UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA

Este exemplar corresponde à redação final da tese defendida por Maria Cecília Calani Baranauskas e aprovada pela Comissão Julgadora em 04 / 02 / 1993.

Orientador

CRIAÇÃO DE FERRAMENTAS PARA O AMBIENTE PROLOG E O ACESSO DE NOVATOS AO PARADIGMA DA PROGRAMAÇÃO EM LÓGICA

autor: Maria Cecília Calani Baranauskas
orientador: Prof. Dr. José Armando Valente

Tese apresentada à Faculdade de Engenharia Elétrica, da Universidade Estadual de Campinas, como parte dos requisitos exigidos para obtenção do título de Doutor em Engenharia Elétrica

Fevereiro de 1993

Para

Cibele, Vitukas e Vitor

o meu referencial de vida

Agradecimentos

Com muito carinho, aos alunos que participaram de meu estudo experimental e se deixaram observar em seus processos de "pensar". Sua participação acrescentou realidade às minhas discussões.

A meus ex-orientados de Iniciação Científica, que contribuiram diretamente na fase inicial de implementação e testes das ferramentas; em especial a Urubatan Rocha Pacheco, Fabricio Torres de Souza, Maurício Mendes, Nascif Abrão Abousalah Neto e Luciana Izumi Kido.

Aos pesquisadores e pessoal de apoio do NIED (Núcleo de Informática Aplicada à Educação) da UNICAMP. A possibilidade de interagir num ambiente intelectualmente estimulante, com pesquisadores de diferentes expertises e estilos foi fundamental para o amadurecimento de idéias presentes nesta tese.

Ao DCA (Departamento de Computação e Automação) da FEE UNICAMP, que me possibilitou um estágio de pesquisa no Weizmann Institute of Science de Israel. Essa viagem abriu novos horizontes para intercâmbio e continuidade da pesquisa.

Ao DCC (Departamento de Ciência da Computação) do IMECC UNICAMP, pelo apoio representado na dispensa parcial de carga didática.

Ao Prof. Dr. José Armando Valente, por acreditar no meu trabalho, pelo respeito à liberdade de criação e pela orientação firme.

Aos amigos que, espalhados pelo mundo e pelo tempo, ficam aqui anônimos. Sua influência despercebida em diferentes contextos e momentos da minha vida, foram o *leitmotiv* para este trabalho.

Sumário

Resumo ix

Abstract x

Résumé xi

1. Introdução 1

2. O Cenário, O Estado da Arte e a Problemática 21

2.1 Depuração como Atividade e como Processo 23

2.2 Ferramentas em Ambientes Prolog 27

2.2.1 Depuração Automática 28

2.2.2 Depuração Guiada 29

2.2.3 Depuração por Monitoramento 32

2.2.4 Ferramentas Alternativas 38

2.3 Discussão e Proposta 39

3. Quadro Teórico 43

3.1 Os Paradigmas das Linguagens de Programação como "Meios" 44

3.1.1 As Origens dos Paradigmas de Programação 44

3.1.2 Os "Meios" Gerados pelos Paradigmas de Programação 48

3.1.3 Discussão 55

3.2 O Ambiente de Aprendizado 59

3.2.1 O Papel das Ferramentas nesse Ambiente 63

3.3 As Opções de Design para o Ambiente Proposto 64

3.4 Fundamentos para as Representações Gráficas Propostas nas Ferramentas 67

3.4.1 O Modelo de Computação de Programação em Lógica 68

3.4.2 O Modelo de Execução de Prolog 70

3.4.3 Os Diagramas de Rêdes Semânticas 73

4. Objetivos e Metodologia **77**

5. O Ambiente Proposto **82**

5.1 O Módulo Declarativo 84

5.2 O Módulo Operacional 88

5.3 Exemplos de Conceitos Abordados pelas Ferramentas 92

5.3.1 Análise Sistemática de Fluxo e Meta-Análise 92

5.3.2 A Explicitação do Processo de *Backtracking* 98

5.3.3 A Explicitação do Mecanismo Recursivo - a "Volta" da Recursão 102

5.3.4 A Explicitação de Aspectos de Eficiência 106

5.3.5 A Explicitação da Estrutura da Solução 108

5.4 Discussão 111

6. O Módulo Operacional **114**

6.1 A Representação Proposta para o Modelo de Execução de Prolog 115

6.2 Descrição Funcional das Ferramentas que Compõem o Mód. Operacional 117

6.3 O Ambiente Gerado pelo Módulo Operacional 129

6.3.1 Ambiente Inicial e Facilidades Básicas 129

6.3.2 O Ambiente da Árvore de Busca 133

6.3.3 O Ambiente do Trace Estendido 138

6.4 Aspectos de Implementação do Módulo Operacional 140

6.4.1 Descrição do Sistema que Implementa o Módulo Operacional 141

6.4.1.1 O Módulo de Interface com o Usuário (MI) 143

6.4.1.2 O Módulo de Controle de Variáveis (MCV) 145

6.4.1.3 O Módulo de Controle de Janelas (MCJ) 146

6.4.1.4 O Módulo Principal 149
6.4.1.5 O Módulo de Meta-Interpretação 151
6.4.1.6 O Módulo Utilitário 153
6.4.2 Estrutura de Dados Utilizada 153
6.5 Síntese 157

7. O Módulo Declarativo 159
7.1 Modelo Proposto para Representação de Aspectos do Significado Declarativo de um Programa 160
7.1.1 Representação de Conhecimento Factual 161
7.1.2 Representação de Conhecimento Implicativo 163
7.1.3 Representação da Base de Dados 166
7.2 Descrição das Ferramentas que Compõem o Módulo Declarativo 167
7.2.1 O Editor Semântico (EDS) 170
7.2.1.1 Descrição Funcional do EDS 171
7.2.1.2 O Ambiente Gerado pelo EDS 172
7.2.1.3 Aspectos de Implementação do EDS 180
7.2.1.3.1 Estrutura de Dados Utilizada 183
7.2.2 O Modo de Representação em Diagramas Semânticos (MDS) 189
7.2.2.1 Descrição Funcional do MDS 190
7.2.2.2 O Ambiente Inicial e Facilidades Básicas 191
7.2.2.3 O Ambiente do Diagrama Semântico 192
7.2.2.4 A Opção Inferência 201
7.2.2.5 Aspectos de Implementação do MDS 203
7.2.2.5.1 Descrição do Sistema que Implementa o MDS 203
7.2.2.5.2 Estrutura de Dados Utilizada 206
7.2.2.5.3 Estratégias Utilizadas no Desenho do DS 209

7.3 Síntese 214

8. Estudo Experimental da Interação do Novato no Ambiente Proposto 216

8.1 Metodologia do Estudo 217

8.2 Resultados da Análise do Grupo I (novato-novato) 221

8.2.1 O Modelo Conceitual Inicial da Máquina Virtual Prolog 222

8.2.2 Os Erros Conceituais dos Sujeitos e o *Feedback* das Ferramentas 224

8.2.3 As Estratégias de Depuração na Ausência de *Feedback* 239

8.2.4 Obtendo Feedback a partir do Uso das Ferramentas 242

8.2.5 A Atividade de Criação de Bases de Dados 245

8.2.6 A Evolução do Modelo Conceitual da Máquina Virtual 253

8.2.7 Aspectos Operacionais x Aspectos Declarativos 256

8.2.8 Síntese da Análise do Grupo I 259

8.3 Estudo de Caso de um Sujeito do Grupo Novato-Novato 260

8.3.1 Síntese do Estudo de Caso 285

8.4 Resultados da Análise do Grupo II (novato-Prolog) 287

8.4.1 Modelos Conceituais Iniciais da Máquina de Inferência 287

8.4.2 O Conceito de Recursão visto em outro Meio e o *Feedback* das Ferramentas 291

8.4.3 A Construção do "Meio" Prolog 303

8.4.4 Síntese da Análise do Grupo II 317

8.5 Síntese de Resultados do Estudo 317

9. Discussão, Conclusões e Perspectivas 320

Referências 347

Apêndice 1 **357**

Apêndice 2 **362**

Resumo

Esta tese tem como tema principal o *design* de ferramentas computacionais que constituem um ambiente de programação Prolog e o estudo da interação do novato nesse ambiente.

A proposta do conjunto de ferramentas é explicitar o paradigma de programação subjacente e enriquecer o *feedback* gerado pelo ambiente Prolog, possibilitando ao novato acesso ao programa não apenas em seu significado operacional, mas também em seu significado lógico. As ferramentas, inseridas no ambiente de programação Prolog, constituem um Módulo Operacional e um Módulo Declarativo. No Módulo Operacional, a máquina virtual da linguagem é explicitada através de uma representação gráfica da árvore de busca de determinada meta em uma base de dados, que denominamos Árvore de Espaços de Busca. No Módulo Declarativo o formalismo clausal do programa é expresso de forma pictórica através de um modelo baseado em diagramas de rêdes semânticas, que denominamos Diagramas Semânticos.

Foi feito um estudo experimental para análise da interação do novato no ambiente proposto, que envolveu dois tipos de novatos: (A) novatos em sua primeira experiência com linguagem de programação (e mesmo com computadores) e (B) novatos em Prolog, mas com bom conhecimento de linguagens procedurais. Nosso objetivo foi investigar os estágios iniciais do processo de aquisição da linguagem Prolog e os efeitos das ferramentas propostas, nesse processo.

Os estudos realizados mostraram que ambos os tipos de novato têm modelos conceituais iniciais da máquina virtual, responsáveis pela sua interpretação a respeito do programa Prolog e do comportamento da máquina virtual da linguagem. As situações de erro podem ser interpretadas a partir desses modelos conceituais correntes, que evoluem ao longo do processo de interação com o ambiente. A atividade de programar mostrou-se como um processo incremental que envolve um ciclo realimentado por respostas fornecidas pelo ambiente de programação. Dessa maneira, depuração é parte do processo de aquisição da linguagem e não uma fase a ser tratada isoladamente. O *feedback* gerado pelas ferramentas provocou uma mudança de perspectiva dos estudantes em relação ao conhecimento sendo representado (programa) e em relação ao interlocutor no processo de programação (máquina virtual), o que possibilitou a modificação nos seus modelos conceituais da linguagem.

Abstract

This thesis concerns the design of computational tools which constitute a Prolog programming environment and the study of novices' interaction in this environment.

The purpose of the tool package is to make explicit the underlying programming paradigm and to reinforce the feedback provided by the Prolog environment, opening to the novice access to the program, not only in its operational meaning but also in its logical meaning. The tools are integrated in the Prolog environment through an Operational Module and a Declarative Module. In the Operational Module, the language virtual machine is shown through a graphical representation of the search tree of a goal within a database, which I named Search-Spaces Tree. In the Declarative Module, the clausal form of the program is represented in a pictorial way, by means of a model based on network diagrams, which I named Semantic Diagrams.

The study of the novice's interaction within the proposed environment was conducted experimentally with two types of novices: (A) novices in their first experience with a programming language (and even with computers) and (B) novices with good practice of procedural programming. Our aim was to investigate the early stages of Prolog programming and the effects of the proposed tools in this process.

The observational studies undertaken showed that both types of novices had an initial conceptual model of the virtual machine. Such models guided their interpretation of the Prolog program and of the behavior of the machine. Their misunderstandings may be understood as resulting from their current models, which evolve as long as they interact in the environment. The programming activity can be interpreted, from the results, as an incremental process which envolves a cyclic process fed by responses of the programming environment. By this way, debugging is part of the language learning process and not an activity to be treated in isolation. The feedback created by the tools provided changes in the students' perspective related to the knowledge being represented (program) and the interlocutor in the process (virtual machine), which lead them to the understanding of the conceptual model of the language.

Résumé

Cette thèse a comme thème principal le *design* des outils computationales constituent un ambiant de programmation Prolog et l'étude de l'interaction du novice en cet ambiant.

La proposition de l'ensemble des outils est expliciter le paradigme sous-jacent et enrichir le *feedback* formé par l'ambiant Prolog facilitant au novice l'accès au programme pas seulement dans la signification operationale, mais aussi dans la signification logique. Les outils insérés dans l'ambiant de programmation Prolog constituent um Module Operationale et un Module Declaratif. Dans le Module Operationale, la machine virtuelle du langue est executé à travers d'une base des donnés que nous appelons Arbre du Espace de Cherche. Dans le Module Declaratif le formalisme clausal du programme est exprimé d'une manière picturale à travres d'un modèle basé en diagrammes des reseaux sémantiques, que nous appelons Diagrammes Sémantiques.

Il a été fait une étude experimentale pour l'analyse de l'interaction du novice dans l'ambiant proposé, avec deux types de novices: (A) novices dans leur première experience avec une langue de programmation (et aussi avec les ordinateurs) et (B) novices en Prolog, mais avec un bon connaissance des langues de procedures. Notre objectif a été étuder les états initials du procés de acquisition de la langue Prolog et les effets des outils dans ce procés.

Les études des observations faites on montré que les deux types de novice ont modéles conceptuelles initiales de la machine virtuelle, qui sont les responsables de leur interpretation du programme Prolog e de la conduite de la machine virtuelle de la langue. Les situations d'erreur peuvent être interprétés a partir de ces modèles conceptuelles courant, qui evoluent dans le procés d'interaction avec l'ambiant. L'activité de programmer s'est montré comme un procès d'incrémentation que comprend une cycle realimenté par reponds donés par l'ambiant de programation. Ainsi, la dépuration est une partie du procés de acquisition de la langue et non une phase que doit être traité isolément. Le *feedback* formé par les outils a été l'origin d'une transformation de la perspective des élèves en relation a la conaissance etant représenté (programme) et en relation au interlocuteur dans le procés de programmation (machine virtuelle), lequel a rend possible la modification de leurs modéles conceptuelles de la langue.

Capítulo 1

Introdução

Capítulo 1

Introdução

"Existe a crença de que só se pode programar o que se compreende perfeitamente. Essa crença ignora a evidência de que a programação, como qualquer outra forma de escrita, é um processo experimental. Programamos como redigimos, não porque compreendemos, mas para chegar a compreender."

(Weizenbaum, 1981)

Este trabalho tem como tema principal o *design*[*1] de ferramentas computacionais que constituem um ambiente de programação Prolog e o estudo da interação do novato nesse ambiente.

Para tornar explícitas as estratégias que nortearam o desenvolvimento desta pesquisa, devemos situá-la no contexto do uso da tecnologia (de computadores), que por sua vez, como toda nova tecnologia, desenvolve-se dentro de uma visão de mundo e de como se entende a natureza humana. Estamos usando o termo *design* (de ferramentas) significando *"interação entre entendimento e criação"* (Winograd, 1988, p. 4), para refletir nossa preocupação com a questão mais ampla de uma sociedade que cria uma tecnologia cuja existência, por sua vez, leva a mudanças fundamentais no que fazemos e em como entendemos a natureza do trabalho humano, alterando aquela sociedade.

---

*1 optamos pela não tradução da palavra *design*. Ela não é usada aqui no sentido de uma metodologia específica para para a criação de artefatos, mas num contexto mais amplo de teoria de *design* (Schon, 1990; Winograd, 1988), onde traduções como "desenho", "projeto" ou "concepção" não captariam o seu significado.

Papert, entre outros cientistas contemporâneos, foi um dos pioneiros a considerar "estudos em tecnologia" como "conhecimento básico" para as novas sociedades, de forma análoga à enorme cultura que surgiu ao redor da linguagem escrita (Papert, 1991). Enquanto que muitos consideram "fluência tecnológica" apenas ao nível de uso de tecnologia, Papert vai além fazendo uma analogia com fluência em leitura, que, além de ser considerada uma habilidade indispensável para a completa participação na sociedade contemporânea, potencialmente enriquece o aprendizado de tudo o mais.

Nossa opção pelo novato como público alvo do design de ferramentas computacionais vem de encontro à preocupação de facilitar o acesso de "pessoas comuns" a uma tecnologia que não ha muito tempo atrás era domínio de uma comunidade de "praticantes de uma ciência". Esse acesso à tecnologia incorpora a noção de Papert de níveis de fluência tecnológica, propondo um acesso mais profundo ao conhecimento, não apenas pela utilização de tecnologia, mas em direção ao aprender pelo "fazer" tecnologia, conforme discutiremos a seguir.

O cenário para nossa pesquisa envolve o novato interagindo com o computador com o objetivo de "resolver problemas". O interlocutor nessa interação é a linguagem de programação Prolog que será usada para a "representação da solução do problema". Nossa intenção é instrumentalizar o novato no cenário especificado, de modo que ele próprio seja o condutor de seu processo de apropriação da linguagem de programação.

O conhecimento de uma linguagem de programação tem sido confundido com a soma dos conhecimentos da sintaxe e da semântica da linguagem. Conhecimento de como "compor" as primitivas da linguagem e como "decompor" o problema segundo essas primitivas. Dessa maneira, "saber programar" é confundido com a soma dessas partes.

Essa fragmentação do conhecimento de programar, ao nosso ver, não traduz a realidade de nosso cenário, onde o conhecimento da linguagem em si, em seu aspecto sintático e semântico não existe isoladamente ou a priori do conhecimento de como "expressar" um determinado problema naquela linguagem. Isso seria equivalente a dizer que basta ao novato conhecer o "manual de referência" da linguagem para ser um "programador" ou que esse conhecimento seria um "pré-requisito" para o novato iniciar a atividade de programação. Em nossa ótica, conhecimento da linguagem em seus aspectos sintáticos e semânticos e conhecimento de como expressar/representar um problema nessa linguagem (através dessa linguagem) são construídos juntos, ao longo de um processo. Estamos usando a expressão "apropriação da linguagem de programação" para representação da solução de problemas para sugerir essa indissociabilidade.

Concordamos com Papert quando ele diz que nossa concepção sobre o que é "programação" mudou tanto com base no desenvolvimento de novos paradigmas, que já não está claro que devamos usar a mesma palavra (Papert, 1991).

No contexto do cenário definido, os termos "problema", "resolução", "solução" existem em dois níveis. Num primeiro nível, existe um "problema-alvo", dentro de determinado domínio do conhecimento, como por exemplo "o cálculo do fatorial de um número inteiro positivo", que será o objeto de representação na linguagem de programação. Nesse nível, "solução" é a resposta para o problema; por exemplo, 120 é o fatorial de 5, e "resolução" é o processo que leva à obtenção da resposta. A atividade de programar pode ser entendida como o processo de criação de representações simbólicas para a solução do problema alvo, que são "interpretadas" em algum nível dentro da hierarquia de graus variados de abstração da máquina. O nível de "resolução de problemas" que estamos focalizando neste trabalho refere-se, portanto, ao processo de resolução de um problema alvo intermediado por uma linguagem de programação.

A Inteligência Artificial (IA) desenvolveu tentativas de formalizar o comportamento de resolução de problemas, sob a influência dos trabalhos de Newell e Simon (1972) em tomada de decisão racional. Segundo a teoria de Newell e Simon, um problema ou tarefa é analisado em termos de um "espaço de problema" gerado por um conjunto finito de propriedades e operações. Uma "solução" é um ponto particular nesse espaço que tem as propriedades desejadas. Resolução de problemas é um processo de busca por uma sequência de operações que conduzirão ao ponto de solução. Esse modelo foi aplicado diretamente em programas como o GPS (the General Problem Solver) e tem influenciado, em um sentido mais geral, a maioria dos trabalhos em IA. Apesar de ter sido responsável por grandes avanços em IA nas últimas décadas, Winograd e Flores apontam as limitações dessa abordagem e a situam historicamente dentro da "tradição racionalista" responsável pelas dificuldades dentro da área em abordar questões como por exemplo o "bom senso" (Winograd, 1988). Num contexto mais amplo, a tradição racionalista em IA aparece como um reflexo do paradigma mecanicista que manteve sua influência sobre o pensamento científico ocidental no século XX e influenciou não só nossa maneira de ver o mundo, mas a nossa própria concepção de "inteligência".

No contexto de ensino/aprendizagem, em particular, desenvolveram-se os sistemas chamados Tutores Inteligentes (Intelligent Tutoring Systems - ITS), programas que utilizam técnicas de IA para representar o conhecimento do domínio do problema e para conduzir a interação com o estudante. Tais sistemas têm sido bem sucedidos em domínios restritos, como por exemplo, para treinamento de habilidades específicas, onde tanto os "problemas" a serem resolvidos quanto os "processos de resolução" são bem definidos.

As limitações dessa abordagem podem ser apontadas no contexto de nosso cenário: frente à tarefa de programar, o novato não tem uma pré-definição simples do problema (o conhecimento necessário para a representação da solução

do problema alvo na linguagem em questão ainda está sendo construído), ou do "espaço de estados" nos quais buscar uma solução (programa). Como muito bem colocam Winograd e Flores, "*a essência da inteligência é agir apropriadamente quando não há uma pré-definição simples do problema, ou não há o espaço de estados nos quais buscar uma solução*" (Winograd, 1988, p. 98). A grande limitação dos ITS, considerando nosso cenário, é que eles só seriam úteis na medida em que sua estrutura formal correspondesse efetivamente à variedade de situações em que o novato pudesse se encontrar durante o processo. Somente de posse dessas situações é que seria possível intervir efetivamente na aprendizagem.

A alternativa que colocamos é o *design* de ferramentas que possam ser integradas à linguagem formando um ambiente de programação com o qual o novato interaja e possa construir o conhecimento não apenas da linguagem em seus aspectos sintático e semântico, mas principalmente da linguagem como "meio" para representação/expressão de soluções para problemas. O estudo da interação do novato nesse ambiente vêm de encontro ao redirecionamento filosófico proposto por Winograd, onde a alternativa não é a de se criar ferramentas que são "inteligentes", para uso de não expertos; mas, uma tentativa de criar um entendimento de como projetar ferramentas computacionais adequadas ao uso e propósito humanos.

O computador como uma máquina para criação, manipulação e transmissão simbólica de objetos, leva-nos a analisá-lo em nosso cenário, sob o aspecto de linguagem. Considerando as linguagens de programação sob seu aspecto de evolução histórica, elas representam, em graus variados, abstrações da arquitetura subjacente, chamada von Neumann. A "pré-história" das linguagens de programação remete-nos ao final da década de 1940 e início da década de 1950, quando os computadores , além dos problemas decorrentes da tecnologia da época, eram difíceis de serem programados. Na ausência de um software intermediário, a programação era feita em código de máquina. Essa maneira de programar tornava os pro-

gramas ilegíveis, além de ser bastante complicado o seu processo de depuração, na ocorrência de erros. Do ponto de vista do programador essa parece ter sido uma motivação importante para a criação das "linguagens de montagem" (assembly) e de seus montadores (assemblers). Por outro lado, as aplicações numéricas da época requeriam o uso de certas facilidades que não estavam incluídas no hardware das máquinas de então (números reais, acesso a elementos de um conjunto por seu índice, por exemplo), surgindo daí a criação de linguagens de mais alto nível, que incluissem tais recursos. As linguagens de programação que evoluiram nessa direção, como por exemplo Pascal, C, Modula-2 são chamadas linguagens "procedurais"[*2] e exemplificam o paradigma procedural de programação, onde a metáfora para programar é "dar ordens" à máquina que as executará sequencialmente.

Apesar do estilo imperativo (procedural) de programar ser bem aceito entre os programadores até os dias de hoje, o vínculo da linguagem com a arquitetura von Neumann, excetuando-se razões técnicas de eficiência, apresentou-se como uma restrição desnecessária: surgiram outras bases para o projeto de linguagens que acrescentaram uma camada de abstração sobre a arquitetura von Neumann e introduziram novos paradigmas de programação. A atividade de programar passa a ser vista, então, dentro de um novo nível de abstração e consequentemente, mudam em função do paradigma, os "significados" para o "programa" e para a "máquina" que o executa.

Papert, apesar de referir-se a linguagens suportadas por novas arquiteturas de computador (arquiteturas paralelas), define paradigma de programação como um modelo básico estruturador [*3] que é subjacente à atividade de programar

*2 Não existe na lingua portuquesa tradução para a palavra "procedural". Estamos usando uma das formas encontradas na literatura de Ciência da Computação para tal tradução.

*3 "framework" no original.

e que muda nossa própria concepção do que é "programação" (Papert, 1991). Segundo nossa ótica, esse "modelo estruturador" subjacente à atividade de programar já existe a partir do nível de abstração que linguagens como Lisp e Prolog, por exemplo, acrescentaram sobre a arquitetura von Neumann, através da representação de funções e declarações, respectivamente.

Programar nos diferentes paradigmas significa, portanto, representar segundo modelos diferentes, o problema a ser resolvido pela máquina. Cada linguagem que suporta determinado paradigma representa, portanto, um "meio", no sentido usado por Papert de "estrutura básica", através do qual o problema-alvo é representado. Dessa maneira, o paradigma subjacente à linguagem "molda" a representação do problema a ser resolvido pela máquina. Nosso domínio de estudo, em particular, compreende o paradigma da programação em lógica através de Prolog.

Duas perguntas devem, ainda, ser respondidas: porquê programação e porquê Prolog.

Vários pesquisadores (Papert, 1991; Pea, 1985; O'Shea, 1983), têm mostrado que programação oferece oportunidades extraordinárias para o "aprender através do fazer". Nossa proposta tem como base a teoria construcionista*4 de Papert que leva a uma visão de conhecimento tecnológico como "reflexivo", cujo valor não existe somente pelo conhecimento *per se*, mas, também pela capacidade de aumentar o aprendizado de muitas outras coisas (Papert, 1991). Essa teoria coloca-se na tradição das teorias psicológicas construtivistas, como a de Piaget, Vygotsky e outros, olhando os aprendizes como "construtores" de suas estruturas intelectuais,

*4 A palavra construcionista está sendo usada como tradução "literal" de *constructionism*. A tradução mais próxima, que seria construtivismo, tem um significado bem definido em teorias de aprendizagem, que não traduz exatamente o significado de Papert para *constructionism*.

como "criadores" de conhecimento. Portanto, a ênfase não está colocada em aprender programação em seu sentido restrito, mas aprender através de programação.

Prolog tem sido usada em educação desde 1981. Foi sugerido em 1980 por Kowalski que lógica poderia ser usada como uma linguagem de programação para crianças. Desde então, ferramentas baseadas em Prolog para uso em educação têm sido tema de pesquisas lideradas principalmente por projetos no Reino Unido, Israel, França, Canadá, Itália, Austrália e Dinamarca.

Não faltam argumentos para resposta à pergunta "porquê Prolog", especialmente entre grande parte da comunidade de pesquisadores em IA e principalmente entre os grupos envolvidos com pesquisas em cognição. Dos vários argumentos que são apresentados na literatura, citaremos alguns, que parecem-nos resumir, em significado, a escolha de Prolog para o contexto de nosso cenário.

Programação em lógica parece ter muitos dos benefícios intelectuais de se estudar Lógica, o que era representado no passado por linguagens como o Grego e o Latim e a Geometria Euclideana. A possibilidade de sua execução potencial no computador viabiliza o trabalho com raciocínios dedutivo, indutivo e abdutivo, sem a necessidade de um conhecimento a priori rigoroso da teoria da Lógica (Scherz, 1986). Esse tipo de prática é realimentada pela resposta da máquina e, colocada em nosso cenário, possibilita o desenvolvimento da meta-cognição, uma vez que requer dos sujeitos que conscientemente estruturem seu pensamento e clarifiquem sua "linguagem", considerando que eles próprios têm o controle do diálogo.

No contexto de Ciência da Computação, Prolog oferece ao estudante uma nova maneira de "enxergar" o computador e a própria "computação". Sistemas baseados em conhecimento representam uma alternativa para as aplicações numéricas tradicionais e sua abordagem procedural. Briggs sugere que esse tipo de envolvimento pode ter uma significação profunda para os contactos que o estudante terá, no futuro, com expertise de modo geral (Briggs, 1988). Além disso, para os

estudantes de Ciência da Computação, Prolog pode ser vista como uma linguagem para implementação, "*a primeira de uma série de ferramentas declarativas com as quais ferramentas de mais alto nível podem ser construídas, possibilitando novas maneiras de pensar sobre o conhecimento*" (Briggs, 1988, p. 120).

Para compreendermos melhor as questões envolvidas com o desenvolvimento de programas no meio gerado pelo paradigma da programação em lógica, devemos situar Prolog no contexto de "programação em lógica".

Embora tenha sido usado "lógica" no projeto de computadores e para pensar em computadores e programas, desde o início da tecnologia de computadores, o uso de lógica diretamente como linguagem de programação surgiu apenas no início dos anos 70. Essa idéia, chamada "programação em lógica" é derivada de um modelo abstrato de computação sem relação ou dependência direta com o modelo von Neumann de máquina.

O enfoque do paradigma da programação em lógica para se representar o problema-alvo a ser resolvido pelo computador, consiste em expressar o problema na forma de um conjunto de axiomas lógicos ("fatos" e "regras" sobre o domínio do problema) que são interpretados como "programas". Dessa maneira, a representação da solução para o problema, muitas vezes confunde-se com a própria representação do problema. Segundo o modelo de programar proposto pelo paradigma da programação em lógica, o significado de um "programa" não é mais dado por uma sucessão de operações elementares que o computador supostamente realiza, mas por uma base de conhecimento a respeito de certo domínio e por perguntas feitas a essa base de conhecimento, independentemente. Existe, por trás do programa uma máquina de inferência, em princípio escondida do programador, responsável por "encontrar respostas" para o problema descrito.

Dessa maneira, podemos definir um "programa em lógica" como um conjunto de axiomas, ou regras, definindo relacionamentos entre objetos. Uma

computação de um programa em lógica é uma dedução de consequências do programa. O significado de um programa é dado pelo conjunto de consequências que ele define. Essa definição pode ser sumarizada pelas seguintes equações metafóricas:

> "programa = conjunto de axiomas
>
> computação = prova construtiva de uma meta de um programa" (Sterling, 1986, p. xviii)

Para ilustrar a definição, consideremos, por exemplo, o programa a seguir, que contém axiomas definindo a relação "máximo divisor comum" (mdc) para dois números naturais:

mdc(X,0,X).
Leia-se "o mdc entre X e 0 é X ".

mdc(X,Y,Z) se resto(X,Y,R) e mdc(Y,R,Z).
Leia-se "o mdc entre X e Y é Z se o resto da divisão inteira de X por Y é R e o mdc entre Y e R é Z".

Os dois axiomas acima constituem, portanto, o programa em lógica para se determinar o mdc entre dois números naturais.
Uma meta poderia ser:

"Existe M tal que mdc entre 8 e 12 é M" ou *mdc(12,8,M)*

Um mecanismo construtivo é aplicado para se tentar "provar" a meta. No caso de ter sido bem sucedido (provado a meta) ele fornece a identidade das incógnitas

(variáveis) mencionadas na meta. Esse é o resultado (resposta) da computação. No nosso exemplo $M = 4$.

Prolog é a linguagem de programação desenvolvida em uma máquina sequencial, que mais se aproxima do modelo de computação proposto em "programação em lógica". Prolog é uma abreviação de "programming in logic" (ou programation et logique), uma idéia que emergiu no início dos anos 70, através dos trabalhos de Robert Kowalski e Maarten van Emden, de Edinburgh e Alain Colmerauer, de Marseilles. O mecanismo de execução de Prolog na máquina sequencial é uma simulação sequencial do mecanismo de computação proposto em "programação em lógica" e é obtido a partir de um interpretador que parte da meta mais à esquerda e substitui a escolha não determinística de uma cláusula na base de dados, por uma busca sequencial de uma cláusula unificável e retrocesso (*backtracking*).

Programação em Prolog consiste em definir relações e questionar sobre relações. Uma relação pode ser especificada por "fatos" (n-tuplas de objetos que satisfazem a relação) e/ou por "regras" sobre a relação. Um "programa" Prolog consiste, portanto, de três tipos de cláusulas: fatos, regras e perguntas (metas). Ao conjunto de fatos e regras que definem o domínio de conhecimento do programa chamamos "base de dados".

Dois tipos de semântica podem ser associadas a um programa Prolog: uma semântica "declarativa" e uma semântica "operacional", atribuindo ao programa dois tipos de "significado": declarativo e operacional. A semântica declarativa é relativa somente às relações definidas pelo programa e, dessa forma, determina "o quê" é computado. A semântica operacional determina "como" as relações são avaliadas pelo interpretador Prolog, para conduzir a um resultado. A diferença entre esses dois significados pode ser observada a partir das diferentes "leituras" que se pode fazer, por exemplo, da cláusula:

A se B1 e B2 e B3 e ... e Bn.

Considerando a semântica declarativa da cláusula, podemos "lê-la" como:

"A é verdadeira se B1 e B2 e ... Bn são verdadeiras" ou
"De B1 e B2 e ...Bn segue A".

Considerando a semântica operacional da cláusula, ela poderia ser "lida" como:

"Para resolver (executar) A, primeiro resolva (execute) B1, depois resolva (execute) B2, ... e então resolva (execute) Bn" ou
"Para satisfazer a meta A, primeiro satisfaça a submeta B1, depois B2, ... e então Bn"

Dessa maneira, a semântica operacional fornece uma maneira de descrever "proceduralmente" o significado de um programa, enquanto que a semântica declarativa é baseada na semântica do modelo teórico da lógica de primeira ordem.

Essa duplicidade de semânticas não têm sido explicitada para o novato no ambiente de programação Prolog e tem sido tratada na literatura, em função da dupla maneira de se "ler" um programa, como responsável por diferentes "estilos" de programar: o declarativo e o procedural.

Segundo nossa ótica essa tem sido a principal problemática que o novato enfrenta no ambiente Prolog. Para ilustrar esse problema, consideremos as definições das relações *ancestral1*, *ancestral2* e *ancestral3*, como a seguir:

ancestral1(X,Y) :- pais(X,Y).
ancestral1(X,Y) :- pais(X,Z) , ancestral1(Z,Y).

ancestral2(X,Y) :- pais(X,Z) , ancestral2(Z,Y).

ancestral2(X,Y) :- pais(X,Y).

ancestral3(X,Y) :- ancestral3(Z,Y) , pais(X,Z).

ancestral3(X,Y) :- pais(X,Y).

A relação "ancestral", pode ser representada por qualquer dos três pares de cláusulas, definidas e expressar o mesmo significado declarativo "Os ancestrais de alguém são os seus pais e os ancestrais de seus pais". Entretanto, cada definição apresenta significados operacionais distintos, refletidos nas respostas para a pergunta sobre a relação ancestral. Por exemplo, considerando incluídas na mesma base de dados as cláusulas:

pai(maria,joao).[*5]

pai(pedro,paulo).

mae(pedro,maria).

pais(X,Y) :- pai(X,Y).

pais(X,Y) :- mae(X,Y).

A pergunta *ancestral1(pedro,X)* é "respondida" com:

X = paulo;

X = maria;

X = joao;

[*5] As bases de dados estão sendo apresentadas na forma sintática aceita pelo interpretador Prolog. Por isso as palavras aparecem sem acentuação.

A resposta para *ancestral2(pedro,X)* é:

X = joao;

X = paulo;

X = maria.

Ancestral3(pedro,X) não é respondida.

Ou seja, a mesma pergunta gera respostas diferentes para cada definição da relação ancestral, refletindo significados operacionais diferentes para bases de dados equivalentes, do ponto de vista de sua semântica declarativa. Isso significa que existe um processo de interpretação, pela máquina, do conhecimento expresso na base de dados, que não está explícito em seu código.

Em nossa visão do assunto, tratar uma ou outra semântica isoladamente não traduz o meio subjacente à linguagem, como uma estrutura básica sobre a qual o problema alvo deve ser representado.

Analisando as metodologias de introdução do novato em ambientes Prolog, é comum encontrar-se, como reflexo da problemática mencionada, abordagens que apontam em direção a dois casos extremos. Por um lado, o novato, em seu primeiro contacto com uma linguagem de programação é encorajado a escrever proposições que captam, apenas, as relações lógicas entre os objetos do domínio do problema, acreditando não ser necessário conhecer como a "máquina de inferência" usa as proposições para deduzir resultados. Esse tipo de abordagem, em geral, leva a uma visão mistificada da máquina como "respondedora" de perguntas e tende a acentuar uma das grandes dificuldades que o novato encontra para diferenciar, como ainda veremos neste trabalho, o conhecimento do "senso comum", do conhecimento

expresso na base de dados; seu processo de inferir resultados e o da máquina. Além disso, gera dificuldades para o novato ao tentar ultrapassar as "portas de entrada" do Prolog para outros domínios, ou seja ultrapassar a fase da construção de bases de dados envolvendo conhecimento taxonômico.

No outro extremo, o novato em Prolog, com experiência em linguagens procedurais, frequentemente tem em mente o comportamento de uma máquina procedural enquanto está representando o problema, e tem dificuldades em expressá-lo declarativamente. Consideremos, por exemplo, a definição da relação "último" elemento de uma lista dada. É comum entre esse tipo de novato, a definição da cláusula:

"*ultimo([X],Y) :- X = Y*" (o último elemento de uma lista unitária é um certo Y onde Y é igual ao elemento da lista unitária)

em vez da cláusula: "*ultimo([X],X)*" (o último elemento de uma lista unitária é ele próprio), onde se percebe que o sujeito está usando como "ponte" para representação do problema, um meio procedural, com elementos presentes no paradigma procedural (atribuição de valores a variáveis, comandos condicionais, por exemplo).

Estes dois tipos de novatos são os sujeitos de nossa investigação e muitos outros exemplos de sua problemática no cenário descrito de início serão apresentados no decorrer deste trabalho.

Se, por um lado apenas o conhecimento declarativo não é suficiente para efetivar a comunicação do novato nesse cenário, por outro lado, o tipo de ferramenta mais comumente encontrada no ambiente Prolog para abordar aspectos operacionais envolvidos em programação tem sido os rastreadores de código (*tracers*). Os *tracers* tradicionais, herdaram o estilo dos *tracers* para linguagens procedurais, e como tal, mostram a execução de um programa como um processo linear.

Eles não tornam explícita a estrutura do processo de inferência de Prolog, dificultando ao novato "enxergar a floresta apesar das árvores".

Em nossa ótica, um ambiente para representação de soluções para problemas, em determinada linguagem de programação deve fornecer elementos que favoreçam o envolvimento do novato no "meio" definido pelo paradigma da linguagem. No contexto de Prolog isso envolve possibilitar ao novato acesso ao programa em seu significado lógico e em seu significado operacional (Baranauskas, 1991a; Baranauskas, 1991b). A conquista da linguagem (Prolog) como "meio" para representar problemas depende de um equilíbrio entre a mistura dos aspectos declarativo e operacional necessários para efetivar a comunicação com a máquina.

Nossa proposta para tratar a problemática envolve o design de ferramentas computacionais baseadas em representações alternativas para o conhecimento expresso em um programa Prolog, que instrumentalizem o novato para a criação de "micro-mundos declarativos" e "micro-mundos operacionais". No micro-mundo declarativo ele tem acesso ao conhecimento declarativo expresso em seu programa, de forma independente de qualquer mecanismo de execução e no "micro-mundo operacional" ele tem acesso a uma representação da semântica operacional de seu programa onde pode acompanhar os processos de inferência da máquina Prolog.

A idéia das ferramentas é fornecer ao sujeito, através de representações gráficas, *feedbacks* de natureza declarativa e/ou operacional, de acordo com sua escolha. Para ilustrar, consideremos, por exemplo, a base de dados a seguir:

```
local(P,L):-visita(P,O),local(O,L).

local(P,L):-em(P,L).

em(alam,sala19).

em(jane,sala54).
```

```
em(bete,escritorio).

visita(davi,alam).

visita(janete,bete).

visita(alberto,davi).
```

Para examinar uma base de dados, independentemente do objetivo que leva o sujeito a fazê-lo (ele pode estar tentando "entender" o programa, antecipando a resposta para uma certa pergunta, procurando o motivo de uma resposta errada, etc.), o sujeito pode proceder a uma análise sistemática do fluxo de execução ou pode fazer uma "meta-análise"*6 do texto. O *expert*, em geral faz ambos (Hook, 1990).

As ferramentas do Módulo Operacional instrumentalizam o sujeito para fazer uma análise sistemática do fluxo de execução de seu programa. A figura 1.1 mostra uma representação da hierarquia de metas usadas pela máquina de inferência na busca de respostas para a pergunta *"local(alberto,X)"*.

*6 meta-análise refere-se a análises feitas nos programas que se baseiam em outros fatores, que não análise sistemática e exaustiva de fluxo de controle (Hook, 1990).

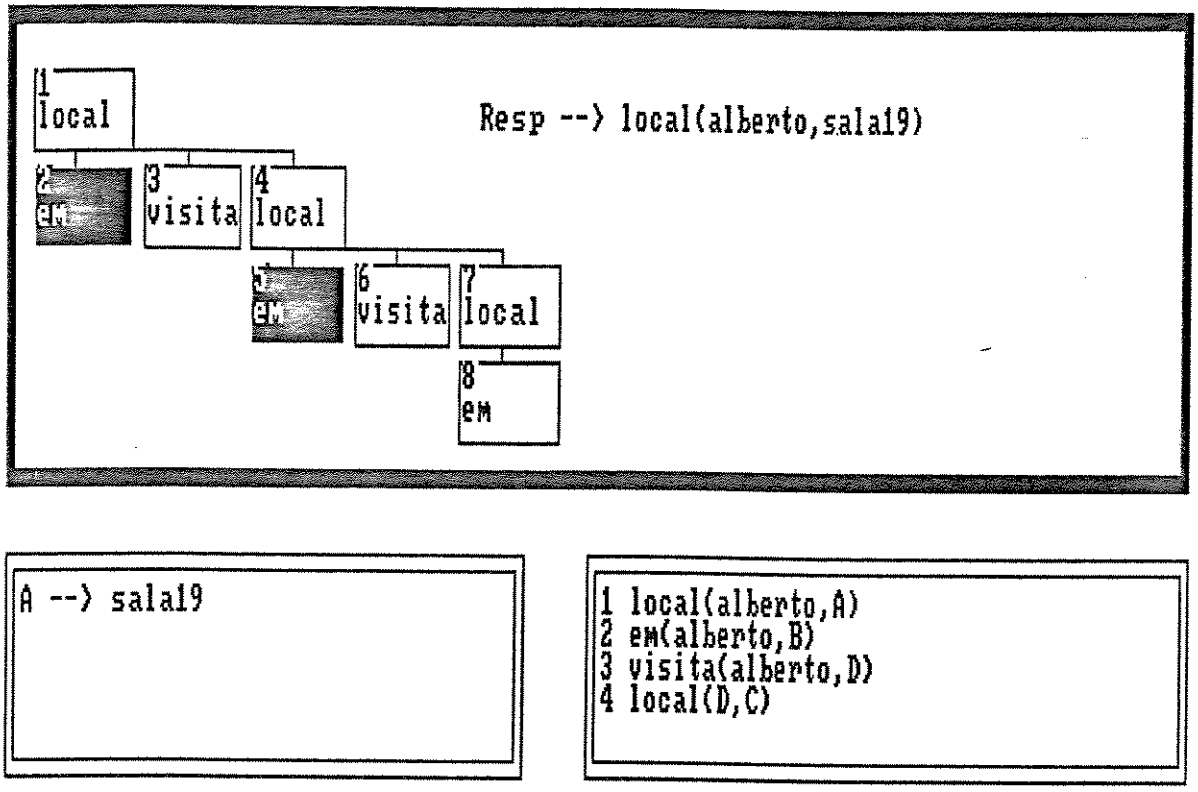


Figura 1.1 Exemplo de Representação usada no Módulo Operacional

As ferramentas do Módulo Declarativo instrumentalizam o sujeito para a meta-análise da base de dados, colocando-o em contacto com as relações presentes entre os elementos da base de dados (ou relações que podem ser deduzidas a partir da base de dados), sem envolver o conhecimento do mecanismo de execução de Prolog. A figura 1.2 mostra porque a resposta à pergunta "*local(alberto,X)*" é "*sala19*", com base na representação do inter-relacionamento entre os objetos da base de dados.

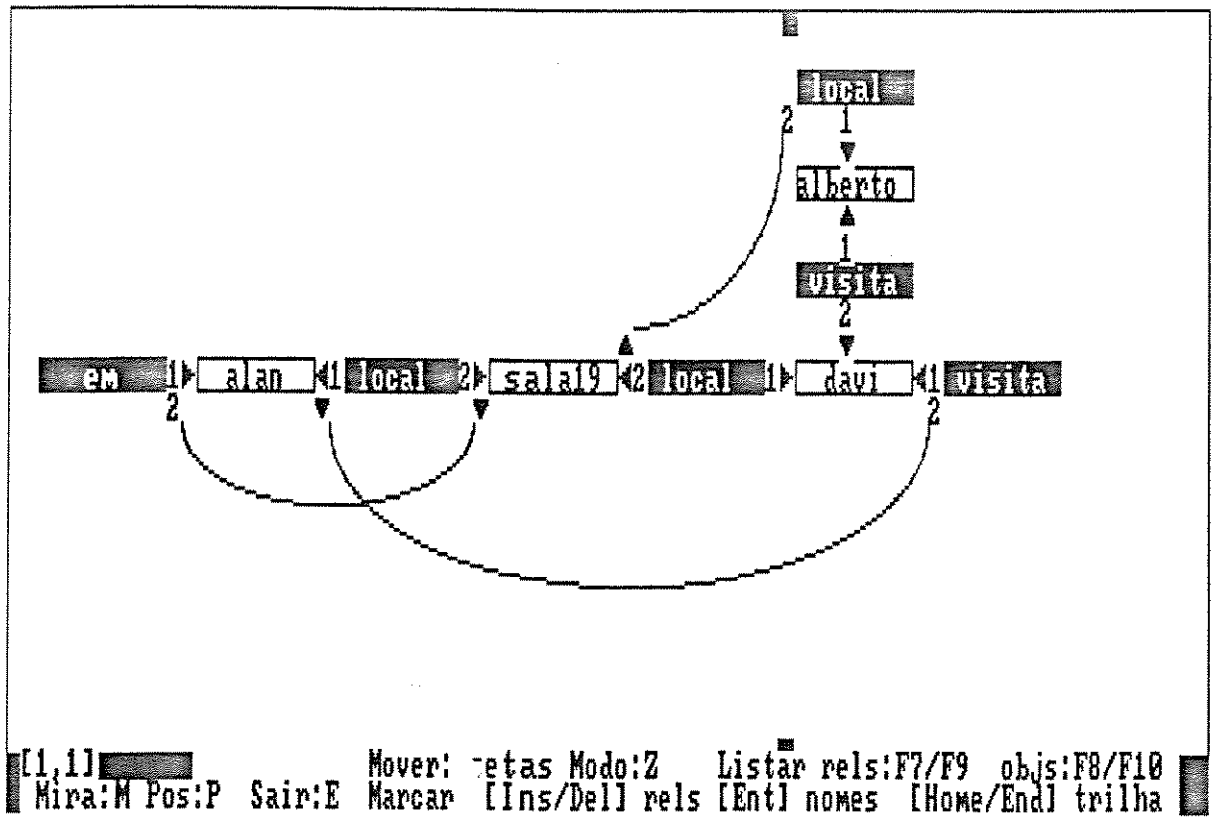


Figura 1.2 Exemplo de Representação usada no Módulo Declarativo

O estudo da interação do novato no ambiente proposto considera o momento em que nos encontramos em relação ao desenvolvimento de novas tecnologias para computadores, enquanto geradoras de novos paradigmas de programação. As mudanças que já começam a acontecer em termos do hardware das máquinas levam a mudanças profundas no tipo de uso dessa tecnologia, que se refletirão na atividade de programar.

Nossa contribuição neste trabalho é em direção ao entendimento dos processos envolvidos na representação da solução de problemas em um dado paradigma de programação, através do *design* de ferramentas computacionais que, ao mesmo tempo em que auxiliam o novato na apropriação da linguagem para re-

presentação de problemas, funcionam como modelos computacionais que possam conduzir a esse entendimento.

No capítulo 2 definimos mais detalhadamente o "cenário" de nossa pesquisa, apresentamos um panorama geral das ferramentas que têm constituído ambientes Prolog e discutimos a problemática que o novato enfrenta nesses ambientes.

No capítulo 3 apresentamos o quadro teórico que suporta nossa proposta de ambiente de aprendizagem baseada no computador e discutimos a questão do paradigma de programação como "modelo" para representação de soluções de problemas. Apresentamos, ainda, as opções de *design* e fundamentos para as ferramentas propostas.

No capítulo 4 apresentamos os objetivos especificos do trabalho e discutimos a metodologia de pesquisa adotada.

No capítulo 5 apresentamos uma visão geral do conjunto de ferramentas que compõem o ambiente proposto e discutimos as principais opções de *design* adotadas.

No capítulo 6 apresentamos o modelo de representação, a descrição funcional e aspectos de implementação das ferramentas que constituem o Módulo Operacional.

No capítulo 7 apresentamos o modelo de representação, a descrição funcional e aspectos de implementação das ferrramentas que constituem o Módulo Declarativo.

No capítulo 8 apresentamos o experimento realizado para uso do ambiente proposto e resultados obtidos do estudo da interação de diferentes tipos de novatos no ambiente composto das ferramentas desenvolvidas.

No capítulo 9 apresentamos uma discussão geral do trabalho, conclusões e novas direções para a pesquisa.

Capítulo 2

O cenário, o Estado da Arte e a Problemática

Capítulo 2

O Cenário, o Estado da Arte e a Problemática

"O problema é que eles ignoram que ignoram." Meta-cognição e reflexão são as chaves de acesso ao conhecimento. (TECFA, 1990)

O cenário para este trabalho envolve um sujeito trabalhando em um ambiente baseado em computador com o objetivo de resolver um problema. O objetivo do sujeito é a apropriação da linguagem de programação para representação da solução do problema. Mais especificamente, o sujeito é um novato e o ambiente baseia-se em Prolog, uma linguagem que exemplifica o paradigma da programação em lógica.

O objetivo deste capítulo é discutir a problemática envolvida em tal cenário, considerando os tipos de ferramentas que encontramos atualmente na literatura e apresentar a nossa proposta sobre o ambiente em consideração.

O uso de Prolog, de início restrito aos interesses de grupos de pesquisa em IA, passou a integrar vários cursos nas universidades, como linguagem de programação. Paralelamente a seu ensino nas universidades, Prolog começou a ser usado também no ensino do segundo grau (Ennals, 1983). Foi observado desde o início que, embora a linguagem fosse muito poderosa e alguns estudantes a usassem com grandes efeitos, outros ficavam bastante confusos (Brna, 1990).

Vários grupos de pesquisa começaram a estudar as dificuldades associadas a Prolog. Alguns dos problemas apontados estão relacionados à idéia de que seria possível escrever programas como afirmações logicamente corretas sem a

necessidade de pensar em como Prolog responde a uma dada pergunta. Essa possibilidade de abstrair aspectos de controle, entretanto, tem como preço a complexidade da máquina subjacente necessária para alcançar esse efeito (Brna, 1990). Os processos de unificação e busca em profundidade implícitos na máquina Prolog são muito poderosos, mas também difíceis de entender, especialmente para um novato.

Tem sido amplamente divulgada na literatura (Brna, 1990; Van Someren, 1990; Fung, 1990), a dificuldade dos usuários com o comportamento de Prolog quando uma meta falha (*backtracking*) e com a forma como Prolog faz casamento de padrões (unificação). Essa dificuldade tem sido explicada como um reflexo da dificuldade do programador em "seguir a execução" de um programa e tem sido associada à atividade de depuração de programas.

A complexidade acrescentada à tarefa de depuração de programas, observada no uso da linguagem desde seu início, deu origem às primeiras iniciativas de criação de modelos de execução para Prolog. O primeiro deles, chamado modelo Byrd Box (Byrd, 1980) tem sido usado, desde então e até recentemente, como base para depuradores usados em ambientes de programação Prolog disponíveis comercialmente. Outras iniciativas foram feitas, no sentido da criação de ferramentas para tornar mais claros os mecanismos computacionais de Prolog, através da apresentação de modelos de como o interpretador chega a uma resposta para uma pergunta (Rajan, 1986; Dewar, 1986; Plummer, 1988; Eisenstadt, 1988).

Os termos "ferramenta" e "ambiente" têm sido usados, na literatura, com significados diferentes, dependendo do contexto. Ferramenta pode designar tanto um dispositivo físico como por exemplo um "*mouse*", quanto um programa "utilitário" com por exemplo um processador de texto. Usaremos o termo ferramenta para designar sistemas (programas) que poderiam ser utilizados no cenário descrito de início, com o objetivo de facilitar ao sujeito o processo de apropriação da linguagem para representação de problemas. Chamaremos "ambiente de progra-

mação" à linguagem básica, no caso Prolog, acrescida de várias ferramentas que podem estar integradas na linguagem, fornecidas em separado como utilitários ou disponíveis em bibliotecas de programas de apoio à linguagem.

As dificuldades dos novatos em ambientes Prolog têm sido analisadas em função dos seus erros mais frequentes. O desenvolvimento de ferramentas para o ambiente Prolog tem sido, desde então, orientado para a atividade de depuração de programas e tem enfatizado os aspectos operacionais envolvidos na tarefa de programar. A maioria das iniciativas tem enriquecido o ambiente Prolog para o profissional; muito pouco tem sido considerado com relação as necessidades do novato. A problemática da aquisição da linguagem de programação não tem sido discutida na literatura, de uma forma mais global.

A análise da natureza da atividade do novato no ambiente de programação é vital para se descobrir que tipo de dificuldades ocorrem, como elas são superadas e que tipo de ferramenta poderia auxiliá-lo no processo de apropriação da linguagem como meio para representação de problemas.

Em nossa ótica, a atividade de programar em Prolog envolve o ciclo criar, questionar e depurar a base de dados. Principalmente se considerarmos a atividade de um novato no ambiente de programação, é difícil separá-la em duas partes (programação e depuração) uma vez que, em geral, o *feedback* do erro é que o levará a refletir e reformular sua base de dados num constante refinamento da representação da solução para o problema, conforme discutiremos a seguir.

2.1 Depuração como Atividade e como Processo

Historicamente, a tarefa de depuração*1 tem sido parte da atividade de desenvolver de programas, desde que o primeiro programa de computador foi escrito. Segundo

*1 *debugging* no original

Seviora o termo foi aplicado pela primeira vez a um defeito no *hardware*: uma traça*2 na circuitaria do MarkII (Seviora, 1987). Entretanto, há pouca teoria formal a respeito da atividade de depuração e tentativas de desenvolver uma metodologia para depuração são raras.

A ótica da depuração como a "atividade pela qual erros no código do programa são detectados e corrigidos", não revela a complexidade do processo, principalmente quando transposta do paradigma procedural para o paradigma subjacente a Prolog. Brna *et al* apontam algumas das dificuldades que o programador encontra ao depurar um programa Prolog:

> - programas são difíceis de compreender num nível operacional, pois a linguagem não tem "marcadores" sintáticos que são encontrados em linguagens procedurais ou funcionais, como por ex. em Pascal ou Lisp.
> - os processos de unificação e backtracking podem dificultar a localização de erros.
> - por causa da natureza lógica de Prolog, nenhum programa sintaticamente correto teria qualquer mensagem de erro associada a ele. (Brna, 1988b, p. 1)

Para descrever as habilidades que programadores necessitam, Brna *et al* propõem um diagrama onde existem quatro níveis nos quais erros podem ser descritos: o sintoma, o mal comportamento do programa, o erro no código do programa e o nível de erro conceitual (fig 2.1).

---

*2 *bug*

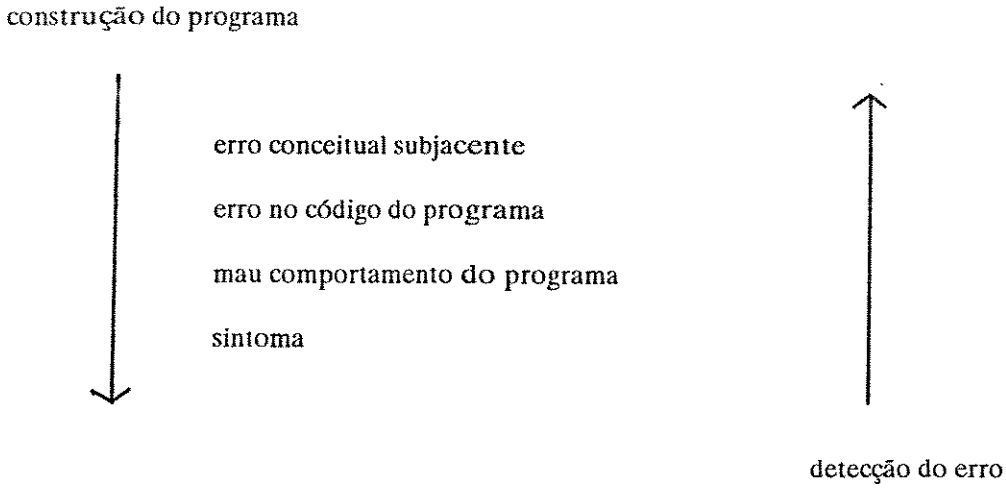


Figura 2.1 Níveis de descrição de erros em Prolog (extraída de Brna 1988a, p. 3)

De acordo com esse diagrama, pa a depurar um programa o programador necessita de uma série de conhecimentos e habilidades, entre os quais são apontados:

> - ter uma compreensão completa da semântica intencional do programa.
> - saber relacionar os sintomas com possíveis erros no código.
> - ser capaz de descrever o mal comportamento do programa em função do erro.
> - ser sensível a captar a descrição do erro no código a partir da descrição do mau comportamento do programa.
> - ser capaz de eliminar possibilidades através de uma seleção especial de testes de casos discriminatórios.
>
> (Brna, 1988b, p. 2).

Em nossa maneira de ver o assunto, em se tratando do contexto do novato, é impossível isolar a atividade de depuração do processo mais amplo de aprender a programar. Mais do que isso, criação e depuração de programas fazem parte de um ciclo

em que o novato se engaja, durante o qual ele constrói o conhecimento da linguagem de programação.

O ciclo em que o sujeito se engaja, no contexto de nosso cenário, envolve a construção de uma base de dados (programa) a partir de "hipóteses" (a respeito da representação da solução do problema intermediada pela linguagem de programação, a respeito do domínio do problema, a respeito do ambiente de programação, etc.). Existe um "*feedback*" a partir de um resultado da execução do programa pela máquina de inferência de Prolog, que pode levar o novato a um processo de "reflexão", necessário para que ele faça o caminho inverso, reformule suas hipóteses e recomece o ciclo. O diagrama representativo das ações do sujeito em nosso cenário pode ser ilustrado da seguinte maneira (figura 2.2):

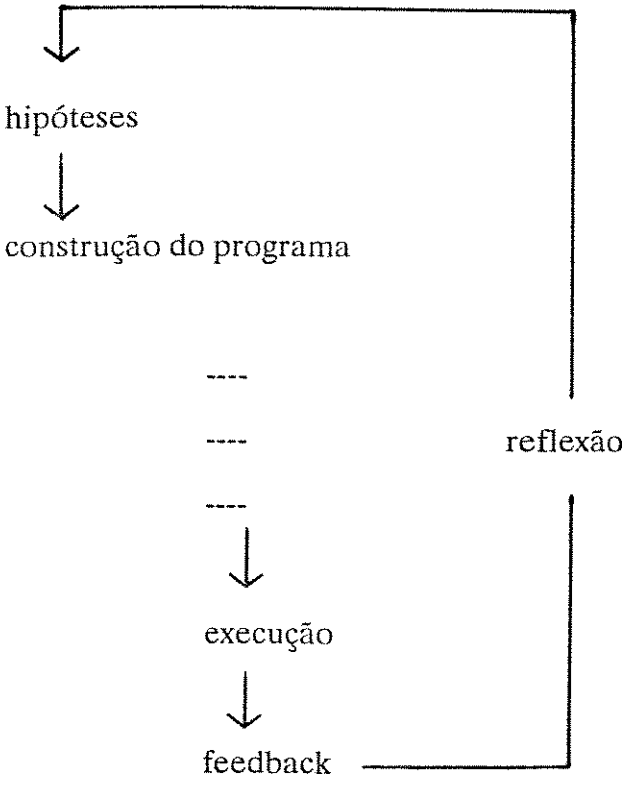


Figura 2.2 O Ciclo Hipótese-Criação-Execução-Feedback

Neste contexto, a depuração deixa de ser vista como simplesmente uma atividade de busca e eliminação de erros, no sentido de ser um "fardo" que se quer eliminar e passa a ser um processo durante o qual o conhecimento vai sendo construído; não somente conhecimento do domínio em questão (o problema que está sendo resolvido), mas, também a respeito do interlocutor (ambiente de programação) e, num nível meta-cognitivo, conhecimento a respeito do próprio processo de "conhecer". Dessa maneira, o conjunto de habilidades necessárias à depuração faz parte do elenco de conhecimento que o novato irá construir nesse ambiente.

Em nossa ótica, não se trata, portanto, de "minimizar" a criação de erros, mas criar ambientes onde a construção do conhecimento da linguagem se dê de maneira natural. À composição do ambiente de programação nesse cenário acrescido do ciclo de ações do sujeito chamaremos "ambiente de aprendizagem".

A seguir, descreveremos os tipos de ferramentas disponíveis em ambientes de programação baseados em Prolog e procuraremos analisá-las sob o aspecto de adequabilidade ao nosso cenário.

2.2 Ferramentas em Ambientes Prolog

As iniciativas de desenvolvimento de ferramentas para depuração de programas têm acontecido, em função de se considerar a atividade de depuração como um "processo ineficiente", com o objetivo de criar ambientes onde se "minimize a criação de erros e maximize a eficiência do processo" (Brna, 1988).

Embora alguns pesquisadores expressem uma visão de programação Prolog relacionando ambos construção e depuração de programas, as ferramentas existentes em geral têm premiado a fase de depuração, isoladamente.

A seguir analisaremos os tipos de ferramentas existentes atualmente e seus diferentes enfoques no processo de depuração, em função do ciclo de ações do novato em nosso cenário. As abordagens têm variado em termos da estratégia aplicada, papel do usuário no processo, flexibilidade, precisão, etc..

2.2.1 Depuração Automática

A idéia de um método completamente automático para detecção de erros no código do programa e possível correção é uma idéia bastante atrativa, do ponto de vista acadêmico, mas criticamente depende do sistema conhecer efetivamente a especificação do programa.

Os "Sistemas Tutores Inteligentes" (ITS), em geral utilizam essa abordagem. Nos ITS a meta é ensinar programadores novatos a escrever programas corretos. Nesse contexto, o sistema pode colocar um conjunto de problemas para os quais a solução é conhecida. O mecanismo de detecção automática pode ser usado para apontar os erros no código e verificar que as correções tenham sido aplicadas. Essa não é uma abordagem que ensina a depurar uma vez que não há apresentação do raciocínio envolvido na identificação ou busca do erro. Além disso, tais sistemas em geral possuem como característica a insistência prescritiva e/ou dialética (Brna, 1988a). Insistência prescritiva é a denominação dada ao comportamento enfático do sistema sobre o erro detectado, que impede o usuário de prosseguir em sua tarefa, a menos que o erro seja eliminado. Insistência dialética, refere-se ao comportamento do sistema em suas interações com o usuário, não permitindo que este influencie o curso da sessão de depuração.

Esse tipo de ferramenta não se aplica ao nosso cenário, onde o sujeito irá construir o conhecimento da linguagem a partir de sua interação no ambiente. O curso da sessão de depuração é parte do ciclo em que o novato se engaja e é con-

duzido pelo próprio sujeito. O "erro" tem papel importante nessa interação, juntamente com as ferramentas que devem fornecer *feedback* suficiente para que ele saiba "o que fazer em seguida". Não se trata, portanto, de "apontar" o erro, mas facilitar ao sujeito a "descoberta" do erro. A tarefa do sujeito na "remoção" do erro é muito mais complexa do que essa idéia sugere, pois envolve reformulação de hipóteses e refinamento da representação, a partir de um processo de reflexão do sujeito.

2.2.2 Depuração Guiada

Neste tipo de sistema ambos, programador e sistema, cooperam na procura do erro através de um diálogo, em geral guiado pelo sistema, sobre a tarefa de depuração.

Os sistemas que requerem resposta do usuário tipicamente procuram o erro enquanto o programa está sendo executado. O programa geralmente pede informação sobre o comportamento desejado de uma meta em particular ou se um resultado particular está correto. A suposição feita é que o programador conhece exatamente quais metas devem suceder, quais devem falhar e como (e em que ordem) variáveis serão associadas a valores. Como é assumido que o programador irá entrar com as respostas corretas (é assumido ser infalível), a origem desta informação é chamada "oráculo". Tal questionamento feito para o usuário permite ao sistema inferir o comportamento desejado do programa e facilitar a tarefa de depuração para o sistema.

Shapiro formalizou e desenvolveu soluções algorítmicas para três tipos de erros e os integrou em um sistema chamado PDS (Prolog Diagnosis System) (Shapiro, 1983). A descrição de erros que Shapiro usa pode ser classificada a nível do código do programa em duas categorias: aqueles devido a um procedimento[*3]

*3 o termo procedimento é usado aqui designando o conjunto de cláusulas que define um determinado predicado.

incorreto e aqueles devido a um procedimento incompleto. Ele mostra que isto pode ser feito e fornece alguma maneira de lidar com (aparente) não terminação.

PDS pode ser usado para investigar programas com resultado incorreto (1), terminação com resultado incompleto (2) e não terminação (3). Para encontrar alguma descrição apropriada de erro no código do programa, os algoritmos de Shapiro que manipulam erros do tipo 1 e 2 interrogam o usuário ao nível do mal funcionamento do programa. Para erros do tipo 3, ele faz uso de checagem do limite de profundidade de execução como um meio de sugerir que um "*loop*" pode existir.

Assim, é feito uso intensivo do conhecimento do programador para prover informação correta para o sistema na forma de respostas a perguntas sobre o comportamento esperado do programa. Para ilustrar, mostramos um exemplo de diagnóstico para erro do tipo 1 numa base de dados definida pelo usuário, que representa o problema de ordenar os elementos de uma lista, usando o método conhecido como "ordenação por inserção".

Base de Dados:

```
isort([X|Xs],Ys) :-isort(Xs,Zs), insert(X,Zs,Ys).
isort([],[]).
insert(X,[Y|Ys],[X,Y|Ys]) :- X <= Y.
insert(X,[Y|Ys],[Y|Zs]) :- insert(X,Ys,Zs).
insert(X,[],[X]).
```

Chamada do sistema:

```
? fp(isort([4,1,2,3,5,6],[1,2,3,5,4,6]),17,C).
```

Exemplo de diálogo:

sistema: isort([2,3,5,6],[2,3,5,6])?

usuário: sim.

sistema: insert(4,[2,3,5,6],[2,3,5,4,6])?

usuário: não.

sistema: insert(4,[5,6],[5,4,6])?

usuário: não.

sistema: insert(4,[6],[4,6])?

usuário: sim.

Resposta do sistema:

C= insert(4,[5,6],[5,4,6]) :- insert(4,[6],[4,6])

(Shapiro, 1983 p. 50)

Existem alguns problemas inerentes ao sistema de depuração baseados em "oráculo", que são agravados principalmente se considerarmos o contexto do novato. Um problema que estes sistemas compartilham é que fornecem mensagens crípticas. Primeiramente eles não mostram ao usuário porquê uma informação particular é requerida pelo sistema (o raciocinio por trás da pergunta). Segundo, seu relatório de diagnóstico pode ser extremamente obscuro. A dificuldade para o usuário, principalmente o novato, está em inferir o contexto no qual a questão está sendo perguntada pelo sistema. Sem informação de contexto fica difícil, para o novato saber "o quê fazer em seguida". Além disso, certos sistemas não permitem aos usuários "mudar de idéia" no meio do caminho e, assim, se uma resposta incorreta do sujeito ou erro de entrada ocorre, o programa também chegará a uma conclusão errada.

2.2.3 Depuração por Monitoramento

A atividade de inspecionar a execução de um programa em um certo nível de detalhe, em geral, pode trazer benefícios ao programador, que incluem: diminuição da carga de memória exigida para exame da execução, possibilidade de focalizar a atenção na execução de partes selecionadas do código, possibilidade de entendimento da semântica operacional do programa.

Os vários sistemas de visualização da execução de Prolog têm evoluído basicamente dentro da concepção do modelo de "portas" também chamado "Byrd Box". O modelo de "árvore e/ou" é bastante recente e pode tornar-se um novo padrão. A seguir descreveremos brevemente essa evolução.

O sistema da DEC-10 (Bowles, 1988):

O *tracer* da DEC, descrito em termos do chamado modelo "quatro portas" de execução de Prolog, tornou-se o padrão. As 4 portas são CALL, EXIT, REDO, e FAIL. O modelo subjacente, também chamado "Byrd Box" é baseado na identificação de uma "caixa" com a invocação de um procedimento específico. Encontrando uma meta, a execução entra pela porta *call* do procedimento que casou. Se a meta sucede, a execução sai do procedimento pela porta *exit*; se ela falha, sai pela porta *fail*. Quando um procedimento deve ser reconsiderado (em função do processo de retrocesso), a execução reentra no procedimento através da porta *redo* e pode novamente sair, ou via a porta *exit* ou via a porta *fail*. A figura 2.3 (adaptada de Brna, 1988), mostra exemplo de um diagrama Byrd Box para a base de dados a seguir.

Base de Dados exemplo:

pais(X,Y):-mae(X,Y).

```
pais(X,Y):-pai(X,Y).

mae(a,b).

pai(c,d).
```

Meta exemplo:

```
pais(X,Y),X==f.
```

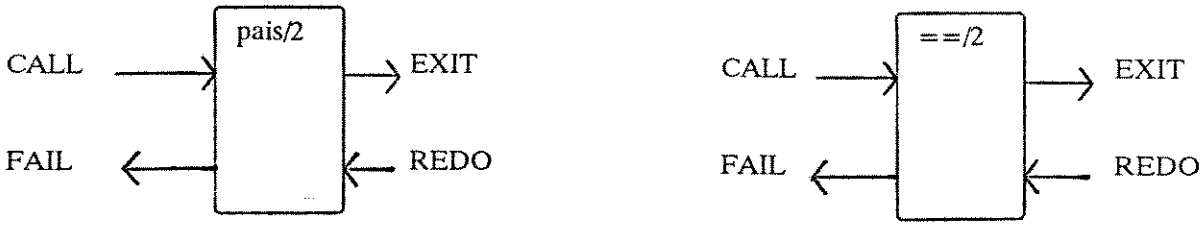


Figura 2.3 Exemplo do Modelo de Portas

A saída de um *tracer* convencional baseado em tal modelo, para o exemplo anterior é mostrada a seguir:

```
CALL pais(X,Y)
CALL mae(X,Y)
EXIT mae(a,b)
Exit pais(a,b)
CALL a==f
FAIL a==f
REDO pais(X,Y)
REDO mae(X,Y)
FAIL mae(X,Y)
CALL pai(X,Y)
```

EXIT pai(c,d)

EXIT pais(c,d)

CALL c==f

FAIL c==f

REDO pais(X,Y)

REDO pai(X,Y)

FAIL pai(X,Y)

FAIL pais(X,Y)

A principal crítica ao *tracer* tradicional é a apresentação "linear" dos passos da execução, herança dos *tracers* para linguagens procedurais. Essa forma de apresentação da execução de programas Prolog não explicita a estrutura do processo utilizado pelo interpretador na busca da resposta (baseado em busca em profundidade e retrocesso). A ausência dessa estrutura interfere no modelo conceitual que o novato cria do processo de execução e pode induzí-lo a certos tipos de erro, conforme será discutido no capítulo 5.

