o código fonte, destacando o trecho em execução;
- o resultado parcial da execução de cada comando;
- uma tabela com as variáveis e os valores já assumidos até aquele instante;
- a solicitação de entrada de dados quando necessário;
- a apresentação das saídas;
- a massa de dados para teste.

O Analisador de Padrões de Qualidade de Programas utiliza métricas da engenharia de software para proceder a avaliação da qualidade e complexidade do programa. Advertências sobre dificuldades de entendimento ou de manutenção posterior podem ser apresentadas ao estudante para que melhoramentos sejam efetuados. O ambiente permite ainda, a análise de um conjunto de programas para indicação de possíveis plágios.

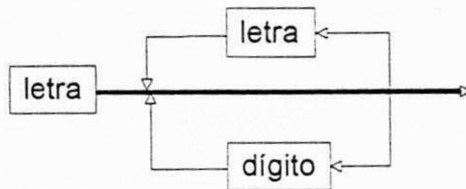
# ANEXO B

SUBCONJUNTO DA LINGUAGEM PASCAL TRATADO PELO VIP

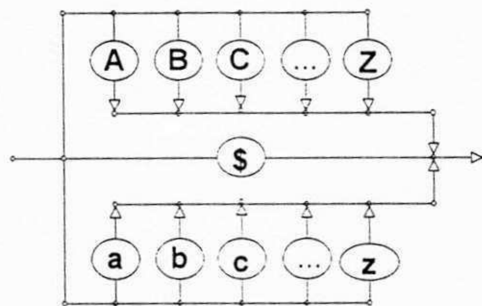
Programa



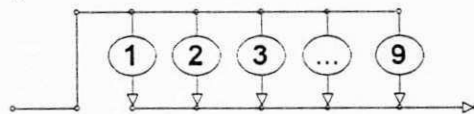

identificador



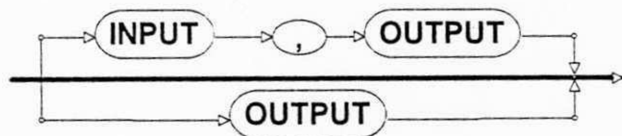

letra



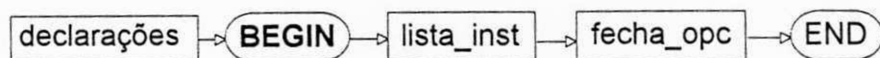

dígito



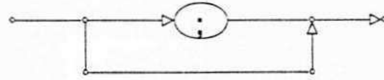

lista_arq



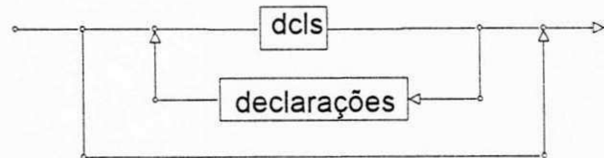

bloco



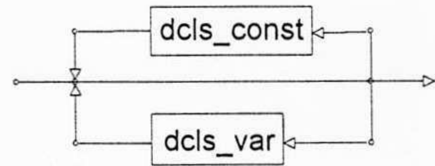
fecha_opc



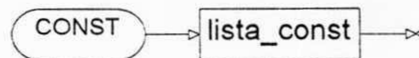

declarações



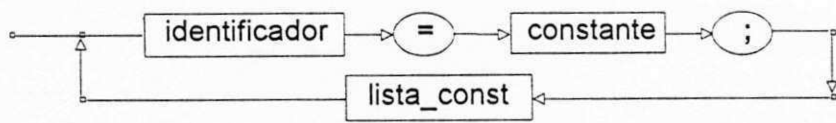

dcls



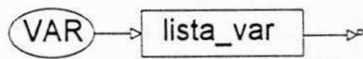

dcls_const



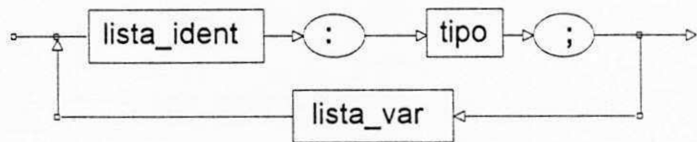
lista_const



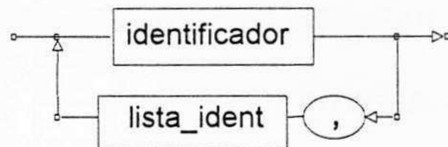

dcls_var



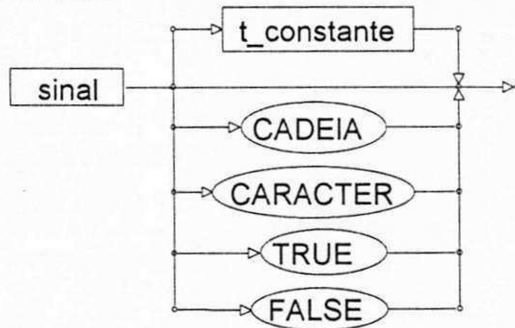

lista_var




lista_ident

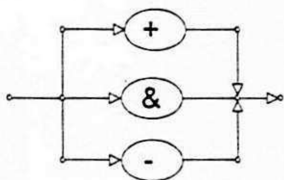



constante

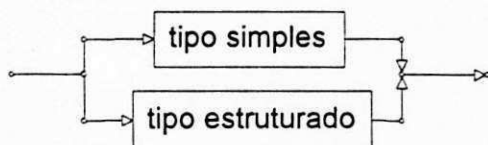



t-constante

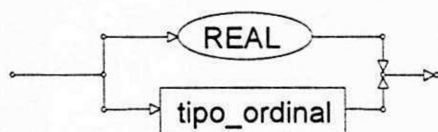
sinal



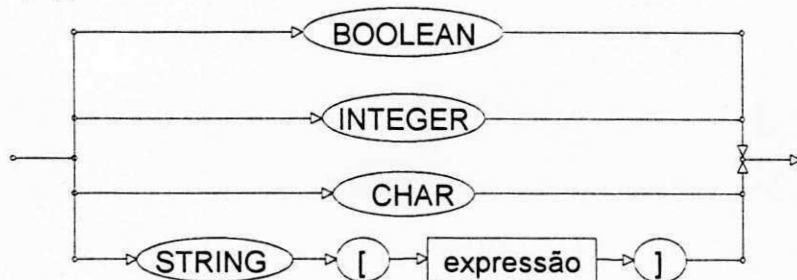

tipo



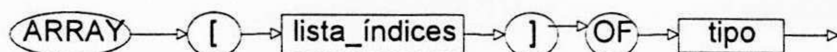

tipo_simples



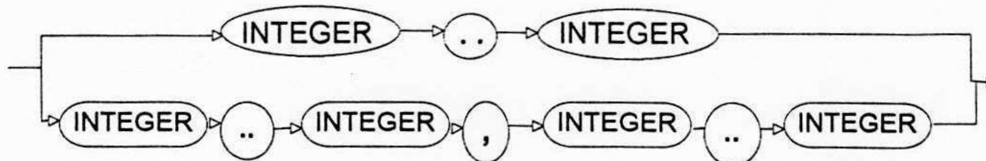

tipo_ordinal



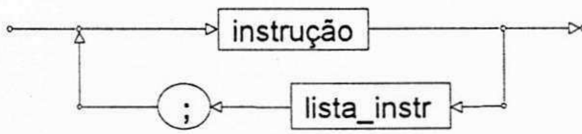

tipo_estruturado



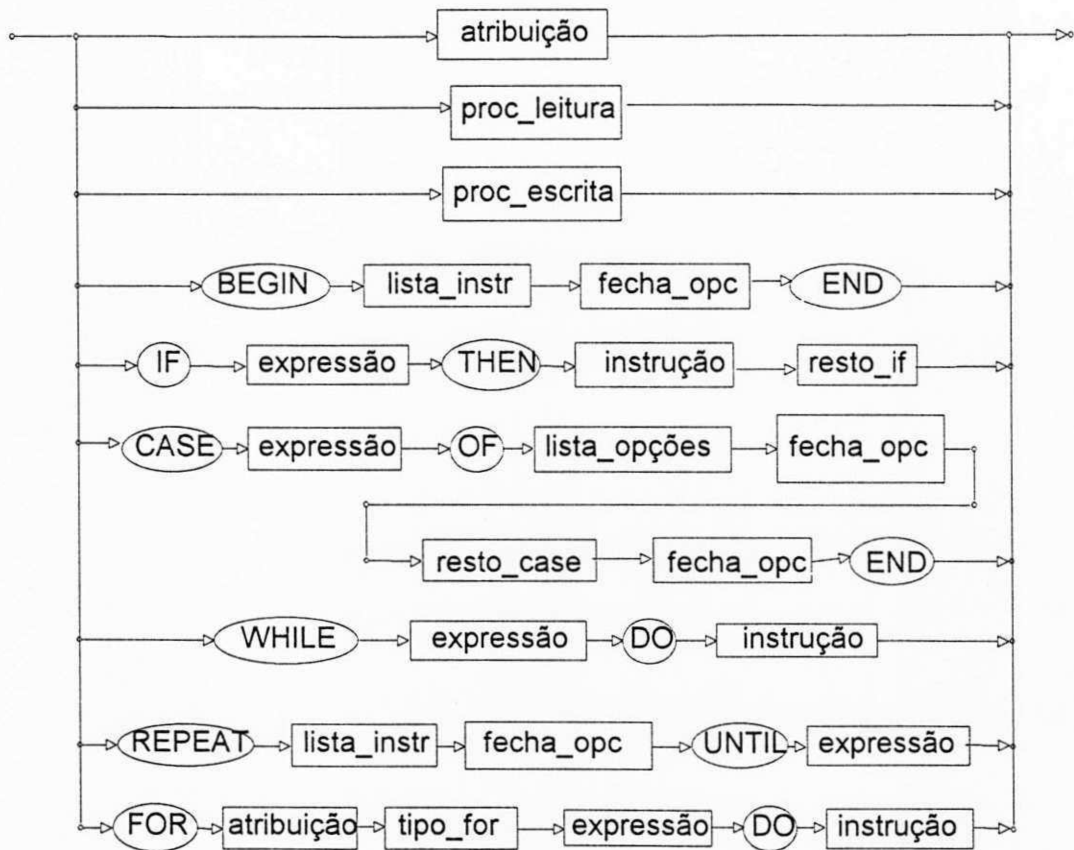