Outros sistemas se seguiram, apresentando pequenas variações no modelo Byrd Box, em geral acrescentando novas portas, que descreveremos brevemente a seguir.

O Sistema CODA (Clause Oriented Debugging Aid), (Plummer, 1987):
Baseado no *tracer* da DEC-10, difere do anterior por ser "baseado em cláusula" em vez de "baseado em procedimento". No modelo CODA a caixa representa a invocação de uma cláusula dentro de um procedimento, explicitando, dessa maneira, informação sobre o processo de unificação. A saída para este modelo pode ser vista como uma extensão do *trace* padrão. Existem 7 portas ao todo: as quatro padrão e

três adicionais: MATCH, EXIT MATCH E FAIL MATCH. O CALL é sempre associado ao corpo da cláusula.

O sistema If/Prolog (InterFace, 1986):
Sistema orientado para telas, com 5 portas: CALL, EXIT, FAIL, REDO E LAST, sendo este último um evento especial, significando a porta EXIT da última sub-meta da meta pai.

O sistema Edinburgh Prolog (Hutchings, 1987):
Também é baseado no *trace* da DEC e usa as mesmas quatro portas, com a porta EXIT renomeada para SUCCEED. Informações adicionais são fornecidas para ambas as portas SUCCEED e FAIL.

O sistema BIM-Prolog (Callebaut, 1987):
Baseado indiretamente no modelo da DEC-10, usa 5 portas onde a porta extra UNIFY é encontrada logo após a porta CALL. Possui, ainda uma exploração da árvore de execução, posterior a execução, que parece ser baseada no trabalho de depuração algorítmica de Shapiro.

O sistema PTP (Prolog Trace Package) (Eisenstadt, 1984):
Essencialmente é o *trace* padrão acrescido de uma grande quantidade de informações como por exemplo,"?" para identificar "chamada de predicado", "]" para identificar " chamada para predicado do sistema", etc. O abandono do modelo das quatro portas em favor de um modelo de 18, além de não eliminar o principal problema dos *tracers* derivados do modelo convencional (a "linearidade"), leva a problemas de sobrecarga de memória para o programador, que tem que tentar relem-

brar o significado de muitos símbolos, a maioria dos quais nem sempre conduzem a uma pista de sua identidade.

O sistema APT (Animated Program Tracer) (Rajan, 1986):
Rajan desenvolveu um sistema de monitoramento baseado na idéia de mostrar a sequência de execução de um programa, em termos do próprio código do programa. Através de indicação em "vídeo reverso", o usuário pode ver as partes de seu código, sendo "rastreadas". Ele acredita que dessa forma, o sujeito pode entender o relacionamento entre as formas estática e dinâmica do programa.

Um aspecto interessante de APT, que não existe nos sistemas anteriores é o fato de o usuário poder relacionar a execução (que é um processo dinâmico) com o código de seu programa (que é uma representação estática). Essa forma de apresentar a execução de programas tem sido adotada em *tracers* para linguagens procedurais. Entretanto, a mesma crítica que fizemos aos *tracers* convencionais, continua válida para o APT: a estrutura do mecanismo de execução não é explicitada.

O sistema TPM (Transparent Prolog Machine) (Eisenstadt, 1988)
O Sistema TPM é um *tracer* gráfico, que teve sua origem no modelo PTP e parece ter surgido como tentativa de resolver o problema colocado por este, fornecendo o mesmo tipo de informação e ao mesmo tempo removendo alguns dos problemas associados com o "achatamento" do espaço de eventos possíveis na execução de um programa.

O modelo introduzido pelo sistema é chamado AORTA (And/Or Tree Augmented). Em adição à estrutura de árvore, ele fornece informação visual sobre se uma sub-meta falhou, sucedeu, ou falhou e então sucedeu; que cláusulas foram tentadas e quantas há para um dado predicado; que variáveis compartilham

valores, associações correntes; o efeito do *cut* e algumas informações sobre invocações prévias de uma tentativa de satisfazer uma meta particular. Ilustramos o modelo, a seguir, com o diagrama AORTA para a mesma Base de Dados do início da seção (figura 2.4). Cada caixa representa uma submeta e contém símbolos que indicam se a submeta sucedeu (V), se falhou (X) ou se sucedeu e depois falhou (W). Na metade inferior da caixa um número indica o número da cláusula, para o predicado em questão, que casou com a sub-meta. Ao lado das caixas são apresentados os textos da sub-meta, da cláusula que casou e indicação da unificação de variáveis.

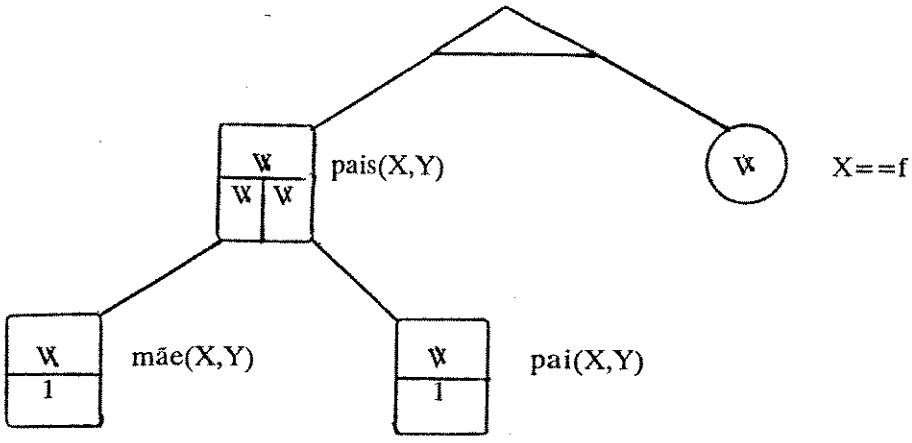


Figura 2.4 Exemplo do Modelo AORTA

O modelo utilizado tem a seu favor a estrutura de árvore, que o deixa mais fiel à representação da semântica procedural de Prolog do que o modelo de portas dos sistemas anteriores. O modelo AORTA, talvez pela influência do PTP do qual se originou, tenta reproduzir na estrutura de árvore, todo o processo de execução, num nível de detalhamento bastante grande. Apesar de seu desenvolvimento recente não ter possibilitado, ainda, uma investigação de sua adequabilidade para uso por novatos, esse modelo parece representar um novo padrão em ferramentas para monitoração em Prolog.

2.2.4 Ferramentas Alternativas

Brna propõe o uso de "técnicas de programação" (Brna, 1991) como maneira de dirigir esforços para a fase de construção de programas. Segundo sua abordagem, existem padrões de código Prolog que são úteis no sentido de serem repetidos muitas vezes em programas. Se um programador tenta criar um predicado com o "padrão" correspondente e faz um erro, então, esse erro no código do programa pode ser descrito em termos das características do padrão utilizado. Esse método fornece um nível mais alto de descrição do erro do que o puramente sintático. Um catálogo de tais técnicas foi proposto como um suplemento para um "editor de estrutura" para Prolog (Brna, 1991). A base para esse trabalho, segundo o autor é a teoria da recursão e tal ferramenta seria usada principalmente para a construção de "programas funcionais".

O objetivo do "editor de estrutura" é evitar que o programador introduza erros, durante o processo de escrever procedimentos Prolog recursivos. A ferramenta permite que somente procedimentos recursivos corretos (a recursão termina e é bem definida) sejam escritos, reduzindo, dessa maneira, o tempo gasto na depuração de programas (Brna, 1990b).

Essa abordagem não parece adequada ao contexto de nosso cenário, pois o próprio conceito de recursão é parte do conhecimento que deve ser construído pelo sujeito. Nesse processo de construção deve haver espaço (oportunidade) para o sujeito elaborar e testar suas hipóteses, criando o código que imagina representar um processo recursivo e testando-o.

2.3 Discussão e Proposta

Vários critérios têm sido usados na análise de tais ferramentas, como por exemplo, quanto do controle é feito pelo sistema de depuração, confiabilidade no diagnóstico de um erro, quanto o sistema precisa conhecer das intenções do programador, quem dirige o diálogo entre o programador e o sistema, tipo de informação apresentada e forma de apresentação, etc..

Somente um pequeno número de sistemas opera sem intervenção do programador embora exista interesse acadêmico no desenvolvimento de sistemas que inferem erros no código de um programa, automaticamente. Na prática, a maioria dos sistemas ou requerem respostas a questões específicas (depuração guiada) ou dão ao usuário uma ferramenta com a qual ele pode localizar e corrigir os erros em seu programa (depuração por monitoramento). A depuração automática tem contra si o problema do custo de encontrar um possível erro no código. Por exemplo, em programas grandes produzidos pelo programador profissional, o custo de encontrar uma cláusula errônea pode tornar-se proibitivamente alto. Em contraste, embora não sem seus próprios custos, monitoramento tem sido muito mais viável. No contexto do novato a identificação de um erro faz parte de uma situação de aprendizado em que ele deve estar engajado. Nesse sentido, o sistema apontar o erro no código de um programa raramente será efetivo no contexto do novato, no sentido de auxiliá-lo a saber "o quê fazer em seguida".

Em relação às ferramentas para monitoramento da execução do programa, com exceção de TPM, essas ferramentas têm conservado o estilo "linear" dos *tracers* de linguagens procedurais. Esse achatamento esconde a estrutura de execução da máquina de inferência de Prolog, dificultando sua construção conceitual pelo sujeito. O modelo Byrd Box mostra o que acontece em determinados pontos do processo de execução do programa, mas não é explícito a respeito de *como* o

processo chegou a esses pontos, o que seria mais útil para o novato. O modelo AORTA, introduzido no sistema TPM tenta satisfazer tanto as necessidades do novato aprendendo Prolog, quanto as necessidades do *expert* com programação em larga escala. Eisenstadt e Brayshaw lidam com as restrições conflitantes impostas por seus objetivos, usando recursos poderosos presentes em estações de trabalho para suportar a construção do espaço de busca para programas grandes e facilitar a seleção de partes e a seleção do nível de detalhes com que o sujeito deseja ver o diagrama AORTA. O novato e o *expert* aparentemente têm objetivos distantes a nível do que esperam de uma ferramenta. Seria desejável um estudo formal a respeito das necessidades de ambos e da utilização de TPM por esses dois tipos de sujeitos, o que ainda não foi apresentado na literatura.

Uma dimensão final pela qual podemos ver uma ferramenta computacional no ambiente de programação é em termos da "carga cognitiva" que ela coloca no usuário. Este problema é ainda maior quando o usuário é um novato em face de aprender uma nova linguagem de programação. Ainda mais se esta é sua primeira linguagem de programação. Por outro lado, formalismos extra podem fornecer uma explicação mais rica tanto do comportamento do programa quanto sobre o processo de entender e depurar um programa. Inevitavelmente existe um compromisso entre o poder potencial de ferramentas para depuração e a quantidade de esforço mental e tempo necessários para dominar o sistema a fim de usá-lo eficientemente e efetivamente no processo de depuração.

Além dessa discussão mais geral, queremos levantar dois aspectos que não tem sido contemplados na literatura a respeito de ambientes de programação: o tipo de sujeito para o qual uma determinada ferramenta é projetada e o tipo de paradigma subjacente à linguagem de programação em questão. Com relação ao primeiro ponto levantado e às ferramentas discutidas na literatura, com algumas exceções (Rajan, 1986; Eisenstadt, 1988), não fica claro a que tipo de sujeito elas

servem. Cabe a pergunta: até que ponto tais ferramentas atendem as necessidades de um programador profissional e de um novato? Parece não haver um consenso a respeito de que tipo de informação é útil para um e desnecessária para outro. Exemplificando essa questão, tem sido argumentado por alguns pesquisadores (Rajan, 1986), a respeito das informações extras fornecidas pelo sistema CODA, (sobre o processo de unificação) que aquele tipo de informação seria útil somente para os novatos. Entretanto, programadores experientes acham tais informações extremamente úteis. Existe muito pouco estudo a respeito da interação de sujeitos com essas ferramentas.

O segundo ponto observado, em relação à discussão anterior, é que os ambientes de programação não têm refletido o "meio" gerado pelo paradigma da linguagem. Em nosso cenário, a interação do sujeito tem como "meio", (intermediador) o paradigma da linguagem de programação. Cada paradigma define um "meio" representado por certas "entidades", através dos quais o problema é moldado. Exemplificando, se o paradigma é o procedural, suas entidades representativas são os conceitos de "variável", "atribuição de valor" e "sequenciação". Se o paradigma é o orientado a objetos, as entidades podem ser os conceitos de "objetos", "mensagens" e "métodos" (Baranauskas, 1991). Dessa maneira, nos diferentes "meios" gerados pelos paradigmas das linguagens existem significados diferentes para "programa" e para a "máquina que o executa". Assim, por exemplo, enquanto no paradigma procedural "programa" é uma "prescrição" de solução para o problema e a "máquina" é um dispositivo que "obedece ordens", no paradigma da programação em lógica, um "programa" é um conjunto de proposições que estabelecem certas relações entre os objetos de um domínio e a "máquina" é uma "máquina de inferência".

A maioria das ferramentas encontradas preocupam-se apenas com a semântica operacional do programa. O *trace* de um programa pode frequentemente contar ao sujeito mais sobre o programa, que justamente localizar um erro. Este en-

tendimento extra pode ser sobre a natureza e contexto do possível erro. Apesar disso, essas ferramentas estarão contando apenas parte da "estória" de Prolog. No contexto do novato, principalmente, é importante considerar a natureza do *feedback* (aspecto operacional, aspecto lógico), pois é através dele que o sujeito será levado a refletir, reformular hipóteses e reconstruir sua Base de Dados. Daí nossa proposta ser em direção ao *design* de ferramentas para investigação/exploração de programas com base no paradigma da linguagem.

Em síntese, a problemática da aquisição da linguagem (Prolog) tem sido traduzida, na literatura, para a problemática da depuração de programas. Daí a ênfase existente em ferramentas para depuração. Argumentamos que, especialmente no caso do novato, depuração é parte do processo de aquisição da linguagem e não uma fase a ser tratada isoladamente. O estudo de como se dá a aquisição da linguagem num ambiente proposto, constitui parte do trabalho de tese. Portanto, nossa proposta refere-se a dois aspectos:

- o *design* de ferramentas computacionais que constituam um ambiente de programação Prolog, onde as entidades do paradigma da programação em lógica sejam "explicitados". Isso envolve fornecer ao sujeito acesso ao programa não apenas em seu significado operacional, mas também em seu significado lógico.
- o estudo da interação do novato com essas ferramentas, para investigação de seu processo de aquisição da linguagem, no cenário proposto.

A necessidade de estudar a atividade de programar, o ambiente de programação e as atividades do novato nesse ambiente, do ponto de vista do paradigma da linguagem, vem de encontro ao momento atual, em que novas arquiteturas de computadores estão surgindo, e com elas novos paradigmas de programação, que moldarão diferentemente a representação de problemas, alterando nossa própria visão da atividade de programar.

Capítulo 3

Quadro Teórico

Capítulo 3

Quadro Teórico

"Gastamos 20 anos trabalhando na direção de sistemas adaptáveis enquanto que a meta real é ajudar os estudantes a tornarem-se, eles próprios adaptáveis." A principal qualidade de um ambiente de aprendizado é ajudar o estudante a adaptar-se a novas situações. (TECFA, 1990)

Tem sido reconhecido que não podemos entender a tecnologia sem termos um entendimento funcional de como ela é usada. Além disso, "*esse entendimento deve incorporar uma visão holística da rêde de tecnologias e atividades na qual ela se ajusta, em vez de tratar os equipamentos tecnológicos isoladamente*" (Winograd, 1988, p. 6). Nosso objetivo neste capítulo é fornecer um referencial teórico para nossa pesquisa como um todo. Isso envolve considerar os aspectos teóricos que são a base das ferramentas sendo propostas (a idéia de considerar as entidades do paradigma de programação subjacente às linguagens no *design* das ferramentas), bem como o contexto teórico em que elas serão utilizadas (o que entendemos por "ambiente de aprendizado baseado em computador").

A seguir, apresentamos nossa interpretação de alguns paradigmas de linguagens de programação, como "meios" onde problemas são representados para serem "resolvidos". Argumentamos que a efetividade do "aprender a programar", no sentido de adaptação do sujeito ao novo meio de representar a solução de problemas, depende da adequação de ferramentas e metodologias aos paradigmas embuti-

dos na linguagem (Seção 3.1). Apresentamos, também, as principais teorias que contribuem à definição do "ambiente" onde essas ferramentas são inseridas metodologicamente (Seção 3.2). Nas Seções 3.3 e 3.4, apresentamos as opções de *design* para as ferramentas e os fundamentos teóricos nos quais baseamos as representações gráficas propostas.

3.1 Os Paradigmas das Linguagens de Programação como "Meios"

3.1.1 As Origens dos Paradigmas de Programação

Várias definições podem ser encontradas na literatura, para "paradigma de programação". Papert, referindo-se a linguagens suportadas por novas arquiteturas, define paradigma de programação como um "quadro estruturador*1" que é subjacente à atividade de programar e coloca que a escolha do paradigma de programação pode mudar notavelmente "*a maneira como o programador pensa sobre a tarefa de programar*" (Papert, 1991, p.8). De acordo com nossa visão esse quadro estruturador já existe no nível de abstração que linguagens como Lisp e Prolog, por exemplo, propuseram sobre a arquitetura von Neumann.

Recuperando o contexto histórico da evolução das linguagens de programação, podemos dizer que elas representam graus variados de abstração da arquitetura subjacente, chamada von Neumann.

Conhecer as origens dos paradigmas de programação envolve conhecer um pouco da história da evolução das linguagens de programação. Os computadores disponíveis no final da década de 40 e início da década de 50, além dos problemas decorrentes da tecnologia da época, eram difíceis de serem programados pela ausência de software. Na falta de linguagens de programação de alto nível, ou

*1 "framework" no original.

mesmo linguagens de montagem, a programação era feita em código de máquina (por exemplo, uma instrução para "somar", deveria ser especificada por um código em vez do seu uso textual). Essa maneira de programar tornava os programas ilegíveis, além de ser bastante complicado o seu processo de depuração. Do ponto de vista do programador, essa foi uma motivação importante para a criação das linguagens de montagem e seus montadores. Além disso, as aplicações numéricas da época requeriam o uso de certas facilidades que não estavam incluidas no *hardware* das máquinas de então, (números reais, acesso a elementos de um conjunto por seu índice,por exemplo) surgindo daí a criação de linguagens de mais alto nível que incluissem tais recursos. O paradigma "procedural", é o que mais se aproxima do uso da arquitetura von Neumann como modelo para representação da solução de um problema a ser resolvido pela máquina. Segundo o paradigma procedural, programar o computador significa "dar-lhe ordens" que são executadas sequencialmente. Em tal paradigma, "representar" a solução de um problema para ser resolvido pelo computador envolve escrever uma série de ações (procedimentos) que, se executadas sequencialmente, levam à solução. Ou seja, o programa representa a prescrição da solução para o problema. As linguagens de programação procedurais (ou imperativas) como por exemplo Fortran, Pascal, C, Módula-2, formam a maior classe das linguagens existentes até então. Várias razões poderiam ser usadas para explicar o crescimento da classe de linguagens procedurais; devemos apontar o papel histórico do uso do computador em aplicações numéricas e o uso do computador dentro do próprio domínio da ciência da computação. O desenvolvimento de tais linguagens tem sido feito por especialistas em computação, para uso de especialistas em computação, dentro do seu próprio domínio, onde questões de eficiência e desempenho são fundamentais. A classe das linguagens procedurais continuará a evoluir, enquanto a arquitetura subjacente dos computadores for a arquitetura von Neumann.

O paradigma funcional de programação surgiu com o desenvolvimento da linguagem Lisp (de List Processing) por John McCarthy em 1958. Lisp foi projetada, portanto, numa época em que só existia processamento numérico, para atender aos interesses dos grupos de Inteligência Artificial no processamento de dados simbólicos. Apesar do estilo imperativo de programar ser bem aceito entre programadores, provavelmente em função do tipo de aplicações realizadas na época, o vínculo da linguagem com a arquitetura von Neumann, excetuando-se as razões técnicas de eficiência, do ponto de vista de metodologia de desenvolvimento de programas, apresenta-se como uma restrição desnecessária (Sebesta, 1988). Surgiu como uma nova base para projeto de linguagens, o uso de funções matemáticas e composição de funções, introduzindo um novo modelo para representação do problema a ser resolvido pela máquina. Segundo o paradigma funcional, programar o computador significa definir funções, aplicar funções e conhecer o comportamento de funções na máquina; os mecanismos de controle, no programa, passam de iterativos a recursivos. Assim, representar a solução de um problema para ser resolvido num ambiente funcional passa a necessitar de uma abordagem completamente diferente dos métodos usados em linguagens imperativas.

Programação orientada a objetos é um novo paradigma, que surgiu em paralelo à criação de uma linguagem de programação, chamada Smalltalk, proposta por Alan Kay em 1972. A idéia básica do paradigma orientado a objetos é imaginar que programas simulam o mundo real: um mundo povoado de objetos. Dessa maneira, linguagens baseadas nos conceitos de simulação do mundo real devem incluir um modelo de objetos que possam enviar e receber mensagens e reagir a mensagens recebidas. Esse conceito é baseado na idéia de que no mundo real frequentemente usamos objetos sem precisarmos conhecer como eles realmente funcionam. Assim, programação orientada a objetos fornece um ambiente onde múltiplos objetos podem coexistir e trocar mensagens entre si.

Programação em lógica é uma teoria que representa um modelo abstrato de computação sem relação direta com o modelo von Neumann de máquina. Prolog (de Programming in Logic), surgiu no início dos anos 70, dos esforços de Robert Kowalski, Maarten van Emden e Alain Colmerauer e é a linguagem de programação desenvolvida em máquina sequencial, que mais se aproxima do modelo de computação de programação em lógica. O enfoque do paradigma da programação em lógica para se representar um problema a ser resolvido no computador, consiste em expressar o problema na forma de lógica simbólica. Um processo de inferência é usado pela máquina para produzir resultados. Segundo o modelo de programar proposto por Prolog, o significado de um "programa" não é mais dado por uma sucessão de operações elementares que o computador supostamente realiza, mas por uma base de conhecimento a respeito de certo domínio e por perguntas feitas a essa base de conhecimento, independentemente. Dessa maneira, Prolog pode ser visto como um formalismo para representar conhecimento a respeito do problema que se quer resolver, de forma declarativa (descritiva). Existe, por trás do programa uma máquina de inferência, em princípio "escondida" do programador, responsável por "encontrar soluções" para o problema descrito.

Programar nos diferentes paradigmas significa, portanto, representar, segundo modelos diferentes, a solução do problema a ser resolvido pela máquina. Cada linguagem que suporta determinado paradigma representa, portanto, um "meio" onde o problema é "resolvido". Enquanto "meio de expressão" e de "comunicação" com a máquina, a linguagem e, indiretamente o seu paradigma, "moldam" a representação do problema a ser resolvido. Assim, na atividade de programar, mudar de paradigma significa muito mais do que conhecer as entidades sintáticas e semânticas da nova linguagem, o processo de pensamento também deve ser mudado, ajustando-se ao novo meio de representação do problema. O psicolinguista

Benjamin Lee Whorf (citado em Johanson (1988)) em seu trabalho com linguagens naturais sugere que existem estruturas inerentes à linguagem que falamos das quais não nos damos conta, mas que têm uma profunda influência em nossos pensamentos. Ele argumenta que os padrões disponíveis nas estruturas das linguagens que usamos são mais importantes que as palavras em si. O trabalho de Whorf, embora não se refira a linguagens artificiais, concorda com o conceito de Papert sobre paradigma de programação como uma estrutura subjacente à linguagem de programação que influencia a maneira como o sujeito encara a tarefa de programar e faz-nos questionar a respeito do tipo de pensamentos que um determinado paradigma desperta, se considerarmos as implicações cognitivas do uso de linguagens de programação em resolução de problemas.

Para situar os diferentes paradigmas como "meios" diferentes onde problemas são representados, procuraremos exemplificá-los através da representação de um mesmo problema, nesses diferentes meios. Apesar de cada paradigma definir uma classe de problemas à qual as linguagens melhor se adequam, as linguagens de programação de alto nível são consideradas "de propósito geral". Assim sendo, correndo o risco de sermos "parciais" a esse respeito, escolhemos como o "problema" a ser exemplificado nos vários paradigmas, um clássico na literatura de computação: o problema de calcular o fatorial de determinado número.

3.1.2 Os "Meios" Gerados pelos Paradigmas de Programação

Os paradigmas das linguagens de programação, interpretados como "meios" onde problemas são resolvidos, apresentam diferentes significados para "programa" e para a "máquina que executa o programa". Consequentemente, tem-se diferentes maneiras de se pensar e representar problemas. A seguir, procuraremos ilustrar essas diferenças.

O meio procedural

O meio procedural pretende "imitar" a máquina von Neumann; o computador é entendido como uma máquina que obedece ordens e o programa como uma prescrição de solução para o problema. O conceito central para representação da solução do problema é o conceito de "variável" como uma abstração para uma posição de memória na máquina, para a qual se pode atribuir um valor. O fluxo de controle da execução pela máquina é ditado por sequenciação e por comandos de repetição.

Assim, para representar a solução do problema do cálculo do fatorial de um número, o usuário precisa prescrever a solução do problema segundo modificações que dinamicamente alteram os conteúdos das variáveis e conduzem ao resultado do cálculo.

Para ilustrar o paradigma procedural consideraremos a "resolução" de fatorial escrita em Pascal na sua forma iterativa pois historicamente a iteração ilustra melhor o paradigma da programação procedural. A seguir nos reportaremos a esse paradigma para comparação com os demais.

```
Procedure Fatorial(n:integer,var fat:integer);
var i:integer;
begin
        fat:=1;
        for i:=1 to n do fat:=fat*i
end;
```

Esse tipo de representação da solução do cálculo do fatorial envolve uma abordagem ao problema com a máquina em mente. Ou seja, o cálculo do fatorial "acontece" distribuído por ações da máquina que manipulam as variáveis *n* (o objeto do cálculo do

fatorial), *i* (variável que auxilia no controle do número de iterações da ação de multiplicar) e *fat* (variável que acumula o resultado da multiplicação). Assim, o cálculo do fatorial de *n* é representado pela ação repetida do comando de atribuição *fat:=fat*i*, cuja semântica explica como é feita a alteração na variável *fat*.

A nível do usuário, a representação da solução do problema nesse meio envolve, além do conhecimento da sintaxe da linguagem, conhecimento da semântica da linguagem ao nível de comando (que modificações nas variáveis um determinado comando provoca) e ao nível de bloco (qual é o "significado" de um conjunto de comandos em relação ao problema em questão).

Portanto, no meio procedural a máquina é tratada como um dispositivo que "obedece" ordens. Assim sendo, a maioria dos comandos é usada para descrever para a máquina "como" resolver o problema. No nosso exemplo, resolver o "problema" Fatorial envolve descrever para a máquina, em detalhes segundo as restrições da sua linguagem, todos os passos necessários para "cálculo" do fatorial de um número.

O meio funcional

Uma função matemática é um mapeamento de membros de um conjunto (domínio) em outro conjunto (contra-domínio). Ou seja, definir uma função envolve especificar, explícita ou implicitamente seu domínio, seu contra-domínio e o mapeamento que "leva" elementos do domínio a elementos do contra-domínio.

A principal característica do meio funcional é "imitar" o comportamento de funções. Assim, no meio funcional, o computador atua como uma máquina que avalia funções e o programa consiste da definição e composição de funções. Nas linguagens procedurais, uma expressão é avaliada e seu resultado, em geral, é armazenado em uma célula de memória representada por uma variável. Uma linguagem de programação puramente funcional não usa variáveis ou comando

de atribuição. A ordem de avaliação de suas expressões de mapeamento é controlada por recursão e expressões condicionais, enquanto que no meio procedural esse controle é feito por sequenciação e iteração. Dessa maneira, o usuário é "poupado" de preocupações com entidades que o levariam a uma metodologia de mais "baixo nível" (próxima da arquitetura da máquina), para desenvolvimento de programas.

A solução do problema do cálculo do fatorial de um número, por exemplo, é representado no "meio" funcional, por uma função recursiva nos moldes de sua definição matemática, como ilustramos a seguir, em Lisp:

```
(defun fatorial (n)
        (cond
                ((zerop n) 1)
                (T (* n (fatorial (- n 1))))
        )
)
```

A idéia básica não é mais a da repetição de sequência de ações, mas de aninhamento de ativações diferentes da mesma função, cada umas delas "retornando" um valor, que é usado pelas demais, em cadeia, como ilustrado na sequência de expressões abaixo:

```
(fatorial 3)=
(3*(fatorial 2))=
(3* (2* (fatorial 1)))=
(3* (2* (1* (fatorial 0))))=
(3* (2* (1* 1)))=
(3* (2* 1))=
```

(3* 2)=

(6)

Portanto, escrever um "programa", segundo esse paradigma, envolve definir funções e aplicação de funções para o problema em questão e o processo de "execução" pela máquina consiste em avaliar as aplicações das funções envolvidas. O computador assume o papel de uma "máquima funcional".

A nível do usuário, o meio para representação da solução de problemas não é mais baseado nos conceitos de variável, atribuição de valor e iteração; mas, é constituído pelos conceitos de função, comandos condicionais e recursão.

O meio orientado a objetos

O meio orientado a objetos pretende imitar o "mundo real", através do papel do computador como uma máquina que simula a interação entre objetos. Nesse mundo, o programa é constituído dos objetos, mensagens e métodos (possíveis mensagens para as quais um objeto pode responder).

No "meio" orientado a objetos, as unidades de programa são objetos. Por exemplo, desde uma constante numérica até um sistema para manipular arquivos, são todos objetos. Mensagens possibilitam a comunicação entre os objetos e é através delas que uma operação de um objeto é requisitada. Um método especifica a reação de um objeto a uma determinada mensagem recebida correspondente aquele método. Sebesta exemplifica esses conceitos mostrando o significado da expressão *21 + 2* , no paradigma orientado a objetos:

> " o objeto receptor é o número 21, para o qual é enviada a mensagem "+ 2". Essa mensagem passa o objeto 2 para o método "+" do objeto 21. O código desse método usa o objeto 2 para construir um novo objeto, o 23" (Sebesta, 1988, p. 465).

No paradigma orientado a objetos, a representação do cálculo do fatorial de um número corresponde a um "método" para objetos "inteiros". Esse método, pode ser invocado por uma "mensagem" do tipo *5 fatorial*, construindo o "objeto" *120*.

Para ilustrar, mostramos a seguir, a "resolução" de fatorial escrita em Smalltalk (extraída de Sebesta, 1988, p. 473):

```
fatorial
  self = 0
        ifTrue: [^ 1].
  self < 0
        ifTrue: [self error 'Fatorial não definido']
        ifFalse: [^ self * (self - 1) fatorial]
```

Apesar da "aparência" convencional do código, sua semântica é bem diferente da semântica das linguagens imperativas. As unidades de programa são os objetos, que são uma abstração de dados e possuem a habilidade de herdar características de objetos de classes ancestrais e de se comunicar com outros objetos enviando e recebendo mensagens. O foco da atenção no meio orientado a objetos está colocado, portanto nos objetos e na capacidade de processamento deles, enquanto que na abordagem de programação tradicional o enfoque está nos processos e na implementação deles.

O meio da lógica

Uma das características principais das linguagens para programação em lógica é sua semântica declarativa. A idéia por trás dessa semântica é que existe uma maneira de determinar o significado de cada declaração que não depende de como a declaração

seria usada para resolver o problema. Isto é, o significado de uma dada proposição num programa é determinado a partir da própria proposição, enquanto que, em linguagens imperativas a semântica de um comando requer informações que não estão contidas ou não estão explicitadas no comando. Por exemplo, o conhecimento das regras de escopo para as variáveis é necessário para se entender o significado de determinado comando em um programa escrito numa linguagem procedural.

Assim, seriam declarações válidas no problema do cálculo do fatorial:

"o fatorial de 0 é 1".

"o fatorial de 1 é 1".

"o fatorial de 2 é 2".

etc.

No meio gerado pela programação em lógica, um programa não contém instruções explícitas à máquina. Em vez disso, ele estabelece "fatos" e "regras" sobre a área do problema como um conjunto de axiomas lógicos, que são "interpretados" como "programas". Ilustrando, um "programa" em Prolog, para cálculo do fatorial de um número inteiro pode ser definido por:

```
fatorial(0,1).
fatorial(N,Fat):- N>0,
                  N1 is N-1,
                  fatorial(N1,Fat1),
                  Fat is Fat1 * N
```

No exemplo dado, as duas proposições podem ser lidas como:

"O fatorial de 0 é 1" (e isso é um fato)

"O fatorial de N é Fat se o fatorial de N-1 é Fat1 e Fat é Fat1 * N" (e isso é uma regra)

Dessa maneira, os programas não estabelecem exatamente "como" um resultado deve ser computado, mas, descrevem fatos e regras que podem levar a máquina à dedução do cálculo do fatorial. O computador assume, portanto, o papel de uma "máquina de inferência", buscando uma prova construtiva para uma meta (pergunta) colocada pelo usuário. A nível do usuário, o computador pode ser visto como uma "lógica" que ele tem acesso para verificação da consistência de suas declarações.

Na seção seguinte, discutiremos algumas implicações da inobservância dos paradigmas subjacentes às linguagens de programação no contexto metodológico do desenvolvimento de programas, principalmente entre novatos.

3.1.3 Discussão

A atividade de "programação de computadores" pode ser vista como uma atividade de "resolução de problemas", onde as linguagens representam, através de seus diferentes paradigmas, os "meios" onde os problemas devem ser resolvidos. Assim, resolver um problema nos paradigmas citados envolve "moldar" a solução do problema segundo as "entidades" representativas de cada paradigma, simplificadamente como:

- "comandos" que são executados passo a passo pela máquina virtual, se o paradigma for o procedural.

- "funções" , que são aplicadas a certos argumentos e "retornam" valores se o paradigma for o funcional.

- "objetos" que se comunicam se o paradigma for orientado a objetos.

- "proposições" assumidas verdadeiras sobre determinado domínio de conhecimento, se o paradigma for o da programação em lógica.

Se para o profissional do domínio da computação essas mudanças de um meio para outro acontecem de maneira natural, o mesmo não pode ser dito a respeito do novato. A interpretação do novato a respeito do *meio* subjacente à linguagem, que ele usa para representar a solução de um problema, tem implicações no seu processo de resolução do problema e de aprendizado da linguagem, conforme será mostrado no decorrer deste trabalho (capítulo 8).

Em geral, o novato num determinado paradigma de programação com experiência anterior no paradigma procedural, usa a sintaxe da nova linguagem sobre uma representação baseada nos elementos do paradigma antigo, no processo de apropriar-se da nova linguagem. A função Lisp[*2], apresentada a seguir, ilustra a dificuldade de sujeitos na passagem de um meio para outro (procedural para funcional, no caso).

```
(defun emaior (l)
(do (maior 0))
 (cond
        ((null l) maior)
        ((>= (car l) (setq maior (car l))
 )
 emaior (cdr l)
)
```

A função definida pretende retornar o maior elemento de uma lista de átomos numéricos. A solução proposta pelo sujeito parece ter sido a de um mecanismo iterativo (*do* na linha 2), mas ao mesmo tempo apresenta uma chamada recursiva para a função (linha 7). A idéia parece ter sido usar uma "variável" *(maior)* para

*2 extraída de teste feito por estudantes em curso regular na Universidade.

"guardar" o maior elemento da sequência, nos moldes que seria feito numa linguagem procedural (o que deveria ser feito na linha 5). No processo de escrever o programa, o sujeito usa elementos do meio procedural com a sintaxe da nova linguagem, misturando estruturas dos dois paradigmas.

Algumas linguagens de programação combinam paradigmas diferentes, como é o caso de Logo, ou incorporam extensões à sua definição original, como é o caso de Object Logo, Lisp, C, por exemplo (que têm extensões para a orientação a objetos). Apesar de Logo ser considerada "procedural" (Mendelsohn, 1990; Johanson, 1988), existe nessa linguagem características do paradigma funcional (através das operações) e do paradigma orientado a objetos (através dos *sprites*). Isso faz de Logo uma ferramenta extremamente flexível e amistosa ao usuário que deseja "sentir o sabor" dos vários paradigmas. Por outro lado, essa multiplicidade de paradigmas, em geral, leva o novato a dificuldades do tipo a usar mecanismos procedurais para "pensar" em funções recursivas (Baranauskas, 1991a). A mudança de um "meio" para outro não parece ser imediata. Em geral o novato não tem consciência dessa multiplicidade de meios para representação de soluções de problemas e tenta usar sempre o meio procedural, independentemente da natureza do problema.

Logo e Prolog têm sido referências obrigatórias quando se fala de linguagens de programação em contextos educacionais. Entretanto, em ambos os casos, a maioria de seus usuários têm grande dificuldade ao ultrapassar o que podemos chamar de suas "portas de entrada": as fronteiras do procedural, incorporadas pela tartaruga, no caso de Logo (Rocha, 1991) e a criação de bases de conhecimento em assuntos de conteúdo puramente declarativo, no caso de Prolog, conforme será mostrado no decorrer deste trabalho (capítulo 8).

A apropriação dessas linguagens como meios de expressão pode ser facilitada na medida em que as metodologias e ambientes de desenvolvimento de programas refletirem e explicitarem seus respectivos paradigmas. A grande dificul-

dade de usuários novatos no trabalho com o paradigma funcional (Logo listas, por exemplo) parece ser relativo ao uso do "meio" procedural (de Logo geométrico) para resolver problemas de natureza funcional (Baranauskas, 1991a; Rocha, 1991). Efeito semelhante acontece aos usuários acostumados com linguagens procedurais quando são introduzidos a linguagens pertencentes a outros paradigmas como no caso de Prolog, por exemplo.

Nos paradigmas citados neste texto, as linguagens de programação requerem em maior ou menor grau, que o usuário adote uma abordagem ao problem com um tipo de máquina (procedural, funcional, etc.) em mente. A aparent abstração de aspectos de controle possibilitada no paradigma da programação ca lógica faz com que usuários iniciantes em Prolog tornem-se envolvidos com a especificação de objetos e seus relacionamentos, de forma que a ênfase é colocada na especificação e na análise lógica do conhecimento envolvido no problema.

Por outro lado, usuários sofisticados, alfabetizados em linguagens procedurais, vêem Prolog apenas segundo sua semântica operacional e muito esforço é requerido deles no sentido de "abstrair" a máquina para representar um determinado problema segundo as restrições do próprio problema. Usuários Pascal, por exemplo, tendem a representar problemas em Prolog com a máquina procedural em mente. Van Someren observa que muitos estudantes parecem ter um algoritmo para converter um programa baseado em Pascal para a forma de Prolog (Van Someren, 1988). Os resultados de van Someren reforçam nossa observação da necessidade de tornar explícitos para o novato os elementos do novo paradigma.

Se por um lado "esconder" do usuário o "raciocínio" que Prolog usa para responder-lhe perguntas (máquina de inferência) traz-lhe uma simplificação que o coloca diretamente envolvido com aspectos relativos ao domínio de conhecimento do problema sendo resolvido, por outro lado, essa prática tem sido uma das maiores responsáveis por levar novatos a interpretações errôneas da linguagem

(Mendelsohn, 1990). Além disso, mostrar gradualmente o processo de inferência de Prolog torna-se necessário à medida em que esse conhecimento amplia a classe de problemas que o usuário consegue resolver (um problema de natureza procedural como "ordenação" de elementos, por exemplo).

Prolog possui uma semântica declarativa, expressa pelas proposições que compõem a base de dados criada pelo usuário, e uma semântica operacional, relativa ao comportamento da máquina de inferência ao buscar uma prova construtiva para uma determinada meta (pergunta), a partir dessa base de dados. A apropriação do paradigma da programação em lógica, para representação da solução de problemas, ao nosso ver, depende da combinação desses dois fatores: o declarativo e o operacional. Levar o novato a ultrapassar a "porta de entrada" de Prolog envolve, a nível de metodologia, definir ambientes e ferramentas que possibilitem ao usuário a construção conceitual do meio gerado pelo paradigma da linguagem. Essa é a abordagem que norteia nossa proposta de ferramentas para o ambiente Prolog.

3.2 O Ambiente de Aprendizado

No contexto onde as ferramentas propostas são inseridas, a aprendizagem desenvolve-se em uma situação social, usando as ferramentas e a *mídia* representacional que a cultura fornece (computador, linguagem de programação e ferramentas), para suportar, estender e reorganizar o conhecimento. Tal ambiente baseia-se no modelo de ambiente de aprendizagem baseado em computador apresentado por Valente*3 (1991) e discutido a seguir. A figura 3.1, ilustra as diferentes nuances do quadro teórico implicado no modelo em questão.

---

*3 para explicar o "ambiente Logo de aprendizagem"

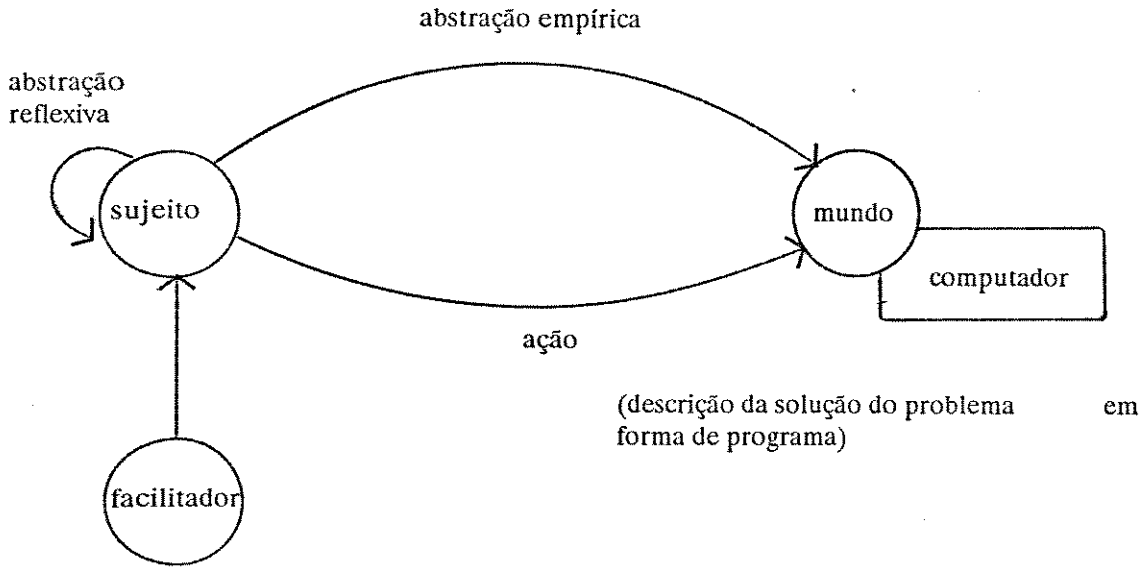


Figura 3.1 Modelo de Ambiente de Aprendizado Baseado em Computador (Valente, 1991)

O quadro teórico adotado para explicar esse modelo considera a teoria construtivista de Piaget, a teoria construcionista proposta por Papert e o aspecto social colocado por Vygotsky. A palavra *construcionismo*, conforme colocado por Papert, representa "*a outra face da moeda*" em relação à palavra *construtivismo* (Papert, 1987). O construtivismo de Piaget é uma teoria psicológica compartilhada por vários teóricos entre os quais Vygotsky, Bruner e Papert, segundo a qual os indivíduos constroem o mundo e a si próprios ("eu"), continuamente através da interação e experiência com esse mundo. A construção do eu e do mundo se ajudam e se influenciam, sendo, portanto, "recriados", "reinventados" a todo instante. Dessa maneira, o conhecimento não é transmitido, ele é *construído*. Cada indivíduo deve reconstruir o conhecimento, embora não necessariamente sozinho. Pode haver a ajuda de outras pessoas e suporte de um ambiente material, de uma cultura e sociedade.

No construtivismo de Piaget o conhecimento é construído internamente a partir das ações do sujeito sobre o mundo, conforme descrito a seguir:

> "a origem do conhecimento não está somente no objeto, nem no sujeito, mas numa interação indissociável entre os dois, de tal modo que aquilo que é dado fisicamente é integrado numa estrutura lógico-matemática implicando a coordenações das ações do sujeito".(Piaget 1977, p. 71)

A contribuição de Vygotsky a esse esquema traz o componente social, através da figura de um "experto"*1 intermediando as ações do sujeito no "mundo". Essa intermediação e essa interação de sujeitos com diferentes expertises é explicada pelo conceito de "Zona de Desenvolvimento Proximal" (ZPD). Newman *et al* argumentam que mudança cognitiva é um processo tanto social quanto individual e explicam o conceito de "Zona de Desenvolvimento Proximal" proposto por Vygotsky, da seguinte maneira:

> "Quando pessoas com diferentes metas, regras e recursos interagem, as diferenças na interpretação fornecem ocasiões para construção de novo conhecimento. As mudanças tomam lugar em interações mediadas socialmente que, seguindo Vygotsky (1978) chamamos "Zona de Desenvolvimento Proximal" (ZPD)" (Newman, 1989 p. 2).

A figura do "facilitador" na figura 3.1 é interpretada através do componente social proposto por Vygotsky, como um mediador das ações do sujeito no ambiente computacional. Nesse quadro, a mudança cognitiva é vista como um processo de construção interativa. Essa interação pode ser interpretada, com base em Newman (1989) da seguinte maneira: o facilitador e o sujeito (novato) engajam-se em um dis-

*1 o "facilitador" da figura 3.1

curso indeterminado no ambiente de computador. Mantendo conversação eles alcançam um patamar comum de compreensão e entendimento (a ZPD). A ZPD media entre o pensamento das duas pessoas. É uma atividade compartilhada, embora isso não signifique necessariamente um entendimento completamente compartilhado do significado da atividade, ou de cada um, conforme descrição de Cazden:

> "Dentro de uma ZPD, objetos não tem uma análise única. Mas, essas diferenças não precisam causar "problema" para o facilitador ou para o novato, ou para a interação social; os participantes podem agir como se seu entendimento fosse o mesmo. Essa "instabilidade"*1 é justamente o que é necessário para permitir que a mudança aconteça, quando pessoas com análises diferentes interagem" (Courtney Cazden, citado em Newman, 1989, p. xii).

Enquanto Vygotsky enfatiza o papel da linguagem e da interação do aprendiz com um adulto informado, através da linguagem, Papert enfatiza a relação entre o aprendiz e os objetos interativos que ele constrói.

O construcionismo de Papert propõe que a melhor maneira de conseguir a "construção do conhecimento", que é um processo interno (acontece "dentro" da cabeça do sujeito), se dá pela construção de algo "tangível" e também pessoalmente significativo ao sujeito (Papert, 1987). O computador entra, então, como uma ferramenta no mundo do sujeito, que possibilita a ele "modelar" o objeto de interesse e o leva a uma "conversa" com esse objeto.

Nesse modelo, o computador não é considerado apenas em seu aspecto "amplificador" do conhecimento, mas como uma "ferramenta cognitiva", conforme será discutido posteriormente. Através do computador, o sujeito participa de um ciclo de ações realimentado por um *feedback*. O sujeito programa (modela o

*1 "looseness" no original.

objeto) e com base no *feedback* gerado reflete e "depura" seu modelo. A presença do computador no "mundo" do sujeito, portanto, altera a dinâmica das ações do sujeito sobre o objeto.

3.2.1 O Papel das Ferramentas nesse Ambiente

Num panorama mais geral devemos endereçar a questão mais ampla de como as tecnologias baseadas em computador são apropriadas pela sociedade de modo geral e pelo indivíduo em particular.

Computadores são comumente vistos como responsáveis em mudar como efetivamente fazemos as tarefas habituais, "amplificando" ou estendendo nossas capacidades, enquanto que as tarefas permanecem basicamente as mesmas. Pea coloca a metáfora da reorganização em oposição à metáfora da ampliação, defendendo que o principal papel do computador é, ao contrário, o de "*mudar as tarefas que fazemos, pela reorganização de nosso funcionamento mental, não apenas pela sua amplificação*", (Pea, 1985, p.168).

Adotando a perspectiva de Bruner do conhecimento como um produto da relação entre as estruturas mentais e ferramentas do intelecto fornecidas pela cultura, Pea denomina essas ferramentas *tecnologias cognitivas*, e as define como a seguir:

> "Uma tecnologia cognitiva é fornecida por qualquer mídia que ajuda a transcender as limitações da mente, tais como memória, em atividades do pensamento, aprendizado e resolução de problemas" (Pea, 1985, p. 168).

Assim, são exemplos de tecnologias cognitivas a linguagem escrita, sistemas de notação matemática, computadores. Ferramentas cognitivas como a linguagem escrita

são colocadas por Bruner (citado em Pea, 1985, p.169) como "*amplificadores culturais*" que podem levar a mudanças qualitativas nas formas do pensamento.

Situamos as ferramentas computacionais que compõem o ambiente sendo proposto neste trabalho, segundo a concepção de Pea de tecnologia cognitiva baseada em computador, enquanto objetivam possibilitar que atividades de resolução de problemas sejam reorganizadas sob a mediação de um paradigma de programação e não apenas "amplificadas".

Essa proposta de criar e simultaneamente estudar mudanças nos processos e consequências no aprendizado com ferramentas cognitivas representa um esforço no sentido de se pensar maneiras pelas quais o computador pode ajudar a servir como tecnologia cognitiva a reorganizar tanto a vida mental individual dos sujeitos envolvidos, quanto o próprio ambiente educacional.

Este trabalho pressupõe um quadro teórico baseado numa síntese dos teóricos mencionados e suas visões idealizadas do processo de aprender. Nossa ênfase está colocada no diálogo que o novato estabelece no ambiente Prolog acrescido de ferramentas que o ajudem a apropriar-se do paradigma de programação subjacente à linguagem em questão.

3.3 As Opções de Design para o Ambiente Proposto

O modelo de *design* como decisão racional, sustentado por Simon*[1] e outros adeptos da tradição racionalista é caracterizado por uma busca heurística entre alternativas em um espaço de possíveis cursos de ação. Segundo esse modelo, o "*designer*" transforma um determinado estado de coisas, um problema, em um estado preferido, uma solução, aplicando regras formais a partir de informações parciais.

---

*[1] apresentado no clássico da literatura em IA "The Sciences of the Artificial", (Simon, Herbert, MIT Press, Cambridge, MA, 1971.

Os trabalhos em Inteligência Artificial e Educação*[2] já foram bastante influenciados pelo paradigma do pensamento racional em IA, que mostrou sua utilidade no estudo e simulação de processos de resolução de problemas em áreas bem definidas. Entretanto, não se pode estender a sua aplicabilidade a problemas que lidam com situações que não são bem definidas. O Manifesto de Genebra sobre ambientes inteligentes para aprendizagem discute a inadequabilidade da visão racionalista do raciocínio humano, para o *design* de ambientes de aprendizagem, justificando: "*Planos são um conceito útil para entendermos as ações humanas, mas eles raramente são construídos para gerar essas ações*" (TECFA, 1990, p. 2).

Um novo modelo de *design* tem sido apresentado (Schon, 1990; Winograd, 1988; Park, 1990), em contraposição ao modelo racionalista, onde as estruturas*[3] de *design*, ao contrário de serem dadas com o problema, são feitas e refeitas no curso do *design*. Segundo esse modelo, em vez de falar sobre "decisões" ou "problemas" devemos falar de "situações de irresolução". O processo que leva da irresolução à resolução tem como característica básica uma espécie de "conversação" guiada por questões relativas a como as ações devem ser dirigidas. "Resolução" refere-se, portanto, à exploração de uma situação, não à aplicação de regras habituais. Problemas bem definidos e técnicas de resolução de problemas tendem a ocorrer, nesse novo modelo de *design*, somente nas suas últimas fases.

É dentro dessa nova abordagem em que *design* significa ao mesmo tempo processo e produto, que situamos o *design* das ferramentas computacionais que constituem o ambiente proposto. As ferramentas foram propostas a partir de um pano de fundo de suposições de como elas serão usadas e como suas respostas serão interpretadas. A questão não é exatamente saber se são boas ou ruins, mas saber como o entendimento e uso das ferramentas determina o que o novato faz no

*[2] denominação usada na literatura de forma abreviada por AI&ED (TECFA, 1990).
*[3] terminologia usada por Schon para designar representações do problema que, juntamente com procedimentos governados por regras, guiam as transformações nessas representações (Schon,1990).

processo de apropriação da linguagem de programação para representação da solução de problemas.

No *design* das ferramentas dois aspectos fundamentais orientaram nossas escolhas que, acreditamos, têm consequências significativas para o processo no qual o novato estará engajado: o uso de um formalismo visual e a multiplicidade de representações para o conhecimento envolvido no programa Prolog.

Segundo Park, a operação de qualquer equipamento é aprendida mais facilmente se o usuário tiver um bom modelo conceitual do sistema. Um dos princípios básicos para o design de equipamentos, é o de se "*usar tecnologia para tornar visível o que, de outra maneira seria invisível, aumentando assim o feedback e a habilidade para manter o controle*" (Park, 1990,p. 192). Esse princípio é usado, por exemplo, nos instrumentos dos automóveis e aviões para tornar visíveis partes dos equipamentos sobre os quais o usuário não tem acesso físico. Também é o mesmo princípio que nos leva a verificar que uma certa unidade de disco foi acionada, num sistema de computador, por exemplo. Esse tipo de informação é importante principalmente em situações de erro, e servem de "pistas" para identificarmos o que está acontecendo a nível físico, com o equipamento.

A metáfora da visibilidade pode ser aplicada à linguagem de programação, enquanto máquina virtual, para fornecer ao usuário acesso ao conhecimento de seu mecanismo. Algumas tentativas já foram feitas no sentido de tornar visível ao novato a "máquina"*[4] da linguagem que eles estão aprendendo (du Boulay, 1981). Visibilidade nos trabalhos citados refere-se a métodos para se ver partes selecionadas e processos dessa máquina em ação. O conhecimento sobre o programa é mostrado através de sua execução na máquina virtual. Tem sido reconhecida na literatura (van Someren, 1990; Pain, 1987) a importância de ferramentas que

*[4] "notional machine", no original, refere-se ao modelo idealizado de computador, implicado pelas construções da linguagem de programação.

forneçam aos estudantes acesso ao processo de execução de programas escritos na linguagem que estão aprendendo. Esse tipo de ferramenta é considerado essencial, especialmente no caso de Prolog, em função da complexidade dos processos de unificação e retrocesso *(backtracking)*, subjacentes. Alguns sistemas, já citados anteriormente (capítulo 2) foram desenvolvidos com esse objetivo, entre outros o de Rajan (1986) e o de Eisenstadt e Brayshaw (1988).

Ao mesmo tempo em que prover o estudante com um modelo de execução detalhado de Prolog é considerado essencial (van Someren, 1990), o entendimento do programa Prolog não acontece apenas através da análise exaustiva e sistemática do seu fluxo de controle na máquina. Hook *et al* mostram que usuários (novatos e *experts*) usam outros tipos de análise (denominadas meta-análise), para entender programas, que não são necessariamente baseadas na reconstrução da execução de Prolog (Hook, 1990). Um dos aspectos básicos que estamos considerando em nossa proposta de ferramentas envolve possibilitar a investigação do programa, baseada também na meta-análise do problema.

3.4 Fundamentos para as Representações Gráficas Propostas nas Ferramentas

Nesta seção apresentamos os fundamentos teóricos nos quais baseamos as representações gráficas propostas para mostrar, respectivamente, aspectos dos significados operacional e declarativo envolvidos na programação em Prolog. Algumas definições básicas, que podem ser necessárias para o entendimento dos conceitos apresentados foram incluídas no apêndice 1.

O significado operacional de um programa Prolog P, é dado pelo processo usado por um interpretador, que chamaremos "máquina de inferência de Prolog" (ou apenas "máquina de inferência", para simplificar), ao responder perguntas com base em P.

A máquina de inferência Prolog P, é uma realização do modelo de computação de programação em lógica em máquinas sequenciais. A seguir descreveremos o modelo de computação de programação em lógica.

3.4.1 O Modelo de Computação de Programação em Lógica

No modelo de computação de programação em lógica, a computação se processa através de reduções de metas. Em cada estágio, existe um *resolvente*, que é uma conjunção de metas a serem provadas. Uma meta do resolvente e uma cláusula do programa são escolhidos de modo que a cabeça da cláusula *unifica* com a meta. A computação prossegue com um novo resolvente, obtido pela substituição da meta escolhida pelo corpo da cláusula escolhida, no resolvente e então aplicando o *unificador mais geral* entre a cabeça da cláusula e a meta. A computação termina quando o resolvente é vazio.

A seguir apresentamos o algoritmo de um interpretador abstrato para programas em lógica.

Entrada: Um programa em lógica, P

Uma meta G

Saída: G*s*, se for uma instância de G deduzida de P

ou Fracasso

Algoritmo: Inicializar o resolvente para ser G, a meta de entrada.

Enquanto o resolvente não for vazio fazer:

- Escolher uma meta A do resolvente e uma cláusula (renomeada) A':-B1,B2,...Bn, n>=0, de P de modo que A e A' unificam com mgu *s* (termina se não existir tal meta e cláusula).

- Remover A e adicionar B1,B2,...Bn no resolvente.

- Aplicar *s* ao resolvente e a G.

> Se o resolvente for vazio, a saída é G, senão a saída é Fracasso. (Sterling, 1986, p. 73)

Para exemplificar, consideremos as cláusulas que definem a concatenação de duas listas, como apresentadas a seguir:

```
concatena([H|T],L,[H|T1]):-concatena(T,L,T1).
concatena([],L,L).
```

Consideremos a cláusula objetivo (pergunta): *concatena([a,b],[c,d],Lista).*

O resolvente é inicializado para ser *concatena([a,b],[c,d],Lista)* e é escolhido como a meta a reduzir.

A cláusula escolhida do programa é a primeira. O unificador entre a meta e a cabeça da regra é: {*H=a, T=[b], L=[c,d], Lista=[a|T1]*}.

O novo resolvente é: *concatena(T,L,T1)* sob o unificador, ou seja: *concatena([b],[c,d],T1).*

Na próxima iteração do *loop*, essa será a nova meta. A mesma cláusula do programa é escolhida (com variáveis renomeadas para evitar conflitos): *concatena([H1|T1],L1,[H1|T2]):-concatena(T1,L1,T2).*

O unificador da meta e da cabeça de regra é: {*H1=b, T1=[], L1=[c,d], T1=[b|T2]*}

O novo resolvente é: *concatena([],[c,d],T2).*

O fato (segunda cláusula) é escolhido (renomeadas as variáveis). O unificador é:
$\{T2=[c,d]\}$

O novo resolvente é vazio e a computação termina.

Para computar o resultado, aplica-se a parte relevante do *mgu* (unificador mais geral) obtido durante a computação:
Lista=[a|T1] da primeira unificação,
T1=[b|T2] da segunda unificação,
T2=[c,d] da terceira unificação,
ou seja, Lista=[a|[b|[c,d]]] ou Lista=[a,b,c,d] é a resposta.

3.4.2 O Modelo de Execução de Prolog

O mecanismo de execução de Prolog é obtido do interpretador abstrato apresentado, escolhendo-se a meta mais à esquerda (em vez de uma arbitrária) e substituindo-se a escolha não determinística de uma cláusula da base de dados, pela busca sequencial de uma cláusula unificável e retrocesso (*backtracking*).

Ao tentar reduzir uma meta, a primeira cláusula cuja cabeça unifica com a meta é escolhida. Se nenhuma cláusula unificável é encontrada, a computação é desviada para a última meta (meta mais recente no processo) e a próxima cláusula unificável é escolhida.

A máquina de inferência aceita como entrada um programa P e uma meta (ou pergunta) G e produz como saída "fracasso" ou "sucesso". No caso de sucesso, a saída inclui, também, uma substituição *s* para as variáveis de G.

Se a saída é "fracasso", significa que não há respostas corretas de G a P e, se a saída é "sucesso", *s* é uma resposta correta de G a P.

A máquina poderá divergir e não produzir qualquer saída, tanto no caso de haver uma resposta correta de G a P, quanto no caso contrário.

Quando requisitada, a máquina pode, ainda, produzir respostas corretas alternativas, se existir mais que uma resposta a G.

A sequência de resolventes produzida durante a computação, juntamente com as escolhas feitas, contém implicitamente uma "prova" da resposta à pergunta (meta) feita, com base em um programa. Uma representação conveniente para essa "prova" é dada pela "árvore de prova", que é definida a seguir, conforme Sterling (1986):

Uma *árvore de prova* consiste de nós e arestas que representam as metas reduzidas durante a computação. A raíz da árvore de prova para uma pergunta simples (meta única) é a própria pergunta. Os nós da árvore são metas que são reduzidas durante a computação. Existe uma aresta dirigida de um nó para cada nó correspondente a uma meta derivada da meta reduzida. A árvore de prova para uma pergunta conjuntiva (conjunção de metas) é a coleção de árvores de prova para as metas individuais da conjunção.

A seguir, apresentamos um exemplo da árvore de prova (figura 3.2) para a meta *insere(3,[1,2],[1,2,3])*, considerando o conjunto de cláusulas a seguir, que definem a inserção de um elemento numa lista ordenada:

```
insere(A,[B|C],[A,B|C]):- A=<B.
insere(A,[B|C],[B|D]):- A>B, insere(A,C,D).
insere(A,[],[A]).
```

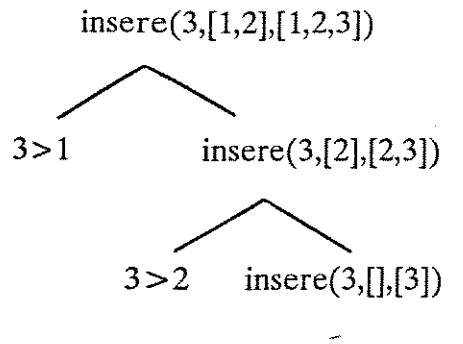


Figura 3.2 Árvore de Prova para a meta *insere(3,[1,2],[1,2,3])*

Representar a semântica operacional de um programa Prolog envolve, portanto, criar uma representação para o modelo de execução de Prolog, conforme será mostrado no capítulo 6.

O significado declarativo de um programa Prolog é herança do significado de um programa em lógica P e é dado pelo conjunto de metas sem variáveis, deduzíveis de P. Assim um programa composto somente de fatos tem como significado declarativo o próprio conjunto de fatos. Um programa composto de fatos e regras, tem como significado o conjunto de fatos presentes no programa e todos os fatos que as regras estabelecem implicitamente; isto é, os fatos que podem ser deduzidos pelas aplicações das regras.

Nossa abordagem para tratar aspectos de significado declarativo de um programa Prolog, envolve explicitar as *relações* definidas pelo programa (*o quê* define o programa), de forma independente de mecanismos de execução ("como" o programa é interpretado). Para tal, estamos usando um modelo de representação do programa, baseado na notação de rêdes semânticas.

3.4.3 Os Diagramas de Rêdes Semânticas

Rêdes semânticas têm tido significados diferentes para diferentes pesquisadores. Segundo Schubert, elas podem ser "*diagramas no papel, ou conjuntos abstratos de n-tuplas de algum tipo, ou estruturas de dados em programas ou mesmo estruturas de informação em cérebros*" (Schubert, 1976, p. 163). Simplificadamente, podemos dizer que, como a lógica, rêdes semânticas são um tipo de formalismo que pode ser usado para representar conhecimento. Rêdes semânticas têm sido usadas tanto para formulação quanto para a exposição da estrutura de informações que denotam, dentro de I.A., principalmente, em aplicações envolvendo entendimento de linguagem natural.

A relação entre lógica e rêdes semânticas tem sido abordada na literatura (Ritchie, 1983). Bons exemplos dessa relação são mostrados em Schubert (1979), em sua abordagem à compreensão de linguagem e em Deliyanni (1979), em uma proposta de rêde semântica estendida, provida com semântica precisa, regras de inferência e interpretação procedural. Ambos mostram que aspectos sintáticos da lógica de predicados podem ser representados na notação de rêdes semânticas.

Schubert *et al* (1976), em sua abordagem à compreensão de linguagem, desenvolve uma notação de rêde para representação do conhecimento proposicional, capaz de mapear certos tipos de sentenças declarativas na representação de rêdes.

Deliyanni e Kowalski (1979) definiram uma forma estendida de rêde semântica como um variante sintático da forma clausal de lógica e mostram que lógica e rêdes semânticas são formalismos compatíveis que têm a contribuir entre si.

Para ilustrar, apresentamos nas figuras 3.3 e 3.4 exemplos de uma proposição atômica nas notações definidas por Schubert e Deliyanni, respectivamente.

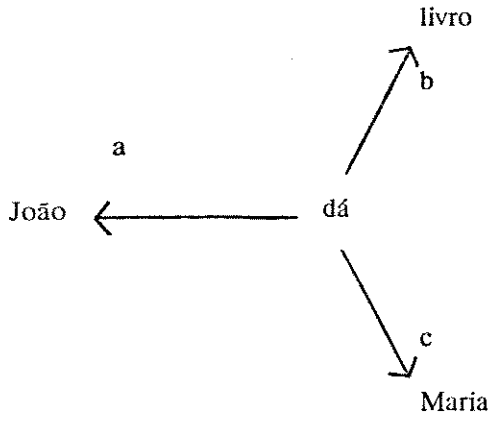


Figura 3.3 Representação da proposição: "João dá o livro a Maria" segundo a notação de Schubert (1976, p. 166)

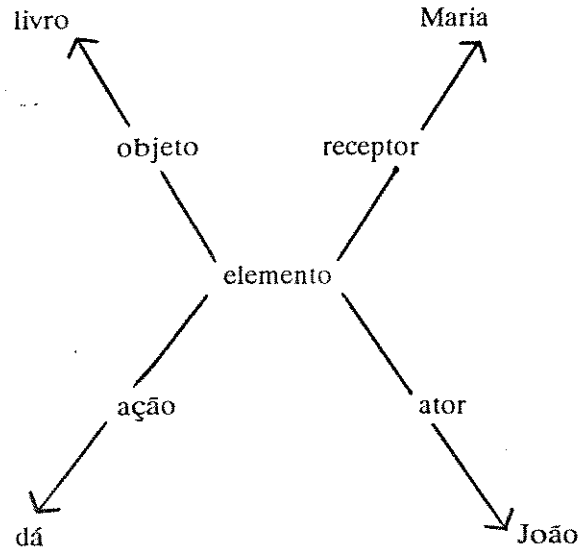


Figura 3.4 Representação da proposição "João dá o livro a Maria", na notação de Deliyanni e Kowalski (1979, p. 187)

A representação de aspectos sintáticos da lógica de predicados depende de uma escolha adequada para os nós e rótulos na rêde semântica (Ritchie, 1983). Para nossa aplicação específica, definimos "rêde semântica" como um grafo orientado cujos nós representam objetos/conceitos e cujos arcos representam relacionamentos entre objetos/conceitos. Um arco é rotulado com o nome da relação que ele representa. Vários arcos podem ter o mesmo rótulo. Entretanto, cada indivíduo (objeto/conceito) é representado por um único nó.

Estamos usando a definição apresentada de rêde semântica, combinada à sua "notação" (isto é, ao simbolismo gráfico usado pelos teóricos de rêdes semânticas), para criar os diagramas que representam graficamente a base de dados (programa) do usuário. Nossa escolha baseia-se em certas características oferecidas pelas rêdes semânticas, que tornam explícito o conhecimento nelas expresso, como comentamos a seguir:

- Existe uma correspondência um a um entre os vários nós da rêde e os objetos/conceitos que denotam.

- Objetos que aparecem em várias proposições conduzem a um agrupamento das proposições através do nó comum (proposições que tratam do mesmo objeto são naturalmente agrupadas).

- O diagrama possibilita uma visão imediata do inter-relacionamento entre objetos (suas conexões pelas sequências de arestas da rêde), e entre proposições que têm objetos em comum.

Representar o programa Prolog de forma independente de como ele é executado envolve, em nossa proposta, propor uma representação gráfica para o "significado" expresso na cláusula Prolog, vista isoladamente ou em conjunto com as outras cláusulas do programa. O modelo que usamos em tal representação é

baseado em rêdes semânticas (definição/notação), conforme será mostrado no capítulo 7.

Capítulo 4

Objetivos e Metodologia

Capítulo 4

Objetivos e Metodologia

O objetivo deste trabalho é o *design* de um ambiente computacional de aprendizado, baseado em programação em lógica, dirigido a novatos, e o estudo de sua interação nesse ambiente. Mais especificamente, estamos propondo um conjunto de ferramentas a serem integradas no ambiente Prolog, que contemplam aspectos relacionados ao paradigma de programação subjacente à linguagem Prolog. Paralelamente estamos interessados em entender os fenômenos relacionados à interação de novatos com essas ferramentas, no sentido de observar, através do uso desse ambiente, os processos envolvidos na tarefa de programar no paradigma da programação em lógica.

Nosso objeto de estudo é, portanto, o cenário composto pelo novato, pelo computador mediado por uma linguagem de programação baseada no paradigma da lógica, pelas ferramentas computacionais propostas e por um experto no domínio de programação em lógica, que chamaremos "facilitador"*[1].

Visto como um ambiente de aprendizado, estamos interessados em investigar os processos nos quais o novato é envolvido interagindo nesse ambiente. O entendimento dos fenômenos que ocorrem nesse cenário, por sua vez, são um importante *feedback* para o design de ferramentas computacionais de modo geral.

O trabalho envolve basicamente a proposta e desenvolvimento de um conjunto de ferramentas computacionais para uso de novatos no ambiente Prolog e um estudo experimental de sua utilização.

---

*[1] "facilitador" é um termo usado pela comunidade de informática educativa, que usa a abordagem Logo, para denominar o indivíduo que interage com o estudante e o ambiente computacional de modo a "facilitar" o envolvimento do estudante com a proposta Logo.

As ferramentas, distribuídas em um Módulo Operacional e um Módulo Declarativo, têm como proposta a explicitação das entidades significativas do paradigma subjacente à linguagem. Conforme discutido no capítulo anterior, o paradigma da linguagem é interpretado como o "meio" onde a solução para o problema é representada. Como tal, a linguagem de programação é vista como mediadora no processo de representação do problema.

A atividade de programar é entendida, conforme discutido no capítulo 2 (dois), como um processo incremental que envolve, a partir de um conjunto de hipóteses (sobre o domínio do problema, sobre a linguagem e sobre o ambiente), a criação e a depuração do programa. Essa "conversa" com a máquina é realimentada por respostas fornecidas pelo ambiente, que inclui as ferramentas sendo propostas.

As ferramentas estão inseridas metodologicamente num contexto de aprendizado com abordagem construtivista, emprestando de Piaget, Vygotsky e Papert, os elementos que compõem o quadro da "construção do conhecimento", conforme descrito no capítulo 3 (três).

A utilização das ferramentas nesse contexto é analisada a partir de um estudo experimental realizado com dois grupos diferentes de sujeitos. O primeiro grupo, que chamamos "novato-novato" é composto de 24 estudantes do segundo grau, em sua primeira experiência com linguagens de programação. As atividades desse grupo foram conduzidas em uma situação de sala de aula. O segundo grupo, que chamamos "novato-Prolog" é composto por 3 estudantes universitários com experiência em linguagens procedurais (principalmente Pascal). As atividades do segundo grupo foram conduzidas em uma situação individual. Em ambos os casos, a metodologia baseou-se em aspectos do método clínico de investigação, onde o papel do pesquisador foi o de tentar entender o "diálogo" que os sujeitos têm com o ambiente, no cenário descrito de início.

Nossa opção por esses dois tipos de sujeitos permite-nos investigar, no caso do novato-novato, em que medida a necessidade dos conhecimentos declarativo e operacional está presente na atividade de programar em Prolog e como as ferramentas propostas respondem a essa necessidade. O grupo de novatos em Prolog permite-nos investigar o processo de transição do paradigma procedural (conhecimento anterior) para o novo paradigma de programação e em que medida as ferramentas colaboram para a apropriação desse novo conhecimento na atividade de programar.

O estudo experimental realizado teve como opção metodológica a pesquisa qualitativa. Essa escolha, em contraposição ao uso de técnicas convencionais, pode ser justificada, em parte, pelo nosso interesse em uma compreensão geral da dinâmica de uso das ferramentas no ambiente proposto, o que se ajusta melhor às características da abordagem qualitativa.

As pesquisas guiadas por técnicas ditas convencionais, continuam prevalecendo em muitas áreas do conhecimento e entre muitos grupos de pesquisa (Monteiro, 1991). Nessa abordagem a quantificação de resultados empíricos é enfatizada e há uma separação entre o sujeito da pesquisa, o pesquisador e seu objeto de estudo. Tal separação supostamente garantiria "objetividade" à pesquisa, isto é "*os fatos, os dados se apresentariam tais quais são, em sua realidade evidente*", segundo Grunwaldt (citado em Monteiro, 1991, p. 28). Entretanto, os fatos e os dados não se revelam gratuita e diretamente aos olhos do pesquisador, conforme comentam Ludke e André:

> "É a partir da interrogação que ele faz aos dados, baseada em tudo o que ele conhece do assunto - portanto, em toda a teoria acumulada a respeito - que se vai construir o conhecimento sobre o fato pesquisado" (Ludke 1986, p. 4).

A atitude de procura de respostas a todas as questões através de técnicas convencionais de pesquisa é criticada por Papert e denominada "cientificismo"*2:

> "Muitas pessoas são enamoradas desses magros experimentos, porque eles são estatísticamente rigorosos e parecem fornecer o tipo de dados "pesados"*3 que se encontra em Física" (Papert, 1987, p. 5).

Experimentos que isolam apenas um fator, deixando o restante inalterado representam uma redução do enfoque do estudo a uma parte do fenômeno, o que pode ser útil para fins de análises específicas, "*mas não resolve o problema da compreensão geral do fenômeno em sua dinâmica complexidade*" (Ludke, 1986, p. 5).

A pesquisa com abordagem qualitativa originou-se na Antropologia e coloca uma dimensão social ao conhecimento científico, que é "*compreendido com sua realidade histórica e não pairando acima dela como verdade absoluta*" (Ludke, 1986, p. 2). Dessa maneira, a pesquisa e o pesquisador estão naturalmente inseridos e são influenciados por um contexto social. Nossa opção pela abordagem qualitativa é função, portanto, da natureza do problema sendo estudado (a interação do novato num ambiente baseado em ferramentas computacionais), e procura ser coerente com o quadro que sustenta a dinâmica do ambiente de aprendizado adotado, onde o componente social está presente.

Dessa maneira, o cenário composto pelo novato resolvendo problemas no ambiente constituído pelo interpretador Prolog acrescido de ferramentas propostas foram a fonte direta de dados em nosso estudo experimental. A análise dos dados seguiu um processo indutivo, partindo de questões de interesse mais amplo

---

*2 Scientism no original.
*3 "hard data" no original.

que se tornaram mais específicos no decorrer da investigação, onde a preocupação com o processo foi muito maior do que com o produto.

Capítulo 5

O Ambiente Proposto

Capítulo 5

O Ambiente Proposto

"Multiplicidade é a alma dos ambientes de aprendizagem".

(TECFA, 1990)

Neste capítulo apresentamos uma visão geral do conjunto de ferramentas que compõem o ambiente proposto e apresentamos as principais opções de *design* adotadas.

Conforme discutimos no capítulo 3, visibilidade é uma metáfora que pode ser aplicada à linguagem de programação, enquanto "máquina virtual" que é alimentada pelo programa. Assim, no nosso caso específico, estamos considerando o *design* de ferramentas que devem facilitar ao novato a construção do modelo conceitual da linguagem enquanto sistema a ser compreendido. Isto requer captar nas ferramentas as partes relevantes da operação do sistema, de forma apropriada ao usuário.

A metáfora da visibilidade é usada, no contexto de nosso trabalho, como forma de enriquecer o *feedback* gerado pelo ambiente e possibilitar ao novato seu engajamento no ciclo de ações que o levam a refletir sobre seu conjunto de hipóteses. Esse *feedback* é importante, principalmente em situações de erro, fornecendo ao novato um espaço de possibilidades para ação. Isso inclui possibilitar ao usuário desenvolver o aprendizado apropriado do domínio no qual os erros ocorrem e também desenvolver as habilidades e procedimentos necessários para reconhecer o que "deu errado" e como lidar com a situação.

Enquanto que visibilidade tem sido aplicada para tornar "transparente" ao usuário a máquina virtual da linguagem, em nossa proposta estamos estendendo essa metáfora de modo a tornar explícito, também, o conhecimento envolvido no conjunto de cláusulas que representa o programa Prolog. Portanto, em nossa abordagem, visibilidade não se refere apenas aos aspectos operacionais da linguagem enquanto máquina virtual. O conhecimento expresso no programa é mostrado, também, de forma independente do mecanismo de execução de Prolog. É assim que entendemos a explicitação do paradigma subjacente à linguagem, onde o aspecto operacional é apenas uma das faces da moeda. Nossa abordagem envolve prover o ambiente com ferramentas que possibilitem ao novato a criação de micro-mundos declarativos para facilitar a meta-análise do problema e micro-mundos operacionais para investigação de como sua base de dados é usada pela máquina, no processo de responder suas perguntas.

A seguir apresentamos a estrutura do sistema computacional desenvolvido, constituído pelas ferramentas (figura 5.1), e mostramos exemplos de representações geradas.

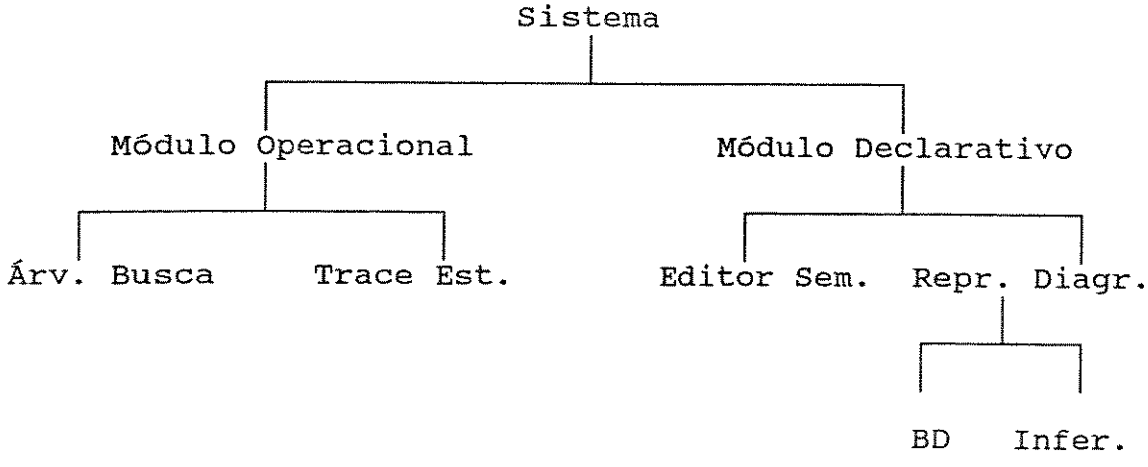


Figura 5.1 Estrutura Hierárquica das Ferramentas que Compõem o Sistema.

O sistema (conjunto de ferramentas) pode ser visto como compondo um "Módulo Declarativo" e um "Módulo Operacional". Nossa abordagem ao modo operacional de investigar um programa é baseada em uma representação gráfica do comportamento da máquina de inferência ao buscar uma resposta a uma pergunta. Nessa representação estamos usando um modelo baseado na estrutura da árvore de busca de determinada meta em uma base de dados, que denominamos "árvore de espaços de busca". O modo declarativo de investigar um programa baseia-se em uma representação gráfica do conhecimento embutido em um programa Prolog, independente de mecanismos de execução. Estamos usando um modelo baseado em diagramas de rêdes semânticas, que denominamos "diagramas semânticos".

5.1 O Módulo Declarativo

No Módulo Declarativo o formalismo clausal de representação de conhecimento é expresso de forma pictórica e pode ser investigado pelo usuário através de facilidades de navegação e busca de informação na representação gráfica.

Através do Módulo Declarativo o usuário tem acesso à representação em diagramas semânticos de sua base de dados, em três níveis diferentes: ao nível de cláusula, ao nível de base de dados e ao nível de inferências geradas a partir da base de dados.

Ao nível de cláusula, o novato pode ter um *feedback* sobre o "significado" de sua sentença, baseado nos conceitos de "objetos" e "relações". Uma das ferramentas do sistema, que chamamos Editor Semântico (EDS) lida com o aspecto de semântica da cláusula. Esta parte do sistema objetiva ajudar na fase de criação da base de dados, através da explicitação dos inter-relacionamentos entre os objetos das proposições. A figura 5.2 mostra um exemplo da representação visual da

sentença *"Y rouba X se Y é ladrao e Y gosta de X"*, representado no formalismo clausal de Prolog por: *"rouba(Y,X):-ladrao(Y),gosta(Y,X)"*.

O diagrama mostra os objetos envolvidos na regra, X e Y, dispostos verticalmente no centro da região de tela reservada para o desenho do diagrama semântico. O predicado da parte consequente (rouba) é representado mais à esquerda dos objetos. Setas pontilhadas associam o predicado a seus argumentos, com numeração correspondendo à ordem dos argumentos no predicado. Os predicados da parte antecedente da regra (gosta e ladrão) são diferenciados do predicado da parte consequente por usarem setas contínuas e serem dispostos mais à direita no diagrama.

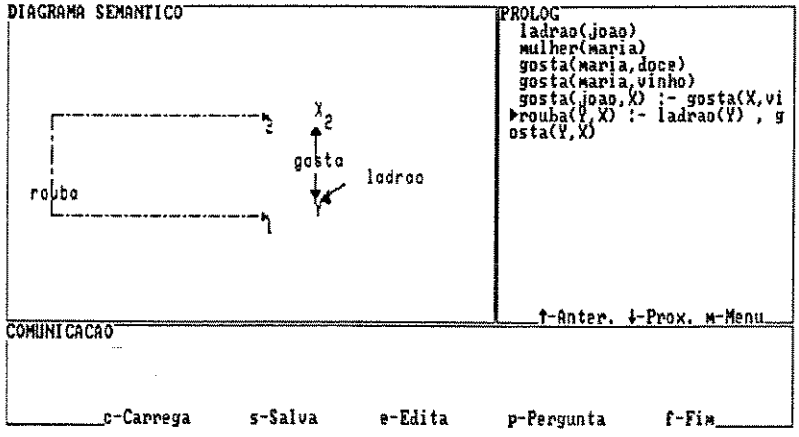


Figura 5.2 Exemplo de Representação de Regra em EDS

Ao nível de base de dados, o sistema provê uma representação em diagrama semântico do conjunto de cláusulas que compõem a base de dados do usuário. Nesse nível, o modelo de representação torna explícitas as inter-relações entre as cláusulas que têm objetos em comum. Com esta parte do sistema, que chamamos "Modo de Representação em Diagramas Semânticos" (ou MDS), o usuário pode localizar objetos ou relações que ele queira investigar. Independentemente do processo de execução,

associações que estão implícitas na base de dados são explicitadas pelo diagrama, como é o caso, por exemplo do "caminho" que liga os objetos *joão* e *maria*, como mostra a figura 5.3a.

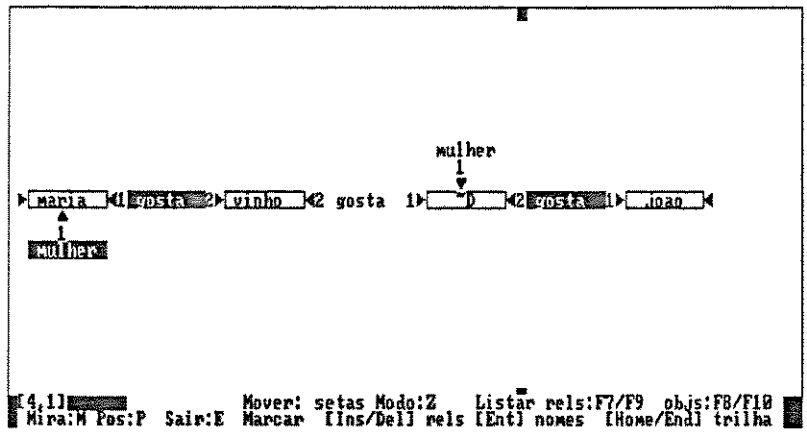


Figura 5.3a Exemplo de Representação da Base de Dados em MDS

As relações estão representadas em fundo preto com numerações indicando a ordem de relacionamentos com os respectivos objetos. Uma janela pode ser acionada para listar relações ou objetos presentes na base de dados (figura 5.3b).

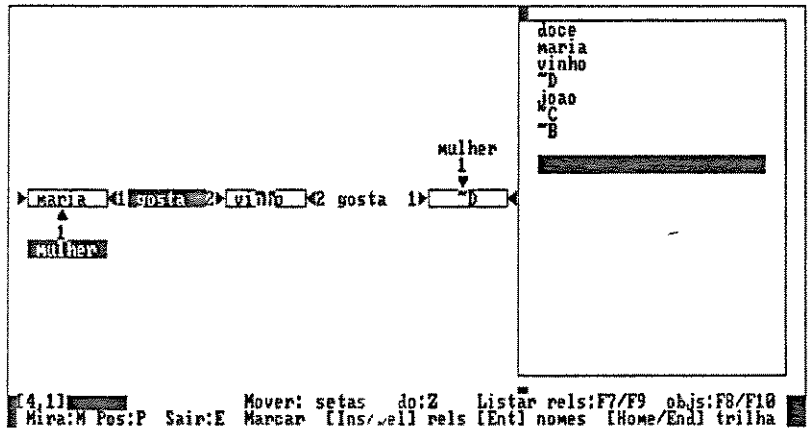


Figura 5.3b Diagrama Semântico e Lista de Objetos

Através dessa janela podem ser feitas marcações e seleções de partes da rêde (figura 5.3c). O diagrama pode ser reduzido, ampliado, movimentado para visualização do todo ou de partes.

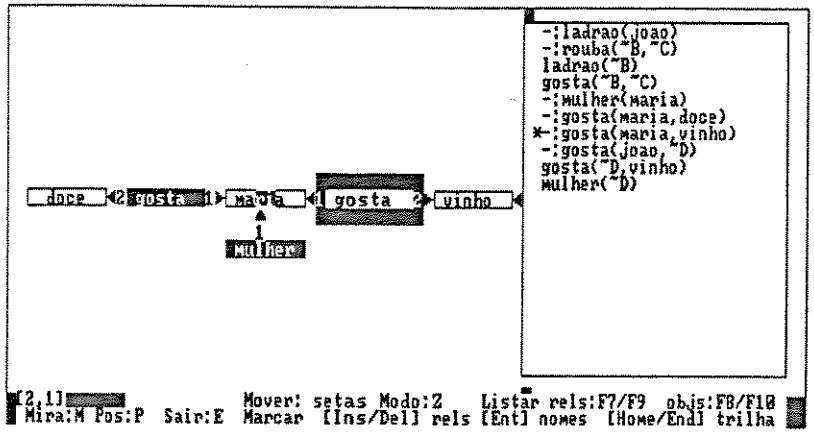


Figura 5.3c Diagrama Semântico e Lista de Relações

Ao nível de inferência, o sistema mostra a representação em diagrama semântico das inferências geradas na busca da resposta a uma pergunta do usuário. Dessa maneira, o usuário pode visualizar fatos implicados por sua base de dados, como por exemplo "*joão gosta de maria*" que combinados a fatos presentes conduzem a uma resposta. A figura 5.4 mostra o diagrama representando a solução para a meta "*rouba(X,Y)*".

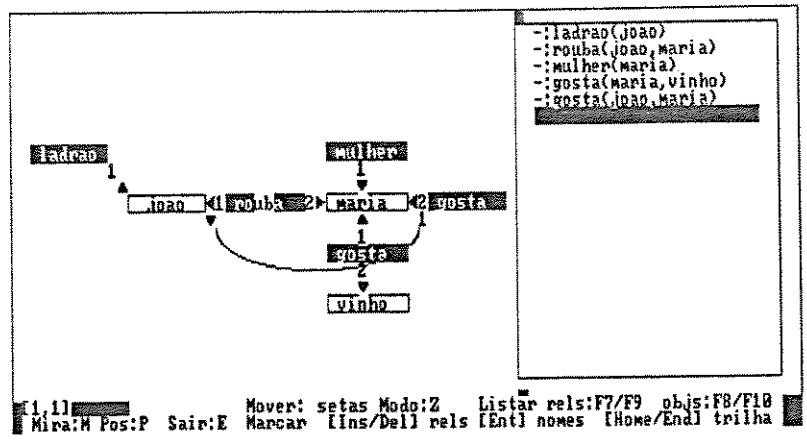


Figura 5.4 Diagrama Semântico de Inferências Geradas

5.2 O Módulo Operacional

O Módulo Operacional lida com a semântica operacional de programas Prolog. Como tal, objetiva ajudar o usuário a construir o modelo conceitual da máquina de inferência Prolog. Nossa abordagem envolve mostrar o processo de execução de duas maneiras diferentes: uma "local", que denominamos "Trace Estendido" e outra "global", que denominamos "Árvore de Espaços de Busca".

Na investigação local o usuário tem acesso à meta que Prolog está tentando provar, à regra que está sendo aplicada, ao resultado da unificação e à próxima sub-meta a ser verificada (figura 5.5). Em cada momento ele pode escolher continuar nesse modo (trace estendido), mudar para o modo global (árvore de busca) ou conhecer o resultado final da interpretação da meta inicial (interpretação), selecionando a opção respectiva.

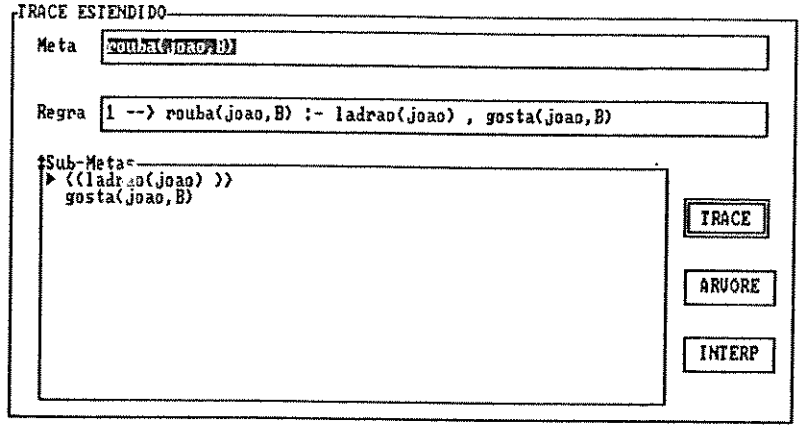


Figura 5.5 Tela do Trace Estendido em MOP

Na investigação global, o usuário tem acesso à construção passo a passo do mecanismo de execução de Prolog. O modelo que estamos usando é uma extensão da árvore de busca de uma meta em uma base de dados, adaptada para mostrar o processo de execução através da estrutura de árvore. A representação mostra a meta cuja prova está sendo buscada no contexto estrutural completo da execução, onde o usuário pode observar a estrutura do processo de execução, através da hierarquia entre metas e sub-metas, as metas que falharam, as metas que sucederam e a árvore de prova, no caso de término bem sucedido do processo. As figuras 5.6a, 5.6b e 5.6c mostram três momentos da sequência de construção da árvore de espaços de busca para a meta *rouba(X,Y)*, com base nas cláusulas:

mulher(maria).

ladrao(joao).

gosta(maria,doce).

gosta(maria,vinho).

gosta(joao,X):-gosta(X,vinho),mulher(X).

rouba(X,Y):-ladrao(X),gosta(X,Y).

A árvore de espaços de busca é construída na janela maior da figura. Cada nó da árvore (um quadrado numerado) representa uma meta cuja redução é tentada. O nó contém o nome do predicado e um número (dinâmico) indicando a ordem de avaliação da meta. Dessa maneira, os nós da árvore constituem a hierarquia de metas e sub-metas usadas pela máquina de inferência na busca de uma solução. Assim, por exemplo, a meta *rouba(joao,X)* (nó 1) tem como sub-metas *ladrao(joao)* e *gosta(joao,X)* (nós 2 e 3 respectivamente). O conteúdo de cada nó é representado de forma completa na janela menor, à direita (janela de predicados) (figura 5.6a).

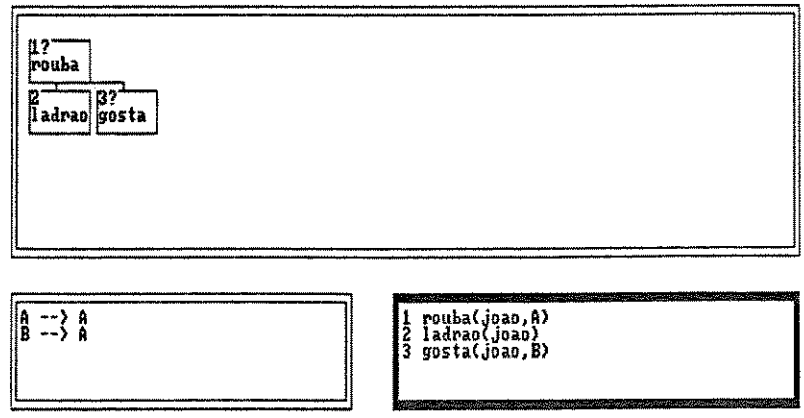


Figura 5.6a Parte da Construção da Árvore para a Meta *rouba(joao,X)*, com Janela de Predicados em Destaque

As instanciações das variáveis são mostradas na janela menor à esquerda (janela de variáveis) (figura 5.6b).

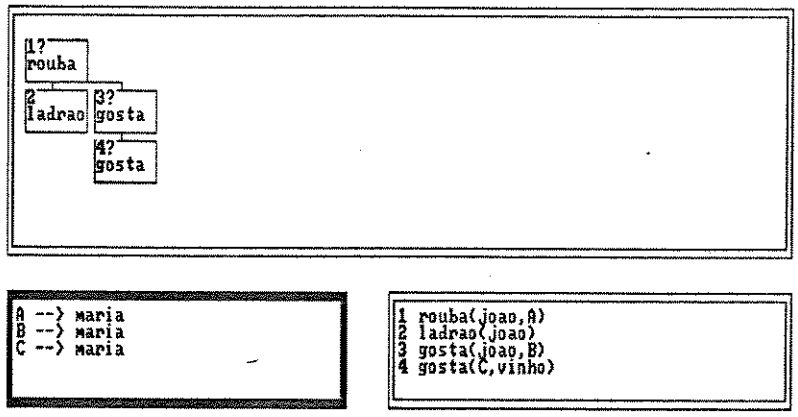


Figura 5.6b Construção da Árvore para a Meta *rouba(joao,X)* (continuação), com Janela de Variáveis em Destaque

As figuras 5.6a, 5.6b e 5.6c ilustram três momentos da construção da árvore de espaços de busca para a meta *rouba(joao,X)*.

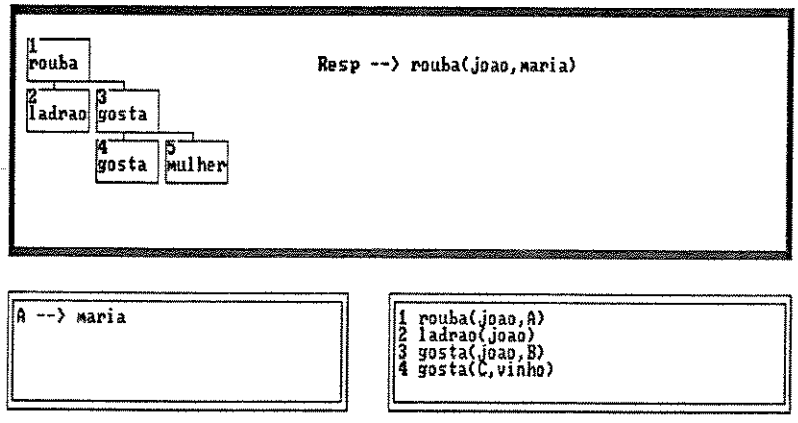


Figura 5.6c Representação Final da Construção da Árvore

Dessa maneira, no modo operacional o usuário tem acesso a elementos necessários para o entendimento da semântica operacional de Prolog, através do conhecimento

de "como" a máquina Prolog está usando sua base de dados para encontrar respostas às suas perguntas.

5.3 Exemplos de Conceitos Abordados pelas Ferramentas

O ambiente constituído pelas ferramentas propostas favorece ao novato a exploração de conceitos importantes, a nível da linguagem (unificação, instanciação de variáveis, *backtracking*), a nível do programa sendo representado (estrutura da solução, definições recursivas, aspectos de eficiência) e a nível da estratégia para investigação do problema.

A seguir mostramos como alguns desses aspectos são explicitados através das ferramentas.

5.3.1 Análise Sistemática de Fluxo e Meta-Análise

A busca do entendimento de um programa Prolog, na tentativa de antecipar ou entender a resposta fornecida a uma dada pergunta, pode ser feita através de uma análise sistemática do fluxo de execução do programa (simulando o que a máquina virtual da linguagem faria) ou pode ser feita por uma leitura baseada em uma meta-análise do programa.

É considerada uma meta-análise, por exemplo, a análise que o sujeito faz de uma determinada cláusula, numa tentativa de "encurtar o caminho" de busca da resposta. Para ilustrar, consideremos a meta a(X) no trecho de base de dados abaixo:

```
a(X):-b(X),c(X).
...
c(3).
```

...

O fato de se ter um único c(3) na base de dados, leva o novato a associar X com 3 e então procurar uma cláusula b que se ajuste a essa restrição (Hook, 1990). Outro tipo de meta-análise é a leitura "ajudada" pelo significado contextual da base de dados (exemplo típico são as relações de parentesco). Outros tipos de conhecimento usados, também, no entendimento de um programa Prolog são relativos ao próprio domínio de programação, como por exemplo a identificação de estruturas de programa e seus mecanismos correspondentes (a identificação de uma definição recursiva, a identificação de clichês, etc.). Esse tipo de conhecimento é usado principalmente (mas não exclusivamente) por sujeitos com conhecimento anterior no domínio de programação.

A meta-análise é considerada uma estratégia poderosa para o entendimento de programas, (Hook, 1990) e é usada com propriedade pelo *expert* como complemento à análise sistemática da execução.

As ferramentas que estamos propondo possibilitam ao novato investigar o programa através dos dois tipos de análise: a análise sistemática do fluxo de execução, através do Módulo Operacional (MOP) e a meta-análise através do Módulo Declarativo (MDS). A seguir, mostramos a representação da base de dados [local] segundo esses dois enfoques.

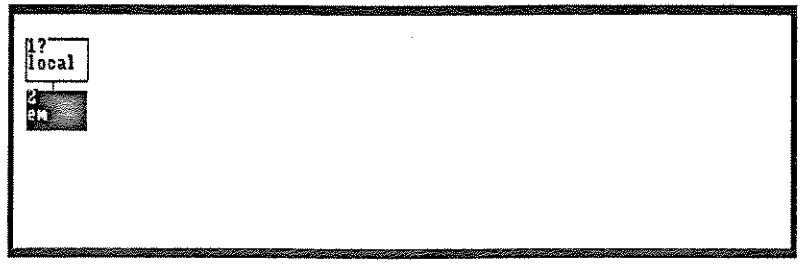
```
local(P,L):-em(P,L).
local(P,L):-visita(P,O),local(O,L).
em(alam,sala19).
em(jane,sala54).
em(bete,escritorio).
visita(davi,alam).
visita(janete,bete).
```

visita(alberto,davi).

?local(alberto,X).

A busca da resposta para a pergunta *local(alberto,X)* pode ser investigada através dos aspectos operacionais envolvidos, conforme trata o MOP ou através dos aspectos declarativos, conforme trata o MDS. A sequência de figuras 5.7 geradas por MOP mostra parte do processo de construção da árvore de busca para a meta *local(alberto,A)*.

A figura 5.7a mostra a primeira tentativa de Prolog satisfazer a meta. Foi aplicada a primeira regra para *local* (*local(P,L):-em(P,L)*), com P associado a *alberto*). A tentativa de satisfazer a sub-meta *em(alberto,A)* fracassou (representado no nó escurecido).



Regra: 1 --> local(alberto,A) :- em(alberto,A)
Meta: em(alberto,A)

Figura 5.7a Parte da Construção da Árvore em MOP

A figura 5.7b mostra, após o fracasso no nó 2, nova regra sendo aplicada à meta inicial: a segunda regra para *local* (*local(P,L):-visita(P,O), local(O,L)*). A próxima meta a ser resolvida (sub-meta de local) é *visita(alberto,D)*.

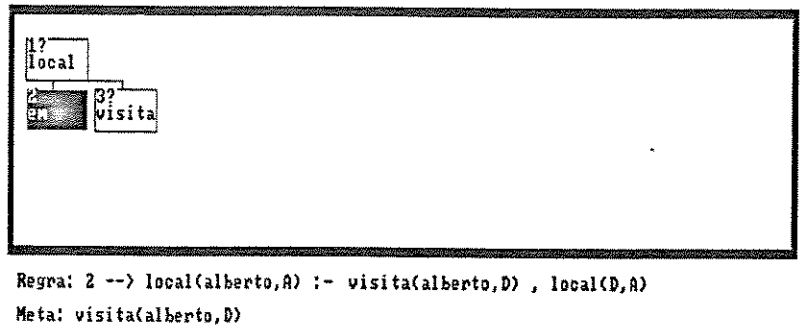


Figura 5.7b Parte da Construção da Árvore em MOP (continuação)

A figura 5.7c mostra o final do processo de busca, com a resposta encontrada (*sala19*). Os nós 1, 3, 4, 6, 7 e 8 constituem a árvore de prova para a meta. Os nós 2 e 5 representam metas que falharam durante o processo.

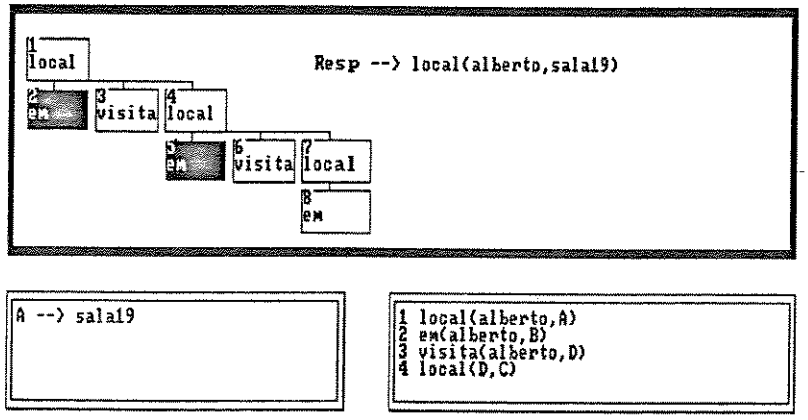


Figura 5.7c Final da Construção da Árvore em MOP

A mesma base de dados, olhada sob a perspectiva do conhecimento declarativo está representada pelos diagramas semânticos das figuras 5.8a, 5.8b e 5.8c. A figura 5.8a

mostra, por exemplo, que existe um inter-relacionamento que liga o objeto *alberto* ao objeto *sala19* (as relações são representadas em fundo preto).

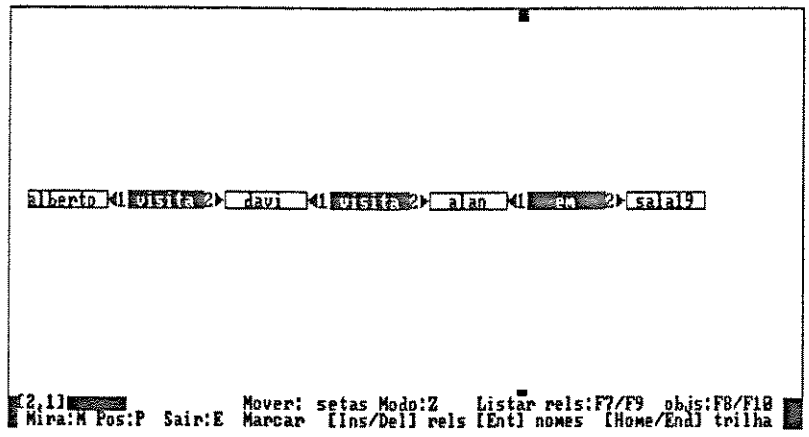


Figura 5.8a Diagrama Semântico de Representação de parte da Base de Dados

As figuras 5.8b e 5.8c mostram as regras que definem *local(Pessoa,Lugar)*.

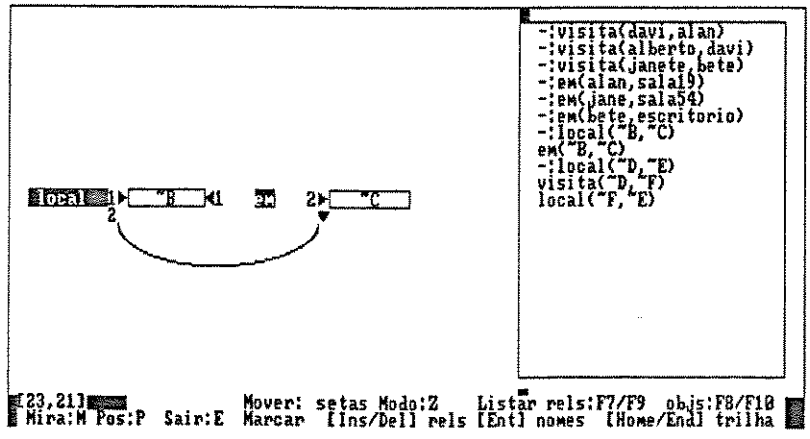


Figura 5.8b Diagrama Semântico de Representação da Regra *local(B,C):-em(B,C)*.

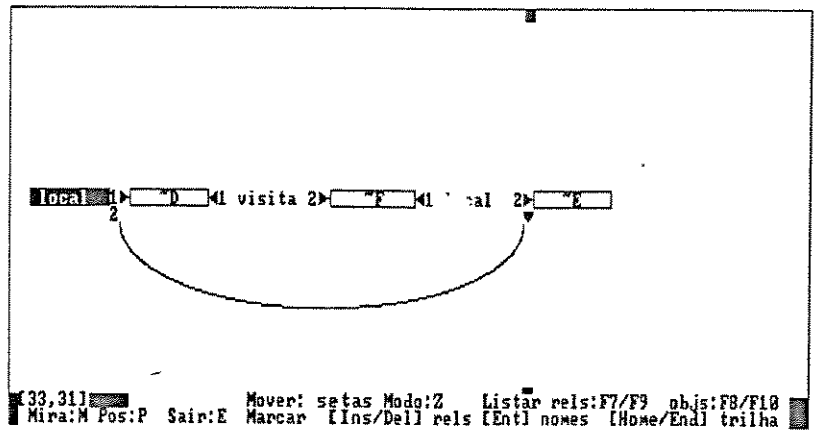


Figura 5.8c Diagrama Semântico de Representação da Regra *local(D,E):- visita(D,F),local(F,E).*

A presença do inter-relacionamento entre *alberto* e *sala19*, visíveis na figura 5.8a não significa, entretanto, que a resposta para *local(alberto,X)* é *sala19*. A resposta é dependente da definição de regras com o significado de *local(Pessoa,Lugar)*, como, por exemplo, as mostradas nas figuras 5.8b e 5.8c. O sujeito pode checar a resposta para *local(alberto,X)* utilizando a opção Inferência de MDS (figura 5.9a) e analisar, no caso de existir solução, a representação do diagrama semântico que representa a solução para a pergunta, como mostra a figura 5.9b.

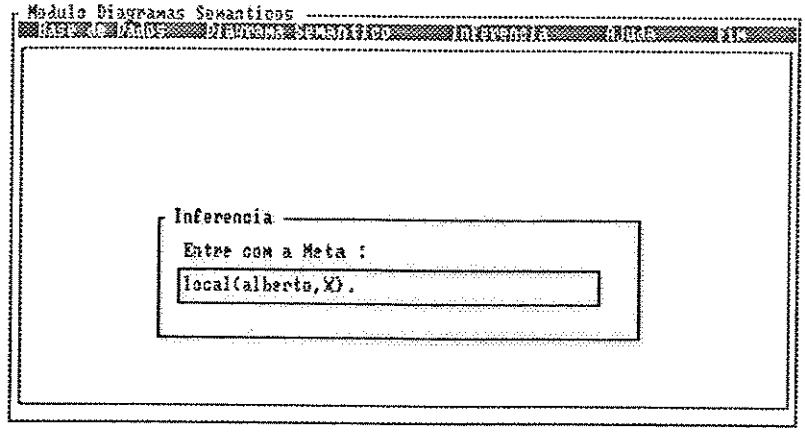


Figura 5.9a Escolha da Opção Inferência de MDS

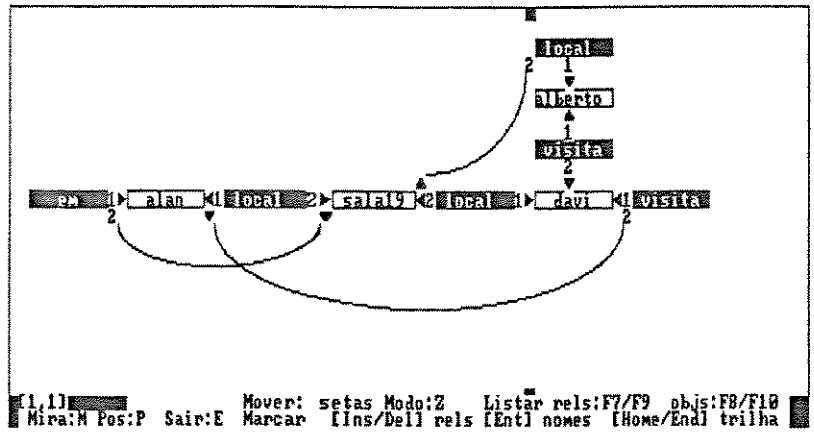


Figura 5.9b Diagrama Semântico de Representação da Solução para a Meta *local(alberto,X)*

O Diagrama semântico da figura 5.9b mostra que o *local de alberto* é *sala19* e todos os fatos relacionados a essa conclusão, como por exemplo: "alberto visita davi", "o local de davi é sala19", "davi visita alan", "alan está na sala19", "o local de alan é sala19", independentemente do conhecimento da máquina de inferência de Prolog.

5.3.2 A Explicitação do Processo de Backtracking

O entendimento da máquina de inferência de Prolog depende do entendimento de seu mecanismo de retrocesso (*backtracking*). Tem sido discutida na literatura a dificuldade dos estudantes e seus erros conceituais relacionados ao processo de *backtracking* (Coombs, 1985; Taylor, 1987; Fung, 1990; Hook, 1990). Dois tipos de erros são encontrados com maior frequência e foram denominados por Coombs e Stell (1985) TOAP ("tenta uma vez e passa") e RDBL ("refaz o corpo a partir da es-

querda")*[5]. Na situação do erro RDBL, o sujeito acredita que quando uma meta falha, Prolog tenta refazer o corpo da cláusula à partir da esquerda (em vez de fazê-lo a partir da direita, começando na escolha mais recente). Na situação do erro TOAP, o sujeito acredita que uma cláusula falha completamente depois de falha em seu corpo. Hook *et al* (1990) sugerem que erros do tipo mencionados podem ser uma influência do modelo Byrd Box usado nos *tracers* tradicionais. Para ilustrar essa questão, consideremos a base de dados [BT1] a seguir:

```
a(X):-b(X),c(X).
b(X):-d(X),e(X).
b(X):-f(X).
d(1).
d(2).
e(1).
e(2).
c(3).
f(3).
        ? a(X).
```

Ao executar a meta a(X) a máquina Prolog escolhe a primeira regra para *b* e em seguida a primeira regra para *d* associando X a 1. Em seguida *e(1)* tem sucesso, mas *c(1)* fracassa. Até esse ponto, o *feedback* gerado pelo modelo Byrd Box seria:

```
(0) call a(_)?
(1) call b(_)?
(2) call d(_)?
```

*5 tradução de "try once and pass" e "redo body from left", respectivamente (Coombs, 1985).

(2) exit d(1)

(3) call e(1)?

(3) exit e(1)

(1) exit b(1)

(4) call c(1)?

(4) fail c(1)

(1) redo b(1)

O "*redo b(1)*" pode induzir o novato à escolha da segunda regra *b* (erro TOAP) ou a recomeçar na mesma regra (primeira regra *b*) a partir da esquerda, tentando *d(X)* antes de *e(X)* (erro RBDL).

No modelo de execução que propomos, o ponto de *backtracking* fica explícito na dinâmica da construção da árvore de busca, conforme ilustrado a seguir, na sequência de figuras 5.10a, 5.10b e 5.10c.

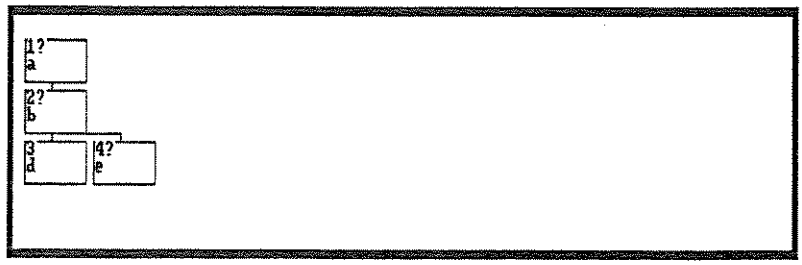


Figura 5.10a Ilustração do mecanismo de retrocesso (momento1)

A figura 5.10a mostra a árvore de busca no momento em que a meta *b* está sendo resolvida (nó 2) e sua próxima sub-meta é *e(1)*. Está sendo aplicada a primeira regra de *b* (b(X):-d(X),e(X)).

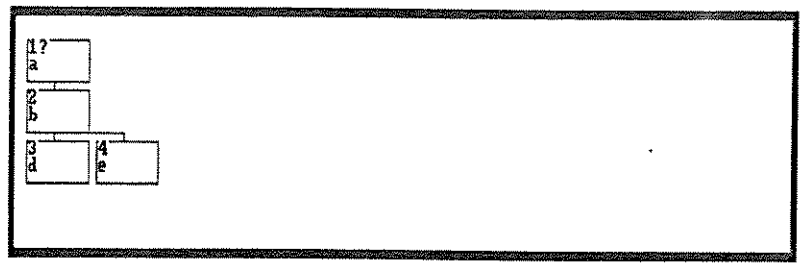


Figura 5.10b Ilustração do mecanismo de retrocesso (momento 2)

A figura 5.10b mostra o momento do processo em que a meta *b* já foi resolvida. Está sendo tentada a meta *a*, através da aplicação de sua primeira regra (a(X):- b(X),c(X)) e a próxima sub-meta é *c(1)*.

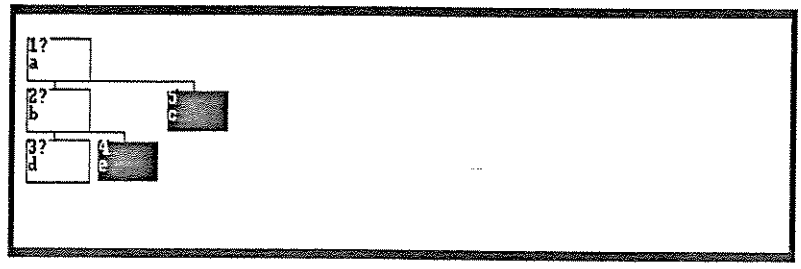


Figura 5.10c Ilustração do mecanismo de retrocesso (momento 3)

A figura 5.10c mostra a falha na meta *c(1)* (representada pelo nó 5 escurecido), o ponto de *backtracking* (nó 4), representando uma falha ao tentar resatisfazer *e(1)* após retrocesso.

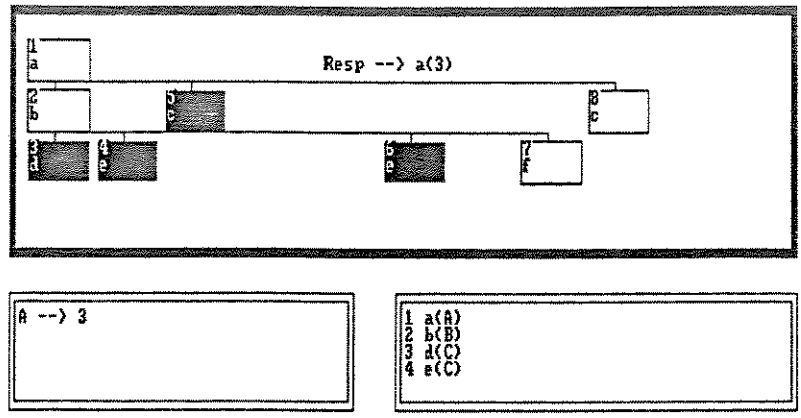


Figura 5.10d Representação Final da Árvore para a Meta *a(X)*

A figura 5.10d mostra a árvore de prova para *a(X)* constituída por *a(3):-b(3),c(3)* e *b(3):-f(3)*.

5.3.3 A Explicitação do Mecanismo Recursivo - a "Volta" da Recursão

A definição de recursão como um procedimento que "chama" a si próprio esconde a complexidade do processo de criar e depurar programas recursivos. Anderson *et al* explicam a dificuldade de novatos com programas recursivos alegando que esse tipo de programa exige atividades mentais que não são familiares e que dependem do desenvolvimento de grande quantidade de conhecimento sobre padrões recursivos específicos (Anderson, 1988). Eles argumentam que os procedimentos que as pessoas usam são iterativos e que a mente humana parece incapaz de fazer o que a máquina faz com recursão: criar uma cópia do procedimento, aninhá-lo em si próprio e controlar o que acontece com as duas cópias. Observando a maneira como estudantes simulam mentalmente a execução de funções Lisp, eles observaram que os estudantes, em geral, não mostram dificuldade quando a recursão é de cauda: "*parece que a mente pode tratar tais procedimentos recursivos de cauda, com estruturas de controle iterativas*" (Anderson, 1988, p. 161).

Esse não é o caso, entretanto, de soluções que envolvem algum processamento na "volta" da recursão, conforme descrito a seguir:

> "...quando eles têm que simular o retorno de resultados e combinar os resultados parciais eles ficam completamente perdidos. Parece que a mente humana, ao contrário do interpretador Lisp, não pode suspender um processo, fazer uma nova cópia do processo, reiniciar o processo para fazer uma chamada recursiva e retornar ao processo original suspenso" (Anderson, 1988, p. 161).

A dificuldade no tratamento do conceito de recursão, a nosso ver, extrapola a questão da linguagem de programação. A dificuldade de novatos e programadores com recursão em Logo e Pascal, por exemplo, foi descrita em (Valente, 1988) e (Rocha, 1991), respectivamente. As dificuldades dos estudantes com recursão em Lisp não são muito diferentes das dificuldades do novato Prolog com cláusulas recursivas, conforme ainda será mostrado no decorrer deste trabalho (capítulo 8).

O modelo de execução que propusemos para mostrar o modelo conceitual da máquina de inferência Prolog lida com esse problema explicitando, na estrutura da árvore de execução, o encadeamento do processo recursivo bem como os retornos correspondentes a cada ativação do processo.

Para ilustrar esse aspecto, consideremos a base de dados que representa o cálculo da potência de um número, como descrito a seguir:

```
pot(X,0,1).
pot(X,N,Y):-N1 is N-1, pot(X,N1,Y1), Y is X*Y1.
```

A sequência de figuras 5.11a, 5.11b e 5.11c ilustram três momentos do processo de execução para a meta *pot(2,3,X)*.

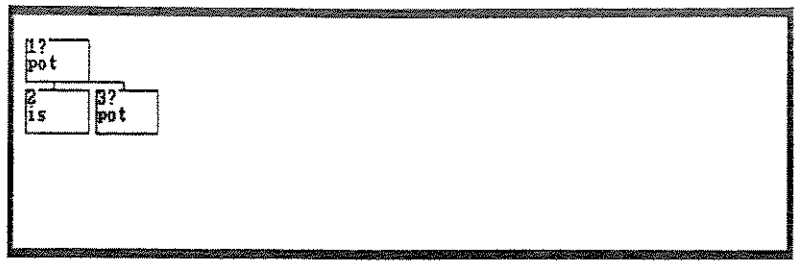


Figura 5.11a Ilustração do Mecanismo Recursivo (momento 1)

A figura 5.11a mostra o momento do processo em que ocorre a primeira chamada recursiva.

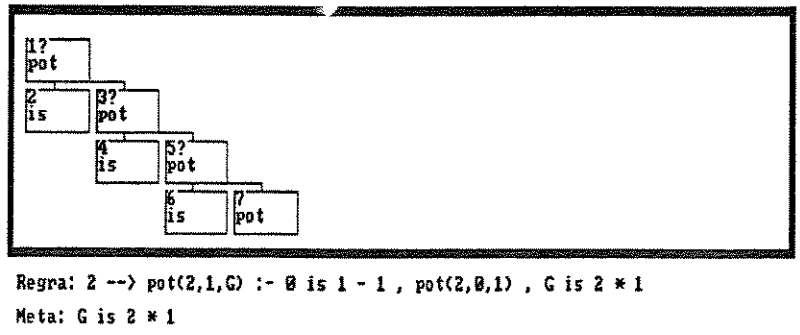


Figura 5.11b Ilustração do Mecanismo Recursivo (momento 2)

A figura 5.11b mostra o momento do processo em que a terceira chamada recursiva já foi resolvida (*pot(2,0,1)*) no nó 7 e o próximo passo representa o primeiro retorno da chamada recursiva, com a execução da meta *G is 2*1*.

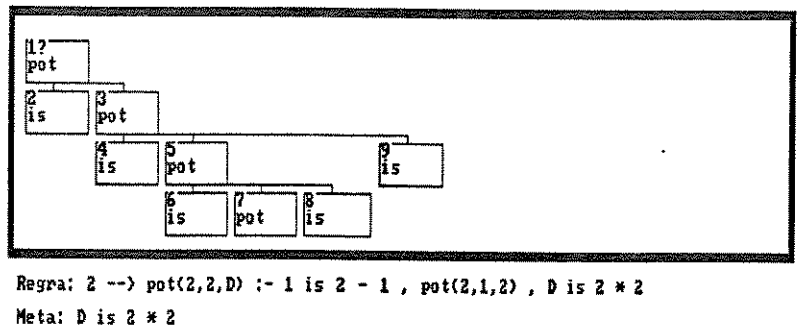


Figura5.11c Ilustração do Mecanismo Recursivo (momento 3)

A figura 5.11c mostra a "volta da recursão", nos nós 8 e 9 que correspondem aos retornos das chamadas recursivas *pot(2,0,1)* (nó 7) e *pot(2,1,2)* (nó 5), respectivamente.

A figura 5.11d mostra o final do processo de execução para a meta *pot(2,3,A)*. Os nós 3, 5 e 7 mostram, respectivamente, as chamadas recursivas *pot(2,2,B)*, *pot(2,1,C)* e *pot(2,0,D)*. Os nós 8, 9 e 10 mostram, respectivamente, os "retornos" das chamadas recursivas, como suas metas "irmãs".

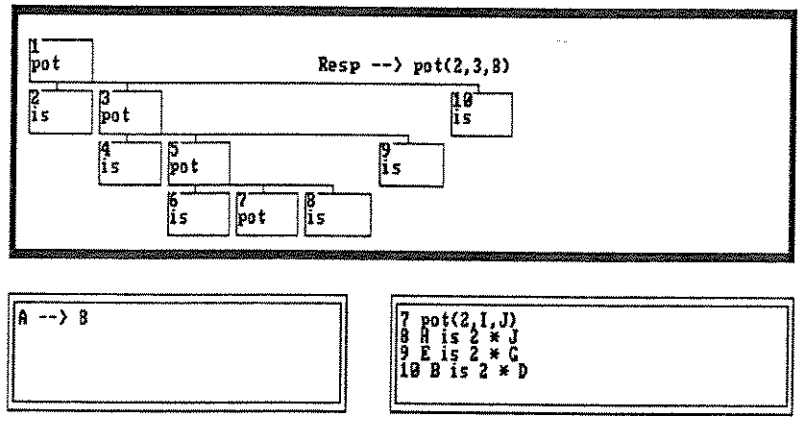


Figura 5.11d Representação Final da Árvore de Execução para *pot(2,3,X)*

A representação visual do processo recursivo pode funcionar como um modelo para o sujeito "pensar sobre" um processo cuja simulação mental é difícil, conforme será mostrado no decorrer deste trabalho.

5.3.4 A Explicitação de Aspectos de Eficiência

Nem sempre é simples, para o novato, o julgamento sobre aspectos de eficiência de um determinado programa. A identificação e julgamento desses aspectos é natural e faz parte da meta-análise usada pelos *experts*, que analisam, por exemplo, a ordem de sub-metas em uma determinada cláusula, conforme exemplo descrito a seguir:

```
parente(Y,Z):-irmaos(X,Z),casados(X,Y).
irmaos(X,Y):-irma(X,Y).
irmaos(X,Y):-irmao(X,Y).
irma(ana.sueli).
irma(ana,pat).
irmao(maria,joao).
casados(maria,jose).
        ? parente(A,B). (Hook, 1990, p. 341)
```

De acordo com Hook *et al* um *expert* tenderia a colocar a sub-meta mais restritiva, *casados(X,Y)* antes de *irmaos(X,Z)*, na cláusula *parente(Y,Z)*, buscando usar a estrutura de controle de Prolog de maneira mais eficiente.

Para o novato, a diferença em considerar a cláusula parente como a apresentada originalmente ou trocando a ordem das sub-metas, fica explicitada na estrutura das árvores construídas, num caso e no outro, como ilustram as figuras 5.12a e 5.12b a seguir.

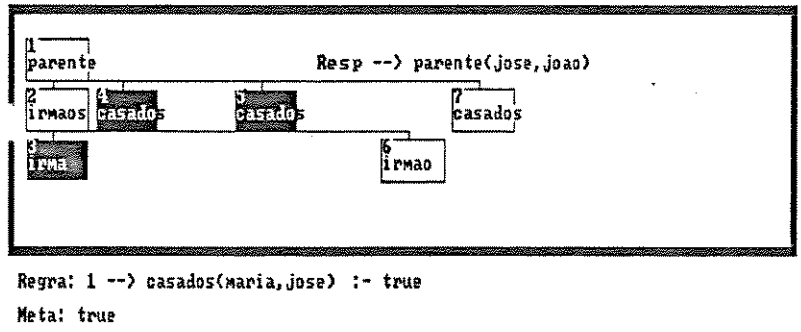


Figura 5.12a Representação da Árvore para a Base de dados com a Regra *parente(Y,Z):-irmaos(X,Z), casados(X,Y)*

Na figura 5.12a, para cada par de irmãs encontrado na base de dados, a sub-meta *casados* é tentada, sem sucesso (representados pelos nós 4 e 5 em fundo preto).

Na figura 5.12b a execução da sub-meta *casados* antes da sub-meta *irmaos* determina um caminho mais curto à resposta.

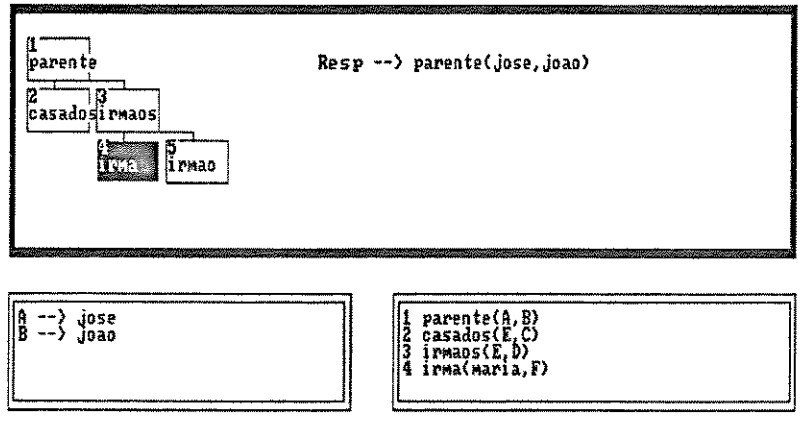


Figura 5.12b Representação da Árvore para a Base de Dados com a Regra *parente(Y,Z):-casados(X,Y),irmaos(X,Z)*

5.3.5 A Explicitação da Estrutura da Solução

A estrutura da execução de programas, mostrada na árvore de busca, reflete a solução para o problema e fornece pistas para entendimento do processo subjacente à solução.

Os problemas de ordenação de elementos em uma lista são exemplos de processos sofisticados, que são refletidos nas estruturas das respectivas árvores de execução. A figura 5.13a mostra a árvore de busca gerada para a meta *isort([3,2,1],X)* usando o método de ordenação por inserção, descrito pela base de dados a seguir:

```
isort([X|Y],Z):-isort(Y,Z1), insert(X,Z1,Z).
isort([],[]).
insert(X,[Y|Z],[X,Y|Z]):-X=<Y.
insert(X,[Y|Z],[Y|W]):-X>Y, insert(X,Z,W).
insert(X,[],[X]).
```

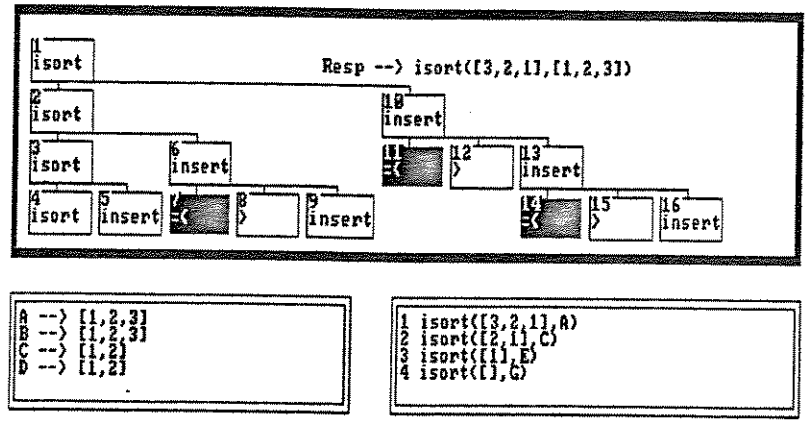


Figura 5.13a Representação Final da Árvore de Execução para *isort([3,2,1],X)*

Na figura 5.13a, a meta original *(isort([3,2,1],X))* é resolvida a partir da resolução das sub-metas *isort([2,1],C)* (nó 2) e *insert(3,[1,2],X)* (nó 10). A sub-árvore cuja raíz é o nó 2, por sua vez, representa uma recorrência no mesmo processo: o nó 2 é resolvido a partir da resolução das submetas *isort([1],E)* (nó 3) e *insert(2,[1],C)* (nó 6) e assim por diante. Dessa maneira, as sub-árvores identificam as partes do processo. Por exemplo, a subárvore cuja raíz é o nó 10 representa a solução para a meta do nó 10.

A figura 5.13b mostra a árvore de execução para a meta *qsort([2,1],X)*, que usa o método *quicksort* de ordenação e corresponde à base de dados a seguir:

```
qsort([H|T],S):-split(H,T,C1,C2),
                        qsort(C1,S1),
                        qsort(C2,S2),
                        append(S1,[H|S2],S).
qsort([],[]).
split(H,[H1|T1],[H1|C1],C2):-H1=<H, split(H,T1,C1,C2).
split(H,[H1|T1],C1,[H1|C2]):-H1>H, split(H,T1,C1,C2).
split(_,[],[],[]).
append([],X,X).
append([X|Y],Z,[X|W]):-append(Y,Z,W).
```

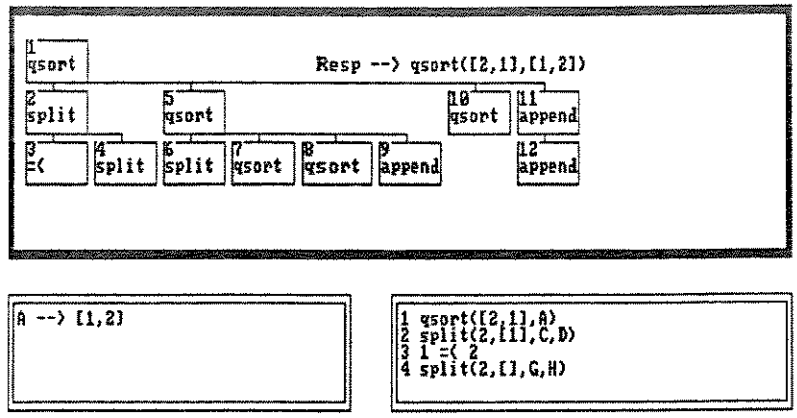


Figura 5.13b Representação Final da Árvore de Execução para *qsort([2,1],X)*

Na figura 5.13b, a meta original *qsort([2,1],X)* é a raíz da árvore, representada pelo nó 1. A meta é resolvida a partir da resolução de suas sub-metas, representadas por suas quatro sub-árvores que têm como raízes as sub-metas *split(2,[1],C1,C2), qsort(C1,S1), qsort(C2, S2) e append(S1,[2|S2],S)* (nós 2, 5, 10 e 11, respectivamente).

A subárvore cuja raiz é o nó 2 é resolvida e associa a C1 a lista [1] e a C2 a lista []. A sub-árvore cuja raíz é o nó 5 representa a recorrência no processo para a lista [1] *(qsort([1],S1))*. O nó 10 representa a sub-meta *qsort([],S2)*. O nó 11 representa a solução para a sub-meta *append([1],[2|[]],[1,2])*.

A visão global do processo, reconstruído na estrutura da árvore de execução pode possibilitar ao novato a exploração das idéias e conceitos discutidos anteriormente, como recursão, eficiência da solução, os processos de unificação e retrocesso, a meta-análise do processo subjacente à solução.

5.4 Discussão

O uso de recursos gráficos na mídia do computador tem sido cada vez mais frequente, exemplificado nas interfaces icônicas e nas propostas de linguagens de programação visuais, à medida em que a tecnologia da computação gráfica suporta essas tendências.

Várias motivações podem ser encontradas na literatura para o uso de representações visuais de informações. Tem sido reconhecido, cada vez mais, o papel da imagem (visual) no pensamento. As pessoas parecem absorver muito mais rapidamente informação visual que informação verbal (O'Shea, 1983). Em resolução de problemas o formalismo gráfico é uma ajuda efetiva, mesmo sem considerarmos a mídia do computador. Dois exemplos clássicos são o uso de desenho para provar teoremas em Geometria e notações como diagramas de Venn para problemas em Teoria de Conjuntos (O'Shea, 1983). Relações conceituais importantes podem ser codificadas no formalismo visual de maneira a fazê-las perceptualmente salientes, diminuindo a carga cognitiva e dessa maneira permitindo lidar com o problema em um nível mais alto (Shniederman, 1987, citado em Holland, 1991).

Além disso, do ponto de vista computacional, representações gráficas podem favorecer a própria atividade de programar. A linguagem Logo é um exemplo bem sucedido de como o uso de formalismos gráficos favorece o entendimento de conceitos, não apenas geométricos, mas do próprio paradigma procedural de programação (Baranauskas, 1991a). Smalltalk é outro exemplo de linguagem que possibilita ao usuário criar e modificar objetos computacionais sob mediação de um ambiente visual.

O uso do formalismo gráfico no design das ferramentas objetiva dar condições ao usuário de pensar em um nível mais alto sobre sua representação do problema (programa) e sobre o que é simulado (execução do programa).

A notação do diagrama semântico, por exemplo, permite organizar a informação contida na base de dados, a nível de cláusula. Através de relacionamentos codificados visualmente pode ser mais fácil reconhecer, por exemplo, uma cláusula "errada". Esse *feedback* é especialmente importante para novatos enquanto estão aprendendo a expressar proposições através do formalismo clausal, como veremos mais adiante no decorrer deste trabalho.

A nível de base de dados, o diagrama semântico pode mostrar múltiplos relacionamentos de um objeto, de forma concisa. No formalismo clausal, um dado objeto aparece tantas vezes quantas ele estiver relacionado com outros objetos e as cláusulas que os contém não estão necessariamente agrupadas. A representação dos objetos repetidos, através de um único nó no diagrama leva a uma síntese da informação contida na base de dados, possibilitando a identificação de situações de "inconsistência".

A árvore de espaços de busca põe o usuário em contacto com a estrutura hierárquica de metas que é usada pela "máquina de inferência" a partir de uma base de dados, para responder perguntas. Nesse processo são explicitados, através do formalismo visual, conceitos envolvidos tanto a nível de linguagem (como por exemplo unificação) como a nível do programa sendo representado (como por exemplo definição recursiva de predicado).

O uso de múltiplas representações ou visões do conhecimento em resolução de problemas foi o segundo aspecto básico considerado. O simbolismo gráfico é proposto nas ferramentas como um complemento e não uma substituição para a notação clausal tradicional. A variação de contexto gerada pelas diferentes representações possibilita ao usuário olhar o problema sob diferentes pontos de vista. Multiplicidade é uma característica importante em ambientes de aprendizagem segundo a abordagem construtivista, na medida em que essa multiplicidade facilita ao novato o uso e teste de suas próprias estruturas de conhecimento. Além disso, se a

questão da aquisição de conhecimento é dependente do contexto*[6] como tem sido discutido na literatura, essa transferência deveria ser "aprendida". A multiplicidade de situações pode promover uma articulação de conhecimentos e, dessa maneira, oferecer um "treino" para a transferência (TECFA, 1990).

As ferramentas propostas, colocadas para o novato de forma integrada ao sistema Prolog, pretendem gerar um "meio" onde ele possa construir o conhecimento do paradigma da linguagem para representação de soluções de problemas. Situamos as ferramentas computacionais que compõem o ambiente sendo proposto neste trabalho, segundo a concepção de Pea de tecnologia cognitiva baseada em computador (Pea, 1985), enquanto podem possibilitar mudanças na própria tarefa de programar.

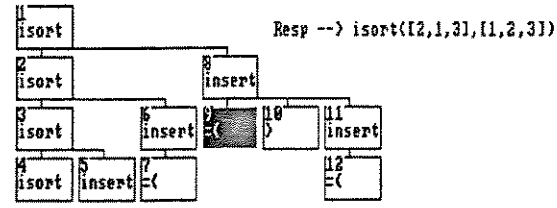
*[6] a questão da transferência do conhecimento em relação ao uso educacional de linguagens de programação é discutida em (Johanson, 1988).

Capítulo 6

O Módulo Operacional

Capítulo 6

O Módulo Operacional



Neste capítulo apresentamos o modelo proposto para representação da máquina de inferência de Prolog e as ferramentas que compõem o Módulo Operacional de nosso conjunto de ferramentas.

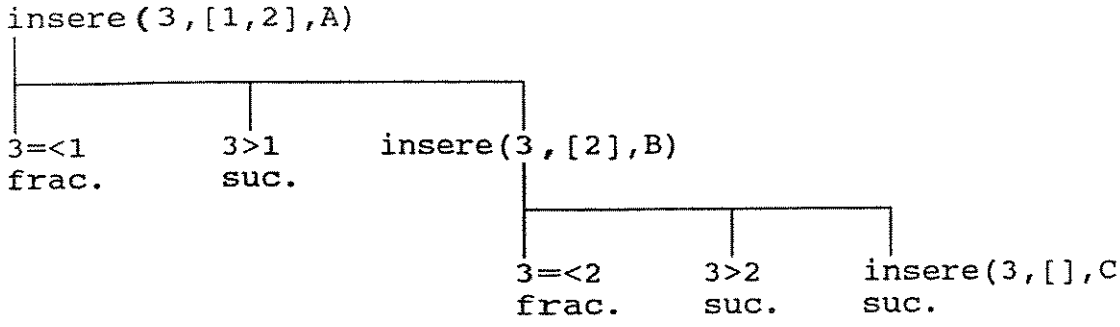
Semântica atribui significados aos programas. O Módulo Operacional (MOP) é constituído de ferramentas que tratam aspectos relativos à semântica operacional de um programa Prolog. Conforme descrevemos no capítulo 3, o significado operacional de um programa Prolog P, é dado pelo processo usado por um interpretador, que chamaremos "máquina de inferência de Prolog" (ou apenas "máquina de inferência", para simplificar), ao responder perguntas com base em P. Representar a semântica operacional do programa Prolog envolve, em nossa proposta, mostrar a máquina de inferência em ação.

6.1 A Representação Proposta para o Modelo de Execução de Prolog

Para representar a máquina de inferência de Prolog, propomos um modelo que tenta mostrar o processo de reduções de metas bem como a estrutura de árvore da execução, como uma extensão à "arvore de prova". A árvore gerada, que chamaremos *árvore de espaços de busca*, é construída dinâmicamente, refletindo as ações da "máquina de inferência" no processo de responder a uma pergunta, ou seja, tentar construir uma "árvore de prova" para a meta. Ao final do processo, em caso da computação terminar com sucesso, a árvore de prova é parte da árvore de espaços de busca que foi construída.

Definimos uma *árvore de espaços de busca* (ou árvore de busca para simplificar) como uma árvore cujos nós e arestas representam as metas que a máquina de inferência tenta reduzir durante a computação. A raíz da árvore de busca é a própria meta (pergunta). Os nós da árvore são metas cuja redução é tentada pela máquina de inferência e que podem ser nós "sucesso" ou "fracasso". Um nó "sucesso" representa uma meta reduzida e um nó "fracasso" representa uma meta cuja redução fracassou. Existe uma aresta dirigida ligando cada nó a uma meta derivada do nó (sub-meta), no processo de computação. Metas conjuntivas são tratadas como "raízes irmãs", ou seja, estão no mesmo nível hierárquico, a exemplo das sub-metas de uma dada meta.