lista_índices



lista_instr

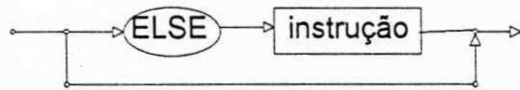



instrução

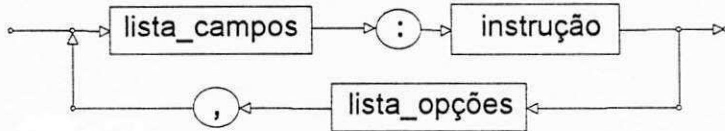



atribuição

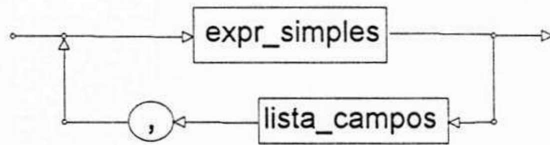
resto_if



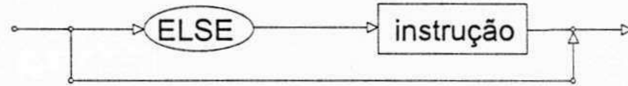

lista_opções



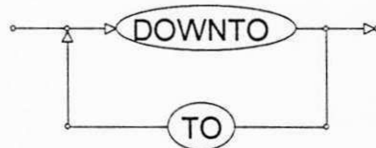

lista_campos



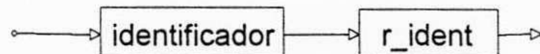

resto_case



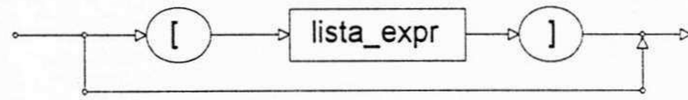

tipo_for



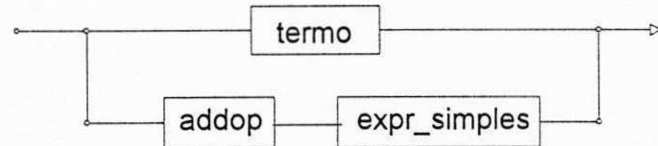

variável



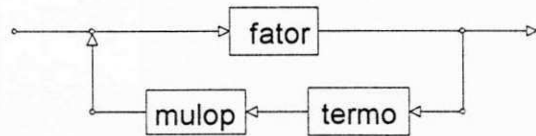
r_ident



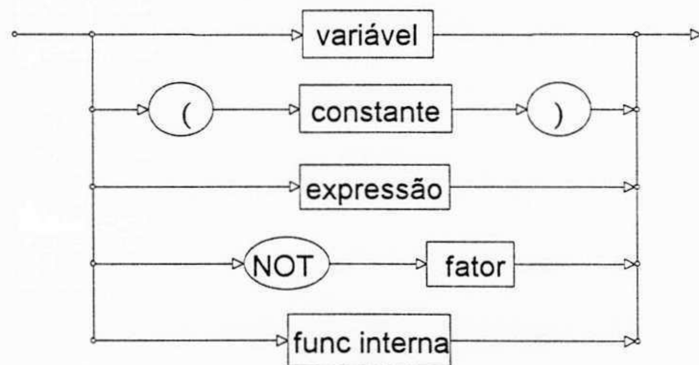

expr_simples



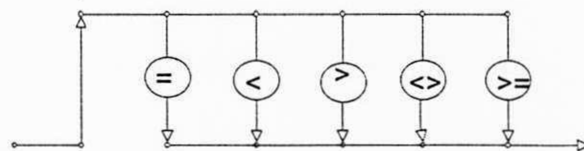

termo



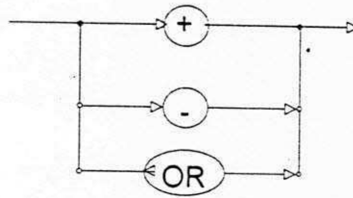

fator



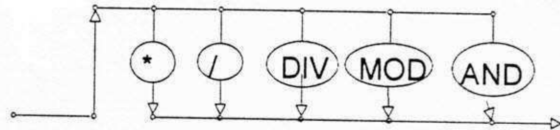

relop



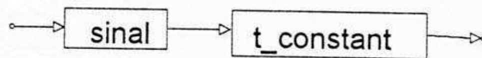
addop



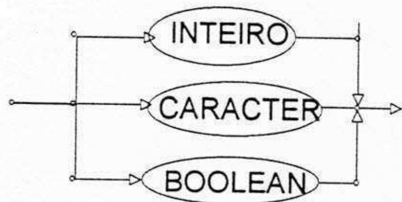

mulop



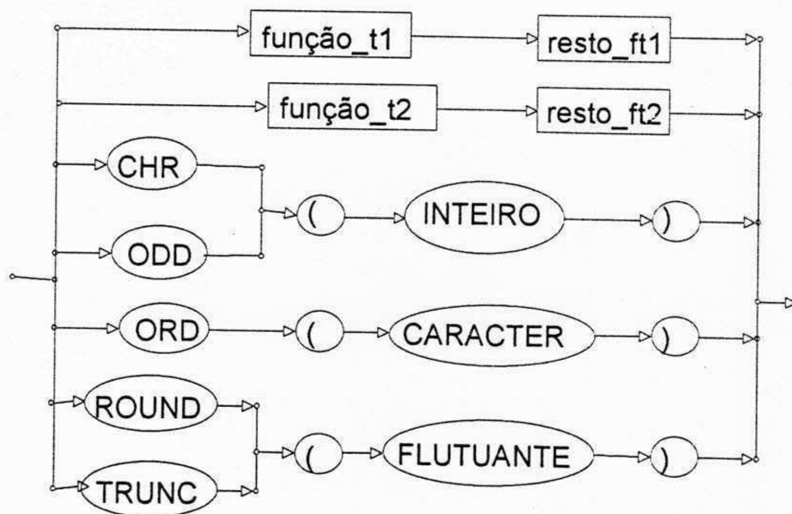

número



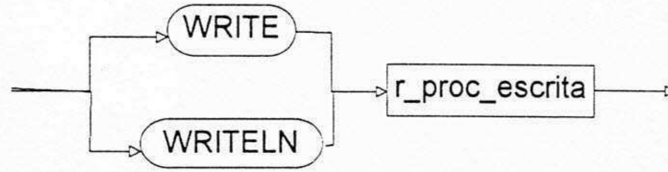

valor




func_interna

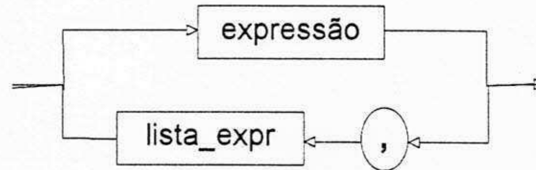


proc_escrita

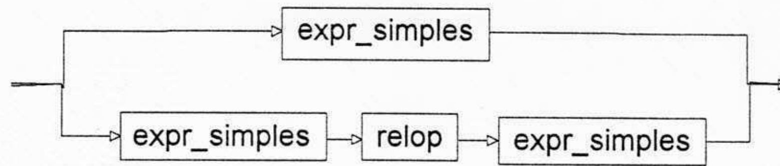



r_proc_escrita

lista_expr




expressão



# ANEXO C

FORMATO DA SOLUÇÃO VIP

## Estrutura de Dados Usada na Solução VIP

Para efeito de verificação da existência de inconsistências lógicas, oriundas do mau uso de conceitos básicos de programação ou do mau entendimento das metas específicas do problema, fez-se necessária a estruturação do programa em três níveis básicos de informação, descritos pelos objetos a seguir.

1. Identificação Geral do Programa - IDENTIFICAÇÃO

---

[OBJETO_ID] - consiste de informações sobre o programa cadastrado para recuperação de suas metas armazenadas pelo instrutor no Banco de Problemas:

- identificação do programa a partir dos códigos referentes à classe, tipo, nível de complexidade e número de sequência correspondentes;
- nome dado pelo estudante ao programa; e
- linha onde se encontra a declaração desta estrutura no programa do estudante.

---

2. Declarações no Programa - DECLARAÇÃO

---

[OBJETO_DCLS] - é opcional para alguns casos de programas do TIPO I (onde não haja necessidade de declaração de dados). Consiste de informações sobre as definições das constantes e variáveis.

As informações referentes às constantes, quando houver, são armazenadas em um objeto específico com o seguinte *layout*:

---

[OBJ_CONST] - contém informações sobre cada uma das constantes criadas no programa:

- tipo;
- forma de definição, isto é, se a constante foi criada com uma declaração ou usada diretamente no contexto de uma expressão;
- linha no programa onde ela é definida;
- tipo da constante (inteiro, real, cadeia de caracteres ou lógico);

- informações sobre os comandos onde a constante é usada:

  [OBJ_USO] - contém informações sobre cada estrutura do programa que utiliza a constante:
  - código individual;
  - tipo;
  - próximo;
  - anterior.

De forma semelhante, as informações referentes às variáveis, quando houver, são armazenadas em um objeto específico com o seguinte *layout*:

[OBJ_VAR] - contém informações sobre cada uma das variáveis encontradas no programa:

- linha no programa onde ela é definida;
- tipo simples (inteira, real, char ou lógica) ou estruturado (cadeia, array-uni ou array-bi). No caso dos tipos estruturados, algumas informações são adicionadas:

  [CADEIA] - contém informações específicas às cadeias:
  - tamanho máximo.

  [ARRAY_UNI] - contém informações específicas aos vetores:
  - índice inferior;
  - índice superior;
  - tipo dos elementos (simples ou estruturado);
  - informações adicionais para os elementos estruturados.

[ARRAY_BI] - contém informações específicas às matrizes:

- índice inferior da linha;
- índice superior da linha;
- índice inferior da coluna;
- índice superior da coluna;
- tipo dos elementos (simples ou estruturado);
- informações adicionais para os elementos estruturados.