A seguir apresentamos um exemplo da árvore de espaços de busca, ao final de seu processo de construção (figura 6.2), para resposta à pergunta (meta) *insere(3,[1,2],X)*, considerando o mesmo programa descrito no capítulo 3, que define a inserção de um elemento em uma lista ordenada.



A=[1,2,3]

B=[2,3]

C=[3]

Figura 6.2 Árvore de Espaços de Busca para a meta *insere(3,[1,2],A)*, ao término do processo de execução

Em cada passo de redução de metas, é mostrada, também, a cláusula do programa que unifica com a meta, sob o unificador. Por exemplo, o terceiro passo de redução de metas mostra:

Meta: insere(3,[1,2],[1|B])

Regra: 2 --> insere(3,[1,2],[1|B]) :- 3>1, insere(3,[2],B).

As ferramentas do Módulo Operacional objetivam mostrar ao usuário a máquina de inferência em ação, através da construção da árvore de espaços de busca de uma meta em uma base de dados.

6.2 Descrição Funcional das Ferramentas que Compõem o Módulo Operacional

O Módulo Operacional é constituído por um conjunto de ferramentas que representam a semântica operacional de programas Prolog. Como tal, ele explora aspectos relacionados ao comportamento da máquina de inferência de Prolog.

Nossa abordagem envolve mostrar o processo de execução combinando duas representações diferentes: uma mais "local" que chamaremos *trace estendido* (ou *trace*) e outra mais "global", que chamaremos *árvore de espaços de busca* (ou árvore de busca).

A representação local (*trace*) mostra as informações que são processadas pela máquina de inferência num passo de redução de uma meta. Através da investigação local, o usuário tem acesso à meta que Prolog está tentando reduzir, à cláusula do programa que está sendo aplicada, ao resultado da unificação e qual é a próxima meta a ser verificada. A cada iteração, no processo de computação, o usuário pode escolher entre continuar nesse modo (*trace*), mudar para o modo global (árvore de busca) ou conhecer o resultado da interpretação da meta inicial. Exemplificando, a figura 6.3 mostra um dos passos da iteração da máquina de inferência, no processo de responder à pergunta *insere(3,[1,2],A)*, do exemplo anterior.

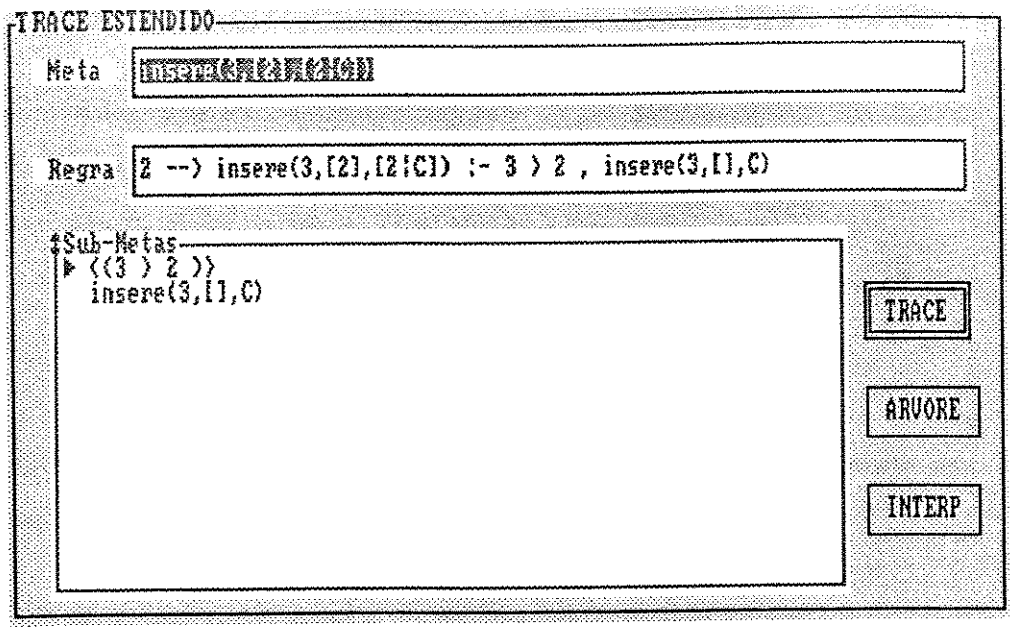


Figura 6.3 Tela da opção Trace Est. mostrando um dos passos da iteração da máquina de inferência

A figura 6.3 mostra o processo de redução de metas quando o resolvente (meta sob o unificador) é *insere(3,[2],[2|C])*, a cláusula do programa que unifica com o resolvente (regra 2), o resultado do processo de unificação e o novo resolvente (3>2, insere(3,[],C)), com indicação da próxima meta (sub-meta 3>2) a ser tentada na próxima iteração.

A representação global (árvore de busca) mostra a estrutura hierárquica de reduções de metas tentadas até o momento em questão da computação. A representação utilizada é uma implementação gráfica da árvore de espaços de busca, definida anteriormente. Através da investigação global, o usuário tem acesso à construção passo a passo do mecanismo de execução de Prolog.

A representação de árvore situa a meta que está sendo tentada, no contexto estrutural da execução. Exemplificando, a figura 6.4 mostra a árvore de espaços de busca correspondente ao momento da computação mostrado na figura 6.3.

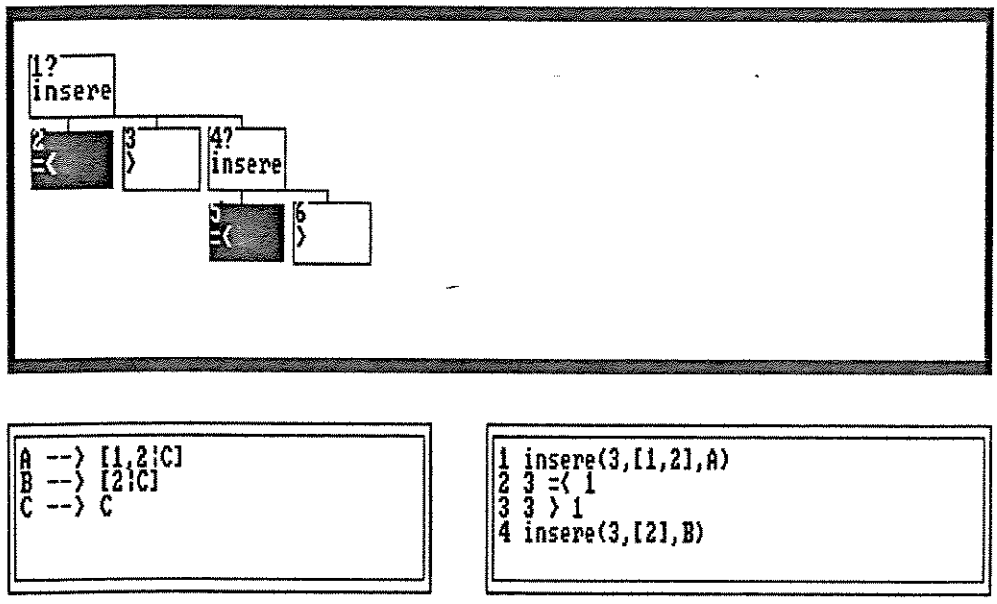


Figura 6.4 Tela da opção Arv. Busca mostrando graficamente a representação do processo de execução até o momento correspondente ao mostrado na figura 6.3

A figura 6.4 mostra a sequência dinâmica de metas tentadas (numeradas dentro do retângulo e mostradas de forma expandida na janela inferior à direita), valores correntes das variáveis (janela inferior à esquerda), metas que fracassaram (retângulos com cores invertidas: 2 e 5), metas que tiveram sucesso (retângulos 3 e 6), metas ainda não "resolvidas" (simbolizadas pela "?" nos retângulos 1 e 4), meta mais recente sendo tentada (retângulo 4), nível hierárquico entre metas (responder à meta 1 depende de responder à meta 4, por exemplo).

Existe uma relação entre as metas pai (meta) e filhos (sub-metas) de um dado nó com uma certa cláusula no programa, onde o nó pai representa a cabeça da cláusula e os nós filhos o corpo da cláusula (ex. os nós 1, 3 e 4 representam uma instância da cláusula insere(A,[B|C],[B|D]):- A>B, insere(A,C,D)).

A seguir apresentamos um exemplo completo das ferramentas mencionadas (sequência de figuras 6.5 e figuras 6.6), mostrando passo a passo as ite-

rações da máquina de inferência para o exemplo sendo discutido. A sequência de figuras 6.5 mostra, passo a passo, a construção da árvore de busca para a meta *insere(3,[1,2],A)*.

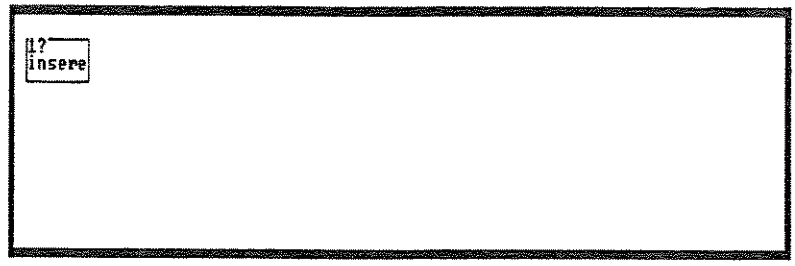


Figura 6.5 Aplicação da regra *insere(A,[B|C],[A,B|C]):-A=<B*

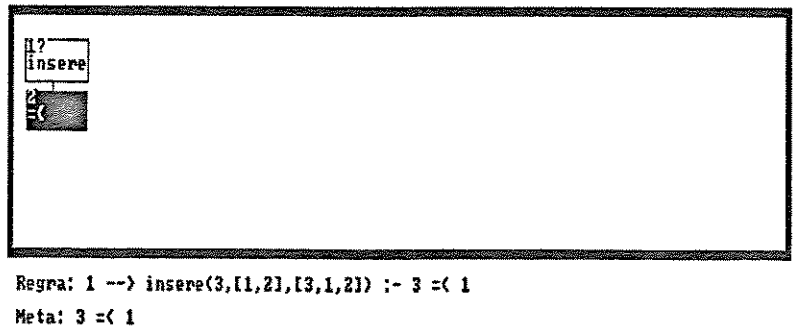


Figura 6.5a Falha da sub-meta *3=<1* (nó 2)

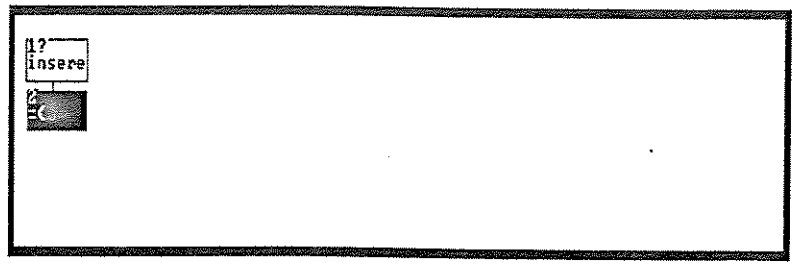


Figura 6.5b Retrocesso e aplicação da regra *insere(A,[B|C],[B|D]):-A>B, insere(A,C,D)*

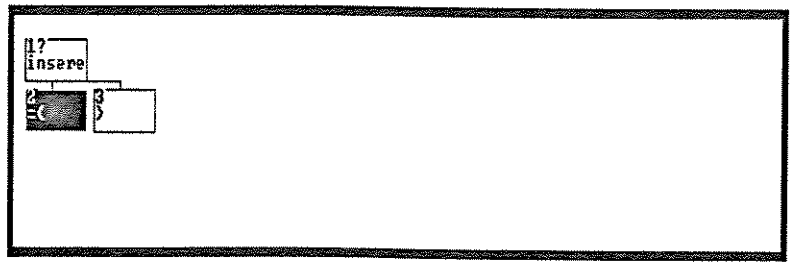


Figura 6.5c Sub-meta *3>1* sucede (nó 3)

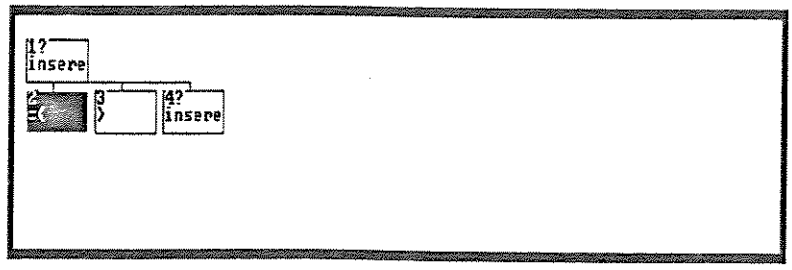
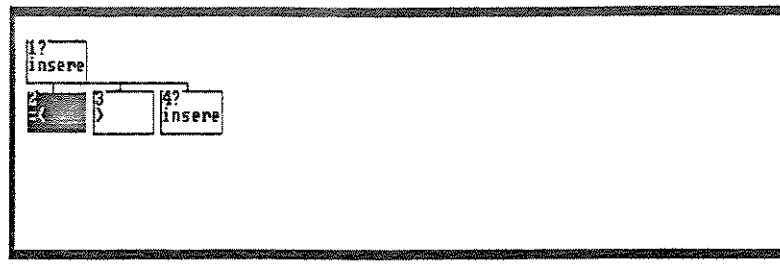
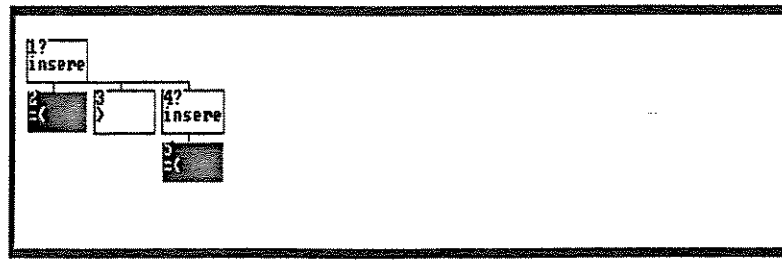


Figura 6.5d Nova sub-meta: *insere(3,[2],B)*



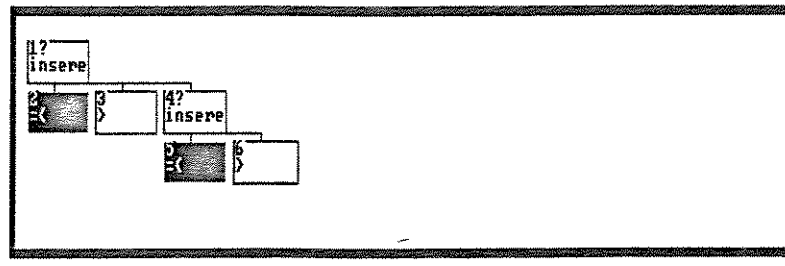

```
Regra: 1 --> insere(3,[2],[3,2]) :- 3 =< 2
Meta: 3 =< 2
```

Figura 6.5e Aplicação da regra 1 à meta representada no nó 4




```
Regra: 1 --> insere(3,[2],[3,2]) :- 3 =< 2
Meta: 3 =< 2
```

Figura 6.5f Falha da sub-meta *3*=<*2* (nó 5)




```
Regra: 2 --> insere(3,[2],[2|C]) :- 3 > 2 , insere(3,[],C)
Meta: 3 > 2
```

Figura 6.5g Retrocesso e aplicação da regra 2 à meta representada no nó 4

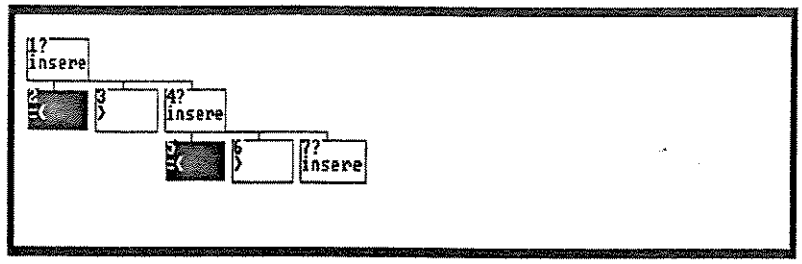


Figura 6.5h Sub-meta do nó 6 sucede (*3>2*). Nova sub-meta: *insere(3,[],C)*

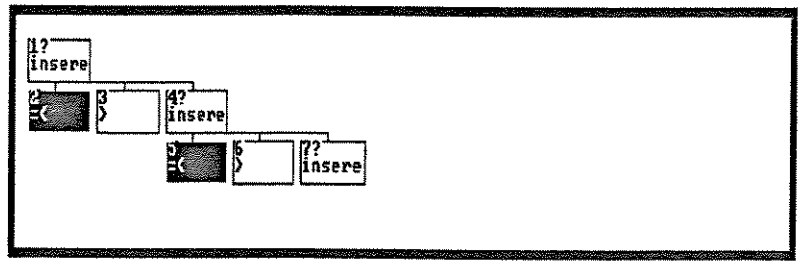


Figura 6.5i Aplicação da regra 3 (*insere(A,[],[A])* à meta representada no nó 7

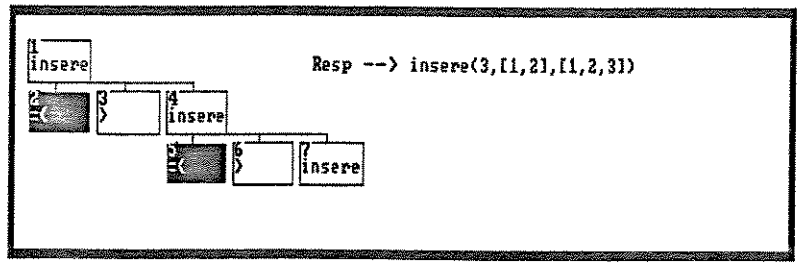


Figura 6.5j Meta representada no nó 7 sucede e a máquina chega à resposta da meta original

Em qualquer momento do processo, as janelas de predicados e de variáveis podem ser acionadas para mostrar os valores correntes das variáveis. A figura 6.5x corresponde ao momento do processo representado na figura 6.5e e mostra os valores associados às variáveis A e B, quando foi aplicada à meta *insere(3,[2],B)* a regra 1.

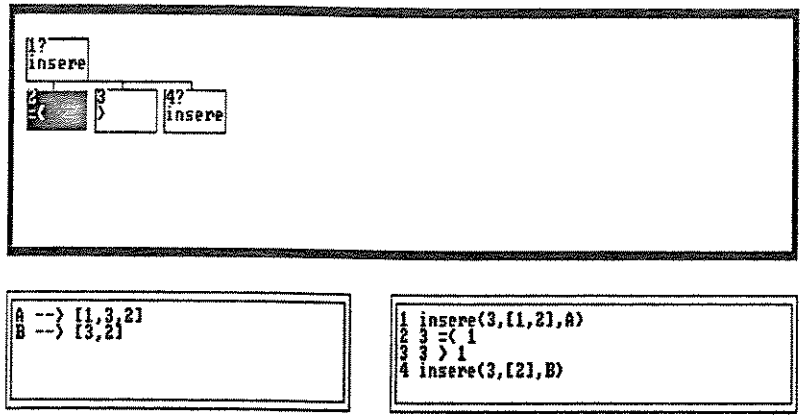


Figura 6.5x Momento do processo equivalente ao da figura 6.5e com as janelas de predicado (canto inferior à direita) e de variáveis (canto inferior à esquerda) acionadas

A figura 6.5y representa o passo seguinte, quando a falha no nó 5 levou a máquina a desfazer a associação de A e B a seus valores e levou a nova associação, como resultado da aplicação da nova regra (regra 2) à meta.

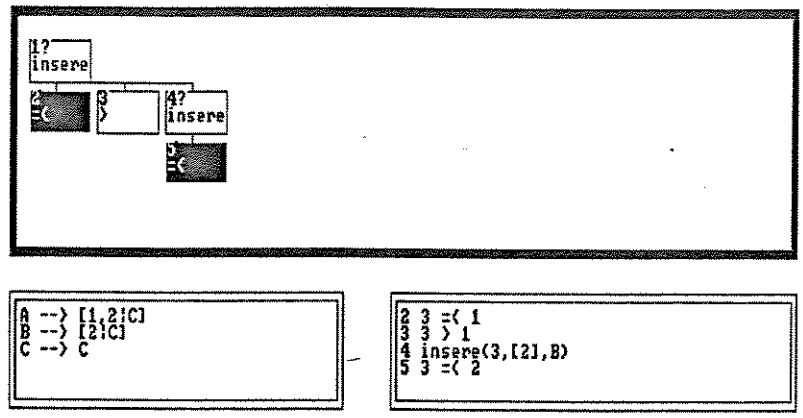


Figura 6.5y Efeito do retrocesso (após falha no nó 5), sobre as variáveis A e B

A mesma sequência de execução pode ser vista de modo local, através do *trace* estendido, como ilustra a sequência de figuras 6.6, a seguir.

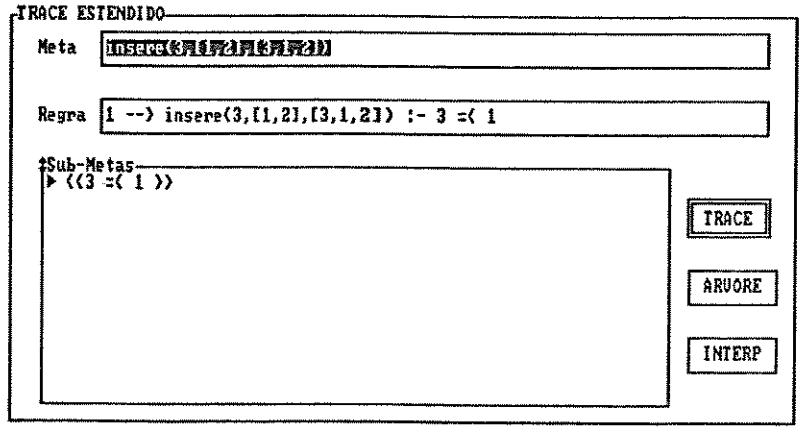


Figura 6.6 Meta (sob o unificador), e cláusula do programa após unificação (regra 1)

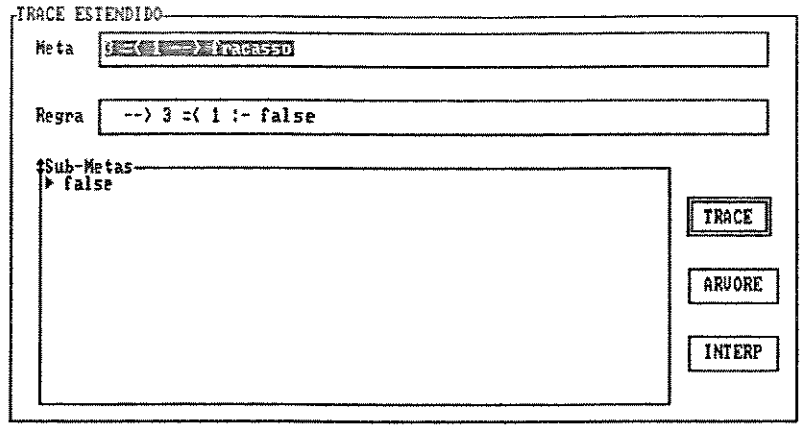


Figura 6.6a Resultado da tentativa de redução de *3=<1*

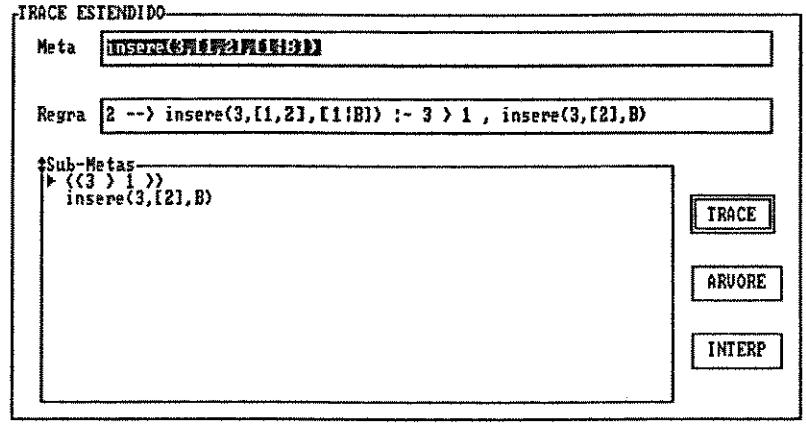


Figura 6.6b Após retrocesso, unificação da meta com nova cláusula no programa (regra 2)

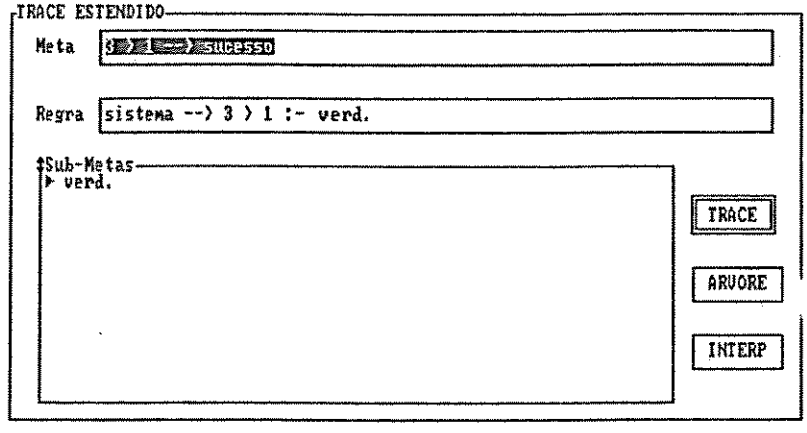


Figura 6.6c Resultado da redução de *3>1*

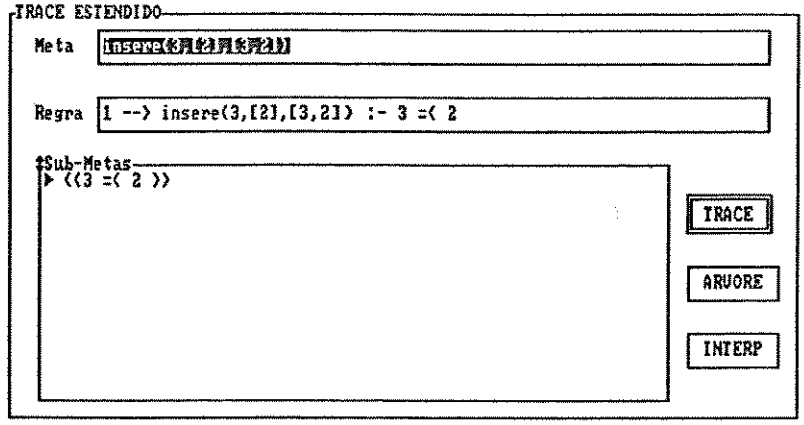


Figura 6.6d Unificação da nova sub-meta (*insere(3,[2],B)*) com cláusula do programa (regra 1)

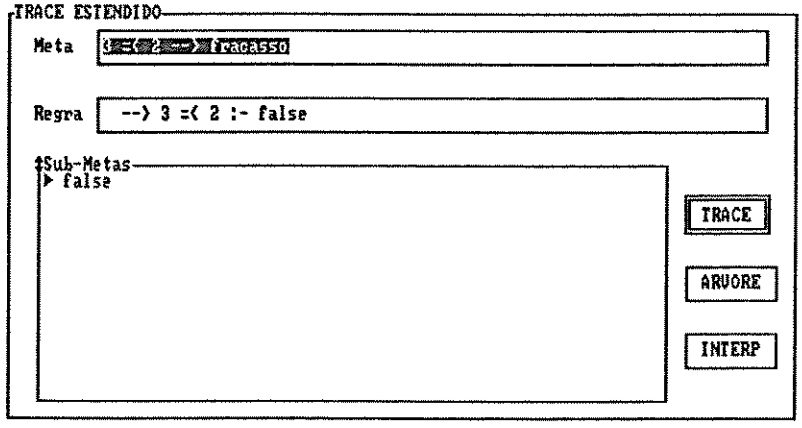


Figura 6.6e Resultado da tentativa de redução de *3=<2*

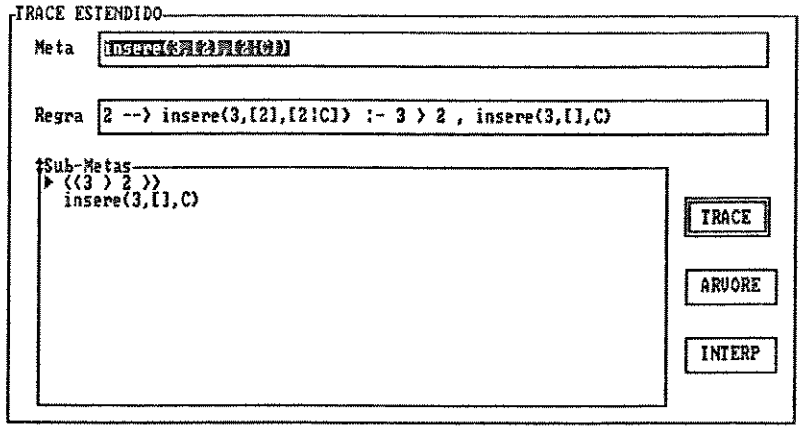


Figura 6.6f Após retrocesso, unificação da sub-meta *insere(3,[2],B)*, com nova cláusula no programa (regra 2)

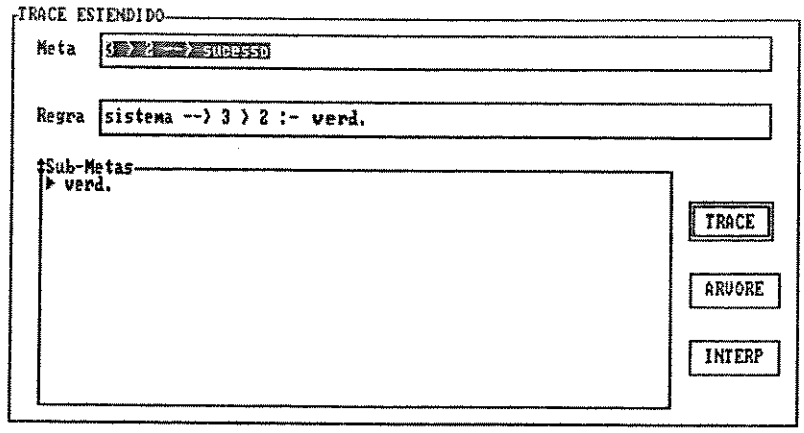


Figura 6.6g Resultado da tentativa de redução de *3>2*

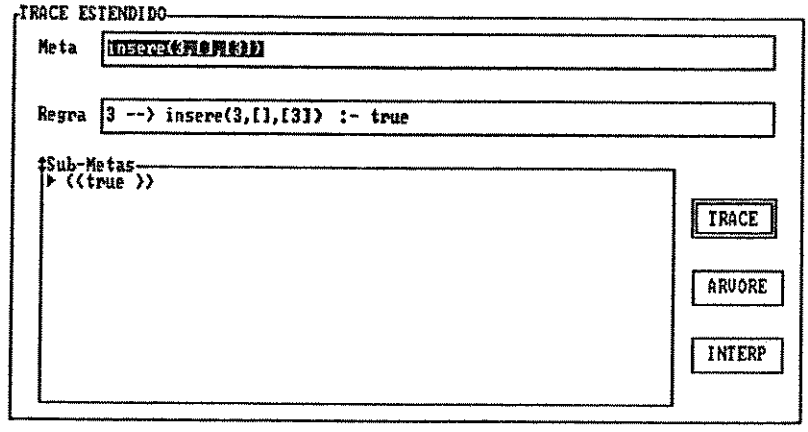


Figura 6.6h Nova sub-meta (*insere(3,[],C)*), após unificação com cláusula do programa (regra 3)

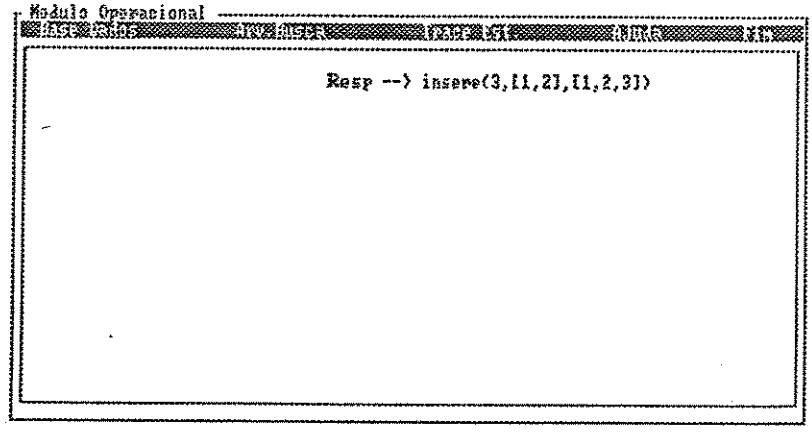


Figura 6.6i Resposta à meta inicial

6.3 O Ambiente Gerado pelo Módulo Operacional

Na seção anterior descrevemos os modos local e global propostos para representação do mecanismo de execução de Prolog. Nesta seção, descreveremos a maneira como o usuário pode interagir com essas representações e outras facilidades contidas no ambiente gerado pelo Módulo Operacional(MOP).

6.3.1 Ambiente Inicial e Facilidades Básicas

A interação com o MOP é essencialmente via seleção de menus. O MOP é acionado no ambiente do sistema operacional, a partir do comando *mop*, que resulta na tela mostrada a seguir, na figura 6.7.

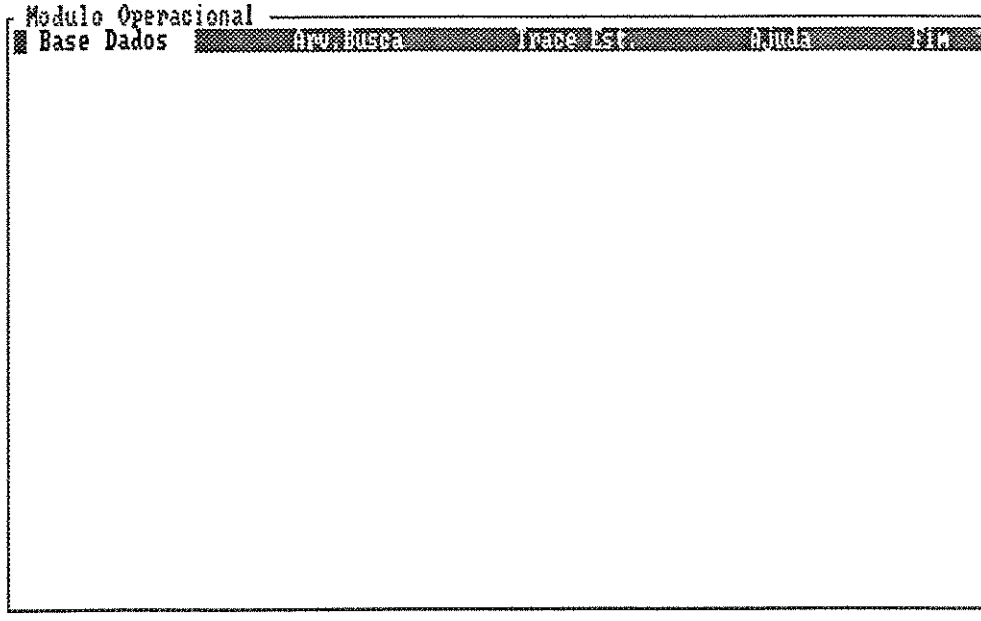


Figura 6.7 Tela inicial do MOP, com menu de opções

A seleção da opção **Base de Dados** dispara um novo menu com as opções **Carregar, Listar** e **Apagar** (figura 6.8).

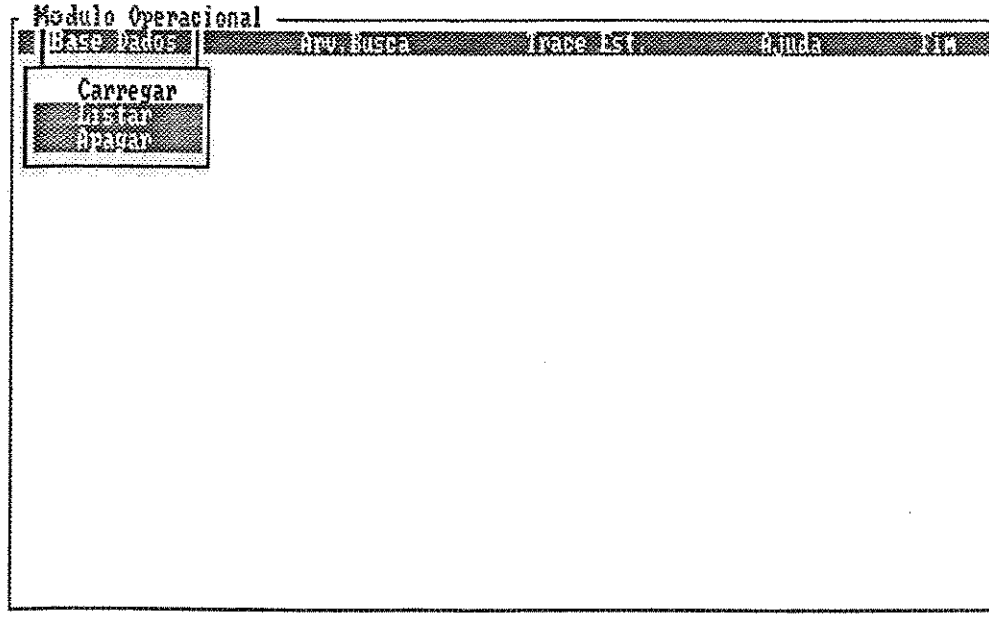


Figura 6.8 Tela de MOP com opções da seleção de Base de Dados

A opção **Carregar** "consulta"[*1] uma base de dados, disparando uma janela de seleção de arquivos nos moldes da janela de seleção do sistema Arity Prolog. Nessa janela, o nome do arquivo que contém a base de dados a ser carregada pode ser teclado no campo especificado ou pode ser selecionado a partir da lista com os nomes de arquivos apresentada, movendo-se as teclas do cursor (tab, seta para cima e seta para baixo), como indica a figura 6.9.

[*1] termo usado para carregar um arquivo de programa para a área de trabalho (memória) do sistema Prolog.

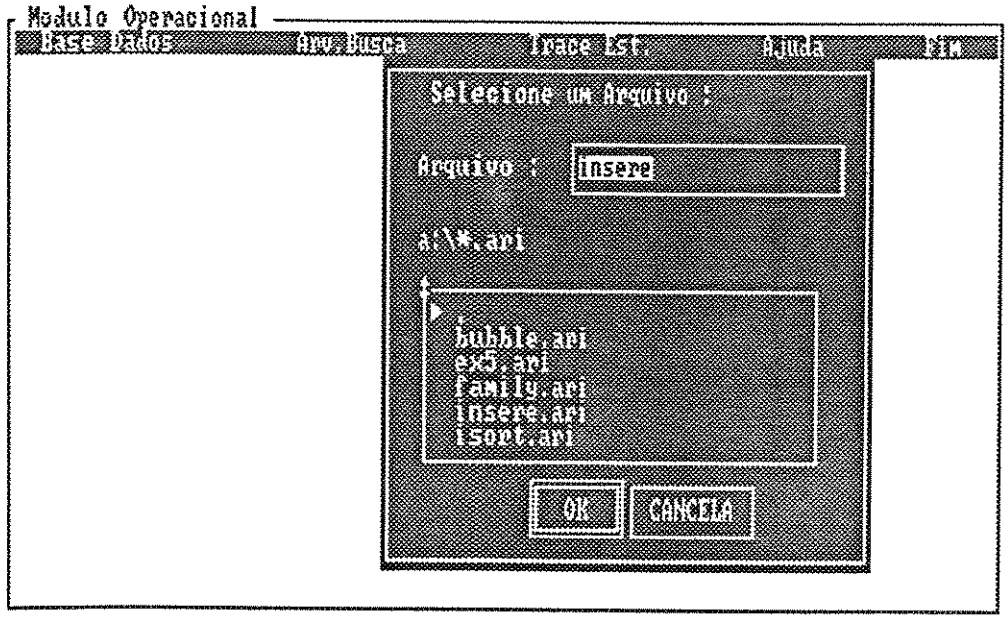


Figura 6.9 Tela de MOP para seleção de arquivo de programa

A opção **Listar**, após seleção de arquivo idêntica à da opção Carregar, aciona uma nova janela que mostra a base de dados contida no arquivo selecionado (figura 6.10). Acionando-se a tecla ESC ("escape"), essa janela é apagada e volta-se ao menu principal.

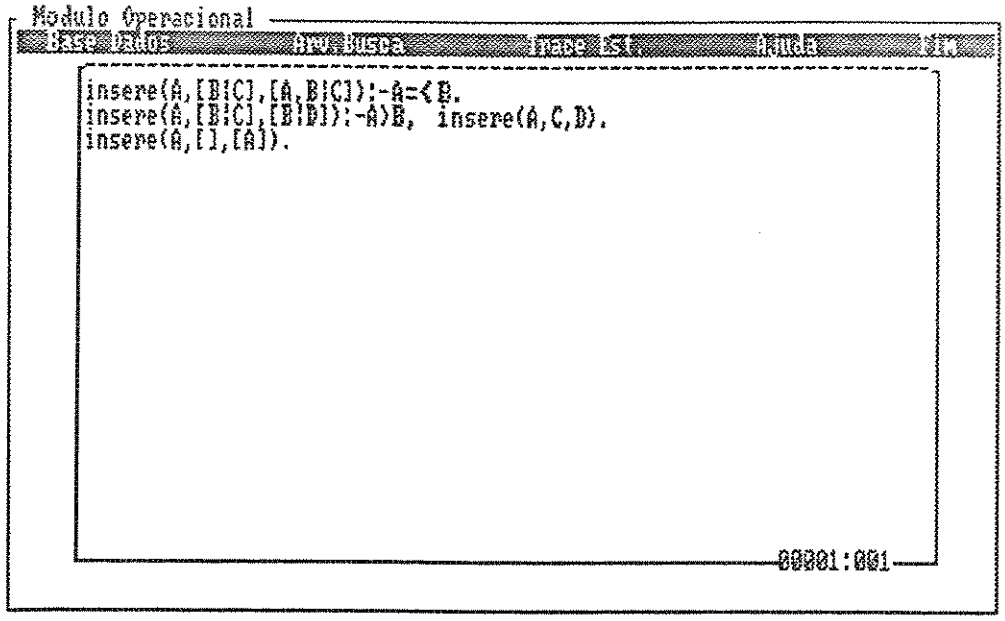


Figura 6.10 Tela de MOP com listagem de programa

A opção **Apagar** elimina da área de memória um ou mais programas previamente consultados, pedindo confirmação do usuário.

Após ação correspondente à seleção de cada opção, é apresentado o menu principal.

A seleção da opção **Ajuda** abre uma janela com acesso rápido a informações sobre o uso das ferramentas que compõem o Módulo Operacional. A figura 6.11 mostra janela de ajuda com parte das informações de ajuda.

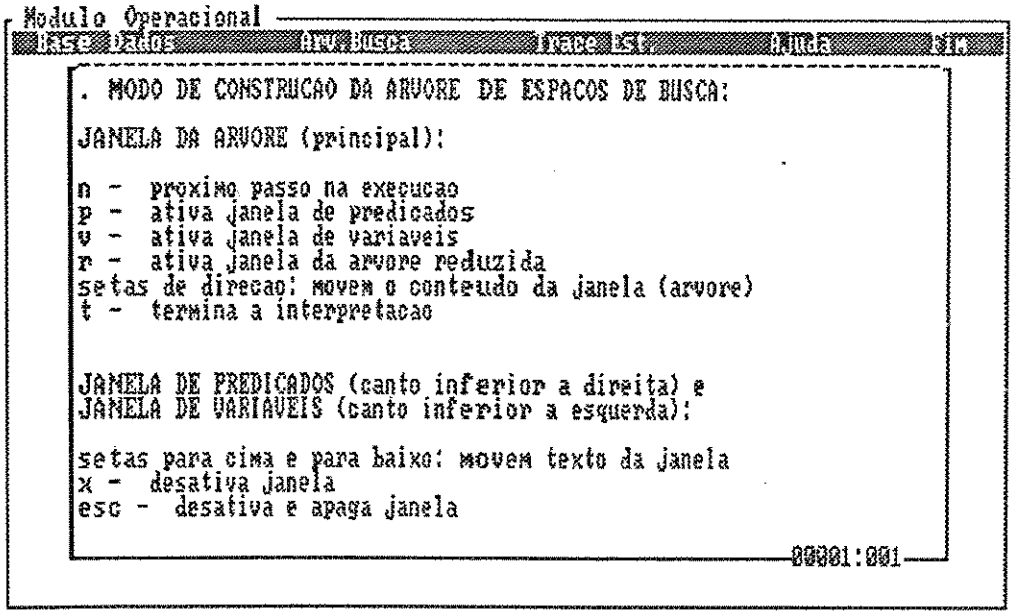


Figura 6.11 Tela de MOP com informações sobre utilização da ferramenta

A opção **Fim** finaliza o uso do Módulo Operacional, retornando ao ambiente Prolog.

6.3.2 O Ambiente da Árvore de Busca

A seleção da opção **Arv. Busca** aciona uma nova janela para que o usuário entre com a meta que deseja investigar (figura 6.12).

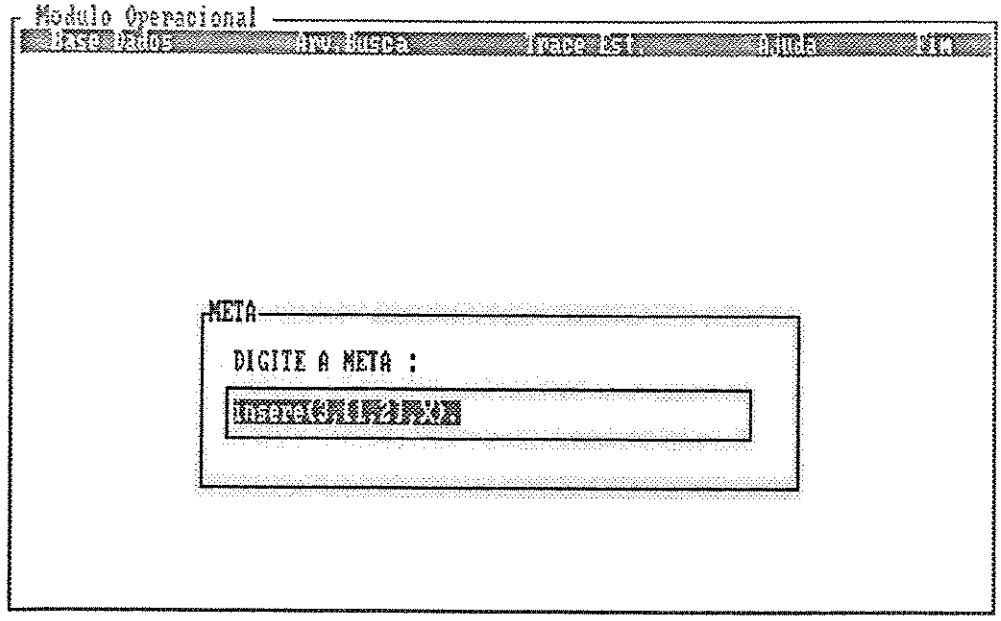


Figura 6.12 Tela de MOP para entrada da meta

A partir de então, cada passo do processo de construção da árvore de espaços de busca é representado numa janela (jarv), sob controle do usuário. Abaixo da janela são mostrados o número da regra que foi aplicada à meta, o resultado da unificação e a próxima meta a ser resolvida.

Os nós da árvore possuem um número (que chamaremos "número dinâmico"), no canto superior esquerdo, que representa a ordem em que a meta é reduzida no processo de computação e o nome do predicado que corresponde a meta. Cada nó apresenta, também, um indicador se a meta ainda não foi resolvida (símbolo "?"), se fracassou (nó com cores "invertidas") ou se foi resolvida com sucesso (ausência de "?"). A figura 6.5c mostra essas situações nos nós 1, 2 e 3 respectivamente.

A representação das informações contidas na árvore é complementada por duas outras janelas: a janela de predicados (apresentada no canto inferior à direita na tela), e a janela de variáveis (apresentada no canto inferior à esquerda na tela), conforme mostram as figuras 6.5x e 6.5y.

A janela de predicados (jclau), acionada pela tecla "p", mostra de forma estendida (completa) o conteúdo dos nós da árvore, associados pelo seu número dinâmico.

A janela de variáveis (jvar), acionada pela tecla "v", mostra o estado das variáveis (com respectivos valores associados, se é o caso) no momento em questão do processo de execução.

Os conteúdos das janelas de predicados e de variáveis podem ser movimentados através das teclas de cursor (para cima e para baixo).

A árvore de espaços de busca também pode ser movimentada usando-se as teclas do cursor (para cima, para baixo, para direita e para esquerda), para possibilitar a visualização de partes da árvore que não caberiam no espaço da janela principal (figura 6.13).

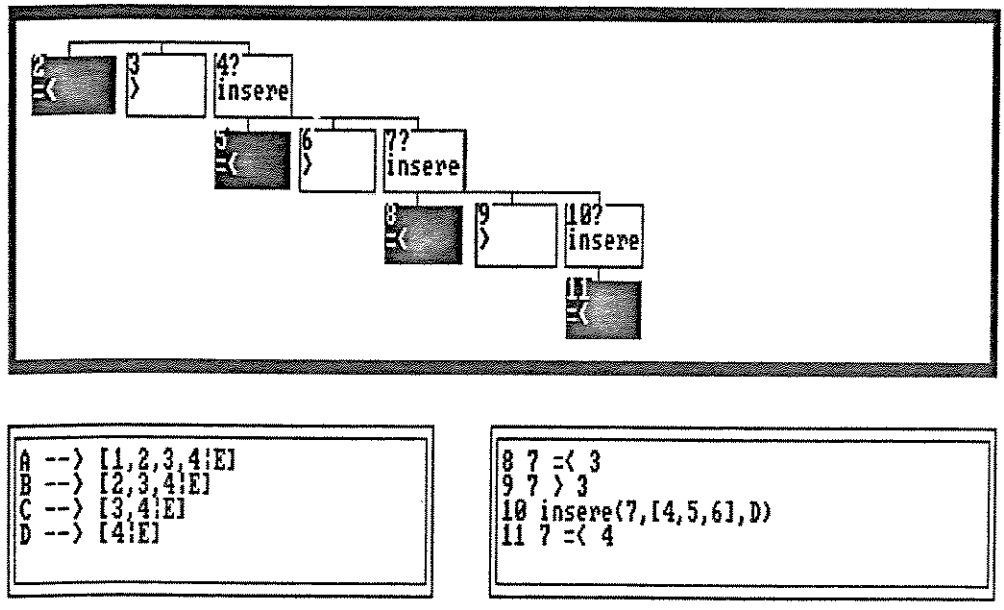


Figura 6.13 Parte de uma árvore de busca após movimentação para cima

Uma quarta janela pode ser acionada, a partir da janela da árvore, para que o usuário não perca a visão global da estrutura da árvore de espaços de busca, em ca-

sos onde ela cresce muito: a janela da árvore reduzida, ativada pela tecla "r". A janela da árvore reduzida (jared) ocupa a região inferior da tela, em lugar das janelas de predicados e de variáveis e mostra a estrutura completa da árvore de espaços de busca. Nessa estrutura, um cursor pode ser deslocado (através das teclas de cursor), para seleção da região da árvore que o usuário deseja ver expandida na janela principal. A sequência de figuras 6.14a, 6.14b e 6.14c mostra respectivamente a árvore reduzida com com região correspondente à árvore sendo mostrada, em destaque, a árvore reduzida com uma região selecionada e a janela principal após expansão da região escolhida.

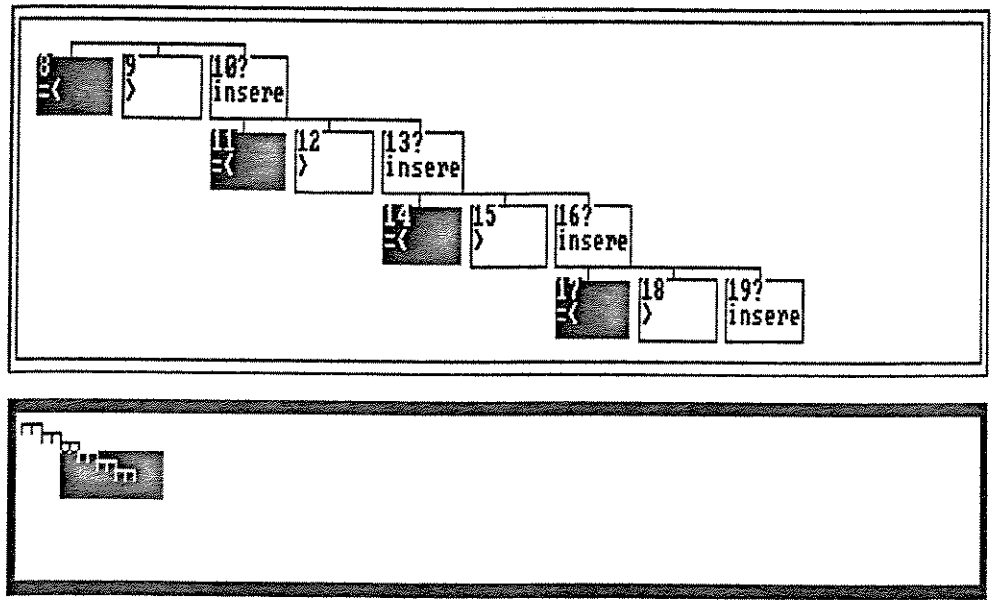


Figura 6.14a Região da árvore mostrada na janela maior, em destaque na janela da árvore reduzida

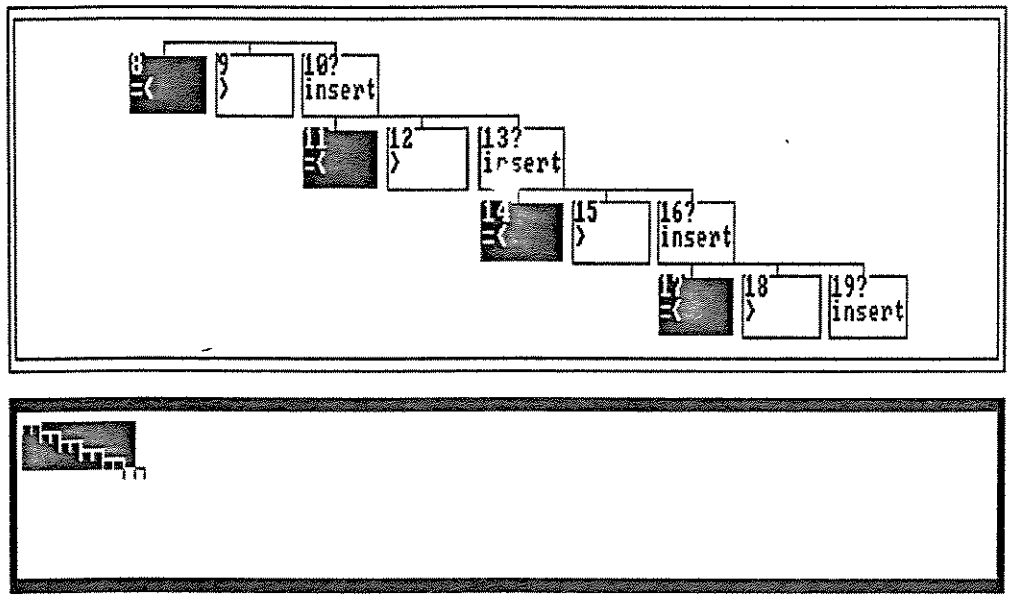


Figura 6.14b Seleção de nova região da árvore, através de deslocamento da região em destaque na árvore reduzida

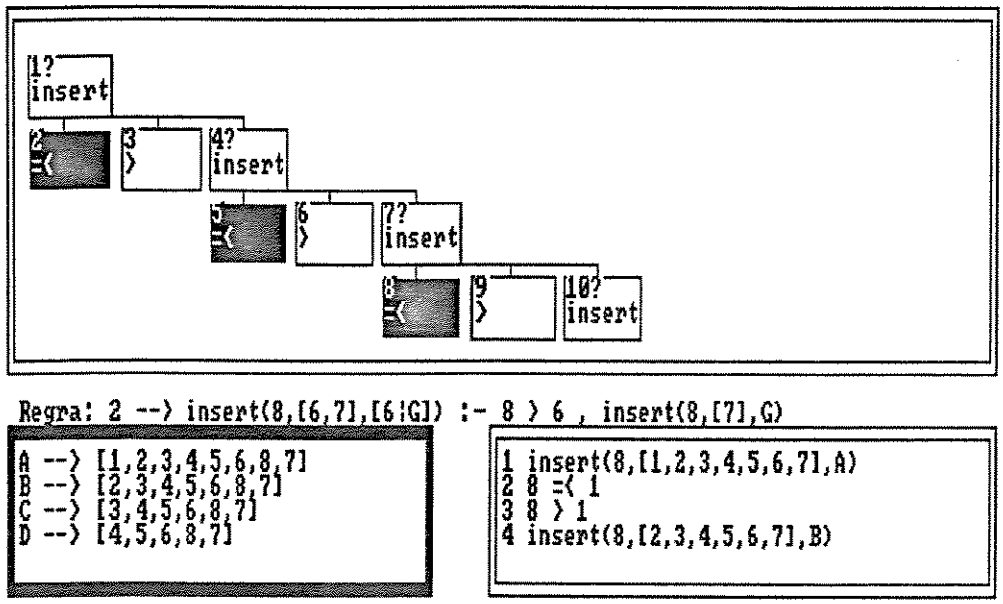


Figura 6.14c Resultado da seleção de nova região da árvore (mostrada na figura 6.14b)

A cada momento, uma das quatro janelas está "ativa"; isto é, responde a seus respectivos comandos. A janela ativa é indicada com bordas preenchidas como no caso da janela reduzida na figura 6.14a. A janela principal está inicialmente ativa e torna-se ativa com a desativação de qualquer das outras janelas (jclau, jvar, jared). As janelas jvar e jclau são desativadas ao se teclar "x" (mantém a janela) ou ESC (apaga a janela). A janela jared é desativada após seleção de uma de suas partes.

O mecanismo de construção da árvore de espaços de busca é mostrado, portanto, através das janelas apresentadas, passo a passo, até a obtenção da "resposta" da máquina de inferência à pergunta inicial (meta). Após apresentação da resposta (figura 6.5j), o usuário pode voltar ao menu principal (teclando "enter"), ou pode acionar o mecanismo de retrocesso da máquina de inferência e continuar a busca por outras respostas, teclando ";"*4.

A qualquer momento, o processo pode ser interrompido (através das teclas ctrl-brk) e pode-se retornar ao menu principal.

6.3.3 O Ambiente do Trace Estendido

Ao selecionar a opção **Trace Est.**, é apresentada ao usuário uma janela para que ele entre com a meta a ser investigada, como no caso da opção **Arv. Busca** (figura 6.12).

Após entrada da meta, a tela mostra, através dos campos "meta", "regra" e "sub-metas", informações locais a cada momento do processo de redução de metas.

A figura 6.6b mostra um momento da computação exibido pela janela do trace estendido. O campo "meta" mostra a cláusula objetivo após unificação com uma cláusula da base de dados. O resultado da computação de uma determinada

*4 como é feito quando se usa o Arity Prolog ao nível do interpretador.

sub-meta é mostrado, também, no campo "meta", como indicado na figura 6.6a (Meta: 3=<1 --> fracasso).

O campo "regra" mostra a cláusula da base de dados após unificação com a cláusula objetivo e uma indicação de qual regra da base de dados foi utilizada (através de numeração). Quando a meta envolve predicados do sistema Prolog, o resultado da computação de tais metas é mostrado no campo "regra", como indicam as figuras 6.6a (Regra: sistema --> 3=<1 :- falso), 6.6c, 6.6e, 6.6g.

O campo sub-metas mostra, em sequência, as próximas metas que serão tentadas no processo de redução de metas.

A cada momento do processo de redução de metas, o usuário pode continuar na opção do trace estendido (sinalizado como *default*), ou selecionar um dos outros campos "*arvore*" ou "*interp*". Os campos de opção aparecem à direita na janela do trace estendido e podem ser selecionados movimentando o cursor através da tecla TAB.

A seleção da opção "*arvore*" muda a representação para o ambiente da construção da árvore de espaços de busca, mostrando a árvore construída até o momento em questão.

A seleção da opção "*interp*" resume a interpretação da meta inicial (cláusula objetivo) e fornece a resposta a essa computação.

A qualquer momento do proceso o usuário pode ver o texto de seu programa, pressionando a tecla de função F2 ou pode obter informações sobre o uso da ferramenta, pressionando a tecla de função F1.

Ao término do processo de redução de metas, a resposta da computação é mostrada ao usuário que pode escolher, como no modo da árvore de busca, voltar para o menu principal ou continuar a busca por outras possíveis respostas.

6.4 Aspectos de Implementação do Módulo Operacional

A implementação das ferramentas que compõem o Módulo Operacional (MOP) foi feita usando uma extensão do Arity Prolog, que inclui a biblioteca de rotinas gráficas APGRAPH (APGRAPH, 1990). APGRAPH é uma biblioteca de primitivas gráficas para o Arity/Prolog v. 5.1. Consiste de um conjunto de predicados implementados em Prolog e em Microsoft C v. 5.1. Através destes predicados, um programa escrito inteiramente em Prolog pode acessar as rotinas da biblioteca gráfica GRAPHICS.LIB e outras rotinas de apoio implementadas em C.

O MOP age como um intermediário entre o programa do usuário e Prolog, usando o conceito de meta-programas: programas que analisam, transformam ou simulam outros programas, tratando-os como dados. Meta-programação é facilitada em Prolog, dada a equivalência entre programas e dados (ambos são "termos" Prolog).

O MOP tem como base a implementação de um meta-interpretador que constitui uma classe especial de meta-programas. Meta-interpretadores possibilitam a criação de um ambiente de programação integrado e acesso ao processo computacional da linguagem. O conceito de meta-interpretador deve-se a Steele e Sussman (1978) e a sugestão de que meta-interpretadores poderiam ser a base para um ambiente de programação foi feita por Shapiro (1983).

A seguir, apresentamos um meta-interpretador bastante simplificado, para exemplificar como ele reflete o modelo de computação proposto por Prolog:

```
interp(true).
interp(Meta1,Meta2) :- interp(Meta1), interp(Meta2).
interp(Meta) :- clause(Meta,Cauda), interp(Cauda).
```

A semântica operacional do meta-interpretador pode ser explicada da seguinte maneira:

" A meta vazia, representada em Prolog pelo átomo *true* está resolvida (cláusula 1). Para resolver a conjunção de metas (Meta1,Meta2), resolva Meta1 e em seguida Meta2 (cláusula 2). Para resolver uma meta, escolha uma cláusula do programa, cuja cabeça unifica com a meta e recursivamente resolva o corpo da cláusula (cláusula 3). "

A seleção da meta mais à esquerda como a meta a reduzir é garantida pela ordem das sub-metas na cláusula que trata metas conjuntivas (segunda cláusula). Busca sequencial e retrocesso para a escolha não determinística da cláusula a usar para reduzir a meta é garantido pelo comportamento de Prolog ao satisfazer a meta "*clause*".

Os meta-interpretadores podem ser caracterizados em termos de sua "granulação"; isto é, dos níveis de informação que são tornados acessíveis à aplicação. Em nossa aplicação, o meta-interpretador deve captar os processos de associação de valores a variáveis e o efeito do retrocesso.

6.4.1 Descrição do Sistema que Implementa o Módulo Operacional

O sistema que implementa o MOP está dividido nos seguintes módulos:

- Principal
- Interface com o Usuário
- Meta-Interpretação
- Controle de Variáveis
- Controle de Janelas Gráficas

- Utilitário

O Módulo Principal (M.P.) gerencia o sistema e utiliza os outros módulos para atender ao usuário quando da interpretação de um programa.

O Módulo de Interface (M.I.) é usado para receber pedidos, pedir informação e mostrar resultados ao usuário.

O Módulo de Meta-Interpretação (M.M.I) é o responsável pela interpretação do programa do usuário. Ele utiliza o Módulo de Controle de Variáveis e fornece ao Módulo Principal os dados necessários para construção da árvore de espaços de busca (ou "árvore de execução" para simplificar).

O Módulo de Controle de Janelas (M.C.J.) é usado pelo Módulo Principal para mostrar ao usuário, através de janelas gráficas ou de texto, a árvore de execução, o conteúdo dos nós da árvore, o valor das variáveis do programa num momento em questão, uma representação reduzida da árvore, etc.

O Módulo de Controle de Variáveis (M.C.V.) é usado pelo Módulo de Meta-Interpretação para manter atualizadas as associações de valores a variáveis.

O Módulo Utilitário (M.U.) é constituído da definição de predicados de uso comum aos outros módulos.

A figura 6.15 mostra um diagrama do inter-relacionamento dos principais módulos que compõem o sistema desenvolvido.

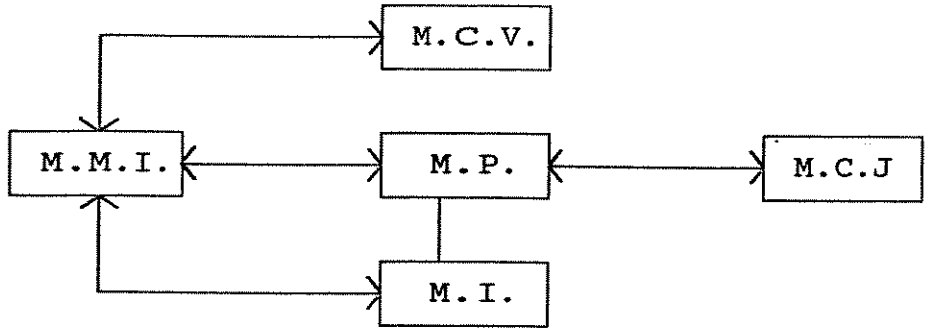


Figura 6.15 Diagrama do relacionamento entre módulos do sistema

Arquivos de dados e de programas que compõem os módulos do sistema:

DIALDEFO.ARI: contém a definição do menu principal (MO_MENU) e as definições dos *dialog boxes* DIALDIR, DIALMETA, DIALTRAC E DIALHELP.

DIALCTRL.ARI: contém as definições dos controles dos *dialog boxes* DIALDIR, DIALMETA, DIALTRAC e DIALHELP.

MOP.ARI: contém o núcleo controlador da interface.

INTERP.HLP: contém as informações de ajuda no uso do sistema.

CTRL_VAR.ARI: contém o MCV.

CTRL_WND.ARI: contém o MCJ.

MPRINC.ARI: contém o MP.

MINTERP.ARI: contém o MMI.

UTIL.ARI, UTIL1.ARI e DESENHO.CGA: arquivos que compõem o MU.

6.4.1.1 O Módulo de Interface com o Usuário (MI)

Para facilitar a integração das ferramentas no sistema Prolog e manter a homogeneidade no seu uso, a interface implementada utiliza os mesmos recursos da interface do Arity Prolog, que é baseada em menus e *dialog boxes*.

Os menus providos pelo Arity Prolog permitem o projeto e gerenciamento de menus específicos para cada aplicação. Especifica-se os conteúdos e atributos de tela de um menu, através de definições armazenadas em arquivos.

O *dialog box* provido pelo Arity Prolog é um tipo especial de janela que pode conter certos tipos de "controles" para realizar várias funções. O *dialog box*, como o nome indica, tem como objetivo sustentar um "diálogo" entre o sistema e o usuário, através do qual o usuário pode fazer certas escolhas ou fornecer determinada informação.

Tanto menus, quanto *dialog boxes* são baseados na arquitetura de trocas de mensagem. Gerencia-se as ações associadas a seleções de menus e ações resultantes de seleção de um controle particular em um *dialog box*, através do uso de mensagens.

O Módulo de Interface contém as seguintes definições de menu e *dialog boxes*:

MO_MENU:

Definição do menu principal. Através desse menu o usuário pode determinar qual programa deseja carregar, listar ou eliminar, em que modo de representação deseja ver a execução de seu programa (árvore de busca ou trace estendido), tem acesso a informações de ajuda e à saída do módulo.

DIALDIR:

Dialog box utilizado polo MO_MENU para escolha do programa a carregar ou listar. A escolha pode ser feita teclando-se o nome do arquivo desejado ou selecionando-o a partir da lista de arquivos do diretório corrente, que é mostrada num dos campos do *dialog box*.

DIALMETA:

Dialog box usado para entrada da meta a ser resolvida.

DIALTRAC:

Dialog box usado para representar cada passo do processo de redução de metas. DIALTRAC possui campos para mostrar: meta (cláusula objetivo após unificação com regra da base de dados), regra da base de dados (usada na unificação) e próximas sub-metas. Permite também selecionar outro tipo de representação (Árvore de Busca ou Interpretação), mudando o foco de seus controles.

DIALHELP:

Dialog box usado para mostrar informações de ajuda ao usuário. Através desse *dialog box* são mostradas informações de uso do sistema e, também, o texto do programa sendo interpretado.

Os arquivos que compõem a interface contém as definições do menu principal, as definições dos *dialog-boxes*, o núcleo controlador da interface e funções auxiliares.

6.4.1.2 O Módulo de Controle de Variáveis (MCV)

O MCV é usado pelo Módulo de Meta-Interpretação para manutenção de uma tabela com todas as variáveis existentes em determinado momento da execução. Para tal, o MCV fornece a lista das variáveis que aparecem numa regra, associa um nome às variáveis e fornece uma cópia da regra onde as variáveis já estão trocadas pelos nomes correspondentes.

Exemplo de utilização:

transforma((a(_0038,_0040) :- b(_0038),_0040 == _0038), Regrast, Lvar, Lvarst).

Regrast= (a("A","B") :- b("A"), "B" == "A"),

Lvar= (_0038,_0040),

Lvarst= ["A","B"].

6.4.1.3 O Módulo de Controle de Janelas (MCJ)

As janelas (*windows*) disponíveis no Arity Prolog são recursos disponíveis apenas para modo texto. Em nossa aplicação foi necessário o desenvolvimento de facilidades semelhantes para o modo gráfico.

A definição deste módulo foi inspirada no modelo de janelas existente para modo texto do Arity Prolog e no processo de trocas de mensagens usado nos *dialog boxes*.

Existe, portanto, um processo de trocas de mensagens entre janelas e o módulo que utiliza o MCJ. Quando, por exemplo, em uma determinada janela um comando é identificado, é enviada uma mensagem "*executa_comando(Janela,Controle)*". O sistema pode interceptar a mensagem, fazendo então o processamento que desejar, e/ou deixá-la ser tratada pelo MCJ, através da mensagem "*default_comando(Janela,Comando)*".

As janelas podem ser divididas em janelas gráficas e janelas gráficas de texto. As janelas são usadas apenas para leitura pelo usuário. A diferença entre janelas gráficas de texto e as janelas gráficas normais está no conteúdo delas. Dentro de janelas gráficas podem ser usados quaisquer dos atributos gráficos definidos no sistema APGRAPH, como por exemplo círculos, retângulos e retas. O uso de texto é permitido, mas não há movimento de texto (*scroll*) automático.