- informações sobre os comandos onde a variável é alterada (read ou assign) ou referenciada:

[OBJ_ALT] - contém informações sobre cada uma das estruturas do programa onde a variável é inicializada ou alterada:

- código individual;
- tipo;
- próximo;
- anterior.

[OBJ_USO] - contém informações sobre cada uma das estruturas do programa onde a variável é usada:

- código individual;
- tipo;
- próximo;
- anterior.

No caso de variáveis estruturadas, aceita-se no máximo matriz de matriz, esta última de elementos de tipo simples. Considerou-se o nível de complexidade das estruturas de dados requeridas nos programas cadastrados no Banco para esta decisão.

3. Bloco de Comando - BLOCO

[OBJETO_BLOCO] - é obrigatório para todos os TIPOS de programas a serem analisados. Entretanto, algumas das estruturas não são usadas nos TIPOS I (programas lineares) e II (programas com ramificações), por exemplo, as estruturas de repetição. O BLOCO é gerado levando em consideração o fluxo de controle dos comandos no programa.

Também é permitida a recuperação de comandos de mesmo tipo pela ordem de aparecimento no programa (por exemplo, a lista dos comandos Read ou For). O uso desta dupla forma de recuperação torna o processo de análise mais flexível.
Para cada comando do BLOCO são armazenadas as seguintes informações:

- identificação individual do comando no programa;
- tipo do comando (read(ln), write(ln), assign, if, case, for, while ou repeat);
- nível de embutimento do comando no programa, isto é, o contexto onde ele se encontra definido. Para o nível principal, associa-se o valor 0 (zero). Para os demais níveis, associa-se como valor a identificação individual da estrutura de comando mais externa. A figura 8 ilustra essa representação;

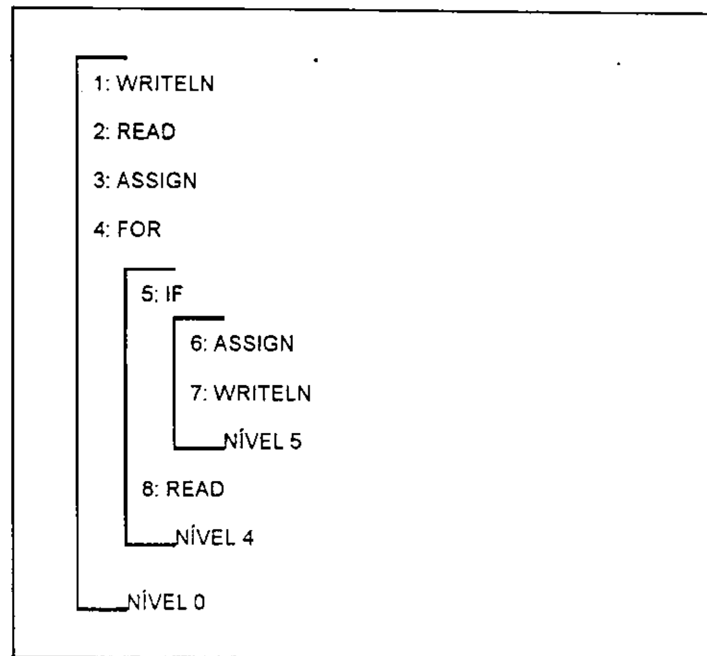



Figura 8: Representação do Nível de Embutimento

- linha onde inicia a definição do comando no programa;
- próximo comando (qualquer) no programa, com o mesmo nível de embutimento;
- comando (qualquer) anterior a este no programa, com o mesmo nível de embutimento;
- indicação do tipo específico de comando. A cada comando é associado um descritor particular, que pode ser qualquer um dos objetos descritos abaixo:

---

[OBJ_READ] - informações sobre o comando read(ln):

- tipo: readln ou read;
- códigos das variáveis lidas;
- apontador para o próximo comando read(ln) no programa;
- apontador para o comando read(ln) anterior a este no programa.

---

---

[OBJ_WRITE] - informações sobre o comando write(ln):

- tipo: write ou writeln;
- expressão(ões) encontrada(s) no write(ln);
- apontador para o próximo comando write(ln) no programa;
- apontador para o comando write(ln) anterior a este no programa.

---

[OBJ_ATRIB] - informações sobre o comando assign:

- código da variável que receberá a atribuição;
- expressão criada para atribuição;
- apontador para o próximo comando assign no programa;
- apontador para o comando assign anterior a este no programa.

---

[OBJ_IF] - informações sobre o comando if:

- expressão da condição;
- lista de comandos da opção then;
- lista de comandos da opção else (se existir);
- apontador para o próximo comando if no programa;
- apontador para o comando if anterior a este no programa.

---

[OBJ_CASE] - informações sobre o comando case:

- expressão seletora;
- próximo comando case no programa;
- comando case anterior a este no programa;
- lista das opções. Para cada opção são armazenadas informações contidas no objeto a seguir:

[OBJ_OP] - informações sobre as opções:

- expressão(ões);
- tipo (normal ou else);
- lista de comandos internos à opção;
- próxima opção;
- opção anterior.

---

[OBJ_FOR] - informações sobre o comando for:

- tipo: crescente ou decescente;
- código da variável de controle;
- expressão indicada como limite inicial para a variável de controle;
- expressão indicada como limite final para a variável de controle;
- lista de comandos internos;
- apontador para o próximo comando for no programa;
- apontador para o comando for anterior a este no programa.

---

[OBJ_WHILE] - informações sobre o comando while:

- expressão da condição;
- lista de comandos internos;
- apontador para o próximo comando while no programa;
- apontador para o comando while anterior a este no programa.

---

[OBJ_REPEAT] - informações sobre o comando repeat:

- expressão da condição;
- lista de comandos internos;
- apontador para o próximo comando repeat no programa;
- apontador para o comando repeat anterior a este no programa.

Apenas um dos objetos, particular ao comando, é armazenado em cada elemento da lista por vez. A lista final apresenta elementos criados na sequência de aparecimento, com indicação do nível de embutimento dos comandos no programa.

# ANEXO D

REPRESENTAÇÃO DAS METAS ESPECÍFICAS DOS PROBLEMAS

### Estrutura de Dados Usada para Armazenamento das Metas Específicas

Para realizar a verificação de inconsistências lógicas, oriundas do mau entendimento das metas específicas ao problema, fez-se necessária a estruturação do programa em níveis primários de informação, descritos nos objetos definidos a seguir. Eles possuem uma estrutura semelhante ao formato VIP para solução do estudante com algumas inserções e simplificações.

1. Identificação Geral do Programa - contém informações gerais sobre as metas armazenadas pelo instrutor para um programa em particular. Difere do formato da Solução VIP pela multiplicidade de soluções corretas armazenadas para efeitos de comparação.

---

- identificação do código do problema, definido conforme estruturação do espaço de resolução de problemas (classe, tipo, nível) mais o número individual;

- indicação da(s) solução(ões) armazenada(s) para o programa, isto é, a solução considerada ótima e, opcionalmente, as soluções alternativas.

---

2. Declarações - opcional para alguns casos de programas de TIPO I. Associam informações sobre as definições de objetos de dados constante e/ou variável. Um pouco mais simplificado que o formato da Solução VIP.

---

- indicação de tabela com as constantes definidas:

  [OBJETO_CONST] - semelhante ao formato da Solução VIP:
  - tipo;
  - valor;

---

- indicação de tabela com as variáveis definidas:

  ---

  [OBJETO_VAR] - semelhante do formato da Solução VIP:
  - tipo;
  - informações adicionais às variáveis estruturadas:

    ---

    [CADEIA] - informações específicas à cadeia:
    - identificação do tamanho máximo permitido.

    ---

    ---

    [ARRAY_UNI] - informações específicas ao vetor:
    - número de elementos;
    - tipo dos elementos;
    - informações adicionais à elementos estruturados.

    ---

    ---

    [ARRAY_BI] - informações específicas à matriz:
    - número de linhas;
    - número de colunas;
    - tipo dos elementos;
    - informações adicionais à elementos estruturados.

    ---

  ---

---

3. <u>Bloco de Comandos</u> - à semelhança do formato da Solução VIP, associa um conjunto de informações sobre cada estrutura de comando (meta) definida no programa. No entanto, a representação das metas difere a partir da associação

do conceito de sinônimos naturais (citado anteriormente) e pelos requerimentos de informação necessários ao processo de análise.

---

[OBJETO_BLOCO] - é construído para todos os TIPOS de programas armazenados pelo instrutor. O BLOCO é gerado levando em consideração a ordem de aparecimento dos comandos no programa.

- código individual;
- tipo do comando;
- nível de embutimento;
- próximo comando (qualquer) no programa, no mesmo nível de embutimento;
- comando (qualquer) anterior a este no programa, no mesmo nível de embutimento;
- descritor do comando específico:

  [OBJ_READ] - informações específicas ao read(ln):
  - tipo;
  - códigos das variáveis lidas;
  - código de erro/advertência contextual apontado pelo instrutor. Por exemplo, "ATENÇÃO: Foram lidas variáveis em excesso".

  [OBJ_WRITE] - informações específicas ao write(ln):
  - tipo;
  - expressão(ões) originais (e/ou sinônimas) indicadas para o write(ln);
  - código de erro/advertência contextual apontado pelo instrutor. Por exemplo, "ERRO: O resultado da impressão não atende ao solicitado no problema".

[OBJ_ATRIB] - informações específicas ao assign:

- variável a ser alterada pela atribuição;
- expressão original (e/ou as sinônimas) indicada para a atribuição;
- código de erro/advertência contextual apontado pelo instrutor. Por exemplo, "ERRO: A expressão atribuída está incorreta".

---

---

[OBJ_IF] - informações específicas ao if:

- expressões válidas para a condição do if. O instrutor apresenta formatos alternativos para construção da expressão e as informações correspondentes a cada contexto de suas utilizações:
    - expressão para a condição;
    - indicação de como os blocos (then e else) devem ser combinados (se 0, bloco1 → then e bloco2 → else; se 1, bloco2 → then e bloco1 → else);
    - próxima expressão alternativa para a condição.

  Esta forma de representação permite que o instrutor indique um conjunto qualquer de expressões para a condição, sem se preocupar com o armazenamento dos blocos.
- comandos internos ao bloco1;
- comandos internos ao bloco2;
- código de erro / advertência contextual apontado pelo instrutor. Por exemplo, "ERRO: A expressão da condição não está adequada ao que foi solicitado".
- informações necessárias à substituição deste if por um case:
    - expressão seletora principal (e/ou sinônimas);
    - expressões principais (e/ou sinônimas) apontadas para as opções;
    - código de erro/advertência contextual apontado pelo instrutor. Por exemplo, "ERRO: Faltam opções na estrutura do case."

A ordem de aparecimento das opções do case deve ser compatível com a estrutura do if, pois os blocos de comandos são considerados na mesma ordem.

---

UFPb / BIBLIOTECA / PRAI

---

[OBJ_CASE] - informações específicas ao case:

- expressão seletora principal (e/ou as sinônimas);
- opções apontadas para o case:

  [OPÇÃO] - informações relacionadas à cada opção:

  - expressão(ões) originais (e/ou as sinônimas) para a opção;
  - tipo da opção (normal ou else);
  - comandos internos à opção;
  - próxima opção;
- código de erro/advertência contextual apontado pelo instrutor. Por exemplo, "ERRO: A expressão seletora não está adequada ao que foi solicitado".
- informações necessárias à substituição do case por um aninhamento de if´s:
  - expressão principal (e/ou sinônimas) para a condição;
  - próxima condição da estrutura de aninhamentos de if´s;
  - código de erro/advertência contextual apontado pelo instrutor. Por exemplo, "ADVERTÊNCIA: A utilização de If nesta meta dificulta desnecessariamente o programa".

A ordem de aparecimento dos if's deve ser compatível com a estrutura do case, pois os blocos são considerados na mesma ordem.

---

---

[OBJ_FOR] - informações específicas ao for:

- código da variável de controle ;
- expressões alternativas para um for crescente e os respectivos contextos de utilização:
  - expressões para os limites inicial e final;
  - próximo conjunto de limites.
- expressões alternativas para um for decrescente e os respectivos contextos de utilização:
  - expressões para os limites inicial e final;
  - próximo conjunto de limites.
- comandos internos ao for;

- informações necessárias à substituição por um while:
  - expressão principal (e/ou sinônimas) da condição;
  - código de erro/advertência contextual apontado pelo instrutor. Por exemplo: "ERRO: Expressão da condição do while não se adequa ao solicitado".
- informações necessárias à substituição por um repeat:
  - expressão principal (e/ou sinônimas) da condição;
  - código de erro/advertência contextual apontado pelo instrutor. Por exemplo, "ADVERTÊNCIA: O uso de um for é mais indicado para repetição automática";
- informações relativas aos comandos de atribuição requeridos pelas estruturas sinônimas (while ou repeat) utilizadas:
  - código da variável da esquerda;
  - expressão principal (e/ou sinônimas) para a atribuição inicial da variável ;
  - expressão principal (e/ou sinônimas) para a atribuição interna (alteração da condição de controle);
  - código de erro/advertência contextual apontado pelo instrutor. Por exemplo, "ERRO: Expressão da atribuição está incorreta";
- código de erro/advertência contextual apontado pelo instrutor. Por exemplo: "ERRO: Expressão do limite é inadequada ao solicitado".