Nas janelas gráficas de texto, somente texto é permitido. A janela, em determinado instante, está mostrando apenas as linhas que cabem em seu espaço. O texto que está "fora" da janela é guardado. As setas de direção (para cima e para baixo) podem ser usadas para movimentar o texto desejado trazendo-o para "dentro" da janela.

Com o funcionamento semelhante ao de *dialog boxes*, quando da execução de um comando pela janela, uma mensagem "*executa_comando(Comando,Janela)*" é enviada de modo que os módulos que utilizam o MCJ possam interceptar as mensagens, caso necessário. Se a mensagem não é interceptada, é enviada uma mensagem "*default_comando(Comando,Janela)*".

O módulo que utiliza o MCJ deve definir quais comandos devem ser aceitos pela janela, como por exemplo:

comandos_aceitaveis(Janela,ListCom,ListComTrad).

Quando a Janela está ativa, caso o usuário pressione uma tecla presente em ListCom, uma mensagem "*executa_comando(Janela,ComTrad)*" é enviada pelo MCJ, onde ComTrad é o comando definido em ListComTrad.
Exemplo:
Considerando "*comandos_aceitaveis(jvar,[1,45,72,80],[esc,exit,up,down])*"
se jvar é a janela ativa e o usuário teclar "x", a mensagem
"*executa_comando(jvar,exit)*" é enviada. Outras teclas pressionadas, não presentes em ListCom são ignoradas.

A seguir apresentamos um modelo de como é feita a definição de uma janela gráfica pelo módulo que utiliza o MCJ (Módulo Principal):

Nome_da_Janela :-

```
        define_gwindow(Nome,Tipo,[L1,C1,L2,C2]),
        current_gwindow(Old,Nome),
        repeat,
        [!
        le_comando_valido(Nome,Com),
        executa_comando(Nome,Com)
        !],
        member(Com,[esc,exit]),
(Com == esc,hide_gwindow(_,Old); current_gwindow(_,Old)).
```

Adaptações são feitas a esse modelo, segundo as necessidades do módulo que utiliza o MCJ.

O MCJ é constituído, portanto, pela definição de predicados que implementam:

- Leitura de comandos para determinada janela.

Quando uma janela está ativa, é feita leitura do teclado. Essa leitura é "filtrada" de modo que somente um comando válido para aquela janela é aceito. Em seguida é enviada uma mensagem para execução do comando solicitado (*le_comando_valido, traduz_comando, valida, traduz, marca_control*).

- Primitivas para uso do MCJ.

As primitivas que tornam possível a manipulação de janelas gráficas por outros módulos podem ser classificadas pelo seu uso como primitivas de:

- definição e uso de janelas gráficas e janelas gráficas de texto por outros módulos (*draw_janela , desativa_janela , erase_janela , drawframe*).

- controle das janelas (*define_gwindow*, *current_gwindow*, *hide_gwindow*, *delete_gwindow*).

- escritura em janela gráfica e de texto (*jgwriteln, pula_linha_janela, jgwrite, acrescenovalinha, jgestouro, jgguarda, gwrite_window, novalinha, aumultlinha*).

- movimentação do conteúdo das janelas (*default_comando, move_janela, sobe_linha_janela, desce_linha_janela, escreve_janela*) .

6.4.1.4 O Módulo Principal

Utiliza as primitivas fornecidas pelo MCJ para definição e manipulação de quatro janelas gráficas:

- janela principal (jarv) onde é desenhada a árvore de espaços de busca.
- janela de predicados (jclau) onde são escritas as metas correspondentes a cada nó da árvore.
- janela de variáveis (jvar) onde são escritas as variáveis correntes e respectivos valores associados.
- janela da árvore reduzida (jared), onde é desenhada a estrutura completa da árvore de busca.

A Construção da Árvore de Busca

A construção da árvore de espaços de busca trata o caso em que a árvore não cabe completamente na janela principal (jarv). Dessa maneira, jarv mostra, a cada momento, a parte da árvore que cabe na janela. As outras partes podem ser vistas na janela, movendo a árvore com as setas de direção (para cima, para baixo, para a esquerda, para a direita), como se movessemos a janela de lugar. Para tal, mantemos

uma árvore-B que guarda todas as informações da parte visível da árvore de busca (dados dos nós que estão em uma posição dentro dos limites definidos pela posição atual da janela principal). A posição atual da janela é mantida numa tabela-Hash (*geral*) sob a chave *posição_janela_árvore*.

Dessa maneira, para desenhar e ligar os nós visíveis na tela usamos a árvore-B *arvore_da_tela*. Para fazer as ligações que aparecem na tela apesar de seu nó não ser visível, usamos outra árvore-B: *arvlin*.

Assim, quando precisamos de informações dos nós visíveis na tela, acessamos a árvore-B *arvore_da_tela*. Quando precisamos das posições dos pais de cada nó, acessamos apenas a árvore-B *arvlin*. Quando todas as informações sobre o nó forem necessárias, acessamos também a base de dados (*arvore*).

A Construção da Árvore Reduzida

A árvore reduzida apresenta, na janela jared, um desenho da estrutura completa da árvore de busca com uma indicação em destaque (mira) sobre a parte da estrutura que está sendo mostrada na janela principal (como encontramos em mapas). Para tal desenhamos um modelo reduzido da árvore de busca, representando apenas as ligações, onde são utilizados dados contidos na árvore-B *arvlin*.

A mira pode movimentar-se pela árvore reduzida para seleção de outras regiões a serem vistas na janela principal.

As Janelas de Predicados e de Variáveis

O controle do conteúdo da janela de predicados (jclau) e da janela de variáveis (jvar) é feito utilizando-se a tabela-Hash *textos_janelas* com chaves jclau e jvar respectiva-

mente. Para mostrar o conteúdo da janela de variáveis é utilizada, também, a tabela *table_var* armazenada na base de dados do sistema.

6.4.1.5 O Módulo de Meta-Interpretação

O Módulo de Meta-Interpretação (MMI) fornece os dados necessários para o Módulo Principal construir a árvore de espaços de busca, enquanto interpreta o programa do usuário.

Cada nó da árvore de busca representa uma meta a reduzir. Assim, cada nó tem como informação o nome do predicado e os parâmetros (constantes e variáveis) da meta. O meta-interpretador deve captar o momento de surgimento dos nós da árvore, quando as variáveis ainda não tem valor associado e seu valor final (após execução da meta).

Devem ser armazenados, então, como são os termos no momento de sua criação. Caso a meta seja interpretada com sucesso, como ficou o nó (valores associados às variáveis). O mesmo é feito para o caso de um casamento (*match*) da meta com uma cabeça de regra, para capturar o processo de unificação.

A árvore de espaços de busca, como uma representação dinâmica do mecanismo de execução de Prolog, leva-nos a considerar três resultados possíveis para interpretação de um nó: o sucesso, o fracasso e o fracasso no retrocesso. Um nó pode ser interpretado com sucesso e mostrado como tal. Contudo, a interpretação continua, podendo ocorrer um retrocesso que leva a resatisfazer essa meta, podendo novamente ocorrer sucesso ou fracasso. Assim, um nó sucesso pode levar a uma falha posterior e consequentemente a um retrocesso, produzindo nós que não pertencem ao caminho da solução final: são nós que pertencem a um caminho tentativa.

O meta-interpretador deve, então, observar todos os processos de associar e desassociar valores às variáveis e o efeito do retrocesso.

A seguir, mostramos uma versão simplificada do meta-interpretador, indicando os pontos onde esses aspectos são tratados.

```
interp(Meta):-
        ctr-set(0,1), interp(Meta,0).
interp(true,Prof) :- !.
interp((A,B),Prof) :- !, write(A), write(B), nl,
        interp(A,Prof),
        interp(B,Prof).
interp(A,Prof) :-
        functor(A,C,D),
        E=C/D,
        system(E),
%       cria nó inicial e em caso de backtracking sinaliza falha.
        cria_no(system),A,!,
        ctr_inc(0,Y),
%       mostra resultado no modo trace.
%       instanciação devida a chamada de primitiva Prolog
        mostra_suces_system(A,Prof).
interp(A,Prof) :-
        write(A),nl,
        ctr_inc(0,Y),
%       cria nó inicial e em caso de backtracking sinaliza falha no nó e no modo trace mostra fracasso
        cria_no(regra),
%       faz a busca na b.d. do usuário por regra a unificar.
        myclause(A,B),
```

```
%       instanciação devido a casamento da meta com a cabeça de regra e desfaz match em caso de
%       backtracking.
        mostra_match_regra(A,Prof),
%       mostra regra após unificação, no modo trace
        Prof1 is Prof+1,
        interp(B,Prof1),
%       elimina "?" do nó, indicando nó-sucesso.
        mostra_suces_regra((A:-B),Prof).
```

6.4.1.6 O Módulo Utilitário

O Módulo Utilitário (MU) é constituído de predicados auxiliares, de uso comum a outros módulos do sistema. Entre esses predicados estão os predicados de auxílio ao desenho das representações gráficas.

6.4.2 Estrutura de Dados Utilizada

Os dados manipulados pelos módulos que compõem o sistema são armazenados temporariamente em estruturas do tipo:

- Árvores-B
- Tabelas-Hash
- Base de Dados

Conceitualmente a árvore-B do sistema Arity Prolog é constituída de nós e folhas. Os nós contém informações para determinar através de qual ramo a busca deve seguir. Cada ramo pode conduzir a outro nó definindo um caminho de busca. O

nível mais fundo da árvore-B é constituído de folhas que são os termos que estão armazenados na árvore-B.

As árvores-B definidas e manipuladas pelo sistema são: *arvore_da_tela* e *arvlin*. A *arvore_da_tela* é utilizada pelo Módulo Principal para armazenar as informações da parte visível da árvore de busca.

Exemplos de definição e de armazenamento de termos na árvore:

defineb(arvore_da_tela,3,true,_),

O segundo parâmetro indica o grau dos nós e o átomo true indica a unicidade das chaves na árvore.

record(arvore_da_tela,Chavelin,[NumNo,A,PosNo,PosPai,PosFilhos,PosAnt,PosProx, Resp]),

O segundo parâmetro é a chave de ordenação e o terceiro parâmetro é o termo armazenado.

Chavelin é dado pela expressão (LinNo-1)*100+ColNo.

O termo é uma lista que contém:

NumNo: número dinâmico do nó.

A: functor do nó.

PosNo: posição (linha e coluna) do nó.

PosPai: posição (linha e coluna) do pai do nó em questão.

PosFilhos: posições dos filhos do nó em questão.

PosAnt: posição do nó anterior.

PosProx: posição do próximo nó.

Res: indicador do resultado da execução do nó.

A *arvlin* é usada no Módulo Principal e no Módulo de Meta-Interpretação para armazenar informações sobre as ligações entre os nós da árvore de busca.

Exemplo de definição de termos na árvore:

defineb(arvlin,Chavelin,[Nref|PosPai]).

Chavelin é dado pela expressão (LinNo-1)*100+ColNo.

Nref é o número de referência de determinado nó na base de dados.

PosPai é uma lista que contém a linha e coluna do pai do nó em questão.

As tabelas-Hash fornecem um meio de classificar um certo número de elementos em uma categoria específica. Consistem, portanto, de um número de categorias separadas chamadas *hash buckets* (números de referência que apontam para termos na base de dados).

As tabelas-Hash, definidas e manipuladas pelos Módulos de Controle de Janelas (MCJ) e Principal (MP) são: *geral, estados_janelas, posições_janelas, textos_janelas.*

Exemplos de dados armazenados nas tabelas-Hash definidas:

recordh(posições_janelas,Chave,[Tipo,L1,C1,L2,C2,Code]).

O segundo parâmetro é a chave de procura (corresponde a um "bucket" hash) e em nossa aplicação pode ser: jarv, jclau, jared, jvar.

O terceiro parâmetro é o termo armazenado na tabela, Tipo pode ser graf ou texto.

L1,C1,L2,C2 são as coordenadas da janela

Code representa o código em que a imagem da janela é gravada.

recordh(posições_janelas,jarv,[graf,2,3,16,80,_]).

recordh(estados_janelas,Chave,Termo).

Chave pode ser jarv, jclau, jared, jvar e o termo armazenado representa o estado da janela num determinado instante (criada, ativa, hide, inativa).

recordh(textos_janelas,Chave,[Antes,Dentro,Apos]).

Chave pode ser jclau e jvar e o termo armazenado é uma lista que representa respectivamente a parte de texto "acima" do texto sendo mostrado na janela (Antes), o texto visível (Dentro) e o texto "abaixo" da parte visível (Após).

recordh(geral,Chave,Termo)

Chave pode ser:

posição_janela_arvore e Termo uma lista com as coordenadas da janela;

janela_ativa e Termo o tipo da janela;

control_ativo e Termo um comando válido para a janela.

Programas Prolog podem fazer uso da própria base de dados para armazenar termos e cláusulas. Cada termo ou conjunto de termos é armazenado sob uma chave (nome pelo qual o termo é conhecido). O sistema Arity Prolog associa um número de referência para cada termo. Ao contrário do caso da chave, o número de referência é único para cada termo armazenado na base de dados.

O Módulo de Meta-Interpretação armazena termos na base de dados, sob a chave "*arvore*", que são usados posteriormente pelo Módulo Principal.
Exemplo de dado armazenado:

recordz(arvore,[NumNo,A,PosNo,PosPai,PosFilho,PosAnt,PosProx,Resp],Nref)

O primeiro parâmetro é a chave de procura, o segundo parâmetro é o termo armazenado e o terceiro parâmetro é o número de referência do termo na base de dados.

Cláusulas também podem ser armazenadas na base de dados e manipuladas por programas (Prolog provê um conjunto de predicados que "entendem" a sintaxe de cláusulas e permitem tal manipulação).

Exemplos de cláusulas definidas e manipuladas pelos Módulos de Interface e de Meta-Interpretação:

table_var(LvarSt,Lvar):

tabela de variáveis contendo nome da variável e seu valor.

meta(Meta)

onde Meta é a meta em execução (cláusula objetivo)

opção(Opção)

onde Opção é a opção de representação (*trace, busca ou interp*).

varst(RegraSt,Lvar):

tabela contendo uma Regra e sua respectiva lista de variáveis.

program(NameExt)

onde Nameext é o arquivo de programa do usuário.

modo(Tipo)

onde Tipo é *modografico* ou *modotexto*

ja_consultou(Arquivo)

para controle de arquivos já consultados.

6.5 Síntese

A proposta do Módulo Operacional é fornecer ao usuário uma representação da execução de Prolog baseada na construção pictórica da hierarquia de metas que representa as ações da máquina de inferência.

A idéia que norteia essa proposta é a de explicitar nas ferramentas as estruturas conceituais subjacentes da linguagem sendo aprendida. A árvore de espaços de busca permite explicitar, através do formalismo visual, conceitos tanto a nível de paradigma da linguagem (busca em profundidade, unificação, retrocesso)

quanto a nível do programa sendo representado (estrutura da solução, definições recursivas).

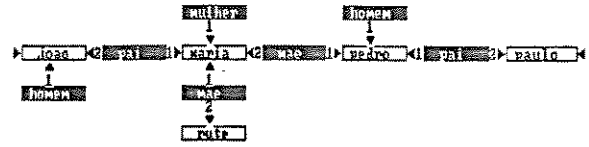
Acreditamos que as ferramentas do Módulo Operacional poderão possibilitar ao sujeito uma base apropriada sobre a qual ele possa pensar sobre os aspectos operacionais envolvidos na programação Prolog. Esse conhecimento de como a máquina virtual "funciona" é um dos componentes necessários para que o sujeito construa o modelo conceitual da linguagem que está aprendendo.

Capítulo 7

O Módulo Declarativo

Capítulo 7

O Módulo Declarativo



Neste capítulo tratamos o programa Prolog segundo uma abordagem declarativa. Apresentamos um modelo de representação para o programa, baseado em diagramas notacionais de rêdes semânticas, que chamamos "diagramas semânticos". Nosso objetivo é obter uma representação gráfica do "conhecimento" expresso nas proposições que constituem o programa do usuário. Em seguida, apresentamos as ferramentas que implementam o modelo de representação proposto e constituem o Módulo Declarativo.

No Módulo Declarativo o programa do usuário é tratado como um conjunto de proposições (cláusulas), que definem relações entre os objetos de certo domínio do conhecimento. Assim, as proposições definem afirmações verdadeiras (fatos) a respeito de objetos e relações, ou definem relações de implicação entre proposições (regras).

O objetivo do Módulo Declarativo é, portanto, criar uma representação que possibilite ao usuário concentrar as atenções no significado das relações

entre objetos definidos em seu programa, de forma independente de como essas relações e objetos são interpretados no processo de redução de metas. As ferramentas que constituem este módulo tratam, portanto, aspectos relativos à semântica declarativa de um programa Prolog.

A definição formal da semântica declarativa de um programa em lógica P, que é baseada no modelo teórico da lógica de primeira ordem foi incluída no apêndice 1. A seguir, apresentamos o modelo baseado na notação de rêdes semânticas, proposto para representação de programas Prolog, segundo a abordagem declarativa.

7.1 Modelo Proposto para Representação de Aspectos do Significado Declarativo de um Programa.

O modelo proposto baseia-se na idéia de se criar uma representação pictórica do "conhecimento" expresso nas proposições que constituem a base de dados do usuário (programa). Tal conhecimento está implícito nas inter-relações entre os objetos que estão presentes numa dada proposição (fato ou regra), ou nas inter-relações entre objetos comuns a várias proposições (excluídas as variáveis, que tem como escopo a proposição em que elas aparecem).

Nossa proposta é usar diagramas notacionais de rêdes semânticas, conforme descrito no capítulo 3, para representar o inter-relacionamento entre os objetos e relações presentes na base de dados, pretendendo, dessa maneira, explicitar esse tipo de "conhecimento" (o quê define a base de dados).

Nosso objetivo é usar rêde semântica como um diagrama para representar graficamente o conhecimento implícito nas proposições que formam a base de

dados do usuário[*2]. O texto de um programa (ou base de dados) Prolog, no Módulo Declarativo é tratado como sendo constituído de cláusulas que expressam dois tipos de conhecimento: factual e implicativo.

Nesta seção apresentamos a notação gráfica proposta, que chamamos "diagrama semântico" e sua correspondência com a notação clausal usada em Prolog.

7.1.1 Representação de Conhecimento Factual

O conhecimento factual está representado no formalismo Prolog pelas proposições assumidas verdadeiras sobre o domínio do problema e corresponde à cláusula unitária Prolog (ou fato). Representar a cláusula unitária Prolog, significa, portanto, representar uma fórmula atômica do tipo p(t1,t2,...tn), n>=0, onde t1,t2,...tn são termos Prolog e p é um átomo no papel de um símbolo predicativo n-ário. t1,t2,...tn correspondem aos objetos a serem representados e p corresponde à relação a ser representada.

Nossa notação diferencia o predicado (relação) de seus argumentos (objetos), pela representação gráfica correspondente. A cada objeto diferente corresponde um nó no diagrama semântico (representado pelo nome do objeto). A relação é representada num nó diferenciado (dentro de um quadrado). Uma aresta (seta) liga a relação a cada um dos objetos, como mostra a figura 7.1. Quando o número de objetos é maior do que um, existe uma numeração próxima a cada seta indicando a ordem do objeto, na cláusula. No caso de objetos repetidos, como no caso da figura 7.3, as setas que ligam a relação ao objeto repetido tem a numeração coincidente com a posição da primeira ocorrência do objeto.

[*2] Ennals sugere em seu livro texto (Ennals, 1983,p. 57) uma abordagem metodológica para a construção de relações familiares em Prolog, baseada no desenho de rêdes semânticas.

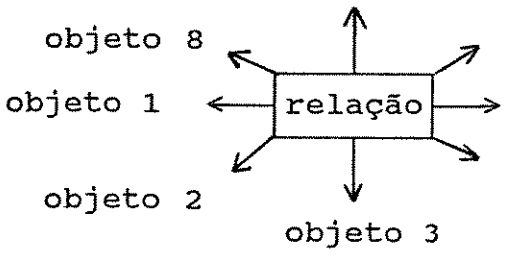


Figura 7.1 Representação de *relação(objeto1, objeto2, objeto3, ...objeto8)*

Exemplos de proposições atômicas são apresentadas a seguir, em sua forma clausal (notação Prolog) e no diagrama semântico proposto (figuras 7.2 e 7.3). A correspondência da notação apresentada com a notação clausal indica a semântica associada aos diagramas semânticos.

Proposição: "João deu o livro a Maria".

em Prolog: deu(joao,livro,maria)

no Diagrama Semântico:

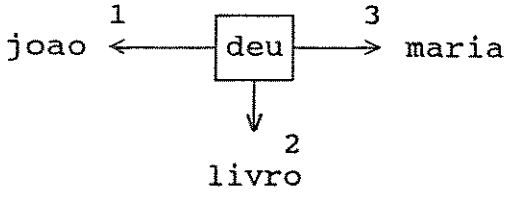


Figura 7.2 Exemplo de representação de cláusula unitária (fato) através de diagrama semântico

Proposição: "A concatenação de [] e X resulta X"

em Prolog: concatenação([],X,X).

no Diagrama Semântico:

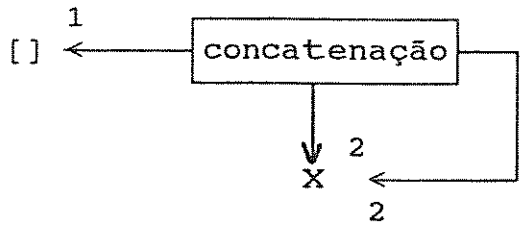


Figura 7.3 Exemplo de representação de cláusula unitária (fato) através de diagrama semântico

7.1.2 Representação de Conhecimento Implicativo

O conhecimento implicativo está representado, no formalismo Prolog, pelas regras de implicação entre proposições e corresponde à cláusula não unitária Prolog (ou regra).

A cláusula não unitária pode ser descrita como sendo constituída de uma parte antecedente e uma parte consequente, como indicado a seguir:

consequente :- antecedente

A parte consequente corresponde à cabeça da cláusula não unitária Prolog, sendo portanto uma fórmula atômica. A parte antecedente corresponde ao corpo da cláusula não unitária, sendo portanto uma conjunção de fórmulas atômicas.

Cada fórmula atômica presente na cláusula não unitária é representada conceitualmente de forma análoga à apresentada na seção 7.1.1.

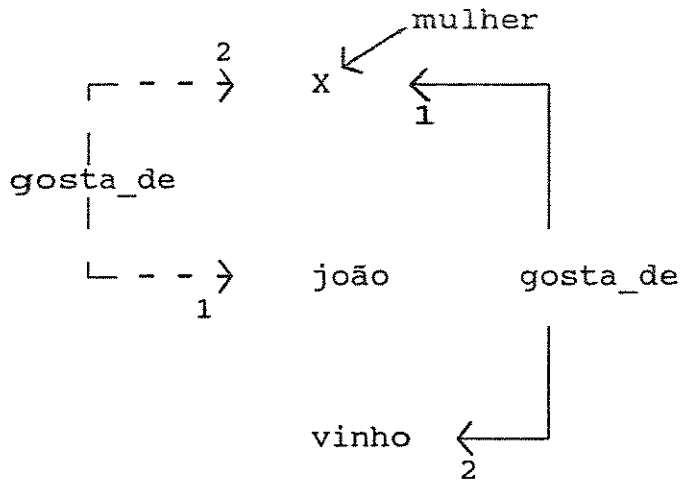
Representar a cláusula não unitária, envolvè manter, no diagrama semântico, a distinção entre sua parte antecedente e sua parte consequente, considerando-se, entretanto, uma representação única para objetos comuns às suas partes antecedente e consequente.

Na notação proposta os objetos são representados pelos termos Prolog que denotam. As relações rotulam as arestas e são representadas pelo predicado da fórmula atômica.

O tipo da aresta que liga a relação da parte consequente aos objetos (seta tracejada) e sua posição na tela (metade à esquerda), diferenciam as partes antecedente e consequente da cláusula.

Exemplos de proposições não atômicas são apresentados a seguir, em sua forma clausal (notação Prolog) e no diagrama semântico proposto (figuras 7.4 e 7.5).

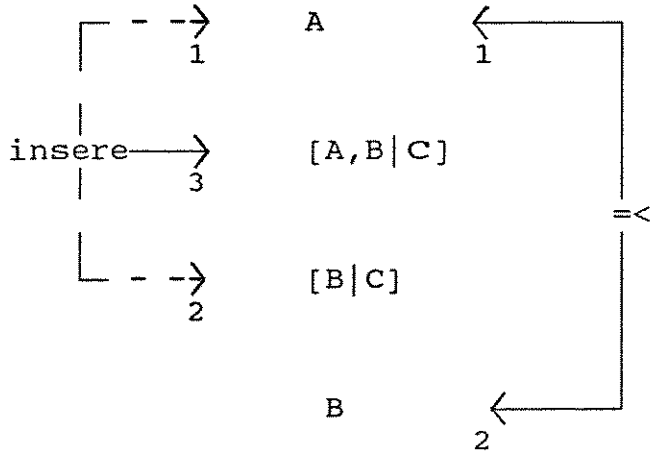
Proposição: "João gosta de mulheres que gostam de vinho"
em Prolog: gosta(joão,X):-gosta(X,vinho),mulher(X)
no Diagrama Semântico:



Proposição: "Inserindo A na lista [B | C] resulta na lista [A,B|C] se A < = B "

em Prolog: insere(A,[B|C],[A,B|C]) :- A =< B.

no Diagrama Semântico:



Figuras 7.4 e 7.5 Exemplos de representação de cláusulas não unitárias (regras) através de diagramas semânticos

7.1.3 Representação da Base de Dados

A notação dos Diagramas Semânticos tem como base a representação da fórmula atômica, que corresponde à cláusula unitária Prolog. Essa notação é estendida de forma a representar a conjunção de fórmulas atômicas que constitui o corpo de uma cláusula não unitária e a fórmula atômica que constitui a cabeça da mesma cláusula (representação da cláusula não unitária).

Num nível mais alto, representar a base de dados como um Diagrama Semântico envolve estender a notação de forma a representar o conjunto das cláusulas que têm objetos em comum, através de um Diagrama Semântico. Para tal, objetos continuam tendo representação única e diferenciada das relações. Os objetos são representados por nós e as relações rotulam arestas numeradas (em fundo preto), com indicação da ordem dos objetos na relação. A disposição dos nós no diagrama é gerada por um processo heurístico baseado no *fan in**3 dos objetos, conforme será descrito no decorrer deste capítulo.

Para exemplificar, mostramos na figura 7.6 o diagrama semântico que representa os fatos da base de dados a seguir:

```
pai_de(maria,joão).
mãe_de(pedro,maria).
pai_de(pedro,paulo).
mãe_de(maria,rute).
mulher(maria).
homem(pedro).
homem(joao).
pais_de(X,Y):-mãe_de(X,Y).
```

*3 número de arestas que chegam ao argumento.

```
pais_de(X,Y):-pai_de(X,Y).

ancestral_de(X,Y):-pais_de(X,Y).

ancestral_de(X,Y):-pais_de(X,Z),ancestral_de(Z,Y).
```

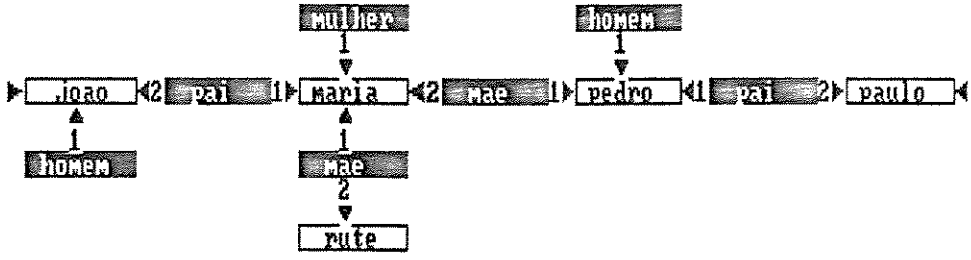


Figura 7.6 Diagrama Semântico de parte da Base de Dados

7.2 Descrição das Ferramentas que Compõem o Módulo Declarativo

O Módulo Declarativo é constituído por um conjunto de ferramentas que organizam e explicitam graficamente os objetos e relações presentes no programa Prolog. Nosso objetivo neste módulo é possibilitar ao usuário, acesso ao significado declarativo de seu programa, através de um *feedback* a respeito da "lógica" envolvida no texto do programa.

No Módulo Declarativo, o formalismo de representação de conhecimento de Prolog é expresso de forma pictórica e pode ser investigado pelo usuário através de "navegação" na representação baseada em Diagramas Semânticos. Através desse Módulo, o usuário tem acesso ao conhecimento expresso em sua base de dados, em três níveis: ao nível de cláusula, ao nível de base de dados (conjunto de

cláusulas) e ao nível de "inferência" (conjunto de cláusulas inferidas a partir da base de dados).

Nossa abordagem para o Módulo Declarativo envolve, num primeiro nível, dar suporte à fase de criação da base de dados, explicitando, ao nível de cláusula, o significado de uma proposição, com base nos conceitos de "objetos" e "relações".

Esta parte do Módulo Declarativo, que denominamos "Editor Semântico" (EDS), possibilita ao usuário ver os diagramas semânticos à medida em que define as cláusulas de sua base de dados. O EDS foi proposto tendo-se em mente principalmente o novato-novato (usuário iniciante em Prolog e em ambiente de computador). A interação do usuário com o ambiente como um todo (sistema Prolog mais ferramentas) nessa fase de criação da base de dados é registrada para análise posterior pelo facilitador ou pelo próprio aprendiz. O EDS possui um módulo que registra as ações do aluno e as respostas do sistema Prolog enquanto ele interage no modo de "perguntas" à base de dados. A figura 7.7 mostra um exemplo de tela do EDS.

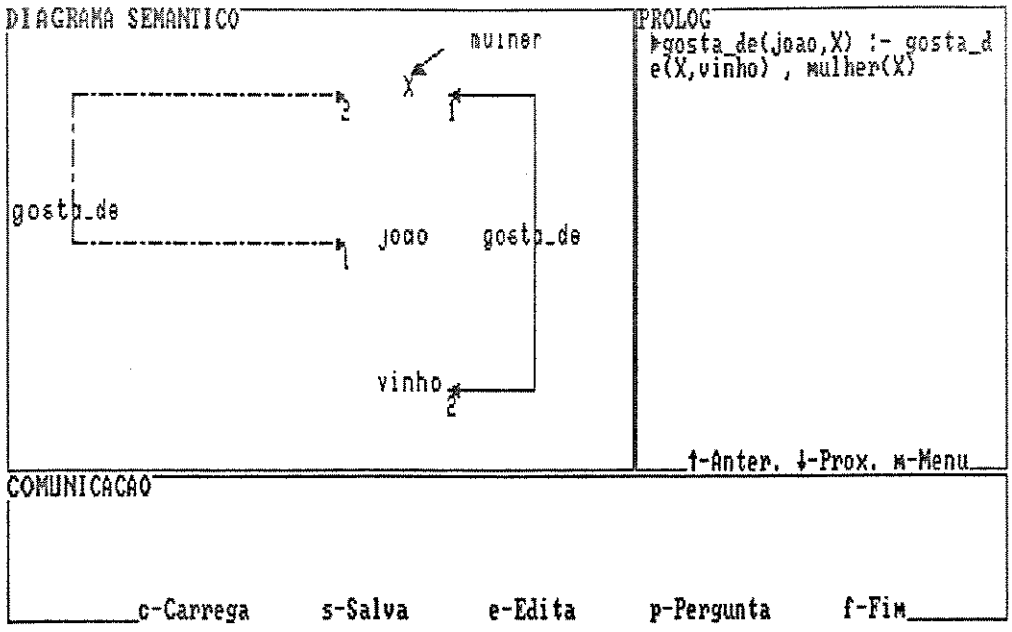


Figura 7.7 Exemplo de Tela de EDS mostrando diagrama semântico correspondente à cláusula indicada na região de tela denominada "Prolog"

Num segundo nível, o usuário tem acesso a ferramentas que traduzem para a representação de Diagramas Semânticos sua base de dados considerada como um todo e proposições inferidas da base de dados a partir de uma pergunta. Essa parte do Módulo Declarativo denominamos "Modo de Representação em Diagramas Semânticos" (MDS). A figura 7.8 mostra um exemplo de uma das telas do MDS.

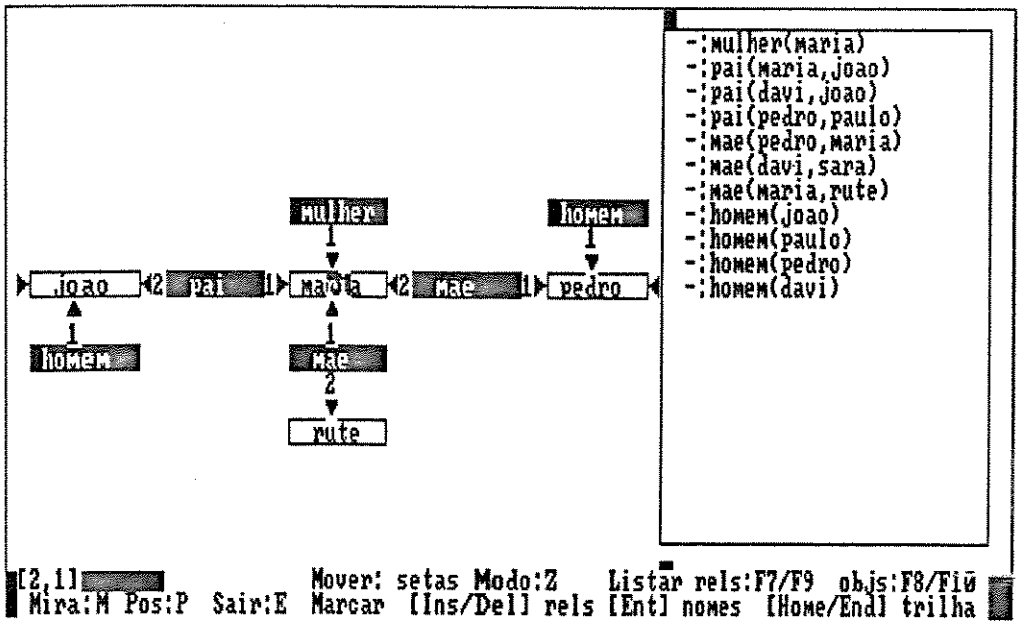


Figura 7.8 Exemplo de Tela de MDS mostrando parte do diagrama semântico, lista de relações (à direita) e menu (abaixo)

A seguir apresentamos a descrição funcional e alguns aspectos de implementação do Editor Semântico (EDS) e, em seguida, do MDS.

7.2.1 O Editor Semântico (EDS)

Uma das primeiras dificuldades dos novatos não sofisticados, no trabalho com Prolog, conforme será discutido nos próximos capítulos, está em expressar através do formalismo clausal, a sentença que ele tem em mente. O objetivo principal do EDS é refletir o "significado" de uma cláusula Prolog traduzindo-a para a representação gráfica de diagrama semântico. A representação fornece um *feedback* a respeito de como os objetos/conceitos presentes na sentença do usuário estão inter-relacionados.

O EDS é, portanto, uma ferramenta que pretende auxiliar o novato na fase de criação da base de dados, considerando principalmente o aspecto de significado apontado anteriormente. No que se refere a característica de edição propriamente dita ele é bastante simplificado. Não era nosso objetivo fazer um editor "completo". Tais facilidades são naturalmente providas pelo editor que compõe o sistema Arity/Prolog.

7.2.1.1 Descrição Funcional do EDS

O EDS possui uma única tela, que é dividida em três regiões: uma região para desenho do diagrama semântico, uma região para a cláusula Prolog e uma região para comunicação com a ferramenta.

Na Região de Comunicação (R.C.) acontece o "diálogo" entre o usuário e o editor. Através desta região da tela, o usuário entra com as cláusulas Prolog que constituirão sua base de dados, recebe informações do editor e envia informações ao editor.

A Região de Prolog (R.P.) é dedicada ao formalismo Prolog e mostra a base de dados sendo editada.

A Região do Diagrama Semântico (R.D.S.) mostra ao usuário o diagrama semântico correspondente à cláusula selecionada na região Prolog.

O menu principal do editor é apresentado permanentemente na Região de Comunicação. As regiões Prolog e Diagrama Semântico possuem menus particulares, no caso de RP, apresentado permanentemente em sua região de tela. A figura 7.9 mostra a tela do EDS e suas respectivas regiões.

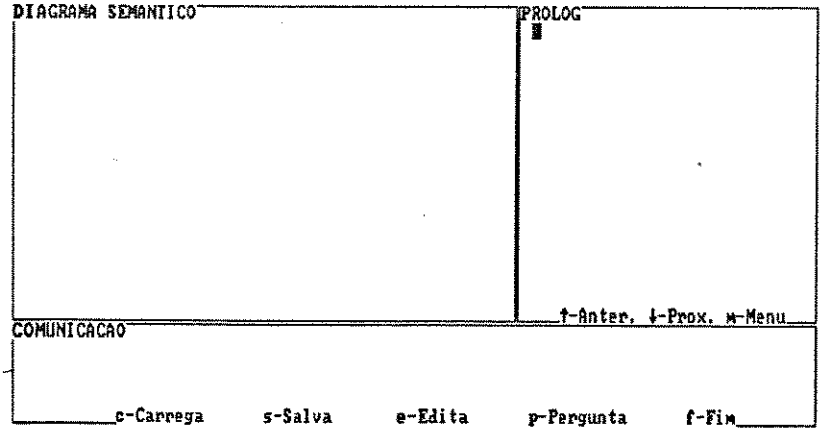


Figura 7.9 Tela de EDS mostrando as regiões do Diagrama Semântico, de Prolog e de Comunicação com menus respectivos

As escolhas que fizemos de tela única e regiões permanentes com menus sempre presentes tem em vista facilitar o uso da ferramenta, considerando principalmente as dificuldades do novato em seu primeiro contacto com ambiente de computador.

7.2.1.2 O Ambiente Gerado pelo EDS

O EDS é acionado a partir do ambiente do Sistema Operacional, através da chamada *eds*, que resulta na tela inicial com informações básicas sobre o uso da ferramenta, como mostra a figura 7.10.

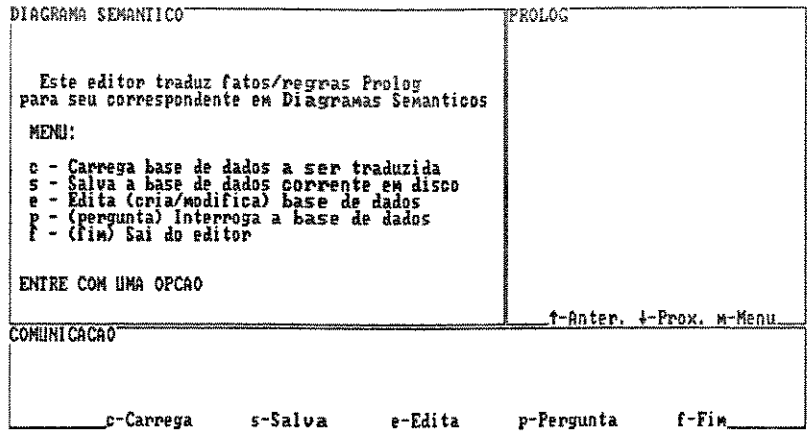


Figura 7.10 Tela inicial de EDS mostrando informações iniciais para uso da ferramenta

A seleção da opção *c-Carrega* é usada para trazer para o ambiente do EDS uma base de dados criada anteriormente e armazenada em um arquivo em disco. A seleção dessa opção mostra na região de tela Prolog, uma lista com os nomes de arquivos Prolog do diretório corrente e pede, via região de Comunicação, que o usuário entre com o nome do arquivo que deseja editar (figura 7.11).

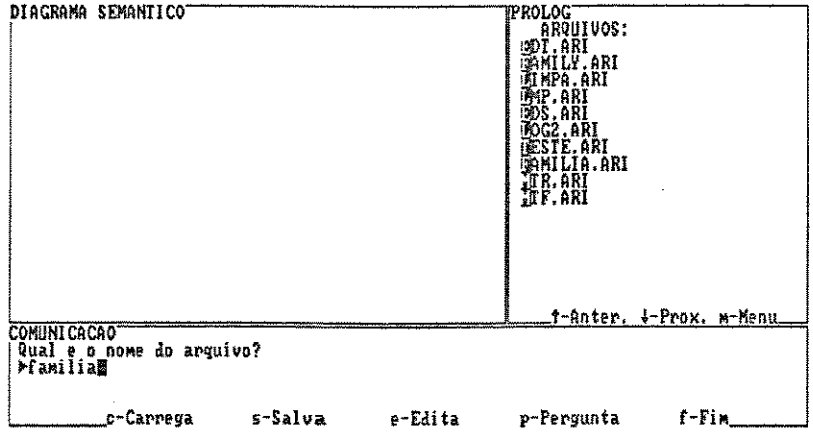


Figura 7.11 Carregando uma base de dados após seleção da opção *c-Carrega*

Após entrar com o nome do arquivo a editar, a região Prolog mostra uma listagem do conteúdo do arquivo carregado (base de dados do usuário). As opções da região Prolog podem então ser selecionadas. Com as teclas de direção (para cima e para baixo) o usuário pode mover o cursor para uma cláusula que deseja ver "traduzida" para a representação de Diagrama Semântico, ou pode retornar ao menu principal (tecla m) para seleção de qualquer de suas opções. A figura 7.12 mostra uma cláusula de RP selecionada e sua representação na RDS.

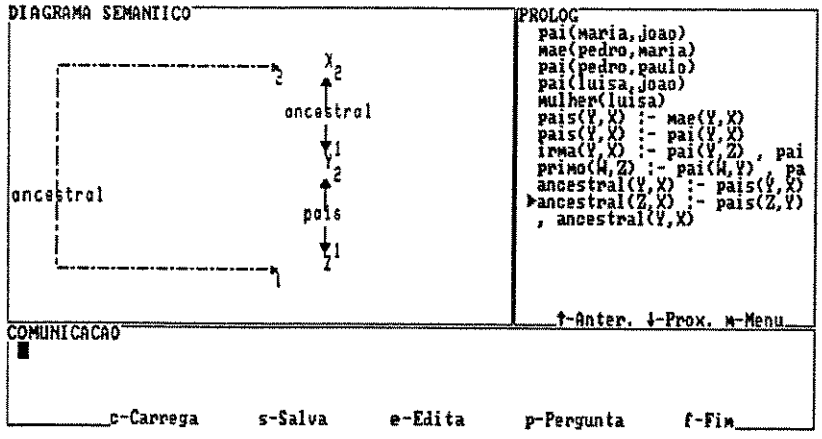


Figura 7.12 Cláusula de RP selecionada e sua representação em RDS

Na região do DS, o diagrama pode ser movido usando-se as teclas de direção (para cima, para baixo, para a direita, para a esquerda) para tornar visíveis partes do diagrama que, eventualmente, não estavam no campo de visão daquela região de tela (figuras 7.13 e 7.14). Retorna-se à RP pressionando-se qualquer outra tecla.

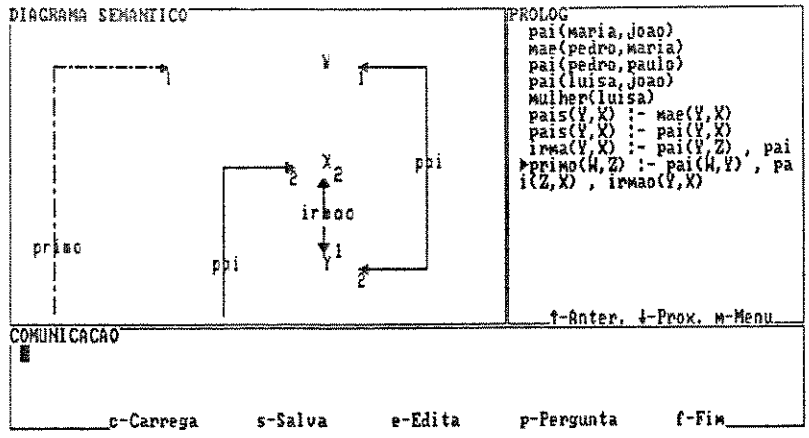


Figura 7.13 Representação de parte da cláusula *primo(W,Z)* (parte 1)

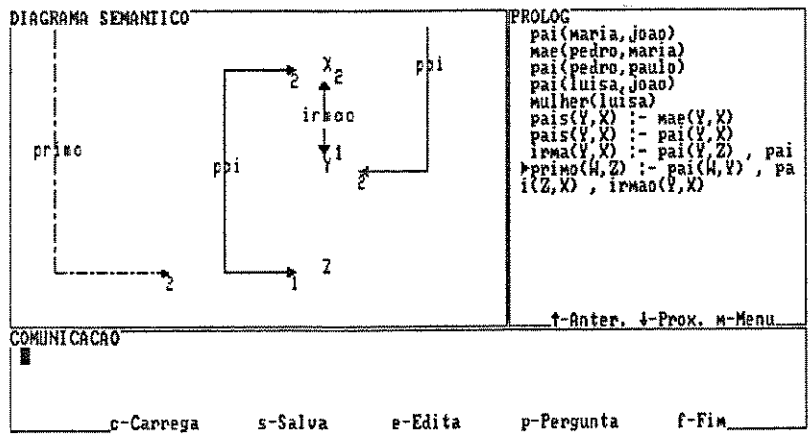


Figura 7.14 Representação de parte da cláusula *primo(W,Z)* (parte 2)

A seleção da opção *e-Edita* é usada para criação de uma nova base de dados ou inclusão de novas cláusulas ou retirada de cláusulas em uma base de dados carregada previamente para o ambiente do EDS. Ao selecionar a opção *e-Edita* o usuário é perguntado se deseja *rever* (tecla r), *incluir* (tecla i) ou *apagar* (tecla a) cláusula(s) na base de dados (figura 7.15). A sub-opção *rever* muda o controle para RP, onde o usuário pode selecionar cláusula e rever sua representação em diagrama semântico.

A sub-opção *incluir* pede que o usuário entre com a nova cláusula, como mostra a figura 7.16.

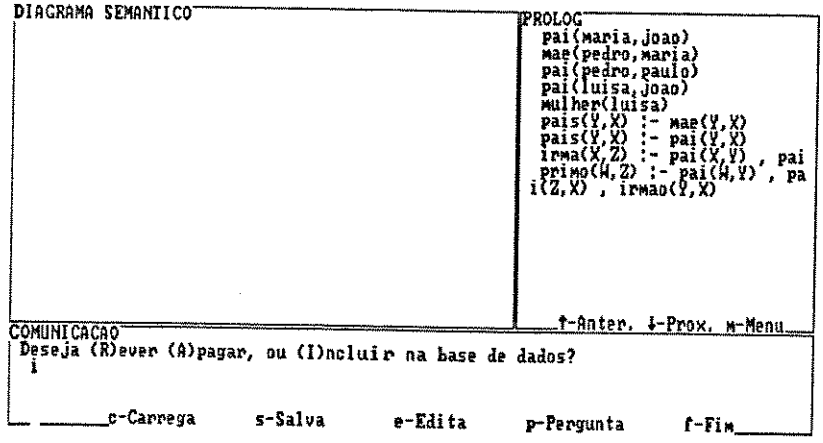


Figura 7.15 Sub-opções de *e-Edita*

Após entrada da nova cláusula, esta é acrescentada à base de dados mostrada na região Prolog e pode ser selecionada para tradução para a representação em DS. As figuras 7.16 e 7.17 mostram, respectivamente, a entrada da nova cláusula (mostrada na RC) e sua seleção para a representação em DS (mostrada em RP e RDS).

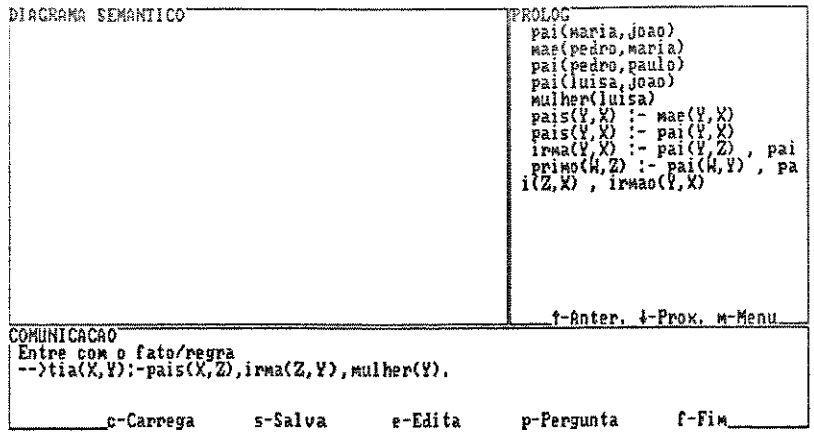


Figura 7.16 Criação de uma nova cláusula

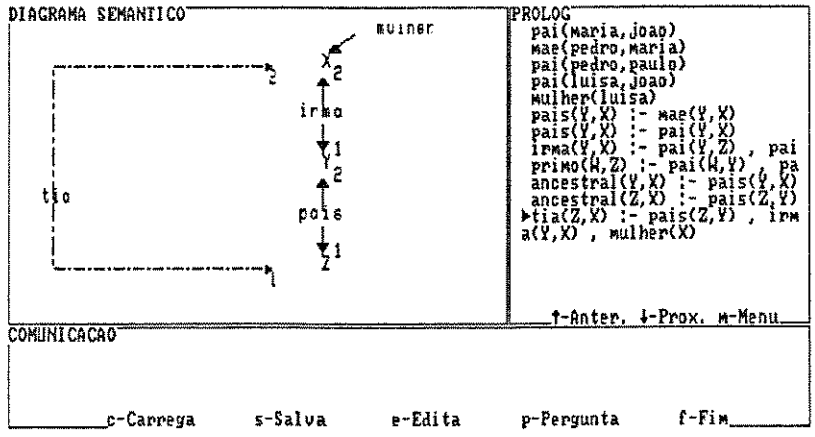


Figura 7.17 Representação em diagrama semântico da nova cláusula

A sub-opção *apagar(a)* permite eliminar uma determinada cláusula, toda a base de dados ou cancelar a operação, como mostra a figura 7.18.

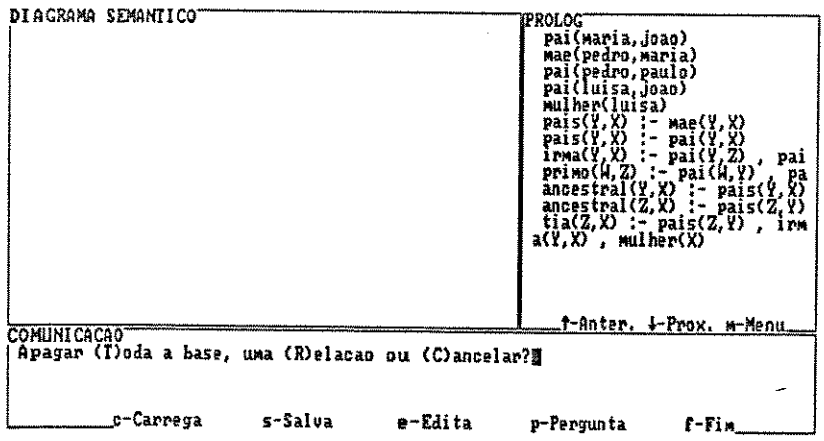


Figura 7.18 Sub-Opções de *Apagar*

Em caso da escolha de apagar uma relação, é pedido ao usuário o número correspondente, na base de dados, da cláusula a ser eliminada. Em caso da escolha de apagar toda a base de dados, é pedido confirmação, para que o usuário não perca seu trabalho inadvertidamente.

A seleção da opção *p-Pergunta* permite ao usuário interagir com o sistema Prolog, interrogando, a base de dados sendo editada. Todo o registro dessa interação é gravado em um arquivo (usr.log), que pode ser analisado *a posteriori* pelo responsável pelo acompanhamento do usuário em seu processo de aprendizagem (um experto ou o próprio usuário).

As figuras 7.19, 7.20, 7.21 mostram uma sequência de interação, após seleção da opção *Pergunta*.

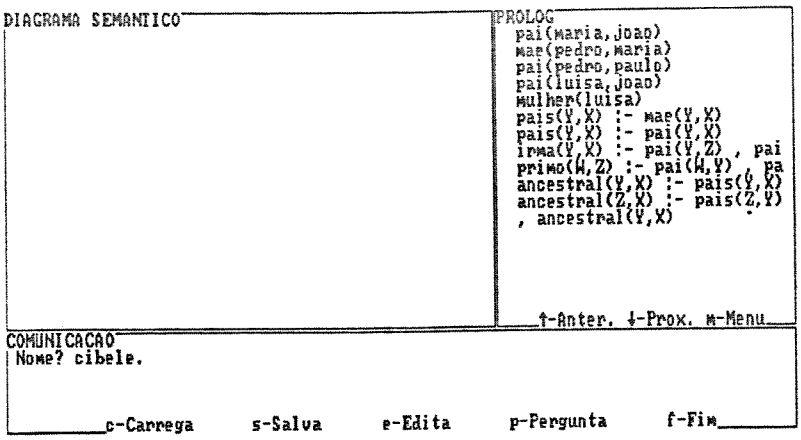


Figura 7.19 Opção *p-Pergunta* (em momento 1)

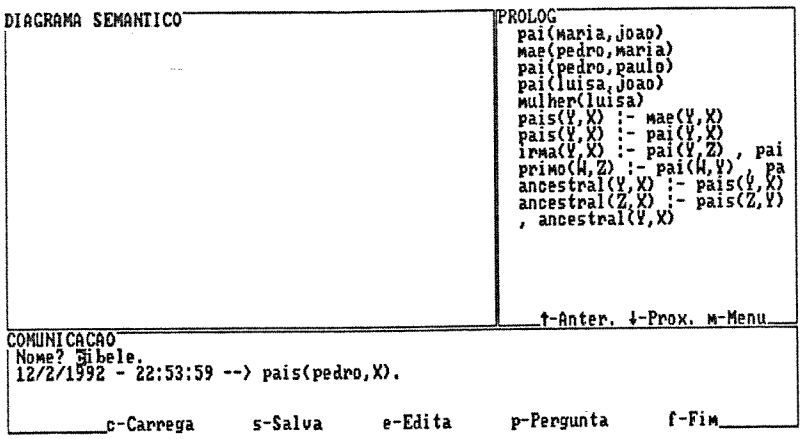


Figura 7.20 Opção *p-Pergunta* (em momento 2)

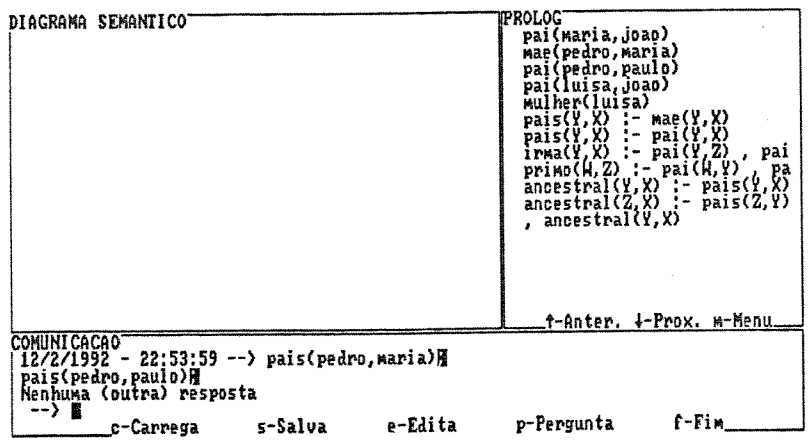


Figura 7.21 Opção *p-Pergunta* (em momento 3)

A seleção da opção *s-Salva*, permite salvar em disco a base de dados sendo editada. Após sua seleção é pedido ao usuário que entre com o nome que deseja dar ao arquivo que será gravado.

A seleção da opção *f-Fim* finaliza o uso do EDS, retornando ao ambiente Prolog.

7.2.1.3 Aspectos de Implementação do EDS

Como no caso das ferramentas que compõem o Módulo Operacional, o EDS foi implementado usando-se uma versão estendida do Arity Prolog, que inclui a biblioteca de rotinas gráficas APGRAPH (APGRAPH, 1990), para computadores da linha PC.

O sistema que implementa o EDS está dividido nos seguintes módulos:

- Módulo Controlador do EDS (MC)
- Módulo de Tradução de Fatos (MTF)
- Módulo de Tradução de Regras (MTR)
- Módulo de Registro de Informação (MRI)

A figura 7.22 mostra um diagrama do relacionamento entre módulos que compõem o EDS.

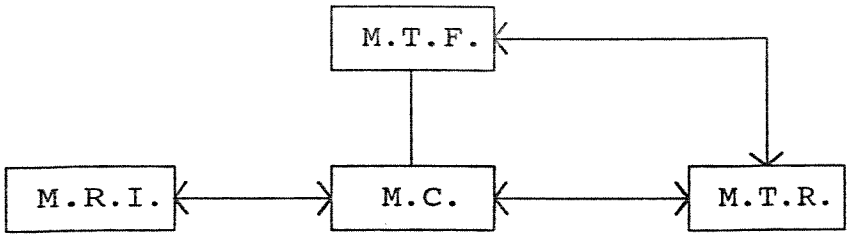


Figura 7.22 Diagrama do relacionamento entre os módulos de EDS

O Módulo Controlador gerencia o sistema e utiliza os outros módulos para sustentar a interação com o usuário. Neste módulo a estrutura de dados utilizada é inicializada e a interação é sustentada pela implementação dos menus principal e das regiões de tela Prolog e da região de tela D.S.. O MC é, portanto, responsável pela definição e controle das ações correspondentes às opções dos menus.

O Módulo de Tradução de Fatos é acionado pelo MC quando o usuário entra com uma cláusula unitária e/ou deseja vê-la traduzida para D.S.. O MTF monta a estrutura de dados necessária para a tradução da cláusula para a representação em diagrama semântico e, quando solicitado, constrói (desenha) a representação gráfica na região de tela do DS.

O DS do fato é feito colocando-se o predicado (relação) no centro da região de tela DS, cercado por um retângulo para diferenciá-lo de seus argumentos (objetos). As arestas saem do predicado em direção aos argumentos, distribuídas em até oito pontos ao redor do retângulo. Isso significa que nossa representação permite relações com no máximo oito objetos. Argumentos repetidos são representados por apenas um objeto, com tantas arestas chegando a ele quantas for o número de repetições desse argumento na cláusula. As arestas são numeradas com a ordem do argumento na cláusula original. A figura 7.3, que representa respectivamente os fatos " João deu o livro a Maria " e " A concatenação de [] com X resulta X ", ilustram o que acabamos de descrever.

O Módulo de Tradução de Regras, também é acionado pelo MC, quando uma cláusula não unitária é identificada. Como o MTF, ele constrói a estrutura de dados necessária para a representação da cláusula através de Diagramas Semânticos e desenha, quando solicitado, o diagrama semântico correspondente na região de tela do DS.

Uma característica da cláusula Prolog não unitária que achamos importante explicitar no DS é a distinção entre a parte consequente (conclusão da regra) e sua parte antecedente (condições da regra). Outra característica observada na cláusula não unitária é que grande parte dos argumentos presentes na regra é compartilhado entre a cabeça e o corpo da cláusula ou entre as proposições que constituem o corpo da cláusula. Assim, a disposição do diagrama semântico na região de tela é feito de maneira a aproveitar essas características. Os objetos são distribuídos verticalmente na tela, a partir do centro e alternados, com base em seu *fan in*. As arestas são rotuladas pelo predicado correspondente à proposição e direcionadas aos argumentos da proposição, com numeração correspondente à ordem dos argumentos naquela proposição. A proposição que corresponde à parte consequente da regra é sempre representada mais à esquerda na região de tela e sua aresta é tracejada. As proposições que correspondem à parte antecedente da regra rotulam arestas de linhas contínuas e são desenhadas, em colunas diferentes (para que não haja sobreposição) e em direção à parte direita da região de tela. Proposições com um único argumento são representadas por arestas desenhadas em sentido diagonal na região de tela e a numeração de tais arestas é omitida. A figura 7.4, que representa a regra "João gosta de mulheres que gostam de vinho" ilustra a descrição feita. Todo o DS pode ser deslocado de posição na tela, para tornar visíveis partes que não cabem completamente na região de tela do DS.

A implementação atual da representação de cláusulas não unitárias limita em sete o número de fórmulas atômicas em uma regra e em cinco o número

de argumentos distintos. A escolha desses parâmetros foi baseada numa estimativa feita a partir da observação de definições de cláusulas em programas de novatos e em programas-exemplo apresentados em livros texto.

O Módulo de Registro de Interação aciona o interpretador Prolog para responder perguntas do usuário, enquanto cria, em disco, um arquivo de registros dessa interação usuário-sistema.

Um "*shell*"[*4] para responder metas Prolog foi implementado e é usado como base para manter um registro de uma sessão de perguntas do usuário à base de dados Prolog sendo criada. São registrados o nome do usuário, data e hora de início e término da sessão, metas entradas pelo usuário e respostas do sistema. Tal registro é importante, do ponto de vista metodológico, para possibilitar uma análise e acompanhamento da interação do aluno com o sistema como um todo. USR.LOG é o arquivo do registro de interações, gerado.

7.2.1.3.1 Estrutura de Dados Utilizada

Os dados manipulados pelos módulos que compõem o EDS estão organizados em árvores-B, tabelas-Hash e na própria base de dados do sistema.

As cláusulas da base de dados do usuário são numeradas e essa numeração é mantida em uma lista, armazenada na base de dados do sistema, através da cláusula *ordem(Lista).*

Associados a cada índice (na numeração de cláusulas da base de dados do usuário), existe um conjunto de informações a respeito da cláusula em questão, armazenados em uma árvore-B de nome "redes". A cada índice de cláusula corresponde, portanto, um nó nessa árvore.

[*4] programa interativo que aceita metas Prolog e as responde.

A figura 7.23 representa a estrutura dos dados armazenados na árvore-B "*redes*".

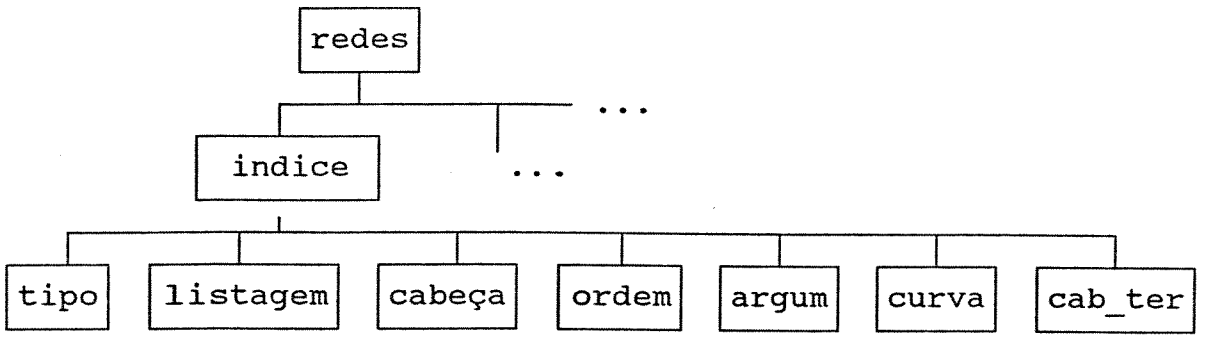


Figura 7.23 Informações armazenadas na árvore-B *redes*

Nas folhas da árvore-B da figura 7.23:

tipo representa uma cláusula unitária (fato) ou uma cláusula não unitária (regra).

listagem armazena textualmente a cláusula (fato ou regra).

cabeça armazena o predicado e sua posição na tela do DS, no caso de uma cláusula unitária e o predicado correspondente à cabeça de regra, sua posição e a lista de argumentos associados a ela, no caso de uma cláusula não unitária.

ordem armazena uma lista com a ordem de colocação na tela do DS, dos argumentos de um fato.

curva armazena a posição da aresta, no caso de argumento repetido no fato.

argum armazena um dado argumento e sua posição na tela do DS.

cab_term armazena o predicado de uma proposição do corpo de uma regra, os argumentos associados e sua posição na tela do DS.

Exemplos de definição e armazenamento de dados correspondentes à cláusula unitária na árvore "redes":

recordb(redes,Ntela,fato).

Ntela é a chave de procura e refere-se ao índice da cláusula na base de dados e *fato* indica tratar-se de uma cláusula unitária.

recordb(redes,Ntela,listagem(X)).

X é a listagem original da cláusula.

recordb(redes,Ntela,cabeça(Cabeça,X,Y)).

Cabeça contém o predicado da cláusula, *X* e *Y* indicam a posição na tela do DS, onde será colocado o predicado.

recordb(redes,Ntela,ordem(L)).

L é uma lista que representa a ordem em que os argumentos são desenhados no diagrama correspondente a um fato.

recordb(redes,Ntela,curva(N,X,Y).

X e *Y* são usados para desenhar uma aresta curva, no caso de argumentos repetidos no fato. *N* representa o número do argumento e *X* e *Y* representam a posição destino para a curva.

recordb(redes,Ntela,argum(Argm,L,C,Xi,Yi,N)).

o terceiro parâmetro representa um argumento e sua posição na tela do DS. *N* é o número do argumento na lista de ordem dos argumentos.

Exemplos de definição e armazenamento de dados correspondentes a cláusula não unitária, na árvore-B "*redes*":

recordb(redes,Ntela,regra).

Ntela é a chave de procura e refere-se ao índice da cláusula na base de dados e *regra* indica tratar-se de cláusula não unitária.

recordb(redes,Ntela,listagem((X:-Y)).

X representa a cabeça e *Y* o corpo da cláusula original.

recordb(redes,Ntela,cabeça(Cab,L_Argc,X,Y)).

Cab corresponde ao predicado da cabeça da cláusula e *L_Argc* representa a lista dos argumentos associados ao cabeça da regra. *X* e *Y* representam a posição onde será colocada a cabeça da regra na tela do DS.

recordb(redes,Ntela,cab_term(Cab,Larg,X,Y)).

Cab representa um predicado do corpo da regra, *X* e *Y* a posição onde será colocado na tela do DS, com base nas posições dos argumentos dessa cabeça, representados pela lista *Larg*.

recordb(redes,Ntela,argum(Arg,X,Y)).

Arg representa um argumento da regra e *X* e *Y* a posição na tela do DS onde ele será colocado.

Três tabelas-Hash, utilizadas como estruturas auxiliares no desenho dos diagramas semânticos correspondentes às cláusulas, são descritas a seguir:

Tabela-Hash *termos*:

Contém os predicados presentes nas cláusulas não unitárias e seus respectivos argumentos.

Tabela-Hash *CabR*:

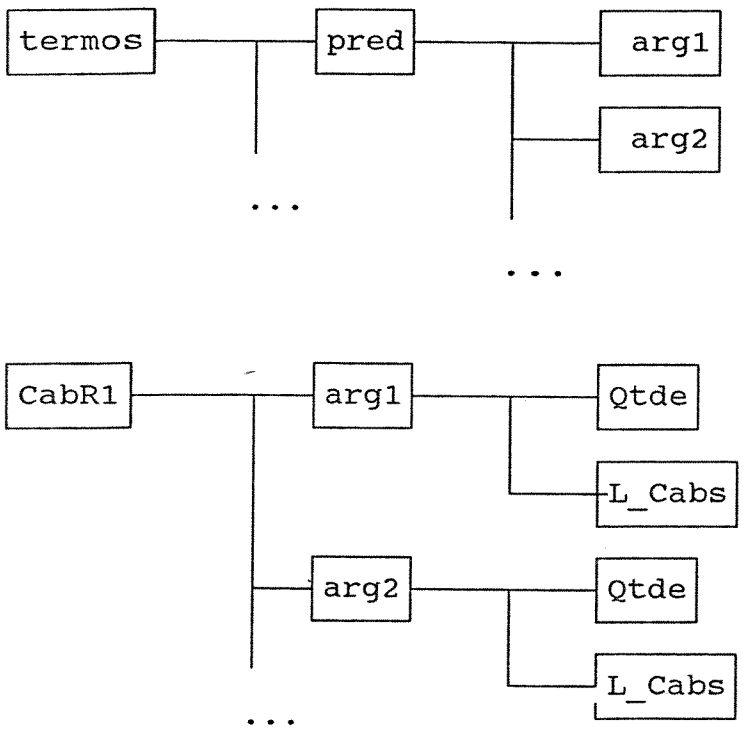
Contém os argumentos, o número de vezes que eles ocorrem numa regra e as listas de predicados associados a ele.

CabR representa o nome de um predicado da cabeça de uma regra.

Tabela-Hash *CabF*:

Contém as coordenadas de início das arestas que saem do retângulo que cerca o nome do predicado da cláusula unitária. *CabF* representa o predicado correspondente a um fato.

A figura 7.24, a seguir, representa a estrutura dos dados armazenados nas três tabelas-Hash, decritas.



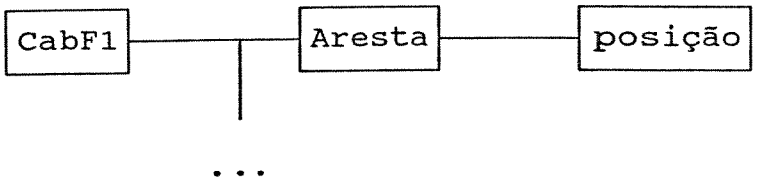


Figura 7.24 Informações armazenadas nas tabelas-Hash *termos*, *CabF*, *CabR*, respectivamente

Exemplos de definição e armazenamento de dados correspondentes às cláusulas, nas tabelas-Hash:

recordh(Nome,Pos,position(X,Y)).

Nome é o predicado correspondente a um fato; *Pos* é um indicador da posição de saída de uma aresta e *X* e *Y* a posição da aresta correspondente a *Pos.*

recordh(termos,Cab,L_arg).

Cab contém o nome de um predicado e L_arg a lista dos nomes dos argumentos associados a esse predicado.

recordh(Nome,Arg,lista(Qtde,L_Cabs)).

Nome representa o nome de um predicado cabeça de regra. *Arg*, o nome de um argumento do predicado; *Qtde* o número de vezes que esse argumento aparece na cláusula e *L_Cabs* a lista dos outros predicados onde o argumento aparece na cláusula.

7.2.2 O Modo de Representação em Diagramas Semânticos (MDS)

O objetivo do MDS é possibilitar ao usuário uma investigação de sua base de dados, com base apenas nos aspectos da "lógica" envolvida entre os elementos de sua base de dados, de forma independente de mecanismos de "controle". Tal exploração é feita sobre a representação em diagramas semânticos, ao nível da base de dados considerada como um todo. Dessa maneira, proposições que no formalismo textual estão isoladas, são inter-conectadas pelos seus objetos comuns, possibilitando uma visão pictórica do conhecimento expresso pelo conjunto das proposições.

Além da exploração da base de dados propriamente dita, podem ser também representadas em DS, proposições inferidas sobre essa base de dados a partir de uma meta do usuário.

O ambiente gerado pelo MDS possibilita examinar várias porções do DS, listando objetos e/ou relações correspondentes, colocando "marcas" no diagrama, buscando relacionamentos diretos ou indiretos entre dois objetos (caminhos), etc.. Os recursos gerados para tal exploração incluem:

- representação reduzida do DS, para visualização completa do diagrama.
- navegação na representação reduzida para seleção de partes a ampliar.
- representação ampliada do DS
- navegação na representação ampliada
- seleção de regiões do diagrama a partir da lista de objetos e/ou relações, ou lista de cláusulas.
- indicação do objeto no diagrama, a partir da navegação na lista de objetos.
- indicação de relações no diagrama, a partir de navegação na lista de relações.
- indicação de região do diagrama, a partir da navegação na lista de cláusulas da base de dados.

- marcações no diagrama, para ressaltar: um objeto específico, uma relação específica, uma relação específica e seus objetos, um caminho entre dois objetos especificados.

A seguir apresentamos a descrição funcional do MDS e aspectos de implementação dessa ferramenta.

7.2.2.1 Descrição Funcional do MDS

A exemplo do Módulo Operacional, a interação do usuário com o sistema se dá essencialmente via seleção de menus. O ambiente gerado pelo MDS se apresenta funcionalmente distribuído em níveis hierárquicos. No primeiro nível temos o ambiente inicial e as facilidades básicas apresentadas pelo menu principal, como operações para manipular a base de dados do usuário (carregar, listar, apagar), apresentação de informações de ajuda ao usuário sobre o uso do sistema, seleção das opções que manipulam os DS e saída do sistema.

Num segundo nível entramos na tela de representação gráfica dos diagramas semânticos, que tem seu próprio menu de opções e operações correspondentes. A figura 7.25 apresenta um diagrama hierárquico de funções do MDS.

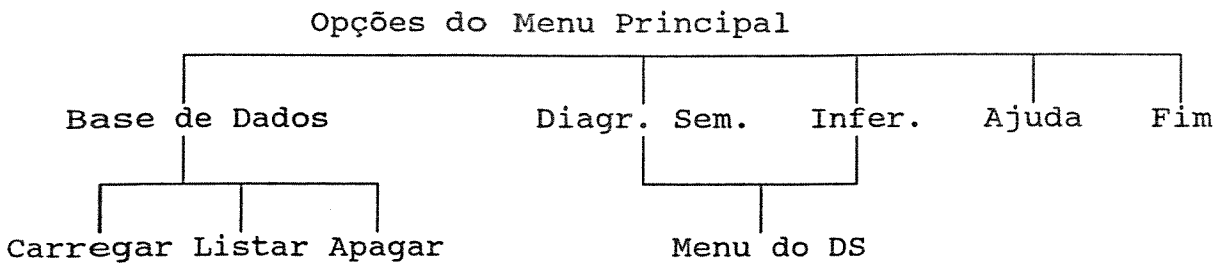


Figura 7.25 Diagrama Hierárquico de funções do MDS

7.2.2.2 O Ambiente Inicial e Facilidades Básicas

O MDS é acionado a partir do ambiente do sistema operacional, pelo predicado *mds*, que resulta na tela mostrada a seguir (figura 7.26).

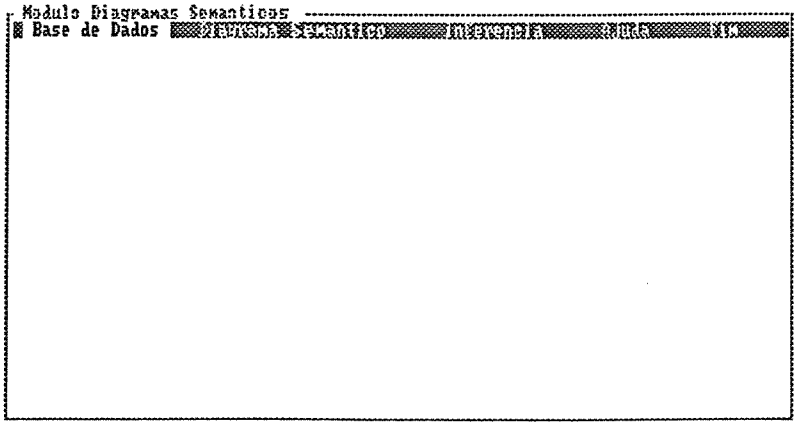


Figura 7.26 Tela inicial de MDS com menu principal

Para manter a homogeneidade do conjunto de ferramentas que compõem os módulos operacional e declarativo, mantivemos o mesmo tipo de estrutura hierárquica de funções e o mesmo conjunto de operações básicas em ambos os módulos.

As operações correspondentes à opção *Base de Dados* (Carregar, Listar e Apagar) são idênticas às do Módulo Operacional e estão detalhadas no capítulo 6. As figuras 6.8, 6.9 e 6.10 do capítulo 6 ilustram tais operações.

A seleção da opção *Ajuda*, a exemplo do MOP, apresenta uma janela com informações sucintas a respeito do uso do MDS. A figura 7.27 mostra a janela de Ajuda, com parte das informações.

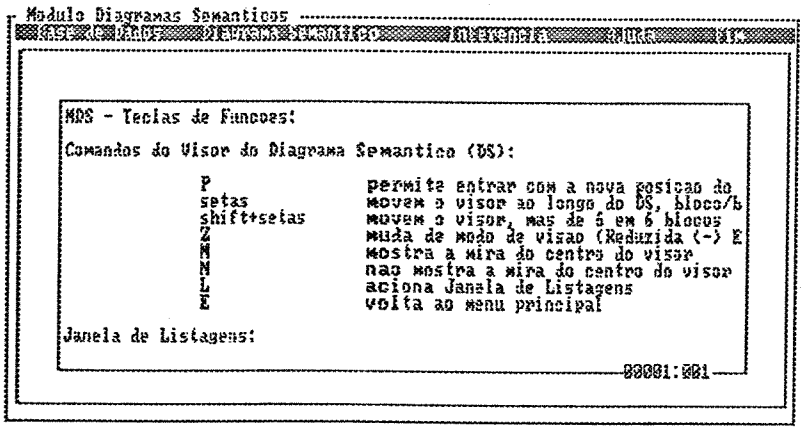


Figura 7.27 Seleção da opção *Ajuda* em MDS

A opção *Fim* finaliza o uso do MDS, retornando ao ambiente Prolog.

A seguir, apresentaremos a descrição funcional do segundo nível de operações, na estrutura hierárquica de funções do MDS, obtidos da seleção das opções *Diagrama Semântico* e *Inferência*.

7.2.2.3 O Ambiente do Diagrama Semântico

A seleção da opção *Diagrama Semântico* aciona a tela gráfica onde é apresentado o diagrama semântico correspondente à base de dados carregada previamente, em sua

forma reduzida, como mostra a figura 7.28. As duas últimas linhas da tela gráfica apresentam o menu correspondente ao ambiente do DS.

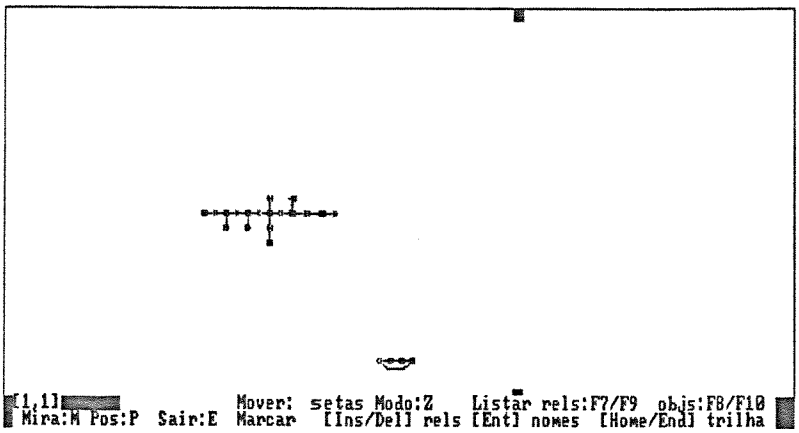


Figura 7.28 Apresentação em forma reduzida do diagrama semântico da base de dados

O modo de apresentação do diagrama pode ser mudado de reduzido para ampliado e vice-versa, usando-se a opção *Modo*:(tecla Z). A figura 7.29 mostra parte do diagrama anterior, de forma ampliada.

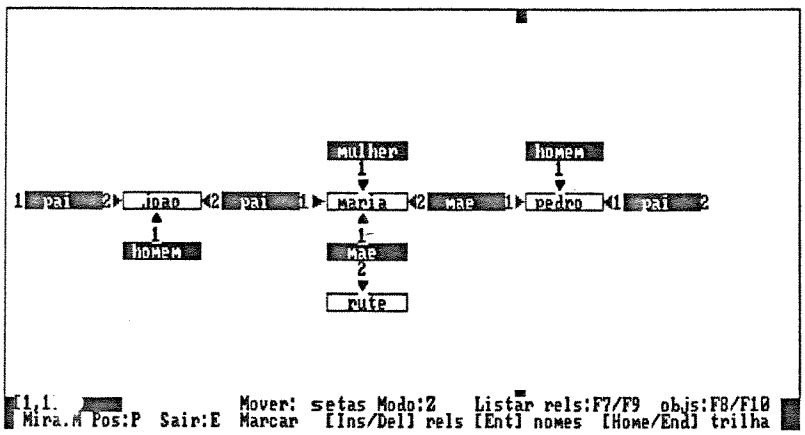


Figura 7.29 Apresentação ampliada de parte do diagrama semântico anterior

A apresentação reduzida do diagrama pode indicar a região do DS que aparecerá na tela, no modo ampliado, usando-se a opção *Mira*:(tecla M). A figura 7.30 mostra o diagrama reduzido, com a mira acionada.

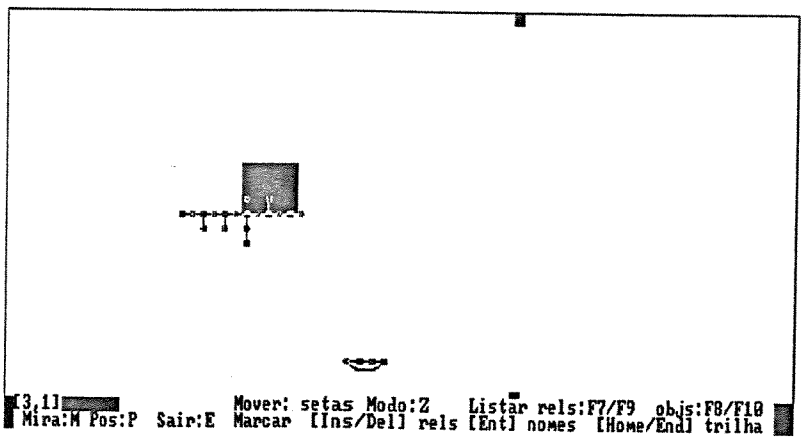


Figura 7.30 Diagrama semântico reduzido com indicação da parte selecionada para ampliação (quadrado em fundo escuro)

O DS pode ser movimentado, usando-se as teclas de direção (setas para cima, para baixo, para esquerda, para direita) para possibilitar a visualização de partes que ficavam fora do campo de visão da tela. A figura 7.31 mostra o diagrama da figura 7.29 deslocado para a esquerda.

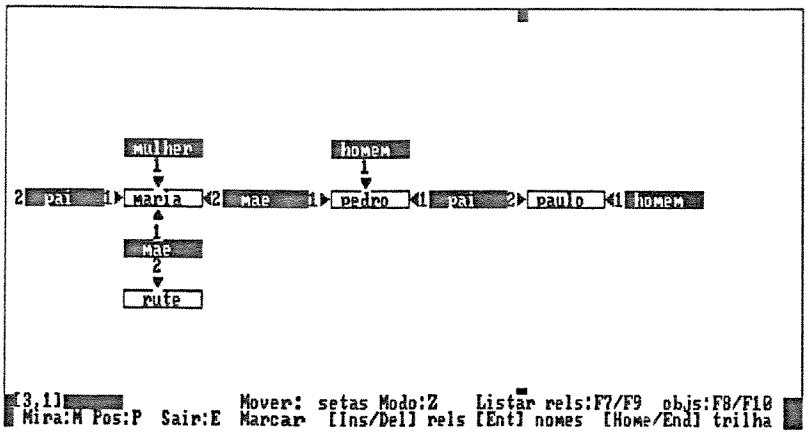


Figura 7.31 Diagrama semântico correspondente à figura 7.29 deslocado para a esquerda

Para auxiliar a visualização dos elementos da base de dados (cláusulas, objetos e relações), pode ser acionada uma janela que se sobrepõe à tela gráfica mostrando a lista dos objetos ou a lista das relações que constituem a base de dados. A listagem inicial *(default)* é a das relações/cláusulas presentes na parte visível do DS e é acionada inicialmente pela tecla "*l*".

A opção *listar rels* (listar relações) pode ser acionada pelas teclas de função F7 e F9. Com a opção *F7* a janela mostra a lista de relações/cláusulas representadas na parte do diagrama visível na tela. Com a opção *F9* a janela mostra todas as relações/cláusulas da base de dados. A figura 7.32 ilustra a escolha da opção F9 para a base de dados sendo mostrada.

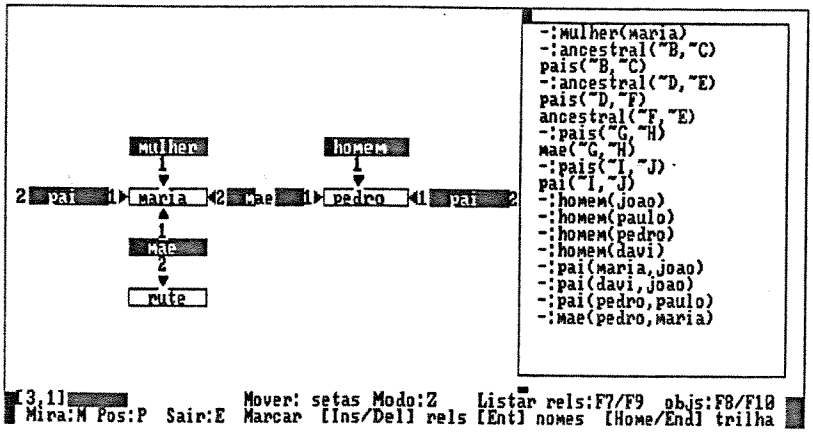


Figura 7.32 Diagrama semântico e lista das relações presentes na base de dados

As fórmulas atômicas que representam uma cláusula unitária (fato), ou uma cabeça de regra (no caso de cláusula não unitária), são antecedidas pelo símbolo " -: ", distinguindo-as das fórmulas atômicas que constituem o corpo de uma regra.

A opção *listar objs* (listar objetos) pode ser acionada pelas teclas de função *F8* e *F10*. Com a opção *F8* a janela mostra a lista dos objetos representados na parte do diagrama visível na tela. Com a opção *F10* a janela mostra todos os objetos presentes na base de dados. A figura 7.33 ilustra a escolha da opção *F10* para a base de dados sendo mostrada.

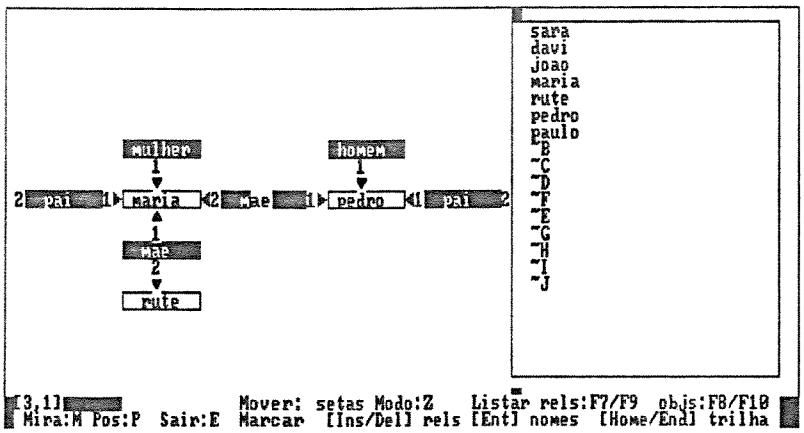


Figura 7.33 Diagrama semântico e lista dos objetos presentes na base de dados

A qualquer momento a janela de listagens pode ser eliminada (tecla "e") deixando visível a parte do diagrama que estava escondida pela sobreposição da janela.

Os objetos que representam variáveis são distinguidos dos demais objetos, nas listagens, pelo símbolo inicial "~"(til). Para facilitar a procura de objetos ou relações no DS, um cursor pode ser movimentado sobre a janela de listagens, enquanto simultaneamente a relação ou objeto sob o cursor são apontados no diagrama. A figura 7.34 mostra uma relação selecionada na lista de relações presente na tela e sua indicação no diagrama.

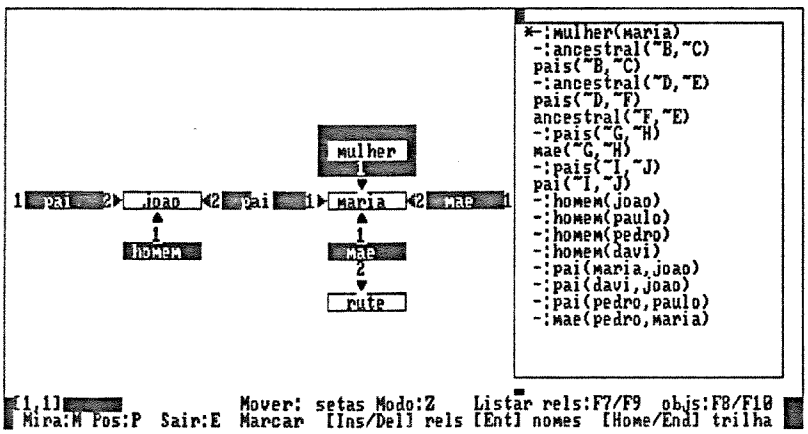


Figura 7.34 Seleção da relação *mulher(maria)*, na listagem de relações e sua indicação correspondente no diagrama semântico (quadrado maior em fundo escuro)

Como já foi descrito anteriormente, o diagrama pode ser movimentado usando-se as setas de direção, ou movendo-se o diagrama reduzido antes de sua ampliação, ou ainda entrando-se com uma dada posição (linha e coluna) de tela, acionando-se a opção *Pos* (tecla P) do menu. Existe, ainda, uma maneira de visualizar regiões do diagrama, com base na seleção de objetos ou relações, a partir de suas respectivas listagens. Navegando sobre a lista de objetos ou relações, posiciona-se o cursor sobre um dado objeto ou relação e aciona-se a opção *Pos* (teclas P). A região do diagrama que contém o objeto ou relação selecionada é trazida, então, para o campo de visualização do diagrama (região central da tela). As figuras 7.35 e 7.36 mostram, respectivamente a tela do MDS antes e após uma seleção de posicionamento para o objeto *davi*.

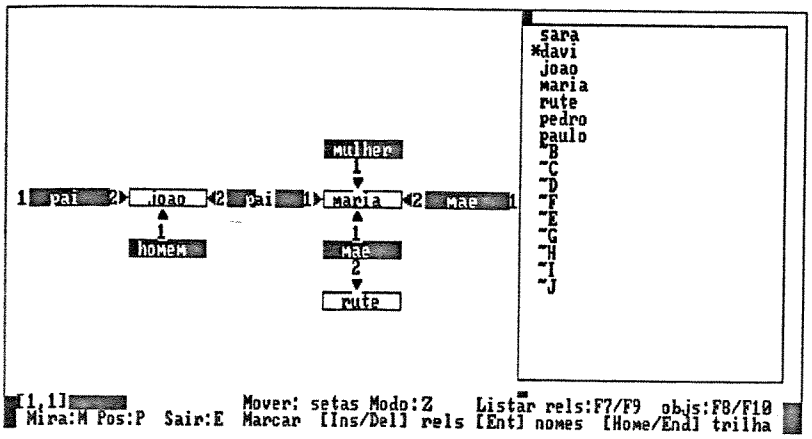


Figura 7.35 Seleção do objeto *davi*, na lista de objetos, para posicionamento de visualização

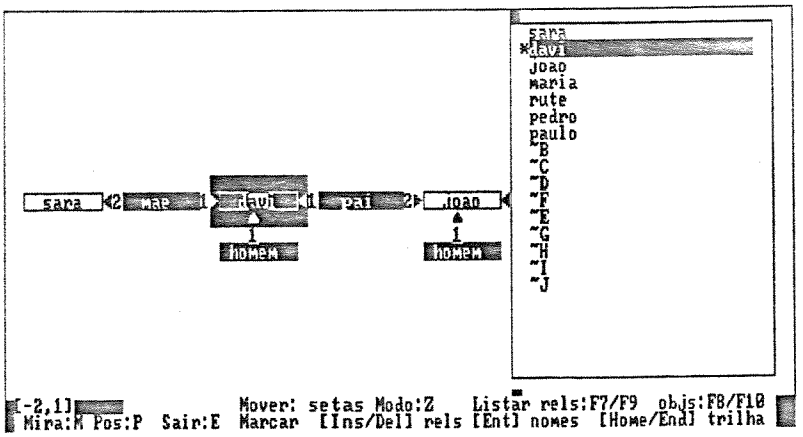


Figura 7.36 Posicionamento do objeto *davi* na região central do diagrama semântico

Outro recurso apresentado para facilitar a exploração da representação é a marcação de nós e/ou arestas do diagrama. É possível marcar objetos, relações ou relações com seus respectivos objetos, como se pintássemos esses elementos no diagrama, para distinguí-los dos demais.

Objetos e relações podem ser marcados no diagrama, selecionando-os na listagem respectiva e teclando *enter*. O mesmo procedimento é feito para desmarcá-los. Uma relação e seus respectivos objetos são marcados selecionando a relação, na lista de relações e acionando-se a opção *Marcar rels* (tecla "ins"). Os mesmos relação e objetos são desmarcados teclando-se "del" após seleção da relação. A figura 7.37 mostra a marcação de uma relação e seus respectivos objetos.

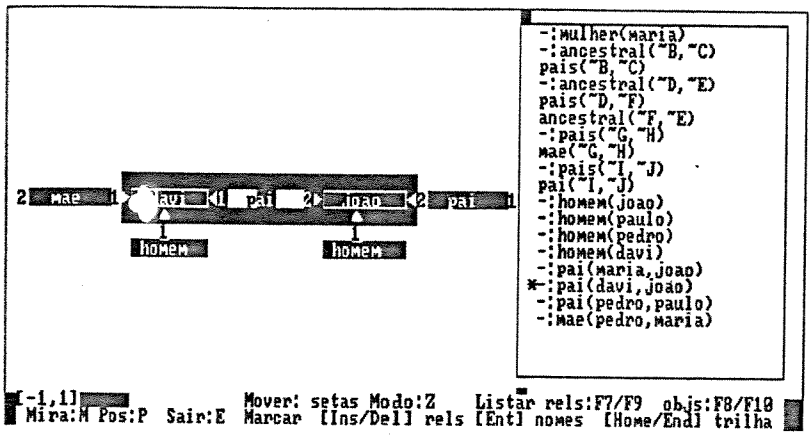


Figura 7.37 Marcação da relação *pai(davi,joão)* na lista de relações e sua visualização no diagrama semântico

Também pode ser investigada a existência de caminhos entre dois objetos do diagrama. Para tal seleciona-se, a partir da lista de objetos, cada um dos objetos. Marca-se o objeto inicial (tecla *home*) e o objeto final (tecla *end*). No caso de existência de um caminho ligando os dois objetos, esse caminho é mostrado, como ilustra a figura 7.38.

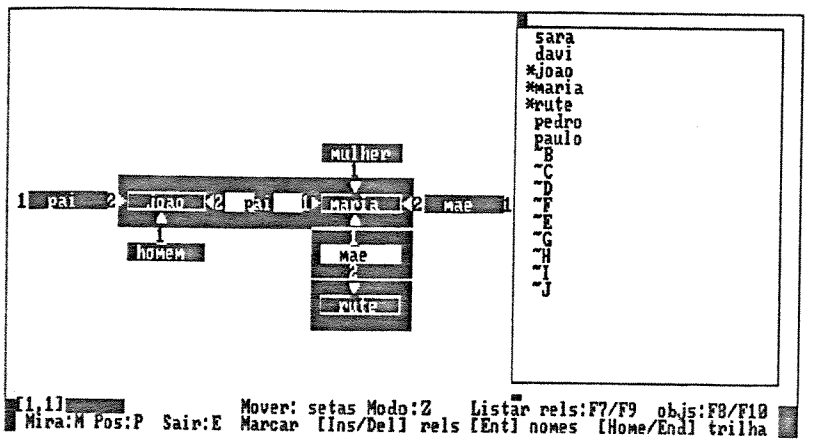


Figura 7.38 Indicação, no diagrama semântico, de caminho ligando os objetos *joão* e *rute* e indicação, na lista de objetos, dos objetos que pertencem a esse caminho

7.2.2.4 A Opção Inferência

A seleção da opção *Infer* usa o ambiente do Diagrama Semântico para mostrar o DS correspondente a proposições inferidas sobre uma dada base de dados, a partir de uma meta colocada pelo usuário. O Diagrama construído pode considerar apenas as proposições inferidas da base de dados durante resposta à meta do usuário ou pode incluir a própria base de dados. As figuras 7.39, 7.40 e 7.41 mostram respectivamete a entrada da meta, o DS correspondente aos fatos explícitos na base de dados ou inferidos a partir da base de dados e o DS das inferências incorporadas à base de dados original.

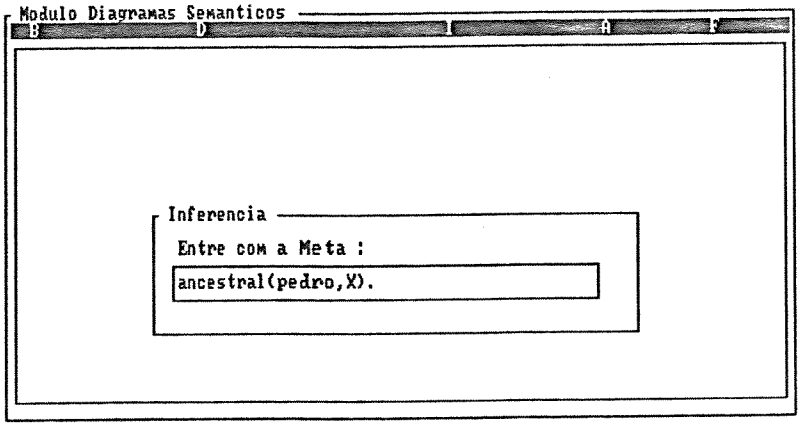


Figura 7.39 Entrada de uma meta para investigação na opção *Inferencia*

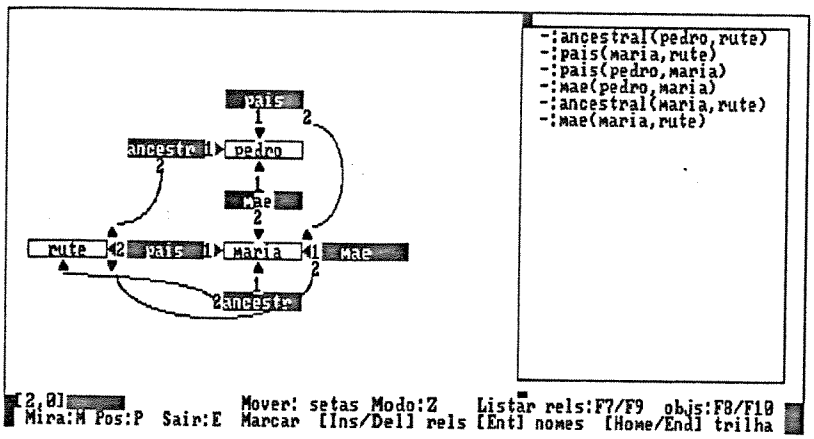


Figura 7.40 Diagrama semântico correspondente à dedução de *ancestral(pedro,rute)*

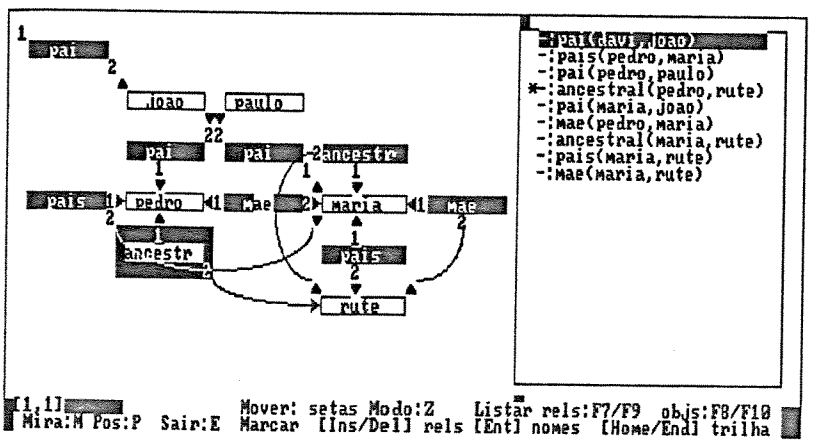


Figura 7.41 Diagrama semântico da opção *Inferencia* combinado ao DS da base de dados

7.2.2.5 Aspectos de Implementação do MDS

A exemplo das demais ferramentas (MOP e EDS), o ambiente de implementação do MDS utiliza a versão estendida do Arity Prolog que inclui a biblioteca de rotinas gráficas APGRAPH (APGRAPH, 1990), para computadores tipo PC.

O sistema é alimentado por arquivos Arity-Prolog contendo os programas do usuário. O produto gerado é a representação gráfica do diagrama semântico correspondente à base de dados do usuário e/ou a proposições inferidas de uma base de dados a partir de uma meta, na tela. A visualização se dá através de uma janela que percorre o plano no qual a base de dados está representada. Ferramentas são oferecidas para facilitar a busca e a compreensão das informações contidas no diagrama.

A seguir, apresentaremos uma breve descrição do sistema que implementa o MDS, as estratégias usadas na construção do diagrama e a estrutura de dados utilizada.

7.2.2.5.1 Descrição do Sistema que Implementa o MDS

O sistema que implementa o MDS está subdividido nos seguintes módulos:

- Principal (MP)
- de Representação:
  - Tradução (MT)
  - Construção (MC)
  - Desenho (MD)
- de Inferência:
  - Busca (MI)

A figura 7.42 mostra um diagrama do relacionamento entre os módulos que compõem o sistema.

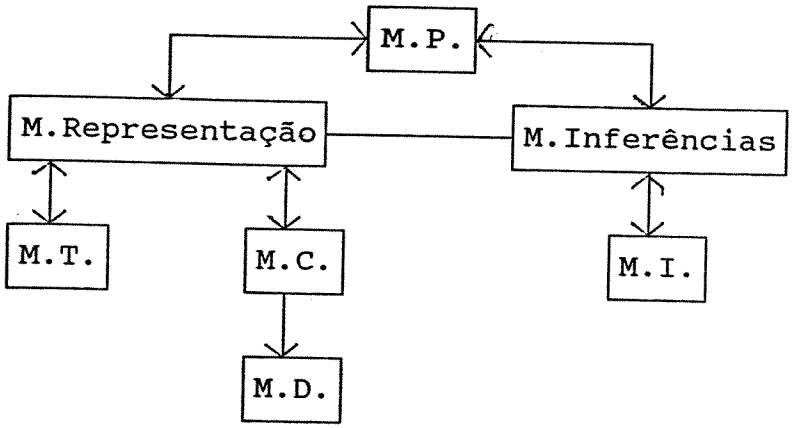


Figura 7.42 Diagrama do inter-relacionamento de módulos de MDS

O Módulo Principal (MP) controla as entradas e saídas, as estruturas de dados usadas e a interface com o usuário. A interface com o usuário utiliza, a exemplo do MOP, os recursos de menus e *dialog boxes* oferecidos pelo sistema Arity Prolog versão 5.1.

O Módulo de Representação (MR) é o responsável pela tradução da base de dados do formato Prolog (clausal) para o formato Diagrama Semântico. Para tal ele é constituído dos módulos: Módulo de Tradução (MT), Módulo de Construção (MC) e Módulo de Desenho (MD).

Inicialmente o programa do usuário é traduzido para uma estrutura de dados que representa o DS. Nessa representação, não há repetições de objetos e estes estão sempre associados a uma ou mais relações. A tarefa de tradução é feita por um sub-módulo do MR: o Módulo de Tradução (MT). O MT gera um arquivo

de nome "*ds_bd.tmp*", contendo a base de dados traduzida e mostra, na tela, as cláusulas traduzidas e indicação de início e fim do processo.

Outro sub-módulo do MR, o Módulo de Construção (MC) é o responsável pela aplicação de uma estratégia para construção do diagrama, a partir da estrutura de dados gerada pelo Módulo de Tradução (MT).

O Módulo de Construção gera uma estrutura de dados que armazena informações para desenho do DS. Essa estrutura é utilizada pelo Módulo de Desenho, para o desenho, propriamente dito, do diagrama semântico.

O Módulo de Representação, a partir da estrutura de dados gerada, permite listar todos os objetos/relações representados e fazer buscas e marcações de objetos/relações no DS, de forma a facilitar a visualização da base sendo representada.

O Módulo de Inferências é responsável por gerar proposições inferidas de uma dada base de dados, pelo interpretador Prolog, no processo de responder a uma meta colocada pelo usuário, para posterior representação em DS. Para tal ele é constituído pelo sub-módulo de Busca. O Módulo de Busca é o responsável pela meta-interpretação do programa do usuário, gerando um arquivo com os fatos intermediários, usados no processo de busca de uma resposta para a meta do usuário.

Os módulos do sistema são compostos dos seguintes arquivos de programas e dados:

Módulo Principal (MP):
arquivos de programas:
MDS.ARI: núcleo controlador da interação do MDS com o usuário
DIALDEFD.ARI: definição do menu principal do MDS e dos *dialog boxes* usados na interface do sistema.
DS.HLP: arquivo "help" do sistema.

Módulo de Representação (MR):

arquivos de programas:

TRAD.ARI: responsável pela tradução do formalismo Prolog para o formalismo DS.

CONST.ARI: responsável pela construção do DS.

DES.ARI: responsável pelo desenho do DS.

DES_CGA.ARI: configuração do desenho para vídeo CGA.

DES_EGA.ARI: configuração do desenho para vídeo EGA.

arquivos de dados (temporário): DS_BD.TMP

Módulo de Inferência (MI):

arquivos de programas:

INFER.ARI: responsável pela meta-interpretação de bases de dados

arquivos de dados:

SOLV*.ARI: soluções geradas a partir do arquivo INFER.ARI

7.2.2.5.2 Estrutura de Dados Utilizada

A estrutura de dados pode ser classificada funcionalmente em dois níveis: existe uma estrutura de dados que armazena a "tradução" da base de dados para a representação de diagramas semânticos do ponto de vista lógico e existe uma estrutura de dados que armazena a representação gráfica do diagrama para sua exibição na tela.

O Diagrama Semântico, do ponto de vista lógico, é construído pelo Módulo de Tradução (MT). Este módulo traduz a base de dados do usuário para uma estrutura de dados que representa as ligações entre os nós do diagrama.

Os dados estão organizados em registros armazenados na base de dados (*nodes*) e em uma árvore-B (*blocks*). Os registros "nodes" contém todas as informações sobre um dado objeto/relação.

Exemplos de armazenamento e recuperação de informações nas estruturas "nodes":

recordz(nodes,Info,Nref)

nodes é a chave de acesso à informação. *Info* representa a informação armazenada e *Nref* é o número de referência associado à informação armazenada na base de dados. A informação armazenada é uma lista do tipo:

[[Nome,Composição,Tipo],Outros,DSinfo]

Nome é o nome que ocupa o nó.

Composição é uma lista que representa com que outros nós o nó em questão está ligado

Tipo é uma estrutura do tipo *tipo(classe)*,onde tipo pode ser: *obj* representando um objeto ou *rel* representando uma relação. Classe pode ser: *atm* representando um átomo; *var* representando uma variável, ou *comp(str)* representando um objeto composto.

Outros é um espaço para novas extensões e não está sendo utilizado na atual versão do sistema.

DSinfo é preenchido pelo Módulo de Construção e contém: *Pos,* a posição do nó no diagrama; *Lista*, a lista de posições da sua perifieria, ocupadas por ligações com outros nós.

Exemplo:

recorded(nodes,[[Nome,Comp,Tipo],Outros,[Pos,Listaligs]],Nref)

[[vinho,

[[rel,~003303DD,2],[rel,~003303E8,2]], obj(atm)],

[],

bloco([9,11],

[3 - [~003303DD],4 - [~003303E8]])]

~003303E5

A árvore-B *blocks* possibilita acesso rápido às informações sobre um dado nó do diagrama. Exemplo de armazenamento e recuperação de informação na árvore-B *blocks*:

recordb(blocks,Chave,bloco(Nref,Composição)).

Chave é a lista *[Nome Tipo]* e tem como informação armazenada, o número de referência do nó na base de dados e com que outros nós ele está ligado.

Exemplo:

retrieveb(blocks,[Nome,Tipo],bloco(Nref,Comp))

[[vinho,obj(atm)],

bloco(~003303E5,[[rel,~003303DD,2],[rel,~003303E8,2]])

O Módulo de Construção (MC), a partir da estrutura de dados gerada pelo Módulo de Tradução, gera uma estrutura de dados para representação gráfica do DS na tela, a ser usada pelo Módulo de Desenho. Os dados para a representação gráfica estão organizados como registros na base de dados (*nodes* e *link*) e em uma árvore-B (*network*).

O registro *link* armazena informações sobre a aresta que liga dois nós distantes. Exemplo: *link([Xi,Yi],[Xf,Yf],Arg_i,Arg_f,_,_).*

O primeiro argumento representa a posição do objeto na tela. O segundo argumento representa a posição da relação na tela. O terceiro argumento representa a posição de chegada da aresta (1-8). O quarto argumento representa a posição de saída da aresta (1-8). O quinto e o sexto argumentos representam respectivamente o código do objeto e o código da relação.
Exemplo: *link([9,11],[12,11],3,1,_,_).*

A árvore-B *network* permite agilizar o acesso às informações sobre um dado nó do diagrama. Tem como chave a posição do nó no diagrama e o seu tipo. Armazena a chave de referência às informações desse nó em "nodes". Exemplo:

retrieveb(network,pos(Pos,Tipo),Nref)

pos([9,11],obj(atm)), ~003303E5

A implementação atual define uma estrutura de dados bastante abrangente no que se refere às informações armazenadas sobre a base de dados, para possibilitar futuras expansões.

7.2.2.5.3 Estratégias Utilizadas no Desenho do DS

Uma das dificuldades para o desenho do DS correspondente a uma dada base de dados está na sistemática de distribuição dos nós e arestas do diagrama semântico, no espaço limitado da tela, de forma a tornar o seu desenho o mais claro possível.

A heurística utilizada baseia-se no princípio de fazer com que as posições das relações sejam dependentes dos objetos associados ao maior número de relações.

A estratégia que adotamos divide o espaço de desenho em blocos, onde cada bloco contém a representação de um objeto ou de uma relação, ou está desocúpado. Para que se efetuem as ligações entre os nós, são acessadas informações contidas nos blocos dos nós origem e destino da ligação (relação e objeto, respectivamente). Os objetos são classificados pelos seus *fan in* (número de arestas que chegam nele) e as relações são classificadas pela soma dos *fan in* de seus objetos. Essa classificação fornece uma ordem de prioridade no desenho do diagrama.

Para dar início ao processo, é escolhido o objeto de máximo *fan in*. Todas as relações associadas a esse objeto são, então, preparadas, ficando disponíveis para representação. Dentre as relações disponíveis a de maior prioridade é selecionada para representação. Quando uma relação é representada, todos os objetos associados a ela passam a fazer parte de uma lista de pares (obj,rel), ordenados pelos seus *fan in*. Esses pares tem peso maior quando um dos seus elementos (objeto ou relação) já foram representados. Associado a essa relação, o objeto de *fan in* máximo é representado e, analogamente, o processo continua até todos os objetos e relações terem sido representados.

A distribuição do espaço a ser ocupado pelos elementos do diagrama deve ser gerenciado de forma a quebrar a tendência de o diagrama "se fechar sobre si mesmo", isto é, ficar confinado às regiões próximas aos objetos envolvidos. A solução adotada para contornar esse problema está baseada nas seguintes estratégias:

- Um novo nó (representando um objeto ou uma relação) é ligado a um nó já existente de forma a ficar o mais distante possível dos outros nós já interligados.

- Quando o diagrama "se fecha sobre si" de forma a não haver posições vagas na periferia de um nó X, para que um novo nó, associado a X seja representado, o algoritmo busca uma posição vazia o mais distante possível dos nós associados a X e alí representa esse novo nó.

Na versão atual da ferramenta, o número máximo de argumentos em um predicado está fixado em 8 (como no EDS) e o número de relações a que um objeto está associado também está fixado em 8. Isso se deve à distribuição física de espaço atual, onde temos no máximo 8 posições (blocos) em volta de um nó. A figura 7.43 mostra a distribuição atual.

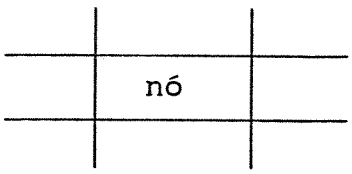


Figura 7.43 Representação da distribuição física de espaço relativa a um nó, na versão atual da implementação

Esse número pode ser expandido em futuras versões do sistema, adotando-se a seguinte estratégia: nós com *fan in* alto seriam representados de forma a ocupar uma área 9 vezes maior, permitindo a um nó ter 16 posições periféricas (e analogamente 32, 64, etc).

A figura 7.44 exemplifica a distribuição física de nós na tela (através de suas posições) e parte da estrutura de dados envolvida no desenho do diagrama que representa as cláusulas:

gosta_de(joao,vinho).

ladrao(joao).

quer(joao,vinho).

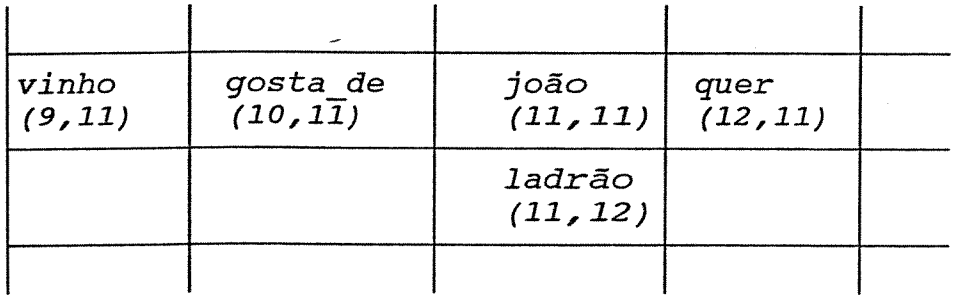


Figura 7.44 Exemplo de distribuição física de nós na tela indicando suas posições

Informações armazenadas em *nodes* com respeito ao nó ocupado pelo objeto "joão", considerando o exemplo acima:

[[joao O nó é nomeado "*joao*". Está associado à:

[[rel,~003303DD,1], relação *quer* (n. de referência na base de dados: ~003303DD)

[rel,~003303E1,1] relação *ladrão* (n. de ref.na b.d.: ~003303E1)

[rel,~003303E8,1] relação *gosta_de* (n.ref. na b.d.: ~003303E8)

obj(atm)] o nó representa um objeto atômico

[],

bloco([11,11], o nó ocupa o bloco de posição [11,11] na tela.

As ligações com os outros nós partem das direções (fig. 7.45):

[2-[~003303E1], 2- liga com relação *ladrão*

8-[~003303E8], 8- liga com relação *gosta_de*

4-[~003303DD]] 4- liga com relação *quer*

~003303E7 n. de ref. do objeto *joao* na b.d.

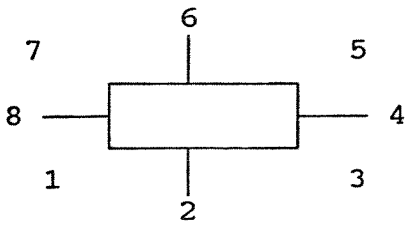


Figura 7.45 Indicação das ligações de um nó com outros

Informações armazenadas na árvore-B *blocks* a respeito do objeto *joão*:

[[joão,obj(atm)], nome e tipo do objeto

blocks(~003303E7, n. de ref. do objeto na b.d.

lista dos nós associados ao objeto joão, respectivamente, relações *quer, ladrão* e *gosta_de*

[[rel,~003303DD,1],

[rel,~003303E1,1],

[rel,~003303E8,1]])

Informações armazenadas na árvore-B *network*, a respeito do objeto *joão*:

pos([11,11],obj(atm)),	posição do nó na tela e tipo
~003303E7	n. de ref. do objeto na b.d.

Informações a respeito da ligação de nós distantes (no nosso exemplo, *vinho* e *quer*), armazenadas em *link*:

link([9,11],	posição do nó *vinho*, na tela
[12,11],	posição do nó *quer*, na tela
3,	direção da aresta, no nó *vinho*
1,	direção da aresta, no nó *quer*
)	

7.3 Síntese

A proposta do Módulo Declarativo é fornecer ao usuário uma representação do programa Prolog, onde o formalismo clausal é expresso de forma pictórica, através de diagramas semânticos.

Os relacionamentos codificados visualmente através da representação proposta levam a uma organização e síntese da informação contida na base de dados.

Acreditamos que através das ferramentas do Módulo Declarativo, o usuário tenha acesso aos aspectos "lógicos" da representação da solução do problema. O formalismo gráfico "organiza" a informação contida na base de dados e, dessa maneira, pode ser mais fácil para o usuário, reconhecer "inconsistências" em seu programa.

A abordagem ao programa de forma independente de como a máquina virtual atua sobre o programa, acreditamos ser fundamental principalmente para o novato-novato, que não teve experiência anterior com o formalismo de linguagens de programação e suas máquinas virtuais.

Capítulo 8

Estudo Experimental da Interação do Novato no Ambiente Proposto

Capítulo 8

Estudo Experimental da Interação do Novato no Ambiente Proposto

"The old order changed, yelding place to new."

A. L. Tennyson

Foi feito um acompanhamento da interação de novatos no ambiente Prolog acrescido das ferramentas propostas, com os objetivos de investigar a dinâmica de uso das ferramentas, investigar como se dá a construção do modelo conceitual da linguagem Prolog nesse ambiente e investigar qual é o papel das ferramentas nesse processo.

Participaram da investigação dois tipos de sujeitos que denominamos "novato-novato" e "novato-Prolog". O novato-novato representa a classe de sujeitos que é introduzida ao formalismo do computador através do ambiente proposto. A classe novato-Prolog representa os sujeitos que possuem uma vivência anterior com a mídia computacional, que supostamente carregam para o ambiente proposto.

Constituem o grupo novato-novato sujeitos sem prévia experiência com linguagens de programação ou mesmo com computadores (em sua maioria). Foram envolvidos nesse grupo 24 estudantes de escolas públicas da segunda série do segundo grau, que participavam de um projeto especial na Universidade[*1] (Projeto de Educação Matemática PFM). As atividades com Prolog foram tratadas como o tópico "computação", dentro do projeto PEM e sucederam uma sessão de três horas,

[*1] PEM - Projeto de Educação Matemática, IMECC, UNICAMP.

em laboratório, sobre conceitos básicos de uso do computador (comandos básicos do Sistema Operacional).

Constituem o grupo novato-Prolog sujeitos novatos no uso de Prolog, mas com boa experiência anterior em linguagens procedurais (Pascal, principalmente) e com computadores. Foram envolvidos nesse grupo três estudantes universitários, no segundo semestre do curso de Matemática Aplicada e Computacional. Os estudantes foram convidados a participar do estudo, não eram pagos e tratavam tais atividades como uma oportunidade para aprender Prolog.

8.1 Metodologia do Estudo

O estudo foi conduzido de forma a se ajustar às peculiaridades de cada grupo e aos objetivos da investigação. A metodologia adotada baseou-se, em parte, no método clínico de entrevistas onde o investigador interage com o sujeito individualmente, para quem é colocada determinada tarefa: algumas vezes escrever um programa (ou parte dele), algumas vezes explicar o que determinado programa "faz", algumas vezes predizer como o programa será executado, etc. Era esperado que as sessões contribuissem para o entendimento de Prolog.

Os sujeitos do grupo novato-novato trabalharam em dois tipos de cenário que denominamos "cenário-1" e "cenário-2". No cenário-1 o sujeito interage com o ambiente Prolog (interpretador e editor), realizando tarefas propostas apresentadas como "folhas-atividade", com um mínimo de interação com um experto (uma monitora[*2]). A monitora supervisiona o trabalho dos sujeitos nos computadores, com o objetivo de facilitar a interação dos mesmos com o ambiente, considerando sua inexperiência com computadores, procurando interferir o mínimo possível nas tarefas que os sujeitos estão realizando. São gravados em disco, as bases de

[*2] A monitora trabalhou como auxiliar de pesquisa, sob orientação da investigadora.

dados criadas (programas) e a interação dos sujeitos durante questionamento dessas bases (protocolos da interação).

No cenário-2 o sujeito interage no ambiente constituído pelas ferramentas e a investigadora, usando as mesmas "folhas-atividade". Aqui o papel da investigadora é interagir com os sujeitos a nível das tarefas que eles estão resolvendo, com o objetivo de investigar o processo pelo qual o novato passa durante as atividades. As interferências da pesquisadora não têm, portanto, o objetivo de ensinar, mas de tentar entender a situação sendo investigada.

O grupo novato-novato participou em uma situação de sala de aula, embora não houvesse aula expositiva ou a intenção de "ensinar" Prolog. Num primeiro momento eles trabalhavam no computador com o ambiente Prolog, em "folhas-atividade" propostas, supervisionados pela monitora (cenário-1). Num segundo momento eles eram expostos ao uso das ferramentas com as mesmas folhas-atividade e interagiam com a investigadora (cenário-2).

As atividades foram conduzidas em quatro sessões de três horas cada, uma vez por semana, conforme era previsto para as atividades do PEM. A sequência de sessões foi realizada em duas vezes, com metade do grupo a cada vez.

Um dos sujeitos do primeiro subgrupo foi convidado, após o término da experiência, a continuar o trabalho no cenário-2. Isto possibilitou observar mais detalhadamente e por um período mais prolongado, o processo evolutivo desse novato, em resolução de problemas no ambiente proposto. Participou, a partir de então de mais 11 sessões de duas horas e meia cada.

O grupo novato-Prolog participou em uma situação individual (cenário-2). O mesmo tipo de atividades propostas ao grupo anterior (novato-novato) foram trabalhadas, começando com as mesmas "folhas-atividade". Esse grupo participou de um total de 9 sessões de duas horas e meia.

Os protocolos gerados para análise são constituídos de: listagem dos arquivos "usr.log" (registro da interação do usuário); listagem de bases de dados (programas) criados e gravados em arquivos "*.ari"; folhas-tarefa com espaços para respostas preenchidos ou outro material escrito produzido pelos sujeitos; observações feitas pela monitora e anotações feitas pela investigadora.

Dinâmica

O grupo de 24 novatos foi dividido em dois subgrupos de 12, trabalhando cenários diferentes. No primeiro subgrupo todos foram expostos ao Módulo Operacional e ao Módulo Declarativo, quando trabalhando no cenário-2. No segundo subgrupo, metade foi exposta somente ao Módulo Declarativo e metade somente ao Módulo Operacional. Essa diferenciação dos grupos no cenário-2 permitiu observar em que medida existe a necessidade do trabalho combinado nos dois aspectos da linguagem: o declarativo e o operacional. Um dos sujeitos do primeiro grupo continuou o trabalho, no cenário-2, para podermos observar, num nível mais detalhado, o processo de uso do ambiente por esse novato.

Material/Tarefas

As "folhas-atividade" são um material escrito apresentado aos sujeitos, com instruções para realização de tarefas e espaço para preenchimento de respostas. As tarefas propostas envolvem: análise de bases de dados apresentadas, previsões para respostas da máquina virtual (sistema Prolog) a perguntas propostas, descrição do processo usado pela máquina virtual, depuração de bases de dados apresentadas, criação de novas bases de dados. Foram definidas quatro folhas-tarefa: Folha-Tarefa I, Folha-Tarefa B, Folha-Tarefa C e Folha_Tarefa A (incluídas no apêndice 2), uma para cada sessão, com tarefas que enfatizam cada uma das atividades mencionadas. Algumas bases de dados trabalhadas são problemas extraídos de

livros texto sobre Prolog (Clocksin, 1984), e de literatura sobre pesquisa de Prolog em educação (Hawley, 1987; Instructional Science, vols. 19,20, respectivamente 1990,1991).

A Folha-Tarefa I é usada para a exploração inicial da máquina virtual Prolog pelo sujeito. São apresentadas bases de dados e é pedido aos sujeitos que analisem as respostas obtidas das "perguntas" formuladas. É pedido também que o sujeito modifique/expanda a(s) base(s) de dados apresentada(s) e formule novas perguntas.

A folha-Tarefa B objetiva possibilitar ao sujeito a formulação de hipóteses sobre o comportamento da máquina, antecipando respostas do sistema Prolog a perguntas. São apresentadas ao sujeito bases de dados e perguntas. É pedido que o sujeito faça uma previsão do que seria respondido, verifique a resposta fornecida por Prolog e descreva o processo usado pelo sistema para chegar a uma resposta.

A Folha-Tarefa C é utilizada para investigar as estratégias de depuração utilizadas pelo sujeito frente à tarefa de identificar, localizar e corrigir erros em bases de dados. São apresentadas ao sujeito bases de dados para que ele verifique a ocorrência ou não de erros, explique a causa e proponha correção.

A Folha-Tarefa A é usada para investigar a tarefa de criação de bases de dados, pelo sujeito. São apresentados aos sujeitos "problemas" para os quais é pedido que ele crie uma base de dados.

A análise do trabalho dos sujeitos nessas atividades permitiu-nos investigar qual é o modelo conceitual inicial que o sujeito tem da máquina virtual, como esse modelo evolui ao longo das sessões de trabalho, quais são os principais erros conceituais (interpretações errôneas que o sujeito faz da máquina virtual) e como as ferramentas trabalham esses aspectos.

8.2 Resultados da Análise do Grupo I (novato-novato)

O grupo novato-novato representa a classe dos sujeitos que não têm experiência anterior com computadores. Dessa maneira é através da intermediação do ambiente proposto que o novato-novato constrói seu conhecimento sobre o computador. Para o novato desse grupo, o computador é entendido em função do funcionamento da sua linguagem de comunicação (máquina virtual). Dessa maneira, o conhecimento sobre o computador é construído simultaneamente ao conhecimento sobre a linguagem de programação (Prolog, neste caso), à medida em que o sujeito interage no ambiente proposto.

O estudo que fizemos para esse grupo de novatos procurou levantar aspectos gerais dos processos nos quais o novato-novato é envolvido interagindo no ambiente proposto em atividades envolvendo "entendimento", depuração e criação de bases de dados. Uma análise da observação do grupo ao longo das quatro sessões levantou vários aspectos que são importantes para nosso entendimento de "quem" é esse tipo de novato e como ele interage num ambiente de computador. Verificamos que esse tipo de sujeito, apesar de sua inexperiência, traz para o ambiente um modelo conceitual inicial da máquina com a qual ele se comunica, que evolui ao longo das sessões. Os erros conceituais do sujeito ao longo do processo são um reflexo do modelo conceitual que ele tem da máquina virtual, num determinado momento. Nas situações de erro, o *feedback* gerado pelas ferramentas do ambiente, contribuem de forma decisiva para a realimentação do ciclo hipótese-criação-execução-feedback, descrito no capítulo 2, e consequente modificação no modelo conceitual da linguagem que ele está aprendendo.

Os resultados de nossa análise, apresentados a seguir, apontam para aspectos que nos ajudam a explicar que tipo de compreensão o sujeito tem da base

de dados Prolog, enquanto objeto de sua comunicação com a máquina virtual e de que maneira as ferramentas propostas interferem nessa compreensão. Nossa discussão é ilustrada por protocolos extraídos das tarefas realizadas pelos sujeitos, pela fala dos sujeitos (identificados por três letras de seu nome) e por figuras que ilustram *feedbacks* das ferramentas durante as atividades.

Um estudo mais prolongado desse tipo de novato foi feito com um desses sujeitos ao longo de onze seções, focalizando mais detalhadamente o processo de resolução de problemas no ambiente proposto. Um estudo de caso desse sujeito será apresentado na sessão 8.3.

8.2.1 O Modelo Conceitual Inicial da Máquina Virtual Prolog

Para esse grupo de novatos, o conhecimento prévio do que é o computador se mistura ao conceito que ele constrói sobre o que é "Prolog". O novato se comunica com o computador, através desse sistema de linguagem onde, a partir de um conjunto de informações sobre determinado domínio (criado pelo sujeito ou por outro) a máquina "responde" a "perguntas" formuladas. Essa intermediação da linguagem faz com que o computador seja visto em função de seu intermediador (a linguagem) e, como tal, Prolog passa a ser uma "máquina virtual". A imagem mais comum que esse tipo de novato costuma ter, previamente, do computador, parece ser baseada no animismo representado na figura do "respondedor de perguntas".

Ao fim da primeira sessão, onde havia sido utilizado apenas o Módulo Declarativo, foi pedido aos sujeitos que descrevessem como eles imaginam que é o processo usado pela máquina para responder as perguntas. A seguir mostramos alguns protocolos, transcritos nas formas apresentadas, que ilustram seus modelos conceituais iniciais.

A máquina virtual dotada de "inteligência":

"Através da lógica o computador responde essas perguntas, como se ele estivesse pensando (impossível), mas parece. Adorei a aula." (Hel)

A máquina virtual como uma grande "memória":

"Acredito que deve haver um arquivo bem detalhado, com todas as perguntas e respostas possíveis." (Ric)

"O computador tem os dados na Base de Dados1. E de acordo com esses dados, que estão na memória, Ele responde com as informações que ele tem. Responde Não a qualquer coisa que não tiver no arquivo." (Fab)

A máquina virtual dotada de um mecanismo:

"O computador, a meu ver, realiza as respostas das perguntas que fazemos através de uma análise combinatória entre os fatos listados na base de dados." (Sol)

A máquina virtual dependente do "conhecimento" expresso nas cláusulas:

"O computador considera cada afirmação que há no seu banco de dados. Cada afirmação é composta de elemento(s) em que há uma relação para ele (ou entre eles). Conforme estas afirmações é que ele responde as afirmações." (Cla)

Neste depoimento o sujeito já menciona os conceitos de cláusula ("afirmação"), objetos ("elementos") e relações. Cada cláusula contém um "conhecimento" que é usado pelo computador para responder as perguntas. Este depoimento sugere um reflexo do trabalho do sujeito no Módulo Declarativo (a ferramenta EDS foi usada na primeira sessão)

8.2.2 Os Erros Conceituais dos Sujeitos e o Feedback das Ferramentas

Os erros conceituais dos sujeitos durante a realização das atividades propostas, em geral, são um reflexo do modelo conceitual da máquina virtual que o sujeito tem no momento em questão e que é modificado à medida em que ele interage no ambiente. Inicialmente, a informação presente na base de dados é estendida de forma a incorporar informações de "bom senso" ou é restringida a informações presentes de forma explícita (excluindo informações que podem ser deduzidas dela), dependendo de como é o modelo inicial que o novato tem da máquina virtual. O formalismo utilizado para escrever as sentenças da base de dados é confundido com uma nova forma sintática para a sentença equivalente em lingua natural. A forma de apresentação da base de dados como um "texto" composto de linhas escritas consecutivamente, influencia a maneira como o sujeito "interpreta" a base de dados. O significado que o sujeito atribui à base de dados é influenciado não somente por sua forma de apresentação, mas também pelo conteúdo veiculado. O entendimento da base de dados tem a interferência das associações que são feitas de seu conteúdo com o discurso em lingua natural (no sentido de ajudar ou de confundir). Esses aspectos são ilustrados e comentados a seguir.

Para facilitar a leitura das próximas sessões, apresentamos a seguir as bases de dados referidas na análise.

[gosto]:

```
gosto(X):-vermelho(X), carro(X).
gosto(X):-azul(X),bicicleta(X).

vermelho(cereja).
vermelho(tijolo).
vermelho(fusca).

carro(mercedes).
carro(santana).

azul(jeans).
```

```
azul(honda).
azul(vidro).
bicicleta(caloi).
bicicleta(honda).
bicicleta(minha).

[local]:

local(P,L):-em(P,L).
local(P,L):-visita(P,O), local(O,L).

em(alan,sala19).
em(jane,sala54).
em(bete,escritorio).

visita(davi,alan).
visita(alberto,davi).
visita(janete,bete).


[flu2]:

p:-a,b,c.
p:-a,b.
a.
a.
b.


[flv2]:

p(X):-a(X),b(X),c(X,Y).
a(jane).
b(jane).
c(sue,helen).
c(helen,fred).


[idade]:

mais_velho(X,Y):- IX > IY,
                  idade(X,IX),
                  idade(Y,IY).

idade(joao,17).
idade(tom,18).
idade(sueli,15).


[familia]:

pai_de(maria,joao).
mae_de(pedro,maria).
pai_de(pedro,paulo).
```

```
pais_de(X,Y):-mae_de(X,Y).
pais_de(X,Y):-pai_de(X,Y).
ancestral_de(X,Y):-pais_de(X,Z),ancestral_de(X,Y).
ancestral_de(X,Y):-pais_de(X,Y).

[soma]:

soma([],0).
soma([X|Y],Z) :- soma(Y,Z1), Z is X + Z1.

[ladrao]:
ladrao(joao).
mulher(maria).
gosta(maria,doce).
gosta(maria,vinho).
gosta(joao,X):-gosta(X,vinho),mulher(X).
rouba(X,Y):-ladrao(X),gosta(X,Y).
```

A Interferência da Linguagem Natural no Formalismo Clausal para Representar Sentenças

O contexto da Linguagem Natural mistura-se ao formalismo de escrever sentenças na forma clausal, interferindo na compreensão do sujeito sobre o formalismo do computador. O sujeito, tem dificuldades, por exemplo, em identificar os elementos básicos desse formalismo; ele usa o mesmo termo, ora como argumento, ora como predicado, as variáveis não são entendidas como tal. Escrever sentenças na forma clausal é confundido com escrever sentenças em lingua natural, respeitando uma certa sintaxe. Exemplos de cláusulas criadas pelo sujeito, que ilustram essa dificuldade:

pode_prender(W,joao):-joao(Y).

gosta_fugir(joao,cadeia):-cadeia(Y).

pode_roubar(joao,W):-ladrao(Y).

gosta(caio,Z):-gosta(H,leite).

pode_pegar(Y,Z):-lapis(X),porque_de(Y,Z).

As figuras 8.1, 8.2, 8.3 e 8.4, a seguir, mostram a representação em diagrama semântico da base de dados composta das cláusulas acima, geradas pelo MDS.

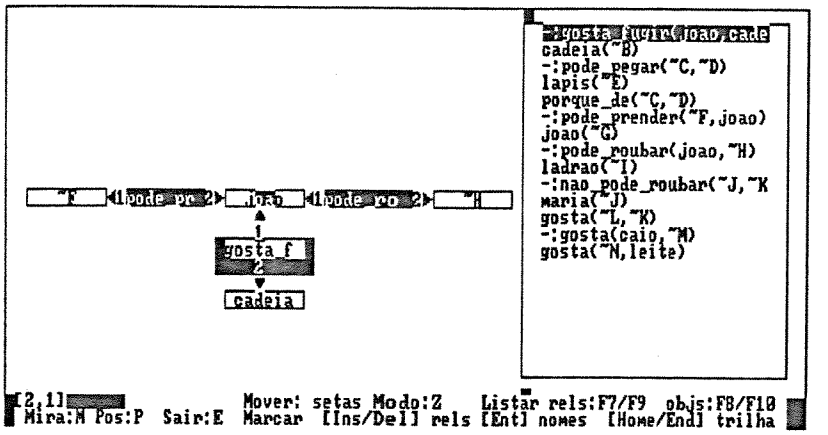


Figura 8.1 Representação em Diagrama Semântico da base de dados do sujeito. Em destaque parte da regra *gosta_fugir(joão,cadeia):-cadeia(B)*

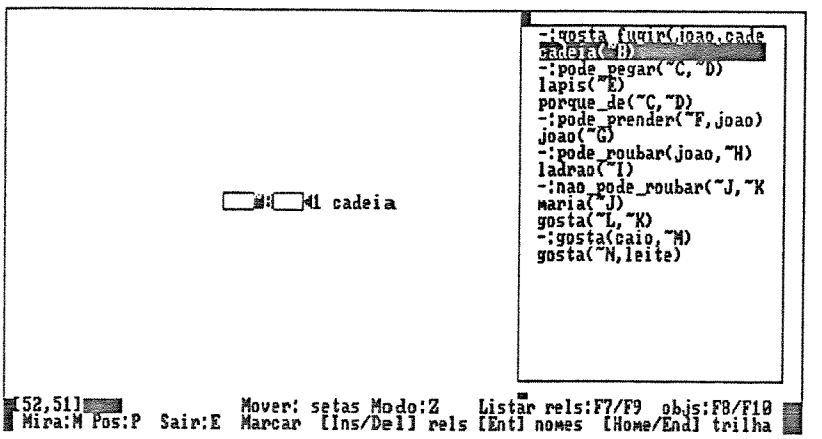


Figura 8.2 Representação do corpo da regra, de forma desconectada de sua cabeça

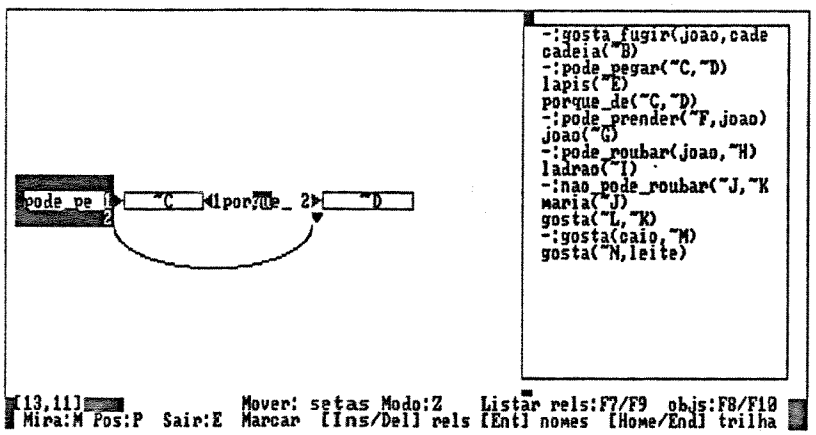


Figura 8.3 Representação de parte da regra *pode_pegar(C,D):-lapis(E), porque_de(C,D)*

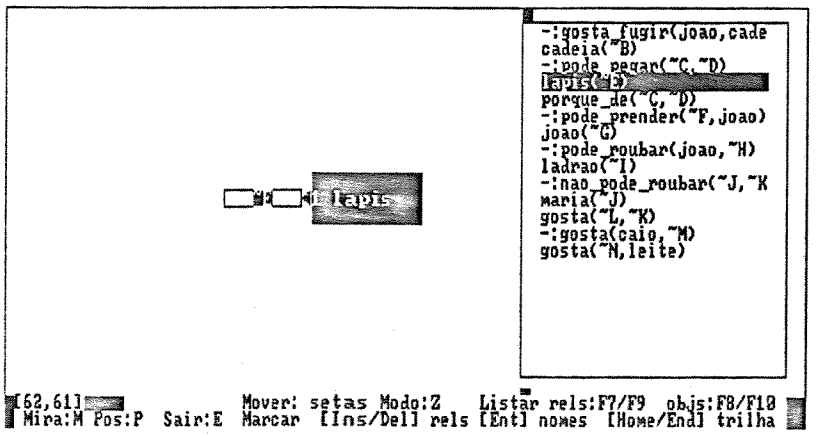


Figura 8.4 Representação da parte *lapis(E)*, desconectada das outras partes da regra

As figuras 8.1 e 8.2, por exemplo, explicitam a falta de conexão entre as partes consequente e antecedente da regra *gosta_fugir(joao,cadeia):-cadeia(B)*, representadas por informações em diagramas diferentes. O mesmo acontece em relação à regra *pode_pegar(C,D):-lapis(E), porque_de(C,D)*, onde parte da informação da regra fica desconectada, como representado nas figuras 8.3 e 8.4 respectivamente.

O mesmo efeito é mostrado, a nível de cláusula, pelo EDS, conforme mostram as figuras 8.5 e 8.6 a seguir. Na figura 8.5 *cadeia* é usado como objeto (representado verticalmente juntamente com os objetos *joao* e *X*) e como relação (apontando para o objeto *X*). Na figura 8.6, a variável *Z* aparece "isolada" das demais e da parte consequente da regra, indicando um conhecimento "desvinculado" dos demais na regra, quando não parecia ser essa a intenção do sujeito.

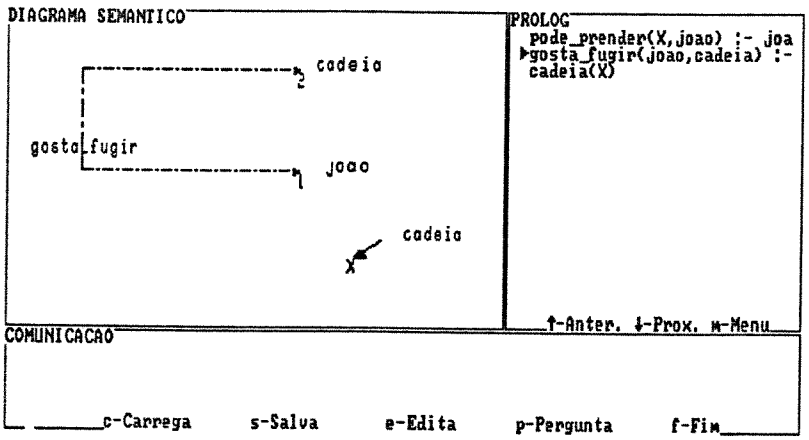


Figura 8.5 Diagrama semântico da cláusula indicada na janela PROLOG, mostrando o isolamento da informação associada a X

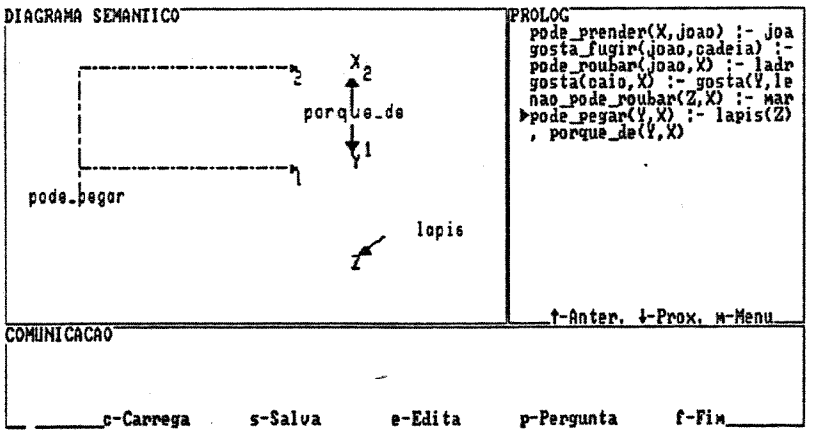


Figura 8.6 Diagrama semântico da cláusula indicada na janela PROLOG, com Z desconectado das outras partes da regra

O Módulo Declarativo "explicita" essa inconsistência, gerando um *feedback* em relação a o que é objeto na cláusula e o que se fala dele (relação). Para o novato o *feedback* do diagrama semântico aparece como uma forma de "organizar" a informação e é visto pelo novato como a forma como Prolog "entende" a cláusula, ou o conjunto de cláusulas.