---

---

[OBJ_WHILE] - informações específicas ao while:

- expressão principal (e/ou sinônimas) para a condição do while;
- comandos internos ao while;
- informações necessárias à substituição por um repeat (opcional):
  - expressão principal (e/ou sinônimas) para a condição do repeat;
  - código de erro/advertência contextual apontado pelo instrutor, quando um repeat é usado. Por exemplo, "ADVERTÊNCIA: O uso de while é mais indicado para esta meta";
- código de erro/advertência contextual apontado pelo instrutor Por exemplo, "ERRO: A condição do while está incorreta".

---

[OBJ_REPEAT] - informações específicas ao repeat:

- expressão principal (e/ou sinônimas) para a condição do repeat;
- comandos internos ao repeat;
- informações necessárias à substituição por um while (opcional):
    - expressão principal (e/ou sinônimas) para a condição do while;
    - código de erro/advertência contextual apontado pelo instrutor, quando um while é usado. Por exemplo, "ADVERTÊNCIA: A condição do while está incorreta";
- código de erro/advertência contextual apontado pelo instrutor Por exemplo, "ERRO: Os comandos internos ao repeat são insuficientes".

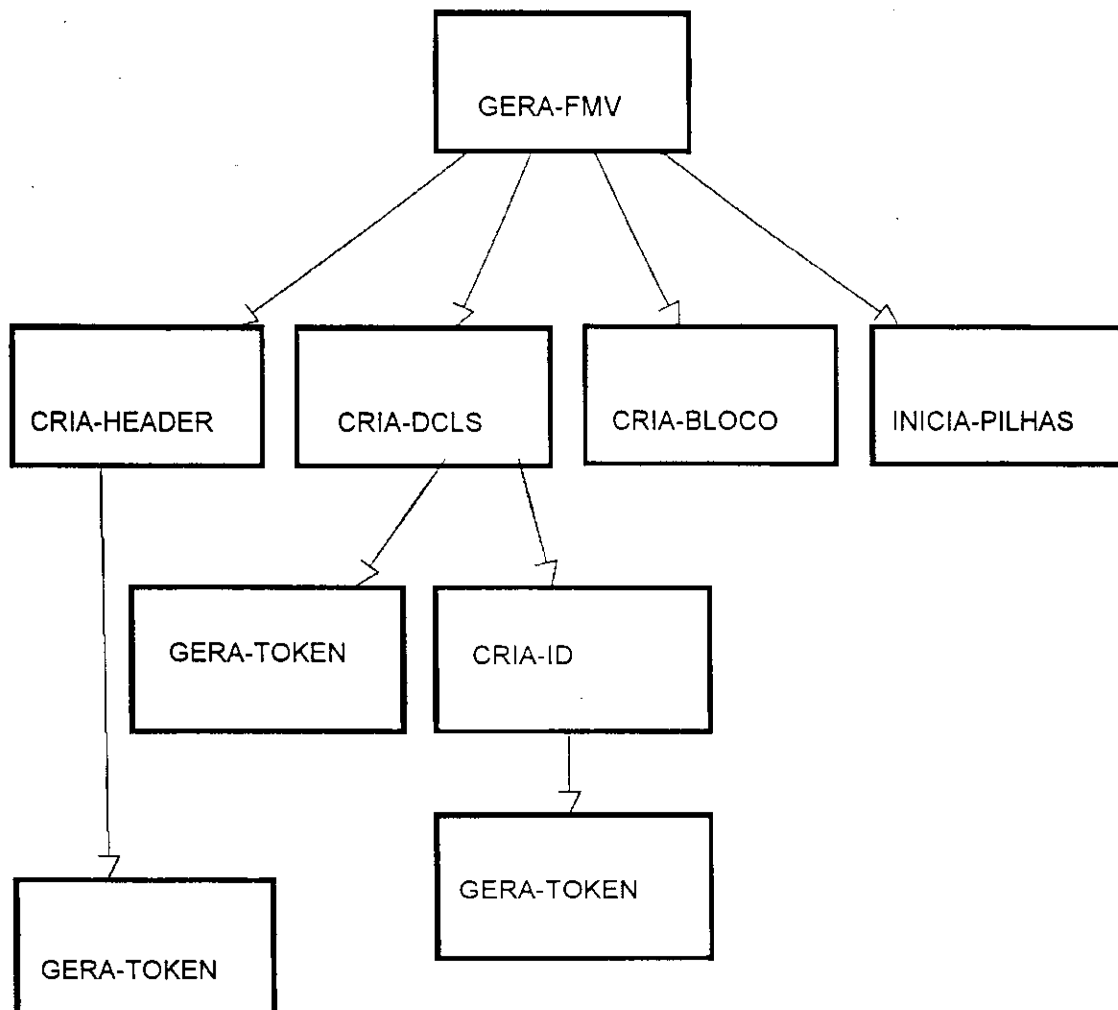
---

É importante ressaltar que os erros apontados contextualmente pelo instrutor em cada programa armazenado, fornece ao estudante uma orientação mais precisa sobre o erro/advertência. O instrutor deve procurar cadastrar mensagens de erros significativas e orientadas ao contexto.