O Contexto Estendido

Estamos denominando "contexto estendido" o uso de informações que não estão presentes na base de dados, mas são, por exemplo, "senso comum". Tais informações são usadas nas previsões de resposta da máquina, como se fossem uma extensão natural das informações contidas na base de dados. O sujeito assume que o "mundo" já está *a priori* na máquina.

Exemplos de protocolos que ilustram esse tipo de interpretação para a base de dados:

em [ladrao], para a pergunta *gosta(maria,chocolate)*, a resposta prevista é baseada na informação de senso comum que relaciona "chocolate" a "doce".

"*sim se chocolate for considerado como doce*" (Cla).

"*sim porque chocolate é doce*" (Hel).

em [gosto], para a pergunta *gosto(X)*, encontramos, por exemplo, tipos de antecipação da resposta, que assumem presente na máquina a informação "fusca é um carro":

"*sim, se X for carro vermelho (fusca);*

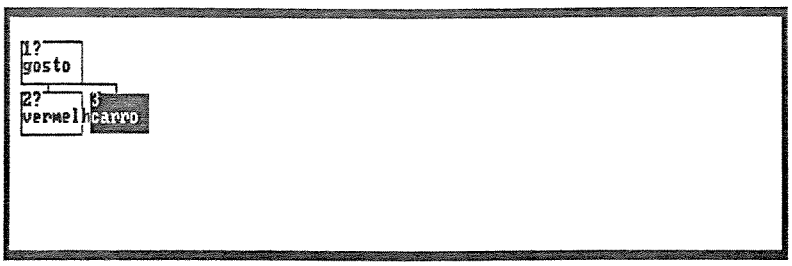
X for bicicleta honda azul " (Hel)

"*vermelho(fusca)*" (Fab).

"*gosto(honda)*

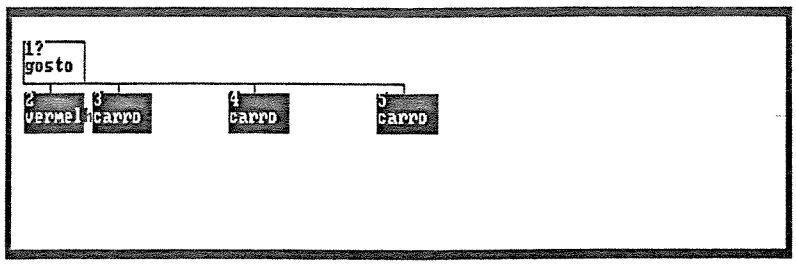
gosto(fusca)" (Taí)

A seguir mostramos parte do processo de inferência gerado pelo MOP, ao buscar resposta para a meta *gosto(X)*.



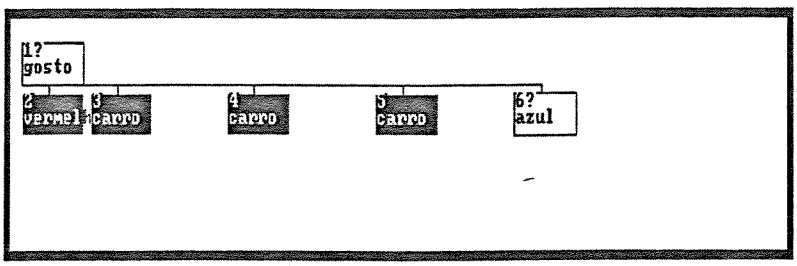

```
Regra: 1 --> gosto(cereja) :- vermelho(cereja) , carro(cereja)
Meta: carro(cereja)
```

Figura 8.7 Aplicação da regra 1 de *gosto* e falha da sub-meta *carro(cereja)*




```
Regra: 1 --> gosto(fusca) :- vermelho(fusca) , carro(fusca)
Meta: carro(fusca)
```

Figura 8.8 Momento da aplicação da regra 1 de *gosto* com falha da sub-meta *carro(fusca)*




```
Regra: 2 --> gosto(A) :- azul(A) , bicicleta(A)
Meta: azul(A)
```

Figura 8.9 Momento seguinte ao da figura 8.8, quando é aplicada a regra 2 de *gosto*

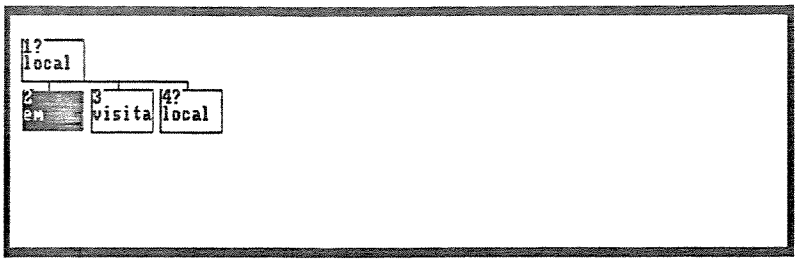
A figura 8.7 mostra a regra 1 sendo aplicada (*gosto(X):-vermelho(X),carro(X)* com *X* substituído por *cereja*) com fracasso da sub-meta *carro(cereja)*, indicado pelo nó 3 em fundo escuro. Os nós 3, 4 e 5 representam, respectivamente, as tentativas de satisfazer as sub-metas *carro(cereja)*, *carro(tijolo)* e *carro(fusca)*. A figura 8.8 mostra o fracasso da regra 1 após o fracasso da submeta *carro(fusca)*, para surpresa do sujeito, que tinha como "certo" que "fusca é um carro". A figura 8.9 mostra o momento do retrocesso, a partir da aplicação da cláusula alternativa (regra 2), dando continuidade ao processo.

As ferramentas do Módulo Operacional mostram as informações da base de dados efetivamente em uso no processo de responder a uma pergunta. Dessa maneira, através de seu uso o sujeito começa a isolar o conhecimento de bom senso do conhecimento efetivo presente na base de dados.

O outro Extremo do Contexto Estendido: O Contexto Restrito

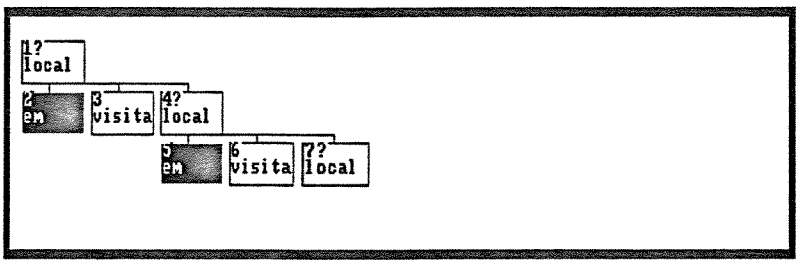
Num extremo temos o sujeito usando "senso comum" como se fosse informação da base de dados e o seu processo de inferência como se fosse um complemento ao da máquina (exemplos anteriores); num outro extremo temos o sujeito que vê o processo da máquina responder perguntas como uma busca direta das informações que estão explícitas na base de dados (fatos).

Um dos sujeitos, (Taí) em [local], responde, como antecipação da resposta para a pergunta *?local(alberto,X)*, *davi*, fazendo referência ao fato *visita(alberto,davi)*, presente na base de dados. A árvore de execução mostra ao sujeito como acontece a busca da informação que não está explícita em um fato na base de dados, conforme mostram as figuras 8.10 e 8.11, a seguir.




```
Regra: 2 --> local(alberto,A) :- visita(alberto,davi) , local(davi,A)
Meta: local(davi,A)
```

Figura 8.10 Aplicação da regra 2 de *local* após falha da regra 1 (representada no nó 2)




```
Regra: 2 --> local(davi,A) :- visita(davi,alan) , local(alan,A)
Meta: local(alan,A)
```

Figura 8.11 Repetição da mesma situação para *local(davi,A)*

A figura 8.10 mostra o momento do processo em que a regra 2 está sendo aplicada, após falha da regra 1 (*em(alberto,A)*), representada pela falha no nó 2. A figura 8.11 mostra que o mesmo processo foi aplicado à sub-meta *local(davi,A)* (nós 5, 6 e 7) mostrando o "desdobramento" do processo recursivo, através da hierarquia de metas.

Tai (o sujeito mencionado anteriormente), ao usar o Módulo Operacional, explica:

"na semana passada eu não tinha entendido que ele ia procurar, se não tivesse escrito na base de dados".

"O computador, para responder a pergunta procura se a informação já é dada diretamente. Se a informação não estiver lá diretamente, ele vai procurando através de outras informações e condições da base de dados, até achar a resposta da pergunta (meta)" (Taí)

O Módulo Operacional mostra ao sujeito que o processo de procura envolve também a busca de informações que não estão explícitas através de fatos, mas que podem ser deduzidas.

O Módulo Declarativo, por sua vez, através da representação em diagramas semânticos, poderia levar o sujeito em questão a fazer uma meta-análise nos fatos e regras contidos de forma textual na base de dados (como aconteceu a outros sujeitos do grupo). A figura 8.12, a seguir, mostra o diagrama semântico de inferências para a meta *local(alberto,X)*, onde pode-se ver representada a rêde de informações que dá origem à resposta *sala19*.

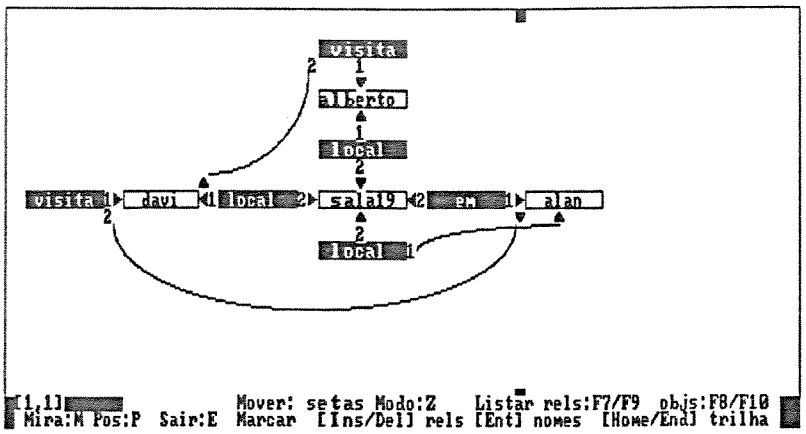


Figura 8.12 Representação em diagrama semântico da solução para a meta *local*(alberto,X)

A diferença de enfoque do Módulo Declarativo em relação ao Operacional objetiva levar o aprendiz à uma análise da "lógica" entre as informações contidas ou deduzidas da base de dados, sem a preocupação com o conhecimento da máquina de inferência como intermediadora do processo.

A informação textual da base de dados

A informação na base de dados é vista, algumas vezes, em função da disposição das cláusulas no texto. Assim, na base de dados [idade], as sub-metas idade, (como *idade(X,IX)*) na regra *mais_velho* são confundidas com fatos por estarem colocados alinhados e em linhas diferentes, de forma semelhante aos fatos *idade* (*idade(tom,18)* por exemplo).

O Módulo Declarativo permite mudar essa perspectiva mostrando a informação textual da base de dados através do formalismo gráfico dos diagramas semânticos. As regras são representadas por diagramas e aparecem agrupadas a

outras cláusulas se tiverem objetos em comum. A figura 8.13, a seguir, mostra parte da representação da regra *mais_velho*, com a sub-meta *idade(B,D)* conectada.

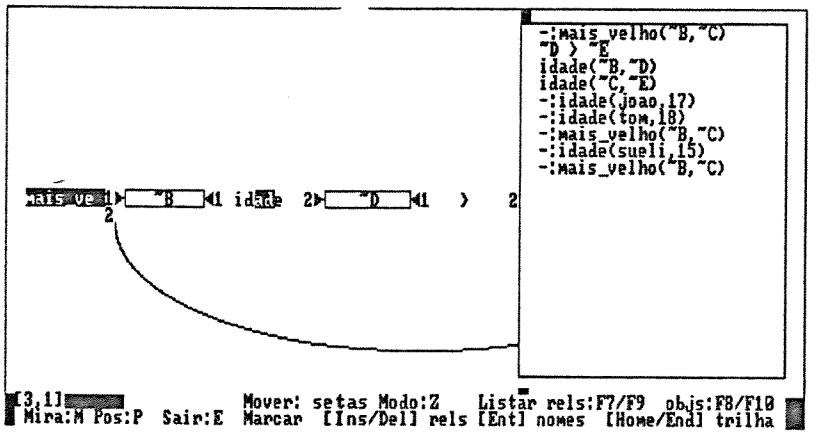


Figura 8.13 Representação da regra *mais_velho*, com a sub-meta *idade(B,D)* conectada

A Imprevisibilidade das Bases de Dados Abstratas

Estamos denominando Bases de Dados Abstratas, às bases de dados sem informação contextual, como por exemplo em [flu2], descrita a seguir:

p:-a,b,c.

p:-a,b.

a.

a.

b.

Em tais bases de dados, muitos sujeitos não conseguem antecipar respostas a perguntas formuladas. Em geral, o entendimento do significado operacional é "influenciado" pela interpretação que o sujeito faz do conteúdo da base de dados.

Na ausência de contexto elas ficam "sem significado" para o sujeito. Um dos sujeitos (Car) explica porque não consegue antecipar resposta a perguntas envolvendo bases de dados abstratas:

"*Se não tiver o computador não tem jeito de responder*".(Car)

Esses sujeitos só foram capazes de fazer previsões para esse tipo de bases de dados após o uso das ferramentas do Módulo Operacional. Nessa atividade, a visualização de como a base de dados é operada pela máquina, muda a perspectiva do novato para o plano do "concreto". Esse resultado concorda com estudos feitos sobre o tipo de "lógica" usado pelo adulto na manipulação do computador, que mostra que "*adultos diante do computador manifestam muitos dos comportamentos que Piaget atribuiu às crianças nos estágios sensório-motor e pré-operacional* (Valente, 1988, p. 33) e precisam manipular o novo conceito de uma maneira concreta para formar teorias elementares a seu respeito.

A figura 8.14, por exemplo, mostra a base de dados mencionada, sendo tratada operacionalmente pela máquina de inferência, para responder à pergunta *?p*.

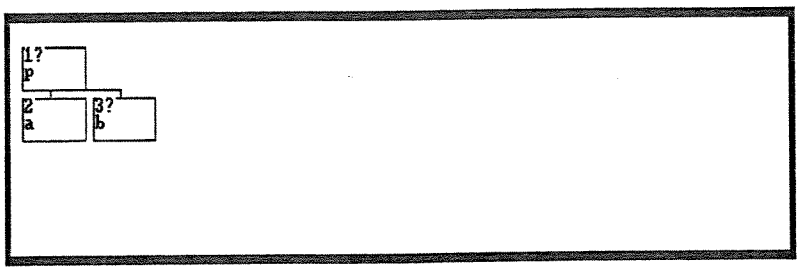


Figura 8.14 Representação da árvore de busca para a meta *p*, em construção

Ou seja, para responder sobre *p* a máquina deve responder sobre as sub-metas *a* e *b* e *c* e assim por diante. As ferramentas instrumentalizam o sujeito para "entender" esse conjunto de cláusulas sob o aspecto operacional, de forma independente de informação contextual.

8.2.3 As Estratégias de Depuração na Ausência de Feedback

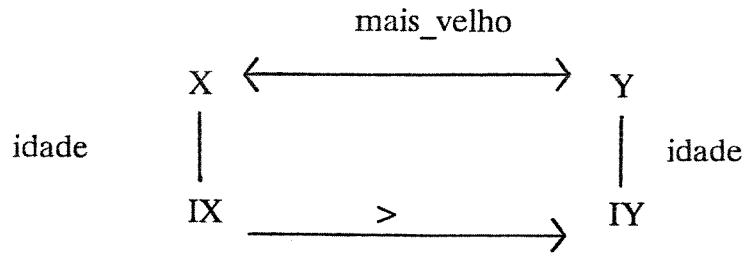
No trabalho de depuração de uma base de dados apresentada ao sujeito, a situação de erro é constatada, em geral, pela resposta negativa de Prolog às perguntas, ou por uma resposta diferente do que seria esperado. Sem o uso de ferramentas, dois tipos de comportamentos foram encontrados entre os sujeitos na situação de erro: interrupção do ciclo (hipótese - modificação da base de dados- execução - *feedback*) e eliminação do sintoma, que ilustraremos a seguir.

Interrompendo o ciclo

Alguns sujeitos, a partir de uma análise da base de dados, não conseguindo formular nenhuma hipótese sobre a causa do erro, interrompem o ciclo e não conseguem prosseguir na atividade.

Por exemplo, em [idade], quando perguntado o que estava errado na base de dados, um dos sujeitos (Rog) responde "nada" e desenha no papel o diagrama semântico da base de dados (mesmo sem usar o Módulo Declarativo), para mostrar que "nada" estava errado com a base de dados em questão. (ver reprodução do protocolo a seguir). Vista pelo aspecto lógico/declarativo não há nada de errado, mesmo. O erro existe do ponto de vista da semântica operacional envolvida (a ordem das sub-metas não permite que o mecanismo de inferência encontre as respostas). Apesar da constatação de erro (Prolog não respondeu às perguntas

como era esperado), o sujeito ficou sem ação, não conseguindo prosseguir, na atividade de depuração.



Reprodução de Protocolo 1 de Rog

Este exemplo mostra o problema em seus dois aspectos: o lógico/declarativo e o operacional e a necessidade de ambos os conhecimentos, para o novato prosseguir. A resposta *no* de Prolog não é suficiente para o sujeito saber o que fazer em seguida. Nesse caso falta a informação do processo que leva a máquina a responder *no*.

Eliminando o sintoma

Outro comportamento encontrado, com frequência na situação de erro, foi a modificação da base de dados no sentido de eliminar o sintoma, isto é, fazer com que a constatação do erro não apareça. Essa, em geral, é a estratégia utilizada pelo sujeito que não tem *feedback* suficiente para localizar a causa do erro, mas apenas seu sintoma.

A "correção" da base de dados [familia], por alguns dos sujeitos, através da retirada da cláusula que leva à situação de erro (a regra recursiva de ancestral), ilustra a eliminação, pura e simples, da causa do erro (*looping*).

Outra maneira de eliminar o sintoma do erro, envolve modificar o texto da base de dados, de forma a "impor" à máquina a resposta desejada. Ilustrando esse comportamento, quando perguntado se havia algo errado na base de dados [idade], um dos sujeitos respondeu:

"*não existe erro*".
"*se IX for 18, o X será tom, ou se o IX for 17 (X,joao), o IY será 15 e o Y a sueli*". (And)

Após constatação de falha, pela resposta *no* de Prolog, o mesmo sujeito propôs como modificação:

"*mais_velho(tom,joao):-18>17*" (And)

Aqui, o sujeito propõe como solução uma regra que "força" a resposta correta (é quase um fato, porque é verdade que 18 é maior que 17). Essa necessidade de "fazer funcionar" é função de, em a "lógica" estando correta, faltar um *feedback* que mostre a base de dados do ponto de vista operacional (como a máquina está usando a informação da base de dados para responder as perguntas).

Outros protocolos que ilustram o mesmo tipo de situação:

explicação do erro:
"*mais_velho(X,Y):-IX > IY, idade(X,IX) > idade(Y,IY). precisa explicar a relação de idade(X,IX) com idade(Y,IY).*"(Sab)

solução proposta:
"*idade(tom,18). idade(joão,17). mais_velho(tom,joão):-18>17.*" (Sab)

Nesta proposta de solução, o sujeito coloca os fatos relativos às idades de *tom* e *joao* antes da regra, interpretando a base de dados como uma sequência (textual) de informações, conforme já foi discutido em 8.2.2.

O efeito da "concretização" (comentado no item 8.2.2) aparece no exemplo de solução, a seguir, onde o sujeito estabelece, no lugar da máquina, as relações de "quem é mais velho do que quem" e porquê:

"*mais_velho(joão,tom):- 17>18.*
mais_velho(joão,sueli):-17>15.
mais_velho(tom,joão):- 18>17.
mais_velho(tom,sueli):- 18>15.
mais_velho(sueli,joão):- 15>18.
mais_velho(sueli,tom):- 15>17"

8.2.4 Obtendo Feedback a partir do Uso das Ferramentas

Enquanto que no ambiente sem as ferramentas (cenário-1) o erro é constatado pela resposta de Prolog apenas, o uso das ferramentas permite ao sujeito uma busca do "porquê" do erro, no sentido de relacionar a resposta indesejada com alguma anomalia vista através das ferramentas, ou no processo, ou na "lógica" da base de dados.

As figuras 8.15, 8.16 e 8.17, mostram a sequência do processo de execução da meta *mais_velho(tom,joao)*, visto através do Trace Estendido, do Módulo Operacional.

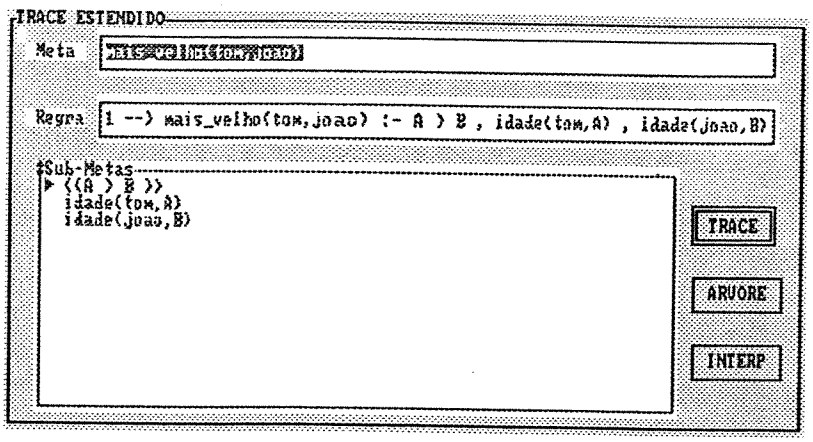


Figura 8.15 Execução da meta *mais_velho(tom,joao)* (momento 1)

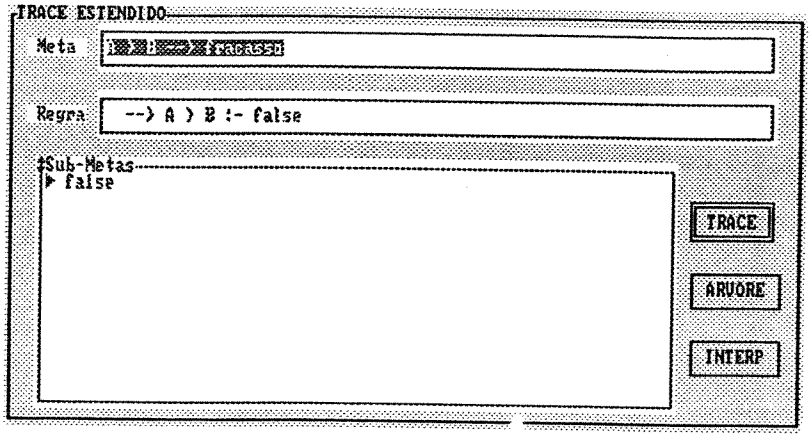


Figura 8.16 Execução da meta *mais_velho(tom,joao)* (momento 2)

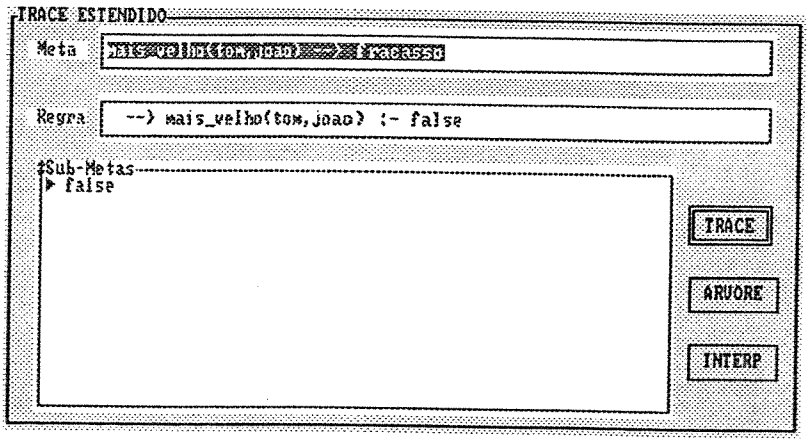


Figura 8.17 Execução da meta *mais_velho(tom,joao)* (momento final)

Rog, o mesmo sujeito mencionado anteriormente no caso de "interrupção do ciclo", após o uso do Módulo Operacional (no cenário-2) apresenta a seguinte resposta à pergunta "que erro foi detectado, qual a causa do erro ?":

"*para o micro associar as informações da regra, este as associa na ordem dada. Portanto, não há relação entre: mais_velho(X,Y):-IX>IY, ocorrendo falta de informações. Para corrigir a regra, deve-se colocar primeiro as informações sobre a idade de cada um [idade(X,IX)] e só depois a relação entre as idades.*" (Rog)

A localização do erro, entretanto, nem sempre é suficiente para a correção imediata do código. Alguns sujeitos, apesar de "localizarem" o problema, não conseguem fazer uma correção efetiva logo em seguida. No exemplo anterior, um dos sujeitos descreve o problema como:

"*a ordem das condições está errada pois o computador precisa, primeiramente, saber a idade dos elementos (X e Y) para depois compará-las. Porém, da maneira que a Base de Dados está escrita, o computador vai primeiro comparar as idades antes de sabê-la. E isso é uma operação ilógica para o prolog e para qualquer um.*"(Sol)

correção proposta:

"*idade(X,IX),*
idade(Y,IY).
mais_velho(X,Y):-IX > IY.
idade(joão,17).
idade(tom,18).
idade(sueli,15)." (Sol)

Apesar de perceber o erro e descrever muito bem a situação do ponto de vista operacional, o sujeito faz uma alteração na base de dados que apresenta outro tipo de erro (a base de dados vista de forma textual, referido anteriormente no item 8.1.2). O importante é que o ciclo não foi quebrado, existe uma hipótese sobre a causa do erro e uma correção proposta, que o realimentam podendo levar o sujeito à correção efetiva, viabilizando o processo de aprendizado do sujeito.

O *feedback* gerado não precisa, necessariamente, acontecer através dos aspectos operacionais envolvidos. Na base de dados [família], por exemplo, um dos sujeitos usou o Módulo Declarativo para ver os diagramas semânticos correspondentes às cláusulas e respondeu:

"*a letra Z está isolada, fora de lugar. O erro está nos ancestrais de X e Y. Z são os pais de X, e Y são os pais de Z* " (Hel).

8.2.5 A Atividade de Criação de Bases de Dados

A atividade de criação de bases de dados pelo sujeito, sugere, em geral, o modelo conceitual da máquina de inferência em uso pelo sujeito no momento em questão, o tipo de *bugs* que persistem e o tipo de ferramenta que poderia ser utilizada para lidar com a situação. A seguir apresentamos alguns protocolos mostrando vários estágios dos sujeitos em relação à construção de bases de dados. O "problema" escolhido pelos sujeitos era o de criar uma base de dados para responder sobre características e informações gerais sobre animais.

Exemplo 1.

"passaro(X). %1

ave(passaro):-setem(X,pena). %2

voa(sabia):-sefor(sabia,leve). %3

sabia(X):-sefor(X,passaro). %4

pena(sabia). %5

animal(Y):-segosta(Y,voar). %6

gosta(Y,voar):-segosta(Y,passaro)". %7

A criação de base de dados representa um exercício onde o sujeito explicita, não apenas suas dificuldades com Prolog, mas suas dificuldades conceituais com relação ao domínio (contexto) sendo tratado. As figuras 8.18, 8.19 e 8.20 mostram os diagramas semânticos das cláusulas 2, 7 e 4 respectivamente.

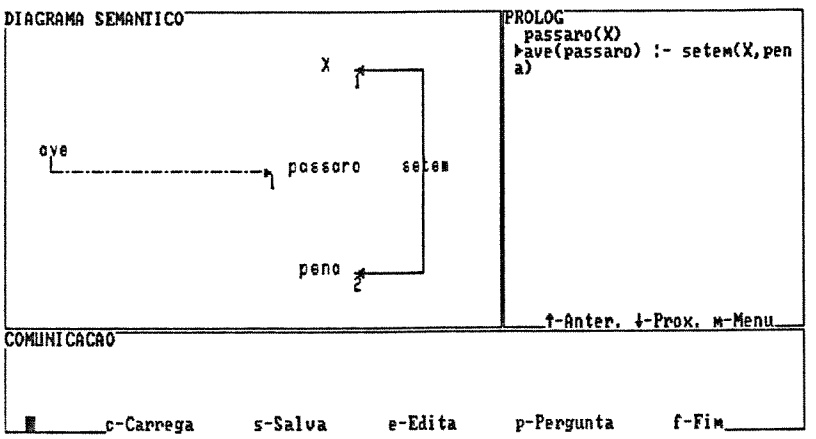


Figura 8.18 Representação da cláusula definida pelo sujeito, indicada na janela Prolog

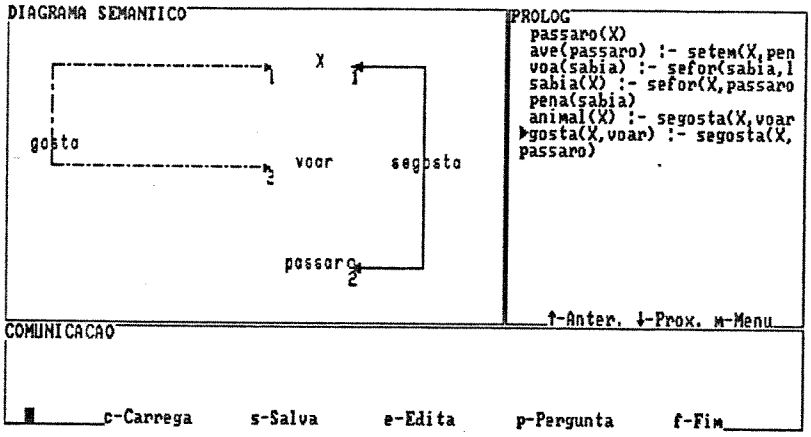


Figura 8.19 Representação da cláusula definida pelo sujeito indicada na janela Prolog

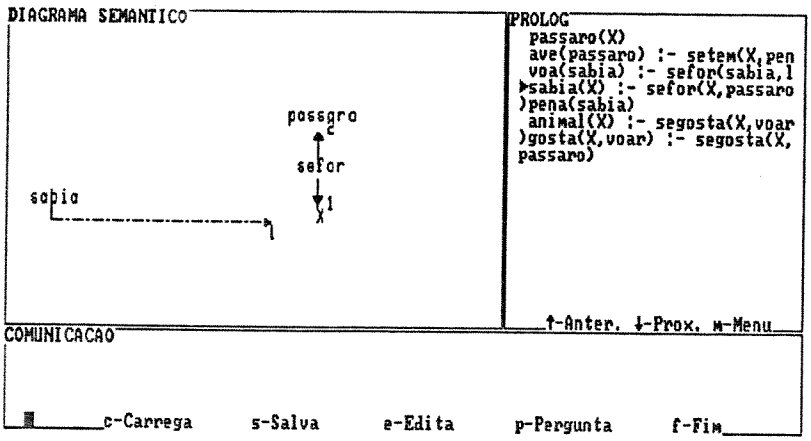


Figura 8.20 Representação da cláusula definida pelo sujeito, indicada na janela Prolog

As figuras acima mostram, refletida nos diagramas, a dificuldade do sujeito em representar seu conceito de "pássaro", no formalismo clausal. A intenção do sujeito ao escrever a cláusula 7 (fig. 8.19) parece ter sido a cláusula representada na figura 8.21, por exemplo. Para a cláusula 4 (fig. 8.20), a intenção parece ter sido uma cláusula do

tipo representado pela figura 8.22. Depurar essa representação implica em depurar o próprio conceito de pássaro sendo representado.

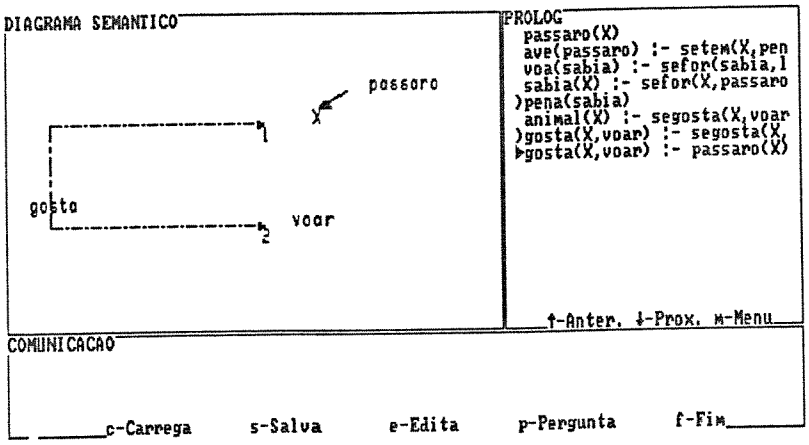


Figura 8.21 Representação alternativa para a cláusula da figura 8.19

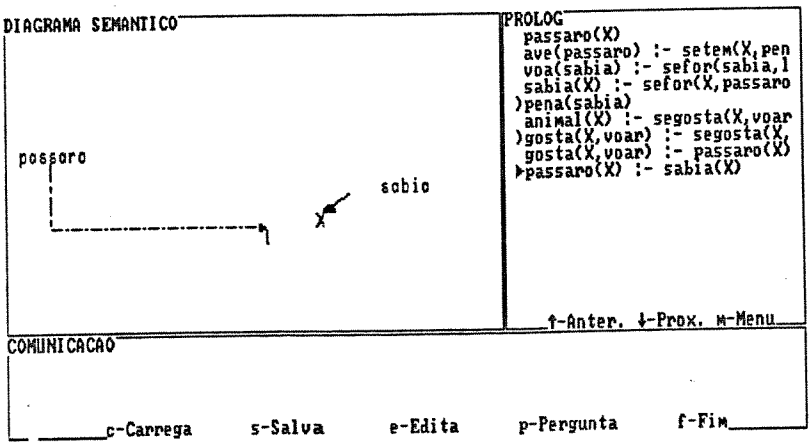


Figura 8.22 Representação alternativa para a cláusula da figura 8.20

Depoimento do sujeito após visualização de sua base de dados através das ferramentas do Módulo Declarativo:

"*Fazendo essa base de dados, quis fortalecer minhas idéias sobre Prolog. Cometi vários erros que me ajudaram a entender melhor o Prolog. Eu colocava, por ex. sabiá como objeto e relação na mesma base de dados e em algumas afirmações não havia uma relação com as outras.*" (Ric)

Esse é um exemplo típico de um estágio no processo de criação de bases de dados onde o depoimento do sujeito mostra o *feedback* que o Módulo Declarativo pôde fornecer. Mais importante do que isso, o sujeito fala sobre sua postura em relação ao erro e seu processo de aprender, envolvendo o ciclo hipótese-criação da base de dados-execução-feedback.

Exemplo 2.

(versão 1)

```
" quadrupede(X):-cachorro(X).          %1
 tempelos(cachorro).
 tempelos(gato).
 quadrupede(X):-tempatas(X,4).
 tempatas(gato,4)."
```

(versão 2)

```
" tempelos(cachorro).          %1
 tempelos(gato).
 quadrupede(X):-tempatas(X,4).
 tempatas(gato,4).
 tempatas(cachorro,4).
 mamifero(X):-quadrupede(X).
 quadrupede(X):-tempelos(X)."          %7
```

Aqui o sujeito mostra, na evolução das versões um para dois, o processo de criação de regras. Na versão 1, por exemplo, o sujeito quer explicitar a dedução de que "cachorro é quadrúpede" (cláusula 1). Na versão 2, o sujeito estabelece relações que possibilitam à máquina deduzir que "cachorro é quadrúpede" (cláusulas 1 e 7).

A figura 8.23, a seguir, mostra a visualização da segunda versão da base de dados do sujeito, através do Módulo Declarativo.

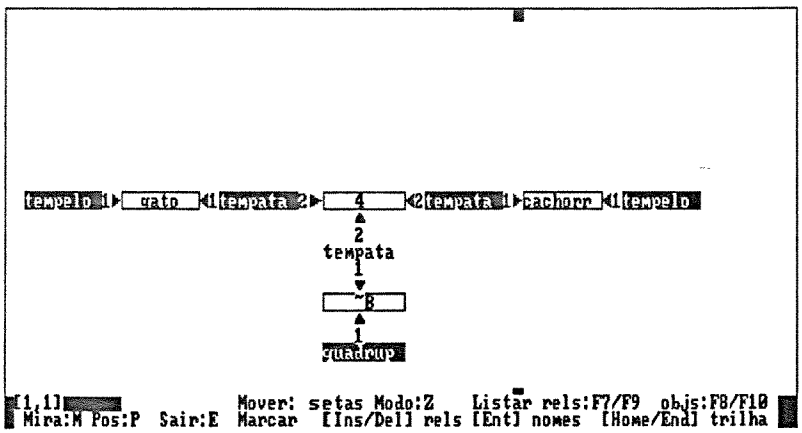


Figura 8.23 Representação da versão 2 da base de dados do sujeito

Depoimento do sujeito ao final da atividade:

"*O computador, no sistema Prolog, trabalha com uma série de informações e regras que constituem a Base de Dados. Ele combina as informações e regras (fatos e regras) para responder à perguntas.*"(Sol)

Exemplo 3.

" *classe_de_animal(X,Y):-carac_da_classe(X,Z),*

carac_do_animal(Y,Z).

carac_da_classe(mamifero,feroz).

carac_da_classe(ave,voa).

carac_da_classe(peixe,nada).

carac_do_animal(onca,feroz).

carac_do_animal(pacu,nada).

carac_do_animal(piranha,feroz).

carac_do_animal(arara,voa)."

? classe_de_animal(X,onca).

X=mamifero.

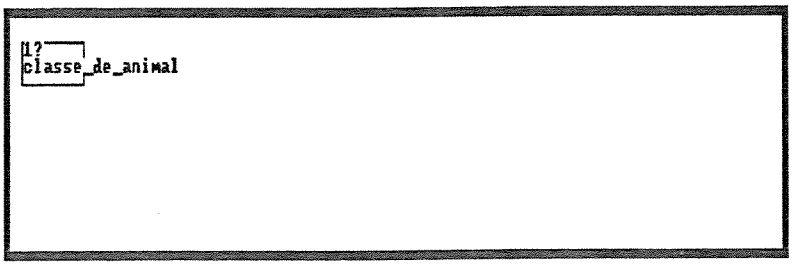
? classe_de_animal(mamifero,X).

X=onca;

X=piranha;" (Rog)

Nesta base de dados o sujeito "resolve o problema", expressando através do formalismo clausal a sentença: "características de determinada classe de animais coincidem com características do animal que pertence àquela classe". Apesar da simplificação do problema, (apenas uma característica comum identifica o animal como sendo da classe em questão) sua definição é simples e flexível o suficiente para identificar tanto os animais de determinada classe, como a que classe pertence determinado animal.

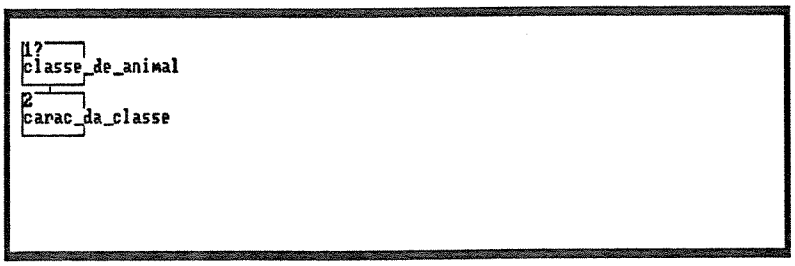
As figuras 8.24, 8.25, 8.26 e 8.27 mostram a interpretação operacional dada à base de dados, através da construção da árvore de espaços de busca para a meta "classe_de_animal(X,Y)".



Regra: 1 --> classe_de_animal(A,B) :- carac_da_classe(A,E) , carac_do_animal(B,E)

Meta: carac_da_classe(A,E) , carac_do_animal(B,E)

Figura 8.24 Representação do processo de execução para a base de dados do sujeito (momento 1)



Regra: 1 --> carac_da_classe(mamifero,feroz) :- true

Meta: true

Figura 8.25 Representação do processo de execução para a base de dados do sujeito (momento 2)

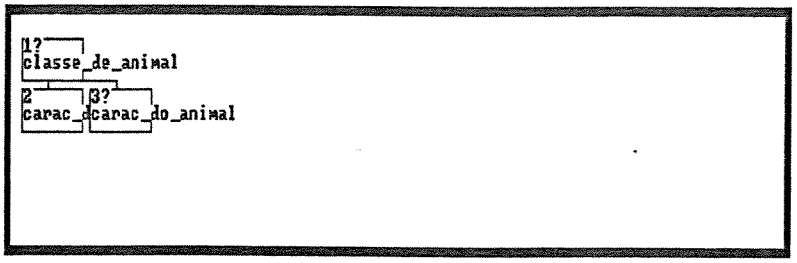


Figura 8.26 Representação do processo de execução para a base de dados do sujeito (momento 3)

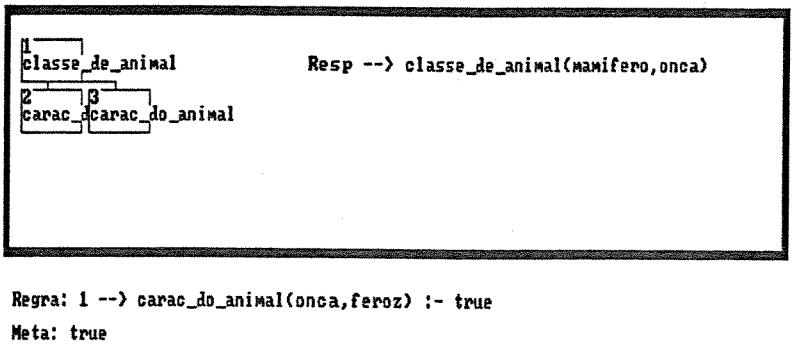


Figura 8.27 Representação da execução para a base de dados do sujeito (momento 4)

8.2.6 A Evolução no Modelo Conceitual da Máquina Virtual

A descrição que o novato faz do processo de inferência usado pela máquina para responder perguntas evolui ao longo das sessões, enquanto o sujeito constrói o modelo de seu interlocutor. Ao fim das quatro sessões esse modelo ainda incompleto, já

mostra os traços dessa evolução principalmente com respeito aos erros conceituais iniciais, conforme ilustrado nos protocolos a seguir.

A consciência da máquina virtual como possuidora de uma "lógica" e "linguagem" próprias.

A imagem mitificada do computador como uma máquina inteligente (que "pensa", à semelhança do Homem), porque é capaz de responder perguntas numa linguagem semelhante, começa a se desfazer. O sujeito passa da posição passiva de espera de respostas à posição ativa de ter que colocar informação na máquina, para que ela seja inteligente. O conhecimento sobre Prolog e sobre a máquina são construídos concomitantemente.

"*Primeiramente temos que fazer nossa b.d. criando informações para esta. Posso dizer que o computador é "burro", pois numa dessas aulas tive um bom exemplo disto: criando minha base de dados coloquei a informação ricardo(nadador) e logo depois perguntei se ricardo nadava**2; *para surpresa tive como resposta não. Então, posso concluir, que após fazermos uma pergunta, esta será avaliada a partir da base de dados. É importante destacar que é tudo "ao pé da letra*"."(Ric)

A consciência da máquina como possuidora de um mecanismo de inferência próprio e independente de sua própria maneira de usar a informação (senso comum).

A interpretação pelo "contexto estendido" das bases de dados começa a ser desfeita, assim como, também, a idéia da máquina como um depósito de informações.

---

*2 gosta_de(ricardo,nadar)

Começa a surgir a idéia do processo utilizado pela máquina para dedução, como um mecanismo próprio e independente.

" *Responde de acordo com as informações dadas (afirmações). Ele faz uma combinação entre as afirmações até chegar numa resposta, dando para ver bem o que ele faz na base de dados [local].*
- se a primeira afirmação não bate ele passa para a próxima e assim por diante.
- o computador só responde com afirmações, quando não possui responde não como no caso do [gosto] => se fosse pela minha dedução, colocaria o fusca sendo um carro, mas como ele não sabe mais do que as informações dadas para ele, o fusca não é um carro tornando a afirmação falsa."(Car).

" *O prolog trabalha estabelecendo uma série de regras, a base de dados, e, baseado nestas regras, ele realiza uma análise combinatória, entre os dados para responder a perguntas. Porém, o computador só se baseia nos dados que possui (na base de dados) e nas ligações que pode estabelecer através desses dados.*" (Sol)

O processo de dedução como um encadeamento de informações

O primeiro progresso observado nas descrições dos sujeitos para o processo de responder perguntas passa da busca por informações presentes explicitamente na base de dados, para um processo de procura direta e indireta. A dedução é vista como um "encadeamento" de informações.

"*Para chegar em uma informação verdadeira, o computador vai verificando e substituindo cada uma delas por outras informações também contidas na base de dados, até chegar na informação correta. Dadas as primeiras informações o computador as veri-*

fica, para ver se elas estão verdadeiras, não conseguindo verificá-la ele passa para outra informação, quando necessário, substituindo por outra informação, até chegar às respostas certas" (Hel).

(a respeito da base de dados [local]) "*O computador faz uma ligação. Por exemplo: alan está na sala 19, essa é a afirmação que ele tem na memória. Davi, pelas informações da memória, não está na sala 19 mas como está ligado, "visita" alan, fica na sala 19, por sua vez alberto se liga a davi por uma outra afirmação (visita) que se liga a alan,e, somente nesse caso, alberto está na sala 19.*" (Fab).

8.2.7 Aspectos Operacionais x Aspectos Declarativos

Há uma discussão na literatura que polariza as metodologias de ensino de Prolog, em duas abordagens aparentemente conflitantes: a abordagem pelo seu aspecto declarativo e a abordagem pelo seu aspecto operacional. Para investigar esse aspecto, o segundo subgrupo de novato-novato teve metade dos sujeitos expostos ao Módulo Declarativo (D) e metade ao Módulo Operacional (O). A seguir comentaremos alguns aspectos observados dessa diferenciação.

A descrição do processo de inferência

Como poder-se-ia esperar, a descrição dos sujeitos expostos ao Módulo Operacional é mais próxima do mecanismo de inferência de Prolog e é baseada no que o sujeito consegue apreender, sobre o processo que ele vê em execução através das ferramentas. Os sujeitos expostos ao Módulo Declarativo, em geral fazem descrições bem próximas do mecanismo que eles próprios usam para dedução e são baseadas na meta-análise da base de dados.

A seguir, mostramos alguns protocolos que exemplificam, na base de dados [ladrao], a descrição do processo que a máquina usa para responder a pergunta *?pode_roubar(joão,Y)*, por um sujeito do grupo D e por um sujeito do grupo O, respectivamente.

" *Analisando a base de dados I:*
- joão rouba Y se gosta de Y.
- joão gosta de Y se Y gosta de vinho.
- maria gosta de vinho.
-> *Y=maria* " (Fab)

"A máquina analisa se joão pode ser X [ladrao(X)] e verifica que sim pois joão é ladrão [ladrão(joão)]. Em seguida a máquina verifica se joão gosta de Y [gosta_de(X,Y)] e observa que joão gosta de Y se Y gosta de vinho [gosta_de(joão,X):-gosta_de(X,vinho)]e, portanto, joão rouba maria porque maria gosta de vinho [gosta_de(maria,vinho)]" (Rod)

Bases de Dados Abstratas

Estranhamente ao que era esperado, a turma D parece ter menos dificuldades com as bases de dados sem contexto do que a turma O. Alguns sujeitos atribuem "significado" (dão uma interpretação) às bases de dados abstratas, para "entendê-la", como mostra o protocolo a seguir:

em [flv2],

"*Define que jane faz e fez algo e sue fez algo e fará algo por helen que fará algo por Fred. Mas um deles só será alguém se um deles fez, faz e fará algo por um qualquer. A resposta para p(X) é não*". (Agn)

Nessa base de dados, o sujeito atribuiu os significados "faz" ao predicado *a*, "fez" ao predicado *b*, "fará" ao predicado *c*, "ser alguém" ao predicado *p*. As variáveis são associadas a "algum", "um deles" e "um qualquer".

Na turma O a falta de "significado" deixa sem "sentido" o processo. Muitos sujeitos não são capazes de fazer previsões sobre bases de dados abstratas, como já foi mostrado anteriormente. Isso sugere que o entendimento do aspecto operacional da base de dados é auxiliado pelo contexto que a base de dados expressa. Essa mesma observação pode ser verificada, também em bases de dados onde o contexto parece interferir de forma a confundir o sujeito, no entendimento do processo, conforme será mostrado a seguir.

A interferência de contexto

A maioria dos sujeitos do grupo D acerta a previsão na base de dados [gosto]. Com o grupo O o resultado foi o oposto: a maioria erra nas previsões. Alguns sujeitos respondem à pergunta *?gosto(X)* com "*vermelho(X),carro(X)*" ou "*vermelho carro*" "*azul bicicleta*".

O Módulo Declarativo parece encorajar a meta-análise, o que explica o sucesso do grupo D. O grupo O ainda não consegue "executar" operacionalmente a base de dados, na ausência da ferramenta. Além do processo de dedução não ser direto (possui várias sub-metas que falham e *backtracking*), a base de dados mistura elementos de domínios diferentes no predicado (*tijolo* e *mercedes*, *fusca* que é nome de carro não aparece com o predicado *carro*, por exemplo). Esses aspectos inserem

complexidade para esse novato que ainda está construindo o modelo de execução da máquina. Daí responderem com o início do processo (aplicação da primeira regra).

O conhecimento operacional começa a ter sentido para o novato a partir de um entendimento da semântica declarativa da base de dados. Por outro lado, o conhecimento operacional é necessário em outros tipos de atividade, como por exemplo na correção de erros decorrentes de considerações operacionais da base de dados, conforme discutido anteriormente, onde a ausência do *feedback* operacional deixa o sujeito sem ação.

8.2.8 Síntese da Análise do Grupo I

Este estudo sobre o novato-novato no ambiente proposto, pretendeu captar as fases iniciais do processo de aquisição da linguagem de programação (no caso Prolog) pelo novato, em seu primeiro contacto com o computador. Os resultados mostram que vários elementos cooperam entre si nesse processo evolutivo: conceitos envolvidos no domínio do problema, o conceito de computador (como interlocutor na representação da solução para o problema) e conceitos da linguagem de programação (como meio de comunicação). No processo cíclico em que o novato se envolve, ele reflete e depura não somente os conceitos de programação envolvidos, mas também seu conhecimento do domínio do problema.

Nesta fase inicial, para o novato-novato, mais importante do que o entendimento da máquina virtual da linguagem é a compreensão do formalismo clausal para expressão do conjunto de proposições que constitui seu programa. Nesse sentido, as ferramentas do Módulo Declarativo responderam melhor às necessidades do novato-novato nessa questão, refletindo, por exemplo, situações de "inconsistência" em sua base de dados. Conforme os resultados apontam, o conhecimento opera-

cional começa a ter sentido para esse novato, a partir de um entendimento da semântica declarativa da base de dados.

8.3 Estudo de Caso de um Sujeito do Grupo Novato-Novato

Dando continuidade à investigação do grupo novato-novato, foi feito um trabalho exploratório, com apenas um sujeito, no cenário-2, usando o método clínico, para observarmos mais especificamente o processo de resolução de problemas no ambiente proposto. Foram trabalhados problemas "clássicos" da literatura de Ciência da Computação (problemas tipo A), problemas envolvendo processamento de listas (problemas tipo B), e problemas "clássicos" em IA, que envolvem representação de conhecimento em Prolog (problemas tipo C).

Os problemas tipo A trabalhados foram: cálculo do fatorial de um número, cálculo de uma certa potência de um número, a sequência de Fibonacci, o cálculo do MDC (máximo divisor comum) entre dois números. Entre os problemas do tipo B trabalhados estão: definição de predicados para retornar o último elemento de uma lista, a soma dos elementos de uma lista, o enésimo elemento de uma lista, verificação da ordenação de uma lista, concatenação de duas listas, inversão de uma lista, produto escalar de dois vetores representados por listas. Os problemas do tipo C trabalhados foram: [caminho] (Folha-Tarefa C, base de dados 4), [tesouro] (Folha-Tarefa A, problema 3) e o problema do FLCA (Fazendeiro, Lobo, Cabra e Alface devem atravessar um rio numa canoa, sob certas condições de segurança).

Mostraremos, nas próximas sessões, os principais "*bugs*" que ocorreram durante o processo de resolução dos problemas dos grupos A e B , o *feedback* das ferramentas nessas situações e uma sessão típica num problema do grupo C.

Inicialmente a visão da linguagem pelo seu aspecto declarativo, impera. A representação da solução para o problema está fortemente influenciada

pela idéia de "escrever proposições verdadeiras a respeito do problema". O conhecimento de como a máquina atua sobre aquela representação vai sendo construído a partir da interação do sujeito no ambiente. É nessa interação que acontece o processo de depuração (da base de dados e do conhecimento do novato sobre Prolog).

O declarativo em excesso

Neste tipo de *bug*, ocorrido anteriormente com outros sujeitos do grupo novato-novato, a base de dados (programa) é interpretada textualmente, como um conjunto de informações, uma em cada linha, que têm um significado quando tomadas em conjunto e em sequência. O "escopo" das variáveis não se restringe a uma cláusula, mas é a base de dados olhada como um todo. No problema de cálculo da potência de determinado número, por exemplo, o novato cria a primeira versão da base de dados, como a seguir:

versão 1:

pot(X,1,X).

pot(X,Y-1,W).

*pot(X,Y,Z):-Z is X*W.*

para representar o processo indutivo:

$x^1 = x$.

$x^{y-1} = w$.

$x^y = z$ onde $z = x * w$.

A seguir, apresentamos a sequência de versões que o novato cria, durante o processo de representação da solução do problema. Até a versão 3, a realimentação do ciclo hipótese-criação-execução-feedback aconteceu a partir da resposta fornecida pelo interpretador Prolog e meta-análise da base de dados, sem o uso das ferramentas. O sujeito encarava como um "desafio" descobrir o erro sem "pedir auxílio" às ferramentas.

(versão 1 não deu certo) "*porquê as cláusulas 2 e 3 estão isoladas*" Rog

versão 2:

pot(X,1,X).

*pot(X,Y,Z):-pot(X,Y-1,W),Z is X*W.*

"*êle entende esse Y-1*"? Rog

versão 3:

pot(X,1,X).

*pot(X,Y,Z):-Y1 is Y-1, pot(X,Y1,W), Z is X*W.*

?pot(3,3,X). (teste da versão 3)

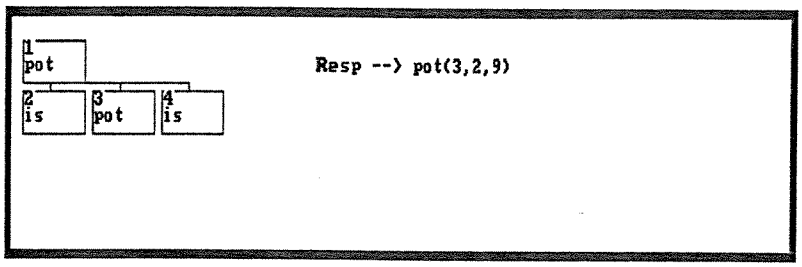
X=9; (sujeito aciona o mecanismo de retrocesso, entrando com ";" após a resposta de Prolog)

"... not enough global stack" (mensagem de erro de Prolog)

"*porquê deu isso?*" Rog

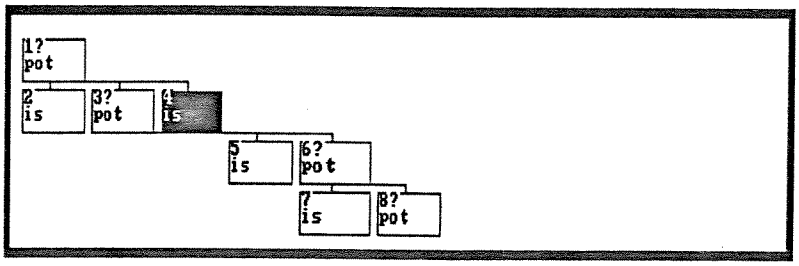
Ao acionar o mecanismo de retrocesso (*backtracking*) para ver todas as respostas possíveis, o novato recebe uma mensagem de erro. A partir de então, apenas com a meta-análise, o novato não consegue detectar a causa do erro. Usa, então o Módulo

Operacional (MOP) e verifica que, após a primeira resposta, está sendo aplicada a regra 2 para o caso de expoente 1, como mostram as figuras 8.28 e 8.29, a seguir, dando continuidade a um processo que deveria parar.




```
Regra: 2 --> pot(3,2,A) :- 1 is 2 - 1 , pot(3,1,3) , A is 3 * 3
Meta: A is 3 * 3
```

Figura 8.28 Primeira resposta à meta *pot(3,2,X)*




```
Regra: 2 --> pot(3,-1,J) :- L is -1 - 1 , pot(3,L,M) , J is 3 * M
Meta: L is -1 - 1 , pot(3,L,M) , J is 3 * M
```

Figura 8.29 Aplicação da regra 2, após a primeira resposta, continuando o processo...

"*então é só colocar Y# 1*" Rog (como condição para aplicação da regra 2)

versão 4:

pot(X,1,X).

*pot(X,Y,Z):-Y>1, Y1 is Y-1, pot(X,Y1,W), Z is X*W*

"*então no fatorial também tem esse problema*" Rog (generalizando para um problema que ele tinha "resolvido" anteriormente)

O novato começa a construir o modelo do processo de inferência, a partir do erro e do *feedback* proporcionado pela ferramenta. Note-se que o sujeito não trata o caso de expoente zero. A investigadora não interfere, assim como não interferiu na sua resolução do fatorial, que ele retomou a partir de então.

Resolução de Problema: criando e "depurando" uma base de dados

A estratégia de representação da solução para o problema, que o sujeito usa é um reflexo de sua interpretação do que significa "criar uma base de dados", conforme comentado anteriormente. A exemplo do que fez no problema descrito anteriormente, o sujeito escreve "pedaços" de cláusulas, no papel, enquanto pensa no problema, depois os "arruma" (coloca no formato clausal). Resolvendo o problema do MDC, ele faz no papel (reprodução de Protocolo 3 de Rog):

Z>0

R>0

R is X mod Y

R1 is Y mod R

versão 1: representação da primeira iteração do algoritmo

mdc(X,Y,Z):- R>0, R is X mod Y, mdc(Y,R,Z).

?mdc(12,8,Z). (teste da versão 1)

no. (resposta de Prolog)

"*acho que é por causa do Z que não está definido*" (sujeito cria hipótese a respeito do erro)

Usando o Módulo Operacional, o sujeito vê o processo de execução através do Trace Estendido, conforme ilustrado nas figuras 8.30, 8.31 e 8.32

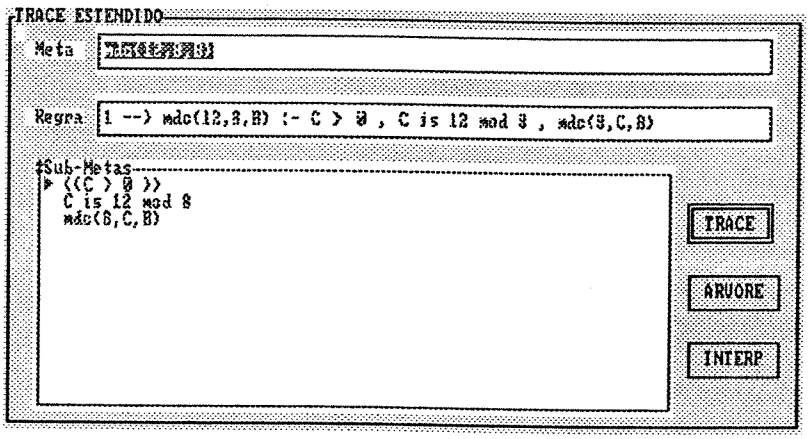


Figura 8.30 Meta inicial e cláusula do programa após unificação

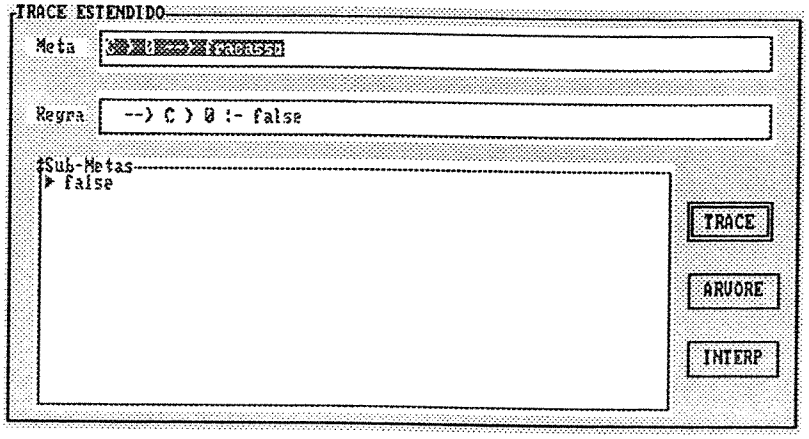


Figura 8.31 Fracasso da primeira sub-meta (C>0)

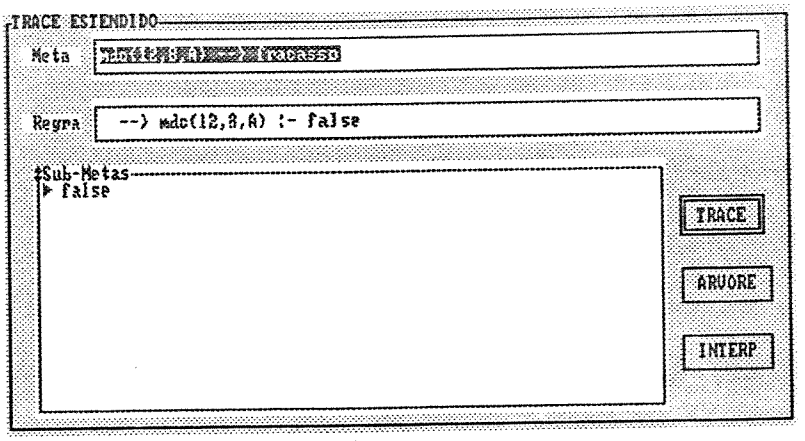


Figura 8.32 Fracasso da meta inicial (não há cláusula alternativa no programa)

"*porquê não tem o R ainda*" Rog (explicando porque deu errado)

O sujeito localiza o erro em termos do processo de inferência, mas ainda exita em modificar o código:

"*não é o segundo argumento que tem que ser >0? porque você vai aplicar o mesmo processo*" Rog

O "você" da fala do sujeito parece referir-se à forma impessoal "aplica-se o mesmo processo". Aqui o sujeito relaciona o conhecimento declarativo a partir da meta-análise que ele faz do problema ao conhecimento operacional (que ele viu no processo de inferência), conseguindo o *feedback* que faltava para a correção.

versão 2:

mdc(X,Y,Z):-Y>0, R is X mod Y, mdc(Y,R,Z).

?mdc(12,8,Z). (teste da versão 2)

no. (resposta de Prolog)

"vai ter uma hora em que Y não é maior que zero, será que tem que dizer isso para o segundo mdc?" Rog

O sujeito já tem uma idéia do aspecto dinâmico da execução e das modificações que vão acontecendo às variáveis durante esse processo. Agora o sujeito usa o Módulo Operacional já com uma hipótese formada, e busca verificação através da ferramenta. Na árvore de espaços de busca, o sujeito vê que foi aplicada a regra 1 para a meta *mdc(4,0,A)* (nó 7), dando continuidade ao processo, quando deveria finalizá-lo, como mostra a figura 8.33.

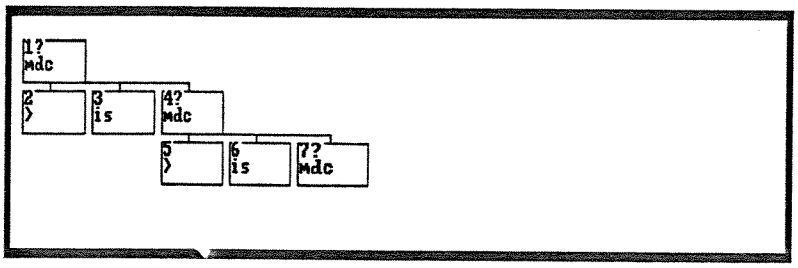


Figura 8.33 Aplicação da regra 1 à meta *mdc(4,0,A)* (nó 7) dando continuidade ao processo

"deveria parar em mdc(4,0,X) e fracassou. não tem resposta para isso. Sabia que com uma cláusula só não ia dar certo". Rog

versão 3:

mdc(X,0,X).

mdc(X,Y,Z):-Y>0, R is X mod Y, mdc(Y,R,Z).

perguntado o que ele achava dessa solução em relação aos problemas anteriores ele diz:

"*achava que tinha que colocar Z is ... por isso demorei a começar. Ele chega na resposta básica os outros usam a resposta básica para (fazer) calcular alguma coisa*"

Os dois últimos depoimentos mostram conceitos computacionalmente sofisticados sendo trabalhados de forma intuitiva: a noção intuitiva da condição de parada para a recursão (uma cláusula apenas não é suficiente para representar a dinâmica do problema) e a percepção da diferença no processo, entre uma recursão de ponta e um processo que envolve a "volta" da recursão (último depoimento). O *feedback* das ferramentas, através de suas representações do conhecimento operacional e declarativo envolvidos na representação da solução, possibilita trabalhar concretamente (visualizar) esses conceitos.

O que o sujeito "entende" e o que a máquina "entende" a respeito da representação da solução do problema.

O processamento de listas é introduzido ao novato, inicialmente, através de problemas já resolvidos. Explorando a notação de listas através da base de dados [soma] (soma dos elementos de uma lista):

?soma([A],C). (teste da solução apresentada)
A=_0038 (resposta de Prolog: variável sem um valor associado)
C= err

"porquê erro? a soma de uma lista unitária não seria ele mesmo?" Rog

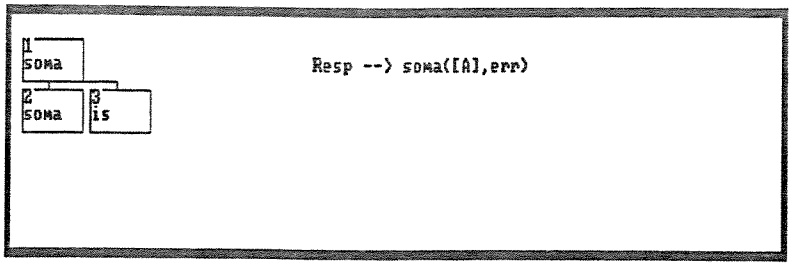
A mesma situação ocorre quando o sujeito entra com a meta *?soma([a,b,c,d],X).* O sujeito esperava como resposta $a+b+c+d$ e não erro. É esperada uma resolução do ponto de vista "lógico" e não quantitativo.

O sujeito, então, simula mentalmente a execução do programa, explicando verbalmente os passos da máquina. Entre eles o momento do processo:

"C is A + 0 que é A"

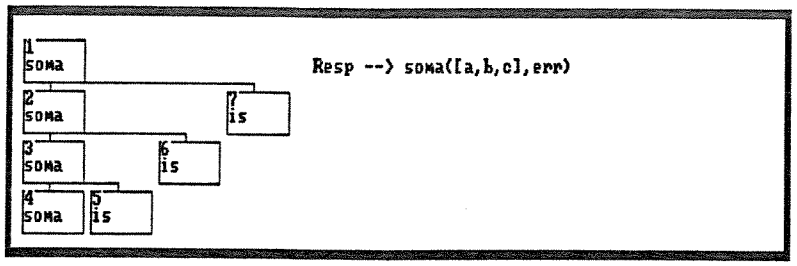
Aqui o sujeito usa seu conhecimento anterior, (de matemática) para interpretar a expressão $A+0$. A variável não é vista como um símbolo que pode estar ou não associado a determinado valor (e só no caso de haver essa associação a soma faz sentido).

Não conseguindo criar uma hipótese da causa do erro suposto, o sujeito, então, usa o Módulo Operacional para ver como as perguntas *?soma([A],C)* e *soma([a,b,c],X)* estão sendo "entendidas" pela máquina. Através do processo ilustrado nas figuras 8.34 e 8.35, o sujeito verifica que a expressão $A+0$ não resulta em A, conforme ele esperava.




```
Regra: 2 --> soma([A],B) :- soma([],0) , B is A + 0
Meta: B is A + 0
```

Figura 8.34 Resposta à meta *soma([A],B)*, mostrando *err* como resultado de *A+0*




```
Regra: 2 --> soma([a,b,c],A) :- soma([b,c],err) , A is a + err
Meta: A is a + err
```

Figura 8.35 Resposta à meta *soma([a,b,c],A)* mostrando os retornos das chamadas recursivas e *err* como resultado de *A is a+err*

Modifica, então, a base de dados em função do significado que o intermediador (a máquina virtual) atribui para a expressão, de forma a obter o resultado desejado:

soma([],0).

soma([X|Y],Z):-Y\==[], soma(Y,Z1), Z is X+Z1.

soma([A],A).

?soma([A],X). (teste)

A=_0038 (resposta de Prolog)

X=_0038

"*isso que eu queria*" Rog

Neste exemplo o sujeito mostra um controle perfeito do processo. Faz com que a segunda cláusula só seja usada para listas com mais que um elemento (inserindo Y\==[]) e cria uma nova cláusula para o caso de apenas um elemento na lista.

O que o sujeito "entende" nem sempre coincide com o que a máquina "entende" a respeito da representação da solução do problema. As ferramentas representam uma janela através da qual o sujeito pode ver como a máquina está "entendendo" a base de dados. Na ausência da perspectiva do interlocutor (como a máquina virtual está "entendendo" determinada informação) o sujeito estaria engajado em um diálogo de surdos com a máquina.

A explicitação do "vale para todos"

Definindo um predicado para verificar se uma lista numérica está ou não ordenada:

versão 1:

```
ord([X|Y]):- N1 is N-1,
             ene([X|Y],N,Z),
             ene([X|Y],N1,Z1),
             Z > Z1.
```

O predicado "*ene(lista,enésimo,resposta)*" já tinha sido definido pelo sujeito para retornar o enésimo elemento de uma lista e é apresentado a seguir:

ene([X|Y],1,X).

ene([X|Y],N,Z):-N1 is N-1, ene(Y,N1,Z).