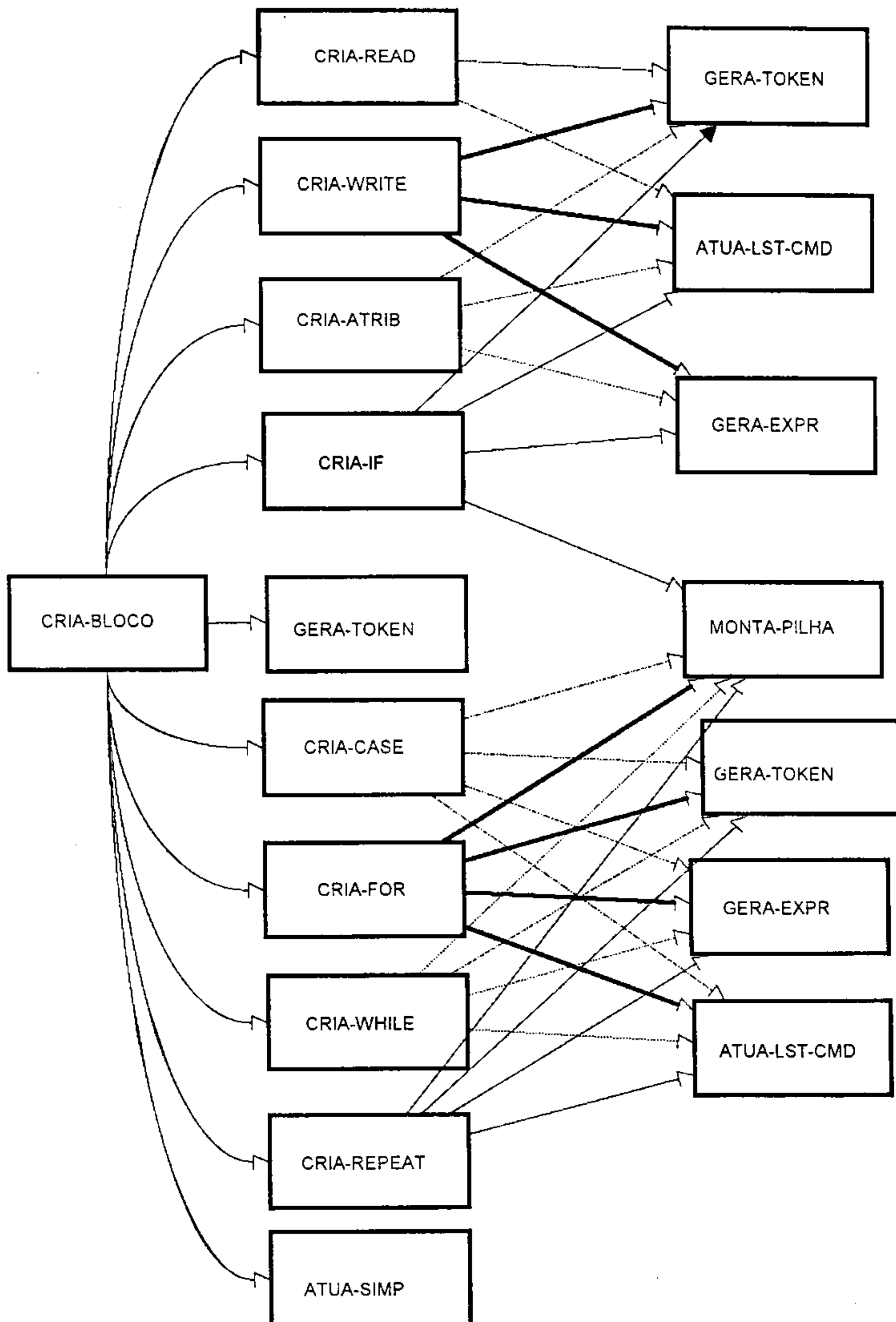
# ANEXO E

ARQUITETURA INTERNA DOS MÓDULOS DO VIP

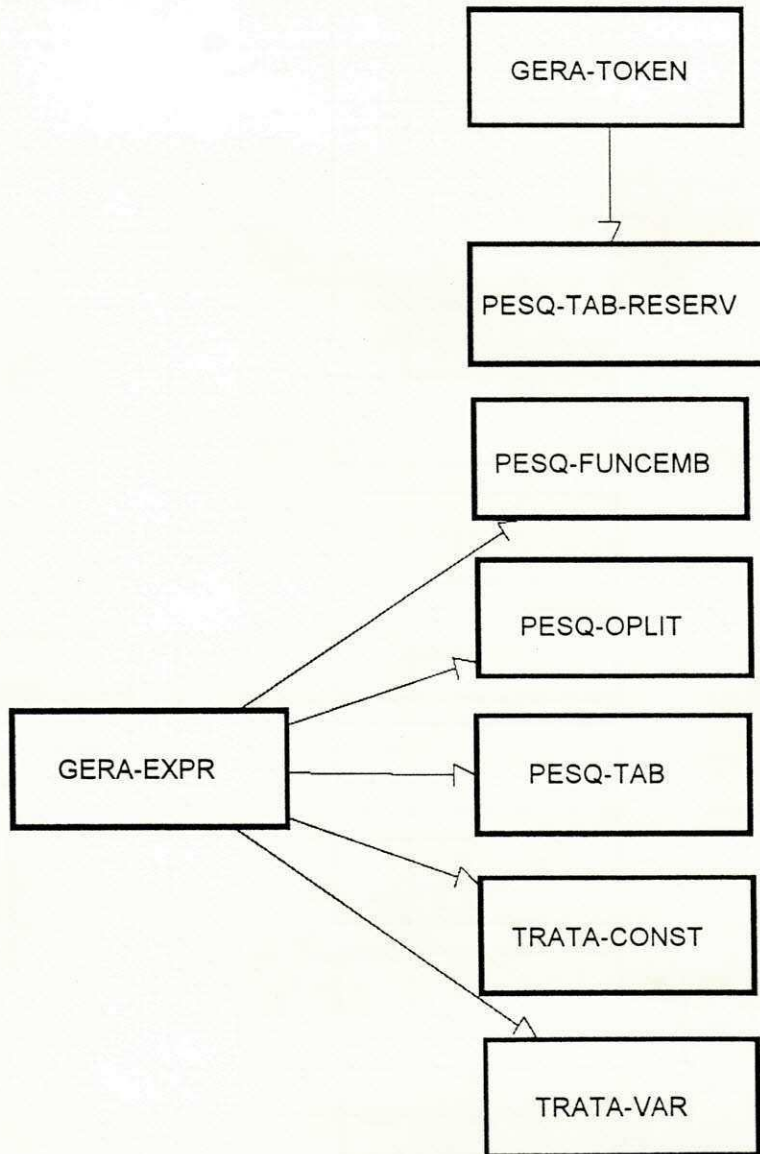
## MÓDULO GERA_FMV



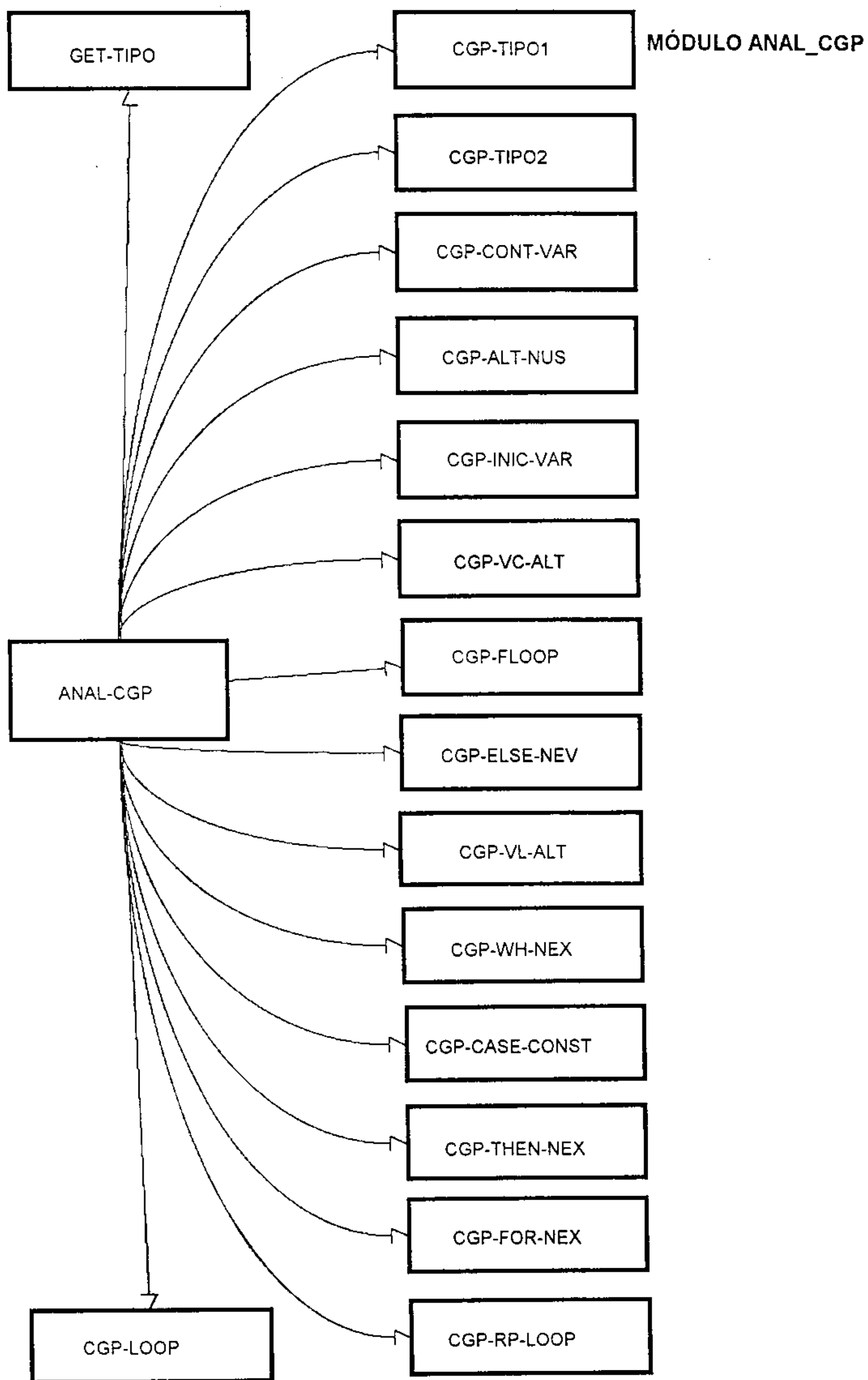
**MÓDULO INTERNO AO GERA_FMV**

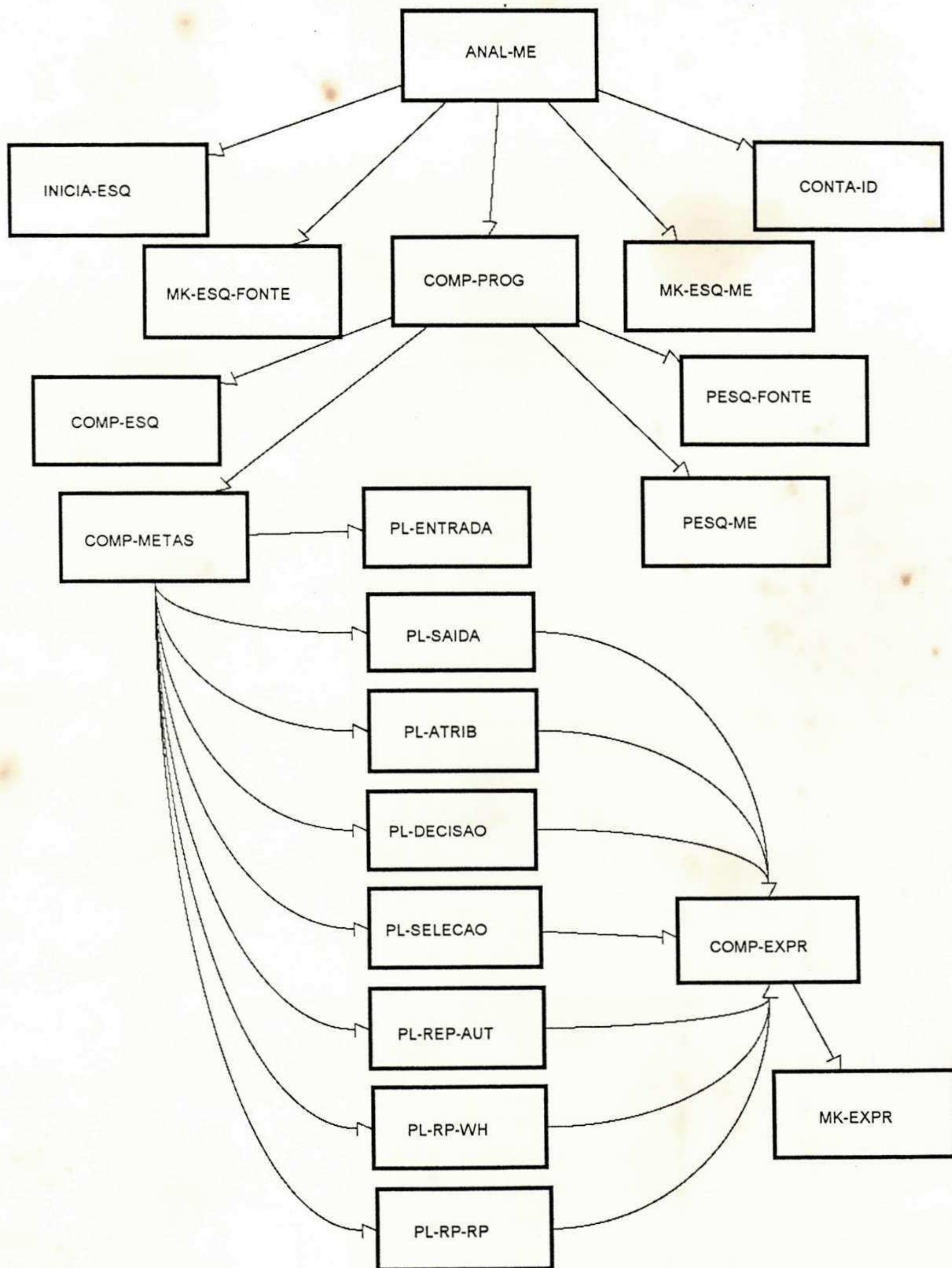


## MÓDULOS INTERNOS AO GERA_FMV



**MÓDULO ANAL_CGP**




MÓDULO ANAL_ME
ANAL-ME
INICIA-ESQ
CONTA-ID
MK-ESQ-FONTE
COMP-PROG
MK-ESQ-ME
COMP-ESQ
PESQ-FONTE
COMP-METAS
PL-ENTRADA
PESQ-ME
PL-SAIDA
PL-ATRIB
PL-DECISAO
PL-SELECAO
COMP-EXPR
PL-REP-AUT
PL-RP-WH
MK-EXPR
PL-RP-RP