?ord([3,2,1]). (teste da versão 1)

no. (resposta de Prolog)

?ord([1,2,3]).

no.

Na tentativa de descobrir o erro, o sujeito "explica" verbalmente sua solução:

" *compara o último com o penúltimo, depois o mesmo para o penúltimo e ante-penúltimo, e assim por diante. Tenho que ver quando para.*" Rog

Enquanto explica descobre que *N* tem que ser um argumento do predicado (para que a máquina possa calcular N-1) e que deve haver uma "condição de parada" para o processo. Coloca a sub-meta N>0 na cláusula, com a intenção de "barrar" a continuação do processo quando N chegar a zero (a exemplo do que já experimentara em outros problemas). O uso do predicado que retorna o "enésimo" elemento de uma lista parece dar conta da repetição que aparece em sua fala e não está refletida na definição do predicado *ord*.

versão 2:

ord([X|Y],N):-N>0,N1 is N-1,ene(...

?ord([1,2],2). (teste da versão 2)

yes (resposta de Prolog)

?ord([1,2,3,4],4).

yes

?ord([5,2,3,3,3],4).

no

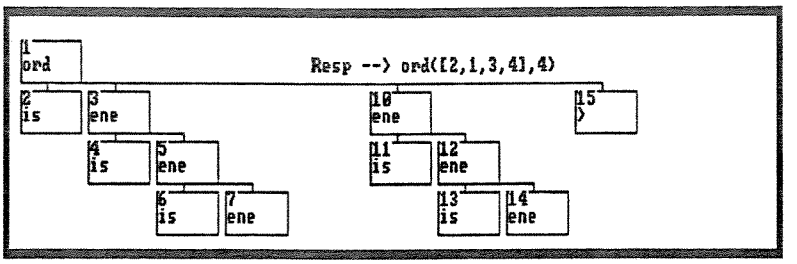
A investigadora sugere a pergunta *?ord([2,1,3,4],4)*. Para surpresa do sujeito a resposta de Prolog é *yes*.

O sujeito novamente explica verbalmente sua solução:

"*uma lista qualquer está ordenada se o enésimo é maior que o enésimo -1*" Rog

Aqui o sujeito mostra novamente o *bug* do "excesso do declarativo" e o uso da representação matemática aparece misturada à solução: o fato de ter colocado "N", seria genérico o suficiente para incluir "todos" (todo o processo).

Usando o Módulo Operacional o sujeito verifica que, operacionalmente, o "vale para todos" precisa ser explicitado. O processo aconteceu apenas entre o último e penúltimo elementos da lista. A figura 8.36 mostra, a hierarquia de metas e sub-metas utilizada pela máquina para chegar na resposta. O predicado *ene* é apenas sub-meta do predicado *ord* e, como tal, não representa a repetição do processo necessária em *ord*.



Regra: 1 --> ord([2,1,3,4],4) :- 3 is 4 - 1 , ene([2,1,3,4],4,4) , ene([2,1,3,4],3,3) , 4 > 3

Meta: 4 > 3

Figura 8.36 Árvore de busca mostrando que a comparação (nó 15) aconteceu apenas entre o último elemento (determinado pela sub-árvore cuja raíz é o nó 3) e o penúltimo elemento (determinado pela sub-árvore cuja raíz é o nó 10)

versão 3:

ord([X|Y],1).

ord([X|Y],N1).

ord([X|Y],N):-N>0, N1 is N-1,

ene([X|Y],N,Z),

ene([X|Y],N1,Z1),

Z>Z1.

Devemos notar que a estratégia usada pelo sujeito para criar a base de dados é a mesma já comentada anteriormente, (problemas da potência, do fatorial) em que ele dispõe em sequência partes da solução. A segunda cláusula (*ord([X|Y],N1*) deveria ser parte da terceira. O sujeito repensa onde colocar esse pedaço de informação (antes ou depois de ene).

versão 4:

ord([X|Y],1).

```
ord([X|Y],N):-N>0, N1 is N-1,
                ene([X|Y],N,Z),
                ene([X|Y],N1,Z1),
                Z>Z1,
                ord([X|Y],N1).
```

A estratégia usada pelo sujeito, em geral, é reaproveitar predicados que ele já definiu antes, embora, como nesse último caso, ele "reconheça" que deve ter uma solução mais simples (quis saber de que outra maneira mais simples o predicado poderia ser definido).

O uso dos aspectos declarativos e operacionais são dosados e combinados na representação que o sujeito faz da solução do problema. A perspectiva declarativa para a versão inicial da solução parece ser dominante. Os aspectos operacionais são trabalhados, posteriormente, nas situações de erro. O uso das ferramentas viabiliza essa combinação.

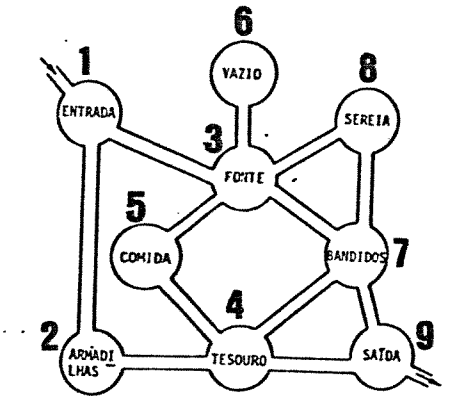
A Representação de conhecimento em Prolog

Existem problemas bem definidos para os quais a solução envolve um processo conhecido *a priori*. As bases de dados para tais problemas representam, na verdade, os processos envolvidos nas soluções de tais problemas (por exemplo, cálculo do fatorial de um número, obtenção do enésimo elemento de uma lista, etc.).

Por outro lado, existe uma classe de problemas que apresentam um salto qualitativo na dificuldade de descrição de sua solução. Em tais problemas é mais visível o ciclo hipótese-edição da base de dados-execução-feedback em que o novato se engaja, enquanto "resolve" o problema. Nesse ciclo, a reflexão é o motor das transformações que levam o sujeito a reformular seu conjunto de hipóteses à

nível do problema que está sendo representado e a nível do conhecimento que ele está construindo do interlocutor (a linguagem). Por outro lado, nem sempre a resposta de Prolog à execução, gera *feedback* suficiente para levar o sujeito a refletir e realimentar o ciclo. Daí o papel das ferramentas em enriquecer o *feedback* do ambiente. A seguir, reconstituímos os principais passos que o novato usou ao resolver o "problema do tesouro" (problema 3 da Folha-Tarefa A, em anexo).

Na exploração inicial, do problema o sujeito faz uma representação do mapa onde cada estado é representado numericamente (como indicado na protocolo reproduzido a seguir). Essa representação é traduzida no conjunto de fatos que estabelece caminhos diretos (com certa orientação) de um nó a um nó adjacente, como descrito a seguir:



Protocolo reproduzido

caminho(1,2).

caminho(1,3).

caminho(2,4).

caminho(3,5).

caminho(3,6).

caminho(3,8).

caminho(4,5).

caminho(4,7).

caminho(4,9).

caminho(7,8).

caminho(7,9).

Ainda na exploração inicial, o sujeito já pensa na representação da solução para o problema em termos de seu interlocutor, quando estabelece o uso de uma "estrutura de dados": o sujeito resolve "usar uma lista para guardar os nós intermediários".

versão 1:

caminho(X,Y,[X,Y]):-caminho(X,Y).

"*o caminho direto entre dois pontos é uma lista só com esses dois pontos, se tiver esse caminho direto*" Rog (sujeito explica sobre a primeira cláusula)

caminho(X,Y,[X|T]):-caminho(X,Z), caminho(Z,Y,T).

"*o caminho entre 2 pontos, o primeiro ponto é X e posso dizer que o último é Y*"
"*tem-se dois pontos. Coloca-se o primeiro na lista, X, se eles não tem caminho direto, tem pelo menos um entre eles, é Z, daí ele vai achar um caminho entre Z e Y, sendo aí uma recorrência*",
"*tem uma hora que vai existir um caminho direto entre um Z intermediário e Y e aí a recorrência cai no primeiro caso*" Rog
(sujeito descrevendo a segunda cláusula, já sob a perspectiva da máquina virtual).

?caminho(1,7,A).

A=[1,2,4,7];

A=[1,3,7];

no

versão 2: evitando armadilhas e bandidos: cláusula 2 é alterada e as cláusulas *perigo* são criadas

```
caminho(X,Y,[X|T]):-caminho(X,Z), not perigo(Z),
                    caminho(Z,Y,T).
perigo(2).
perigo(7).

?caminho(1,9,A).
no.
```

"*acho que não coloquei o caminho (5,4)*" Rog

versão 3: inserindo mais uma cláusula para representar os sentidos inversos dos caminhos entre dois pontos, em vez de colocá-los como fatos.

```
caminho(X,Y,[X|T]):-caminho(Z,X), not perigo(Z),
                                  caminho(Z,Y,T).

?caminho(1,8,A).
A=[1,3,8];
A=[1,3,5,3,8];
A=[1,3,5,3,5,3,8];
A=[1,3,5,3,5,3,5,3,8];
```

...

versão 4: para eliminar o "*loop*" 3,5,3,5,3,5,...sujeito define o predicado pertence(Objeto,Lista).

"*manda seguir só se Z não estiver na lista*" Rog

```
pertence(X,[X|Y]).
pertence(X,[Y|Z]):-pertence(X,Z).
```

Neste ponto aparece o aspecto da dinâmica da representação da solução para o problema. Para dizer que Z não pertence à lista "no momento imediatamente anterior" devem ser considerados dois instantes representados por duas listas (para o momento imediatamente anterior e para o momento atual), refletidos em mais um parâmetro na definição do predicado:

```
caminho(X,Y,[],[X,Y]):-caminho(X,Y).
caminho(X,Y,[X|T],L):-caminho(X,Z),not perigo(Z),
                       not pertence(Z,[X|T]),
                       caminho(Z,Y,[X|T],L).

?caminho(1,5,[],A).
no
?caminho(1,2,[],A).
A=[1,2];
no
```

A partir deste momento, as respostas da máquina virtual às suas perguntas não são suficientes para sugerir ao sujeito novas modificações em sua base de dados. O uso das ferramentas começa a ser necessário, como uma janela para os aspectos dinâmicos da representação da solução para o problema. O sujeito constata, através do MOP, que a regra 2 de sua base de dados nem é tentada, no processo de inferência, como mostram as figuras 8.37, 8.38 e 8.39, a seguir.

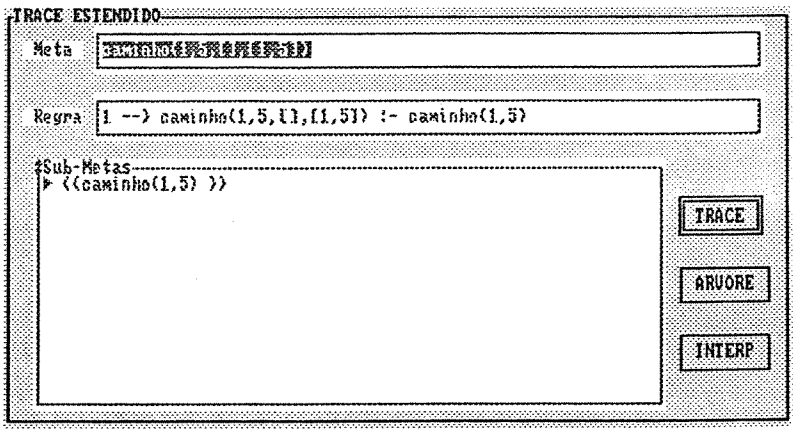


Figura 8.37 Meta *caminho(1,5,[],A)* (sob o unificador) após unificação com a cláusula *caminho(X,Y,[],[X,Y])* (regra 1)

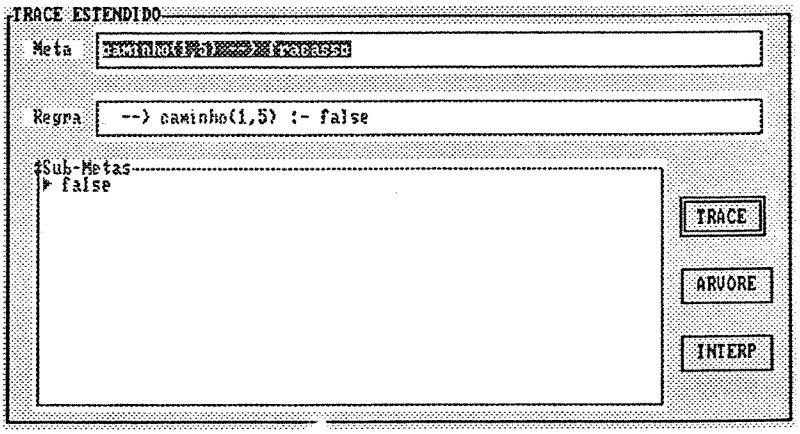


Figura 8.38 Resultado da sub-meta caminho(1,5)

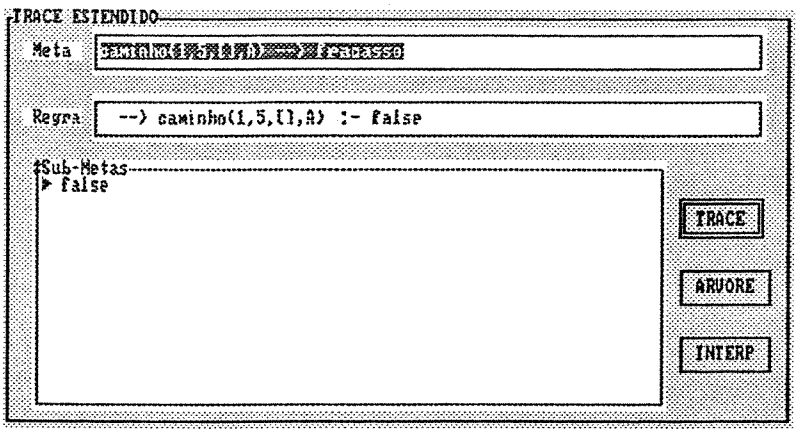


Figura 8.39 Resultado da meta inicial (nenhuma outra cláusula do programa unifica com a meta)

"*terceiro parâmetro tem que ser genérico para encaixar a lista vazia na segunda regra*" Rog

Nessa observação ele se refere ao processo de unificação da máquina de inferência de Prolog. O terceiro parâmetro da segunda cláusula tem que ser escrito de maneira a unificar com a lista vazia, para que a regra seja aplicada.

versão 5: cláusulas 2 e 3 ficam modificadas para:

```
caminho(X,Y,[],[X|Y]):-caminho(X,Y).
caminho(X,Y,T,T1):-caminho(X,Z), not perigo(Z),
                                 not    pertence(Z,T),
                                 caminho(Z,Y,[Z|T],T1).
caminho(X,Y,T,T1):-caminho(Z,X),not perigo(Z),
                                 not pertence(Z,T),
                                 caminho(Z,Y,[Z|T],T1).
```

Sujeito vê processo de execução através do MOP, com a meta caminho(1,5,[],A). Durante essa investigação o sujeito verifica que a segunda regra foi aplicada (figura 8.40).

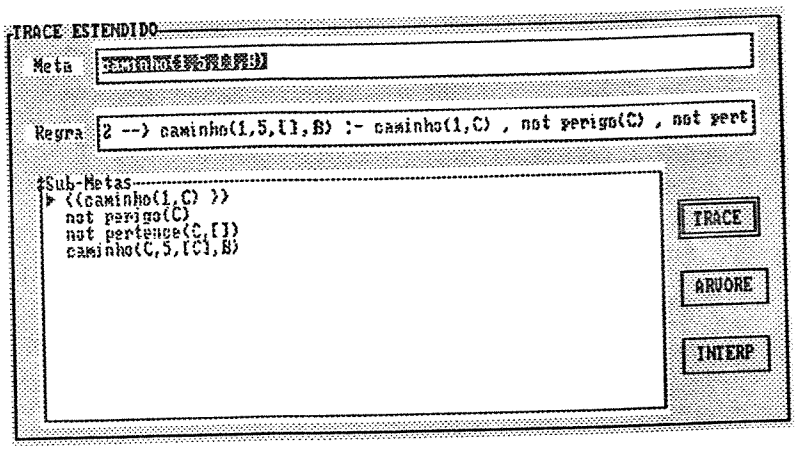


Figura 8.40 Na nova versão da base de dados a meta inicial unifica com a cláusula 2

O sujeito nota que "*ele*" (máquina virtual) não colocou o 1 (estado inicial) na lista; e que o processo "*não está parando*". Quando chega na meta *caminho(5,5,[5,3],F)*, a máquina aplica a regra 2 novamente, em vez de parar, conforme ilustra a figura 8.41.

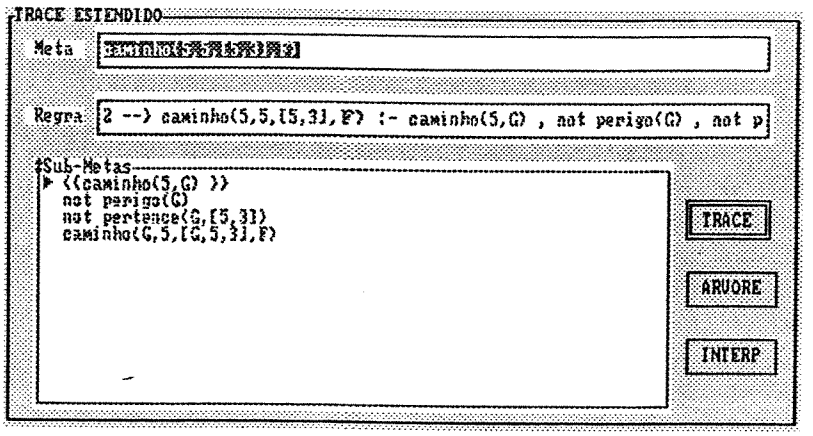


Figura 8.41 Quando o processo chega à meta *caminho(5,5,[5,3],F)*, em vez de terminar, a regra 2 é aplicada novamente

A figura 8.42 sugere o *loop* através da repetição das mesmas sub-metas na estrutura da árvore de execução.

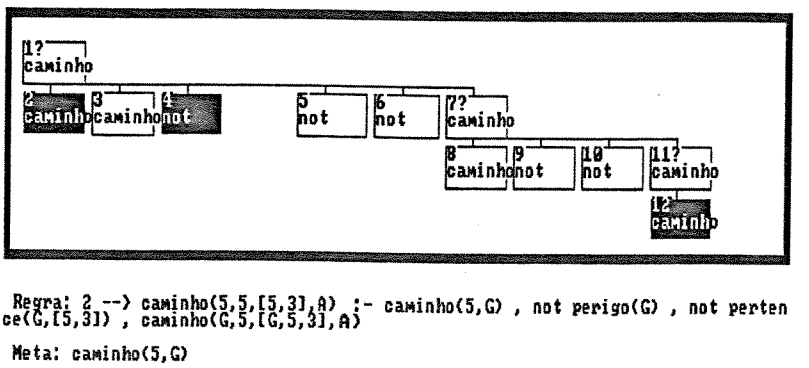


Figura 8.42 Estrutura indicando a repetição das metas representadas pelos nós 1, 3, 5, 6, nos nós 7, 8, 9, 10 e novamente em 11, ...

As modificações sucessivas na base de dados são, a partir de então, verificadas através das ferramentas do Módulo Operacional, como por exemplo a necessidade do "caminho do nó ao nó", que não é intuitivo, sob o aspecto declarativo, mas cuja necessidade fica evidenciada através da visualização do processo de inferência.

Na última versão da base de dados o sujeito coloca a nova cláusula "*caminho(X,X,R,R)*". Elimina a primeira cláusula antiga e justifica, "*porque a segunda cláusula inclui a primeira*", e inclui uma cláusula num nível hierárquico superior, só para facilitar a entrada dos argumentos na pergunta.

última versão:

```
cam(X,Y,R):-caminho(X,Y,[],R1),insere(Y,R1,R).

caminho(X,X,R,R).
caminho(X,Y,T,T1):-caminho(X,Z), not perigo(Z),
                   not pertence(Z,T),
                   caminho(Z,Y,[X|T],T1).
caminho(X,Y,T,T1):-caminho(Z,X), not perigo(Z),
                   not pertence(Z,T),
                   caminho(Z,Y,[X|T],T1).

caminho(1,2).
caminho(1,3).
caminho(2,4).
caminho(3,5).
caminho(3,6).
caminho(3,8).
caminho(4,5).
caminho(4,7).
caminho(4,9).
caminho(7,8).
```

caminho(7,9).

caminho(3,7).

perigo(2).

perigo(7).

pertence(X,[X|Y]).

pertence(X,[Y|Z]):-pertence(X,Z).

insere(X,[Y|T],[X,Y|T]).

8.3.1 Síntese do Estudo de Caso

Na visão que o sujeito tem da linguagem, inicialmente, o aspecto declarativo é dominante. A representação da solução para o problema é influenciada pela idéia de "escrever proposições verdadeiras a respeito do problema". A partir da primeira versão que o novato constrói do programa, ele tenta usar a resposta de Prolog a suas perguntas, para "confirmar" a correção de sua solução. Quando a resposta de Prolog não confirma suas expectativas, a procura de inconsistências através da leitura de sua base de dados se apresenta como um "desafio", para o sujeito. Num primeiro momento, as ferramentas do ambiente são acionadas quando sua leitura não é suficiente para sugerir inconsistências em seu programa e ele não sabe o que fazer em seguida. A escolha da ferramenta é feita por ele ou sugerida pelo facilitador, de acordo com a situação do erro em questão. Num segundo momento, o sujeito usa as ferramentas já com uma hipótese formada sobre a situação e busca verificar essa hipótese.

É a partir da interação do sujeito com as ferramentas do Módulo Operacional, que seu conhecimento de como a máquina atua sobre sua base de dados, vai sendo construído. O *feedback* operacional é importante para o novato-novato depois que ele já consegue fazer uma meta-análise do programa, pois apresenta o programa sob a perspectiva da máquina virtual. É durante essa interação que ele começa a relacionar o conhecimento declarativo (a partir da meta-análise que ele faz do problema) ao conhecimento operacional (que ele vê no processo de inferência).

A escolha das ferramentas do Módulo Operacional é mais frequente nesta fase do trabalho do novato-novato, provavelmente em função de suas necessidades correntes estarem mais relacionadas à necessidade de conhecer como a máquina de inferência atua sobre sua base de dados. Nos problemas trabalhados ele já consegue fazer uma meta-análise de sua base de dados, abstraindo o uso do Módulo Declarativo.

8.4 Resultados da Análise do Grupo II (novato-Prolog)

À semelhança dos sujeitos do grupo novato-novato, os novatos em Prolog também criam um modelo conceitual inicial para entender a linguagem enquanto máquina virtual. A diferença é que esse grupo de sujeitos tem uma experiência anterior de interação com o computador através de linguagens de programação que os deixa mais perto de um modelo formal de linguagem.

Esses modelos iniciais dos sujeitos formam seu conjunto inicial de hipóteses sobre a linguagem (enquanto máquina virtual). Esse conhecimento do intermediador no processo de resolver problemas numa dada linguagem vai sendo modificado durante as atividades que o sujeito realiza no ambiente acrescido das ferramentas. Traços desse processo de construção do sujeito é que tentamos ilustrar nesta análise.

Os resultados mostram que a leitura inicial que o sujeito faz do programa é função do modelo conceitual que ele tem da linguagem em questão. Conceitos já conhecidos dos sujeitos (recursão, por exemplo) não são identificados de início e são "reconstruídos" no novo meio. O processo de resolução de problemas no contexto da nova linguagem é fortemente influenciado por sua experiência anterior no paradigma procedural.

8.4.1 Modelos Conceituais Iniciais da Máquina de Inferência

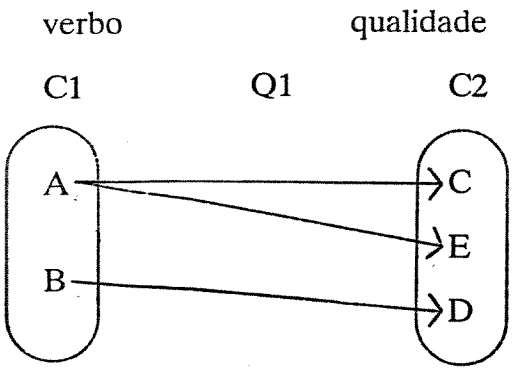
Dos três sujeitos mencionados neste estudo, *And*, cria um modelo inicial da máquina virtual Prolog baseado no conceito de função/relação. Apesar de sua experiência anterior com linguagens procedurais, o sujeito busca teorias matemáticas para criar um modelo dentro do qual possa entender a máquina.

Os outros dois sujeitos criam modelos computacionais para explicar a máquina virtual. *Ren* explica funcionalmente a máquina com elementos que lem-

bram a própria arquitetura do computador (memória e processador). *Ric* cria um modelo baseado na idéia de "casamento de padrões", que se aproxima do modelo Prolog, embora use elementos de sua experiência anterior com linguagens procedurais (campos, variável, compilador, booleanas), para expressar seu conceito.

A seguir, descrevemos em A, B1, B2, protocolos e depoimentos que ilustram os modelos conceituais iniciais dos sujeitos *And*, *Ren* e *Ric* respectivamente.

A - O modelo matemático (relacional)



Q1(C1,C2)
Ex. Qualidade(A,B)

(reproduzido do Protocolo1 de *And*)

"*Para cada verbo, o Prolog separa 2 conjuntos com elementos (ou outros conjuntos) e relaciona os elementos por "vetores" ou "arestas" se for reflexivo.*" And

Na base de dados [ladrao], ao observar as respostas para a meta

?gosta(maria,X).

X=doce;

X=vinho

o sujeito pergunta: "*leva o quê em quê?*" And

Na base de dados [familia], o sujeito usa o modelo funcional para olhar para a cláusula, como sugere a sua fala: *"Quem são os pais de pedro não precisa de mais que um argumento "*

Modifica, entao, a regra *pais(X,Y):-...* para *pais(X):-....*

?pais(pedro). (teste)

yes

"Disse que tem pais, mas não disse quem são os pais" And

O *feedback* da resposta de Prolog leva o sujeito a refletir e a refazer suas hipóte aproximando-o do modelo Prolog.

Na atividade de antecipar respostas a perguntas sobre a base de dados [local], o sujeito prevê como respostas da máquina para as metas:

?local(alberto,X).

no

?local(X,escritorio).

bete

?local(X,Y).

alan sala19

jane sala54

bete escritorio

Os predicados são vistos como relações que "associam" seus argumentos entre si. Assim, o sujeito responde com os elementos da base de dados que são pessoas que estão explicitamente associados a um lugar. A "leitura" que o sujeito faz da base de dados é função de seu modelo conceitual inicial da máquina virtual. A definição recursiva, na cláusula apresentada, não é percebida pelo sujeito, apesar de recursão

não ser um conceito novo para ele (que já trabalhara com recursão no meio Pascal) O modelo de função que o sujeito usa não dá conta das respostas que seriam obtidas a partir da definição recursiva.

B1 - O Modelo Computacional (procedural)

A máquina virtual é vista por *Ren* como uma grande "tabela" que armazena todas as informações contidas na base de dados (memória). Um programa de busca nessa memória (processador) é o responsável por encontrar as respostas às perguntas.

"*Funcionamento do Prolog:*
- *Faz uma grande Tabela com todas as relações possíveis;*
- *Procura na Tabela;*

Ex. ladrao{joao}
gosta {vinho ←
(maria) {doce

gosta---------- *quem gosta de vinho?*

(reproduzido de protocolo 1 de Ren)

Esse modelo não é, em princípio, muito diferente do modelo inicial usado por alguns sujeitos do grupo novato-novato. O que o sujeito acrescenta de novo, com base em sua experiência anterior com computadores, é uma descrição mais completa de como ele acredita ser o procedimento de busca da informação na "Tabela".

B2 - O Modelo de Casamento de Padrões

Ric usa a idéia de "casamento de padrões" para explicar a máquina virtual Prolog embora o faça usando elementos de sua experiência anterior com linguagens procedurais (campo, variável, compilador, booleano).

"*Prolog tem como base a comparação de sentenças. Compara a pergunta feita com as sentenças da base de dados. Se um dos campos da pergunta for uma variável o compilador procura algum outro campo que se encaixe na sentença. Se ambos os campos forem variáveis todas as sentenças da base de dados são apresentadas. Quando uma sentença está completa (sem variáveis) o Prolog diz se a sentença é verdadeira ou não. Trabalha, ao meu ver, basicamente com comparações e respostas booleanas.*" (grifo da pesquisadora)
(reproduzido de Protocolo 1 de Ric)

Essa parece ser a idéia que o sujeito usaria para "implementar" o mecanismo de Prolog no meio procedural. Sua descrição sugere que valores booleanos são obtidos como resposta a partir de comparações que a máquina realiza entre sentenças. É através desses modelos conceituais iniciais que o sujeito faz sua "leitura" da base de dados. Seus erros conceituais iniciais podem ser explicados, portanto, em função desses modelos, que vão se modificando a medida em que ele interage no ambiente.

8.4.2 O Conceito de Recursão Visto em outro "Meio" e o Feedback das Ferramentas

O meio constituido pela nova linguagem faz com que conceitos já conhecidos dos sujeitos sejam redescobertos (reconstruidos) nesse novo contexto. O trabalho dos sujeitos em bases de dados envolvendo definições recursivas, por exemplo, mostra o

uso de seus modelos iniciais da máquina virtual, onde a recursão parece não ter sido percebida nas cláusulas da base de dados e casos onde ela é entendida num contexto procedural. Os exemplos a seguir ilustram alguns desses casos.

Exemplo 1.

Na tarefa de antecipar (prever) quais seriam as respostas da máquina para *local(X,Y)*, na base de dados [local], *Ren* responde:

alan sala19

jane sala54

bete escritório

davi sala19

A previsão feita pelo sujeito leva a supor que ele fez uma busca direta na base de dados (possivelmente em função da definição *local(P,L):-em(P,L)*), para responder: *alan sala19, jane sala 54, bete escritório* e uma busca indireta de um nível como se usasse a definição *local(P.L):-visita(P.O),em(O,L),* que leva às respostas: *davi sala19, janete escritorio.* Aparentemente a recursão não é percebida como tal pelo sujeito. O mecanismo recursivo da definição da base de dados original é substituído por uma iteração de um nível em função de seu modelo inicial da máquina virtual (o modelo de execução do sujeito não dá conta de níveis mais profundos).

Exemplo 2.

Na tarefa de depuração da base de dados [familia], as dúvidas iniciais dos sujeitos *And* e *Ren* refletem o uso do meio procedural, onde a informação textual é interpretada como uma sequência onde a "parada" da recursão deveria aparecer antes da chamada recursiva.

"porquê a cláusula recursiva aparece antes da não recursiva?"

"falta relação entre Z e as outras (variáveis) na primeira cláusula de ancestral"

"é preciso organizar as variáveis"

Tentativas de correção da base de dados, feitas pelos sujeitos:

ancestral(X,Y):-pais(X,Y),ancestral(Y,Z).

ancestral(X,Y):-pais(X,Z),pais(Y,W),ancestral(W,Z).

ancestral(X,Y):-pais(X,Z),ancestral(Z,W).

versão 4: invertendo a ordem das cláusulas ancestral e mudando a cláusula recursiva:

ancestral(X,Y):-pais(X,Y).

ancestral(X,Y):-pais(X,Y),ancestral(Y,W).

?ancestral(pedro,X). (teste da versão 4)

X=maria;

X=paulo;

X=maria;

no

versão 5: eliminando a cláusula 1

?ancestral(pedro,X). (teste da versão 5)

no.

Após eliminar a cláusula 1 e observar a resposta "no" de Prolog, o sujeito conclui que essa informação *"não é dispensável"*, pelo efeito que a sua eliminação causou, mas ainda não sabe dizer porquê.

Sugiro o uso de EDS para trabalhar a semântica das cláusulas. O *feedback* gerado pela ferramenta para a segunda cláusula de ancestral é apresentado na figura 8.43.

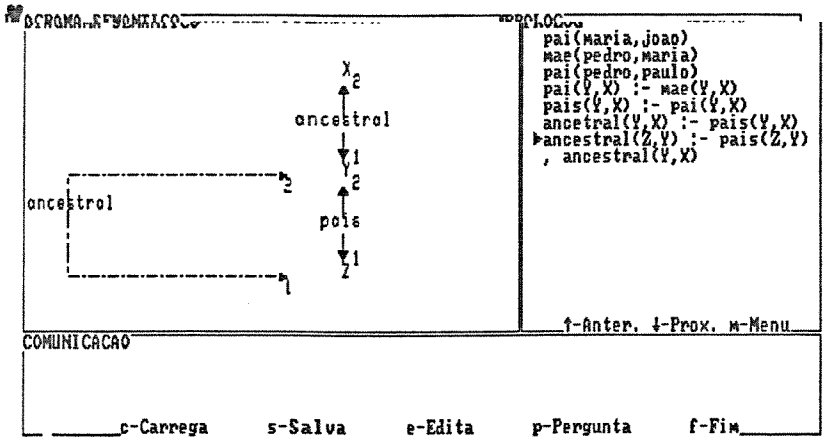


Figura 8.43 Diagrama semântico da cláusula *ancestral(Z,Y):-pais(Z,Y), ancestral(Y,X)*

Os sujeitos percebem que a seta pontilhada (2), da conclusão deveria estar em X e modificam a cláusula 2 para:

ancestral(Z,X):-pais(Z,Y),ancestral(Y,X).

Apesar de resolvido o problema, *And* quer saber "*porquê a cláusula 1 é necessária*". O sujeito não vê a necessidade da cláusula 1, como se a sub-meta pais(X,Y) na cláusula recursiva fosse suficiente para representar a base da recursão. Ele não relaciona a cláusula 1 como sendo a "base" do processo, ou a "condição de parada" da recursão, embora não sendo a recursão um conceito novo para ele.

Esse efeito pode ser explicado, também, como uma consequência do "contexto" da base de dados que, como no caso do grupo novato-novato, parece interferir na leitura que o sujeito faz da base de dados. No contexto do cotidiano, parece óbvio que "seus pais são seus ancestrais". Essa informação só é necessária no

contexto artificial, quando se tem que pensar em termos da operacionalidade de "produzir" os ancestrais. Prolog leva o sujeito a ter que explicitar a relação ancestral na sua forma completa (as duas cláusulas são necessárias para formalizar a definição de *ancestral*). Nesse ponto era necessário que o sujeito visse a leitura que Prolog faz da base de dados, na ausência da cláusula 1. O sujeito usou MOP para observar o processo de execução na ausência da cláusula 1. A figura 8.45 ilustra parte do processo de execução da base de dados mostrada na figura 8.44, onde a cláusula não recursiva de *ancestral* foi transformada em comentário. O sujeito constata que a máquina de inferência não dá resposta após encontrar o "avô de pedro" (joão), continuando o processo.

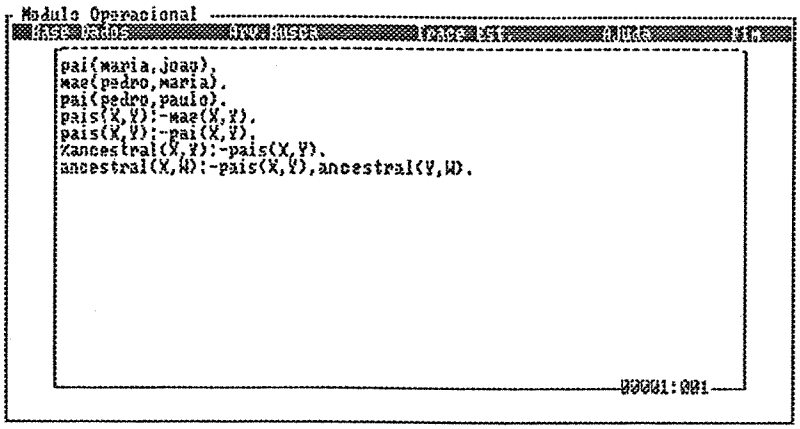


Figura 8.44 Base Dados em questão

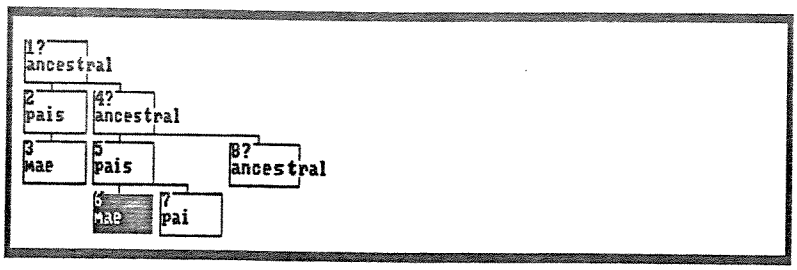


Figura 8.45 Representação da árvore de busca para *ancestral(pedro,A)* (nó 1). O nó 4 representa a meta *ancestral(maria,A)* e o nó 8 representa a meta *ancestral(joão,A)*

Exemplo 3.

Outro exemplo da interferência do contexto, mesmo para sujeitos que já conhecem recursão apareceu no trabalho de *Ric*, na base de dados [grafo], da Folha-Tarefa C, reapresentada a seguir:

caminho(No,No).

caminho(No1,No2):-adjacente(No1,X),adjacente(X,No2).

adjacente(a,b).

adjacente(a,c).

adjacente(b,d).

adjacente(c,d).

adjacente(d,e).

adjacente(f,g).

Primeira tentativa proposta pelo sujeito para correção da cláusula 2:

caminho(No1,No3):-caminho(No1,No2),caminho(No2,No3).

Para alguém que já trabalhou com recursão, em outros contextos, não era esperado esse tipo de solução (recursão dupla). Ao ver o processo de construção da árvore de busca para a meta *caminho(a,e)*, verifica que a mesma situação ocorre nos nós 1, 3, 5 e 7, com a aplicação da regra 2, após aplicação da regra 1 nos nós 2, 4 e 6 respectivamente, como mostra a figura 8.46, ou seja a máquina está em "*looping*".

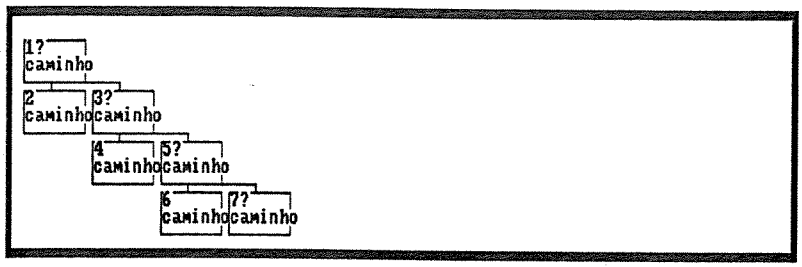


Figura 8.46 Árvore de busca mostrando processo em *looping*

Provavelmente o contexto da lingua natural mescla-se ao formalismo do meio Prolog, "camuflando", num certo sentido, a recursão implícita na definição. Após ver a árvore de execução, mostrada em parte na figura 8.46, o sujeito acrescenta uma terceira cláusula à base de dados:

caminho(N01,No2):-adjacente(No1,X),adjacente(X,No2).

?caminho(a,e) (teste)

"*loop de novo*" Ric

Ao examinar novamente o processo de execução através de MOP, o sujeito constata que a inclusão da cláusula 3 não alterou o quadro de "*looping*" em que a máquina se encontrava. A regra 3 nem chegou a ser usada no mecanismo de inferência.

O sujeito verifica, ainda, que o uso da regra 1 pela máquina (caminho(No,No), representado na árvore pelos nós 2, 4 e 6, é que conduz à repetição do processo. Sua proposta de solução é a eliminação da cláusula:
"*e se tirar o primeiro caminho?*" Ric

Como no caso do exemplo anterior, a cláusula "base" da recursão não é entendida como tal. A idéia de "parada da recursão", tão conhecida no meio procedural parece "sem sentido" no meio declarativo de Prolog. Observa novamente a construção da árvore de execução, sem a primeira cláusula. A figura 8.47 mostra parte do processo, onde o "*looping*" continua, agora na tentativa de satisfazer a primeira sub-meta da regra.

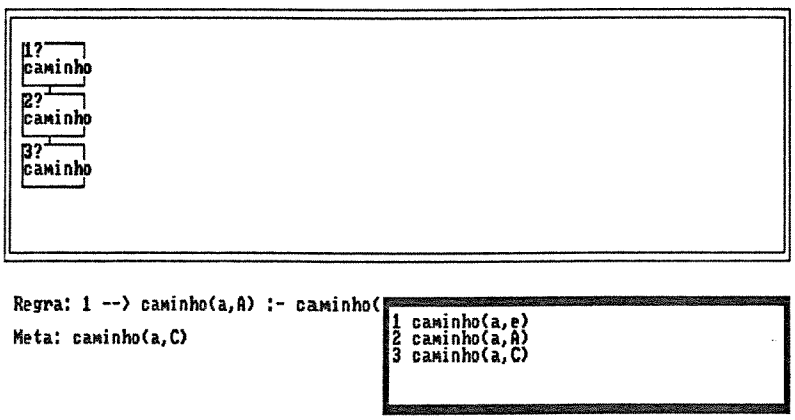


Figura 8.47 Árvore de busca para *caminho(a,e)* na base de dados sem a cláusula *caminho(No,No)*

Em sua quarta tentativa o sujeito substitui a primeira sub-meta da segunda cláusula por "adjacente(No1,No2)", eliminando parte do "*looping*" e volta a incluir a cláusula

1 (caminho(No,No)), que passa a ser o elemento de "parada" do processo recursivo como um todo.

Assim como no caso anterior, para testar a necessidade da cláusula 1 os sujeitos a eliminam e observam a resposta de Prolog. Entretanto, esse *feedback* não é suficiente para compreender a sua necessidade. Os sujeitos buscam essa compreensão observando o aspecto operacional envolvido no problema, através das ferramentas do Módulo Operacional.

Exemplo 4.

No problema do cálculo de um número elevado a uma certa potência (problema 1 da Folha-Tarefa A), *Ric* usa um meio procedural e iterativo para representar o problema, sem pensar em sua definição formal. Quando sugiro que ele pense em "prova por indução" para definir as equações de recorrência, ele não consegue expressar a solução para o problema nesse "meio" e confessa "*apesar de ter feito uma disciplina no semestre anterior onde o tempo todo tinha que fazer isso*[*3]".

versão 1:

pot(X,1,X).

pot(X,0,1).

*pot(X,Y,Z):-Y1 is Y-1, pot(X,Y,Z), Z is X*X.*

Em sua primeira versão, a idéia presente parece ser de iterações onde a cada iteração, *X* (o elemento a ser elevado a uma certa potência) é multiplicado por si próprio (*Z is X*X*).

versão 2: o sujeito muda terceira cláusula para:

[*3] provas por indução

*pot(X,Y,Z):-Y1 is Y-1, pot(X,Y1,Z1), Z1 is Z*X.*

Em sua segunda versão, aparece a idéia da multiplicação de *X* por um certo *Z* que seria a variável que acumula as multiplicações a cada iteração (*Z1 is Z*X*).

versão 3: *pot(X,Y,Z):-Y1 is Y-1, pot(X,Y1,Z), Z1 is Z*X.*
versão 4: *pot(X,Y,Z):-Y1 is Y-1, Z is Z1*X, pot(X,Y1,Z1).*

As versões 3 e 4 são tentativas de colocar no formalismo Prolog a expressão da idéia de iteração para resolver o problema, até que o sujeito "esgota" seu conjunto de hipóteses:
"*agora não sei mais o que está acontecendo*" Ric

As figuras 8.48, 8.49 e 8.50 representam o início da sequência de execução para a meta *pot(2,3,B)*.

Através de MOP (trace estendido), o sujeito detecta o erro na segunda sub-meta: *Z is Z1*X*, conforme ilustra a figura 8.50
"*então Z1 não tem valor*" Ric.

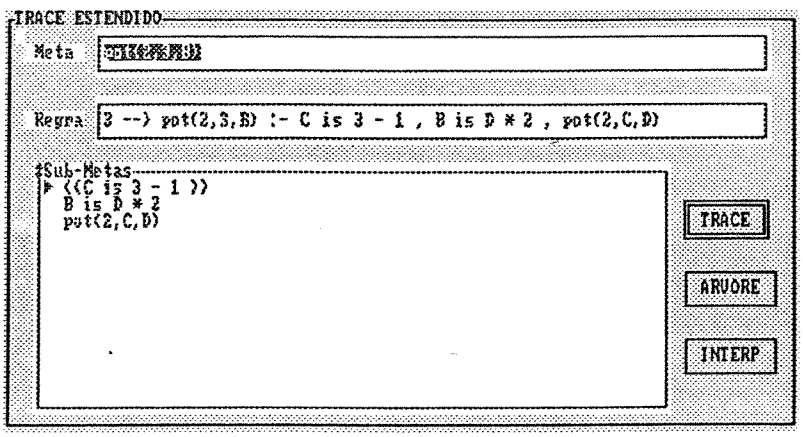


Figura 8.48 Representação da meta inicial, aplicação da terceira cláusula

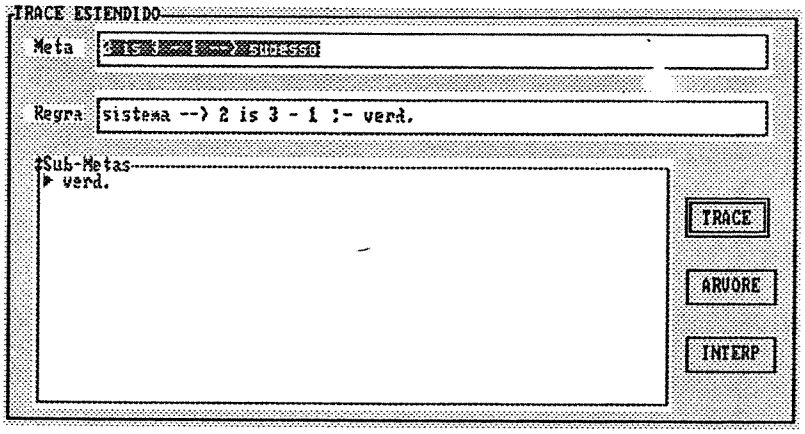


Figura 8.49 Resultado da execução da primeira sub-meta (*C is 3-1*)

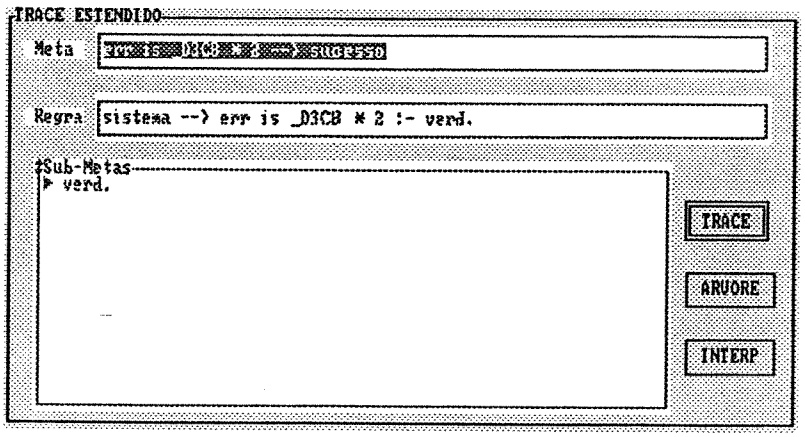


Figura 8.50 Execução da segunda sub-meta (B is D*2)

versão 5:

*pot(X,Y,Z):-Y1 is Y-1, Z1 is Z*X, pot(X,Y1,Z1).*

versão 6:

*pot(X,Y,Z):-Y1 is Y-1, pot(X,Y1,Z1), Z is Z1*X.*

?pot(2,3,X). (teste da versão 6)

X = 8;

...

"loop" `Ric`

Através de MOP, aciona o mecanismo de retrocesso (através do ";") e vê que a máquina de inferência continua o processo aplicando a regra 3 para a sub-meta *pot(2,0,K),* quando deveria ter terminado o processo (tendo como sub-metas L is 0-1, pot(2,L,M), K is M*2), como mostra a figura 8.52.

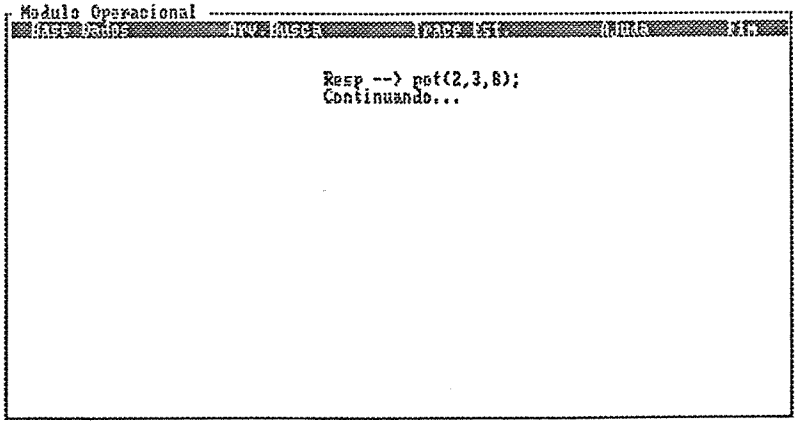


Figura 8.51 Acionando o mecanismo de retrocesso, após resposta

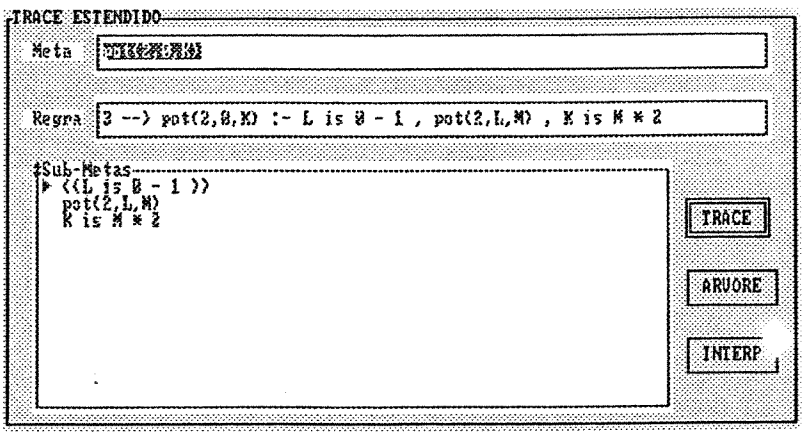


Figura 8.52 Continuação do processo para *pot(2,0,K)*

O sujeito "resolve" o problema dizendo ser necessário "*um teste antes*" , referindo-se a colocação de *Y>0* como primeira sub-meta na regra em questão.

versão 7:

pot(X,0,1).

*pot(X,Y,Z):-Y>0, Y1 is Y-1, pot(X,Y1,Z1), Z is Z1*X.*

Após terminar de resolver o problema, o sujeito consegue definir "matematicamente" a recorrência:

$x^0=1$

$x^y=x^{y-1} * x$

O sujeito usa a sua estratégia "operacional" para conseguir "generalizar" (escrever as equações de recorrência). Em vez de pensar, de início, no problema do ponto de vista conceitual, o sujeito "experimenta", no papel do interpretador, como a potência de um número é calculada. A maneira encontrada pelo sujeito para resolver o problema passa primeiro por sua introjeção no mecanismo que a máquina de inferência usa para resolver o problema e a partir daí ele consegue, através de um distanciamento da máquina, pensar de forma analítica na representação formal da solução para o problema.

8.4.3 A Construção do "Meio" Prolog

A construção do "meio" Prolog pelo sujeito envolve construir o conceito das entidades significativas do novo paradigma de programação e descrever a representação da solução para o problema em função do modelo viabilizado por essas novas enti-

dades. O estudo piloto mostra que o processo de construção desse novo meio pelo novato-Prolog acontece a partir de seu trabalho em um "meio" baseado em seu modelo inicial da máquina virtual. Esse modelo, por sua vez, é fortemente influenciado por sua experiência anterior com linguagens (artificial e natural). A seguir mostramos os traços do processo de construção do *meio* Prolog pelo sujeito.

O Uso do Conhecimento Anterior

Assim como Ren e Ric usam o seu conhecimento procedural para pensar no problema e representar sua solução, And mostra a influência de seu modelo inicial (funcional), quando, para o problema da "lista sem o último elemento", ele descreve: "*tirar o último de uma lista é o mesmo que tirar o último do resto da lista*" e escreve:

tira-ultimo([X|Y],[X|Y1]):-Y1 is tira-ultimo(Y).

Outro exemplo da influência do "funcional" na representação da solução de problemas pelo mesmo sujeito:

No problema do "ordenação por inserção", ele faz:

insort([X|Y],insere(X,Y)):-ordenada(Y).

supostamente para expressar "se Y está ordenada, a resposta é alcançada inserindo X (o primeiro elemento) em Y.", mostrando a semântica funcional num formato declarativo, de forma análoga aos sujeitos do grupo novato-novato que usam a sentença em linguagem natural (seu conhecimento anterior) no formato clausal. Essa mesma idéia é usada por Ren em:

ordena([X|Y],Z):-ordenada(Y), insere(X,Y,Z)

usando o "*ordenada(Y)*" como se fosse uma função booleana, sem expressar a idéia da recursão (são dois predicados diferentes).

A aproximação do "meio" Prolog se dá a partir da construção de uma ponte que o sujeito faz com sua experiência anterior (procedural, funcional, lingua natural, etc.). Daí a importância da multiplicidade de visões no ambiente, onde o ciclo em que o sujeito se envolve é realimentado a partir de sua escolha da ferramenta que melhor se ajusta a sua maneira de "enxergar" a base de dados.

A Influência do Paradigma Procedural na Representação do Problema

Inicialmente o sujeito tenta identificar as entidades significativas do paradigma procedural no novo contexto. Os conceitos de variável e comando de atribuição de valor a variável, são os elementos centrais do paradigma procedural que o sujeito tenta identificar no código da nova linguagem. Por exemplo, analisando a base de dados [mdc], um dos sujeitos busca identificar na nova linguagem o elemento central do paradigma procedural: a atribuição de valor a variável:

"*Como o K recebe um valor?*" Ren

A seguir, apresentamos uma série de casos que ilustram o uso inicial que o sujeito faz do conhecimento procedural na representação da solução de problemas.

Exemplo 1: Definindo um predicado para "retornar o enésimo elemento de uma lista"

Ren começa pensando no problema através de Pascal (diz estar "*pensando em Pascal*") e escreve , no papel, o pedaço de código:

"*for N:=1 to size do*

if ch=mat[N] then halt exit"

A lista é representada pelo vetor (array) *mat*, supostamente um vetor (array) de elementos do tipo "char" (o nome da variável é ch). Aparentemente ele foi buscar um clichê para busca (procura) de um determinado elemento em um array, o que

não é exatamente o que é pedido no problema. O sujeito associa o conceito de lista com o conceito de vetor unidimensional (array) e busca um clichê de "percurso" nessa estrutura (usando array diretamente a solução seria mat[N] simplesmente). Escreve a primeira versão em Prolog, como:

versão 1:

ene([X|Y],1,X).

ene([X|Y],N,T):-N>1,N1 is N-1, ene(Y,N1,T).

? ene([a,b,c,d,e,f,g,h],5,X). (teste da versão 1)

X=e;

? ene([a,b,c,d,e],N,e).

no

"*porque ele não conhece N*" Ren

O sujeito mostra sua busca de generalidade para a solução (entrando com o elemento "e" e querendo como resposta sua posição na lista). Muda, então, o problema para: "*dado o elemento, voltar a posição do elemento*".

versão 2:

elem([X],1,X).

elem([X|Y],N,E):-X\==E, elem(Y,N1,E), N is N1+1.

O sujeito começa resolvendo o problema como se fosse um "outro problema" e depois junta o código de *ene* e *elem*, mudando o nome *elem* para *ene*, tornando genérica a solução para o problema original.

"*É bem lógico: junta com "ou""* Ren

? ene([4,6,2,3,7],4,X). (teste)

X=3;

X=3;

X=3;

X=3

O sujeito, então, tira um dos casos "base" (havia um caso base herdado de *elem*) , para eliminar a repetição da resposta e faz uma reflexão sobre seu processo de resolução de problemas no novo meio:

"Não dava para pensar em Pascal" Ren

Exemplo 2:

Essa ponte entre o procedural e o código Prolog, pode ser vista também, na base de dados para definir a pertinência de um elemento em uma lista, que Ric cria:

existe([X],X).

existe([X|Y],Z):- X==Z.

existe([X|Y],Z):- existe(Y,Z).

A cláusula 1 dá conta de listas com apenas um elemento, onde esse elemento é membro da lista e, portanto, é a resposta.

A cláusula 2 dá conta de listas com mais de um elemento: o primeiro elemento dessa lista é membro da lista, escrito ao estilo procedural: "se Z é igual a X, então ele é membro da lista".

A cláusula 3 dá conta da chamada recursiva que supostamente representa a idéia: "se Z está na cauda da lista, então Z é membro da lista".

Exemplo 3: O mesmo sujeito resolvendo o problema de gerar "a lista sem o último elemento"

"faz falta um repeat until, while..." Ren (rindo)

versão 1:

exclui([X],[]).

exclui([X|Y],[X|Z]):-Y\==[], exclui(Y,Z).

"*como fazer para passar os elementos para a lista Z?*" (atribuição de valor a variável)

"*assim eu vou acabar incluindo todos na lista Z*" Ren

(essa fala sugere que o sujeito vê o caso base dissociado das outras cláusulas da base de dados e não como parte que será usada no processo de execução)

"*quando Y tiver 1 elemento ele cai fora disso*" Ren

(referindo-se ao *loop* gerado pela recursão de ponta da cláusula 2).

De Pascal a Prolog, com o uso das ferramentas

Quando o conhecimento inicial do sujeito o leva a buscar clichês de resolução do problema segundo o paradigma procedural, em geral as ferramentas do Módulo Operacional é que são usadas pelo sujeito, para transportar sua perspectiva do meio procedural para o meio Prolog. Os exemplos a seguir ilustram o processo pelo qual o sujeito passa, de sua perspectiva inicial ao código Prolog.

Exemplo 1: Ric resolvendo o problema de eliminar o último elemento de uma lista.

"*vou chamando até ter só um na lista, aí desprezo ele*" Ric

O "chamando até ter só um na lista" sugere um clichê de chamadas recursivas onde a cada chamada tem-se a lista anterior sem o primeiro elemento. O sujeito propõe uma solução para o problema vista segundo a sua dinâmica: a imagem operacional da busca da solução. O sujeito assume o papel da máquina: "aí desprezo ele", colocando-se no papel do interpretador (introjeção) para pensar na representação da solução do problema.

versão 1:

lista([X],[]).

lista([X|Y],[X]):-lista(Y,Z).

Da imagem do sujeito com sua introjeção na máquina, para o código Prolog, existe uma longa distância. O "caso mais simples", representado pela cláusula 1 é escrito imediatamente pois não envolve o aspecto dinâmico mencionado. É visto como um caso isolado e não como parte da dinâmica (indispensável para que a dinâmica funcione).

? lista([a,b,c,d,e],D). (teste da versão 1)

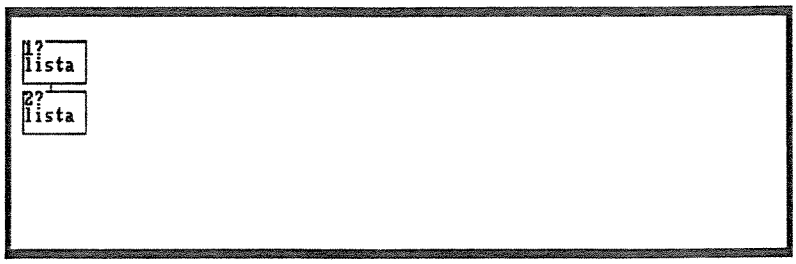
D=[a]

"*porquê ?*" Ric

O sujeito lê o código e diz: "*teoricamente está certo*".

A leitura que o sujeito faz é sempre em função de sua perspectiva inicial. Daí a importância das ferramentas, em levar o sujeito a uma nova perspectiva para "leitura" de sua base de dados. A ferramenta funciona como um óculos que possibilita ao sujeito uma re-leitura de sua solução para o problema.

Após uso de MOP, por sua escolha, ele diz: "Z *está solto*", referindo-se ao código de sua segunda cláusula. As figuras 8.53, 8.54, 8.55 e 8.56 mostram que os elementos da lista não estão sendo "guardados" na lista resposta (segundo parâmetro), o que leva o sujeito a reformular o segundo argumento ([X|Z]), como mostra sua versão de código posterior.




```
Regra: 2 --> lista([a,d,c,d,e],[a]) :- lista([d,c,d,e],B)
Meta: lista([d,c,d,e],B)
```

Figura 8.53 Primeiro passo da construção da Árvore de busca, mostrando o valor associado ao segundo parâmetro da meta ([a])

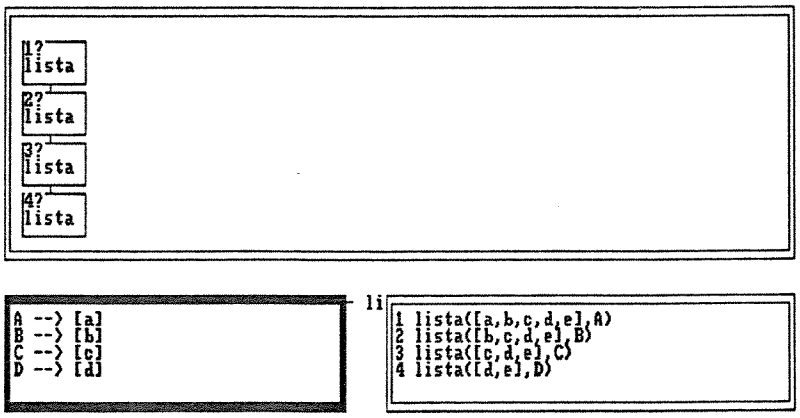


Figura 8.54 Árvore de busca mostrando que cada sub-meta tem uma resposta "independente"

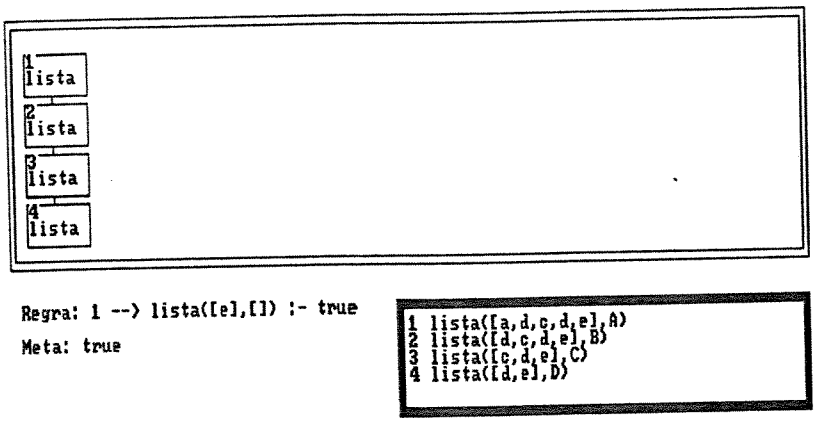


Figura 8.55 Sub-meta final do processo recursivo

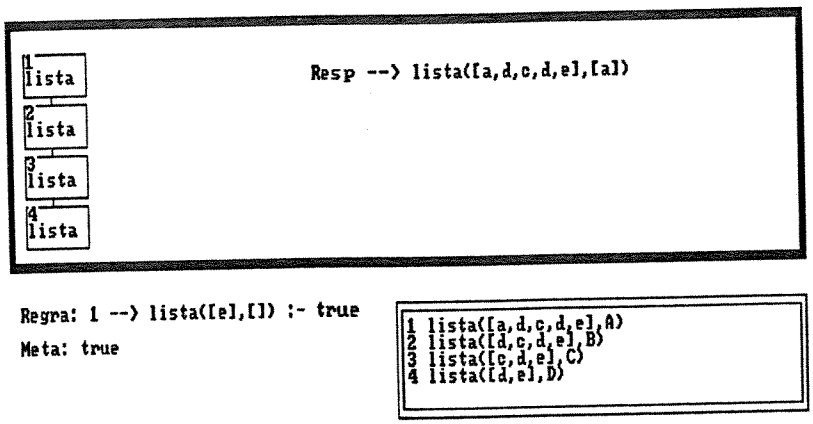


Figura 8.56 Resultado final do processo

versão 2:

lista([X],[]).

lista([X|Y],[X|Z]):-lista(Y,Z).

A versão 1 refletiu seu primeiro depoimento, uma vez que na chamada recursiva "lista(Y,Z)", o primeiro argumento (Y) representa a lista anterior sem seu primeiro elemento e, na leitura que o sujeito fazia, [X] no segundo argumento era lido como "coloca o primeiro elemento na lista resposta", o que dinamicamente era coerente.

Faltava uma leitura "declarativa" da cláusula, que associasse a cabeça da regra com o seu corpo. O interessante a notar é que o sujeito foi buscar o *feedback* usando o Módulo Operacional, ao invés do Declarativo. O sujeito usa o micro-mundo que melhor se ajusta a seu conhecimento anterior (a maneira "procedural" de olhar para o problema).

Exemplo 2: O enésimo elemento de uma lista por Ric.

versão 1:

elemento([X],1,X).

elemento([X|Y],N,Z):-

"*preciso escrever algo para caminhar na lista, só que aqui não sei como caminhar numa lista. Como pegar um elemento nessa lista?*" (grifo da pesquisadora).

Este exemplo mostra o sujeito no "meio" procedural, querendo prescrever para a máquina como "caminhar" na estrutura de dados, "pegar" elementos dessa estrutura, colocado no lugar da máquina ("não sei como caminhar numa lista"). Continuando:

N>0,

elemento([Y|Y2],N-1,Z).

A primeira cláusula é vista de forma "isolada" (resolve o caso mais simples) e não como o final do processo recursivo.

A sua maneira de "caminhar" na lista é uma chamada recursiva para a lista sem o primeiro elemento e o parâmetro N decrementado.

?elemento([a,b,c,d],3,X). (teste da versão 1)

no

versão 2:

elemento([X|Y],1,X).

elemento([X|Y],N,Z):-N>0, elemento([Y|Y2],N-1,Z).

? elemento([a,b,c,d],3,X). (teste da versão 2)

no

"agora não sei o que é. O que eu pensava furou." Ric

O sujeito usa o trace estendido do MOP e justifica: *"porque dá mais detalhe"*. Em geral essa opção é escolhida quando o sujeito assume conhecido o conhecimento estrutural do processo de execução.

O sujeito verifica que no terceiro passo do processo de execução o primeiro argumento da meta não está sendo associado a determinado valor (primeiro argumento de *elemento([G|H],3-1-1,I)*), como ilustra a figura 8.57, a seguir.

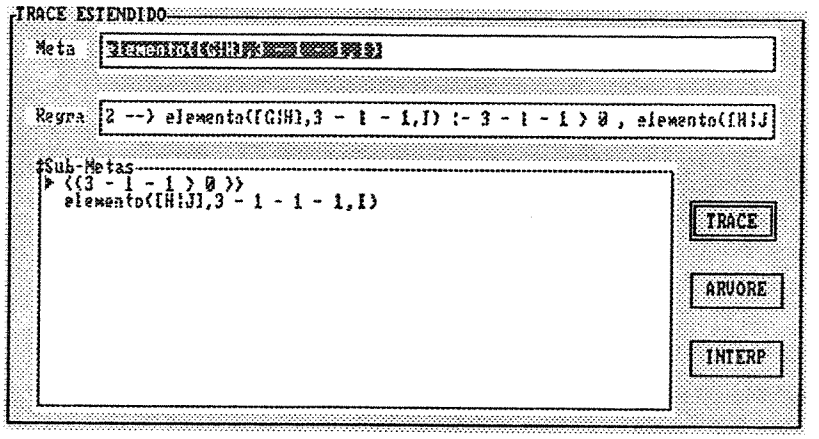


Figura 8.57 Ilustração do terceiro passo da iteração

Através da escolha da opção Trace Estendido, o sujeito pôde focalizar melhor sua atenção no processo de unificação, que acontece a cada passo, localizando o "*bug*" na descrição do primeiro argumento da chamada recursiva (elemento([Y|Y2],...)).

versão 3:

elemento([X|Y],1,X).

elemento([X|Y],N,Z):-N>0, elemento(Y,N-1,Z).

?elemento([a,b,c,d],3,X). (teste da versão 3)

no

"*ainda não*" Ric

O sujeito lê texto, executa mentalmente, e diz: "*quando N for 1 Z vai receber um valor!*" Ric

Usando MOP novamente o sujeito observa que, quando N (segundo argumento) chegou em "1" (3-1-1) a meta não casou a primeira regra de seu código (*elemento([X|Y],1,X)*), como era esperado, mas foi aplicar a segunda, como mostra a figura 8.58.

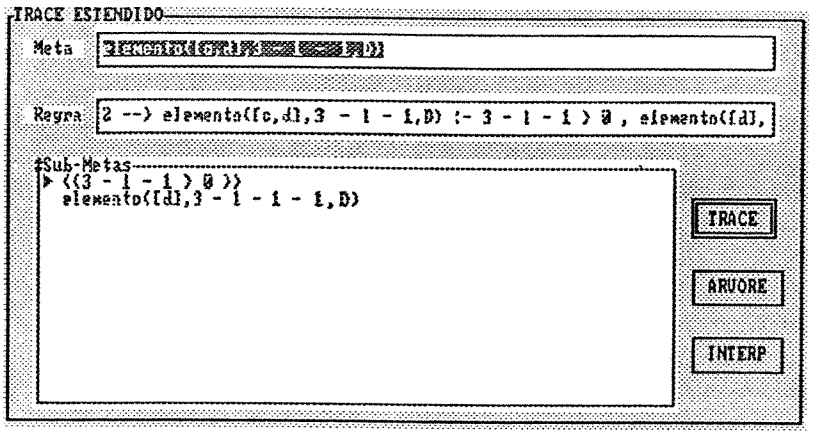


Figura 8.58 Passo do processo de execução em que deveria ser aplicada a primeira regra (segundo parâmetro é *3-1-1*)

Aí lembrou-se, também, de que a mesma situação havia acontecido nos problemas do fatorial, da potência e diz: "*não pode colocar N-1 no parâmetro*" e resolve o "problema".

Exemplo 3:

O mesmo sujeito, após trabalhar no predicado que fornece o enésimo elemento de uma lista, peço que "dada a lista e o elemento, predicado que forneça sua posição":

"*generalizando...*" Ric

sujeito mantém a primeira cláusula:

elemento([X|Y],1,X).

cria uma nova cláusula:

elemento([X|Y],N,Z):-N2 is N+1, elemento(X,N2,Z),

"*até que N2 fosse igual a N.*

não: até que X=Z" Ric

A idéia que o sujeito parece usar para resolver o problema é a de iteração: "N vai sendo incrementado até que Z seja igual a X (primeiro elemento da lista)".

Refazendo:

elemento([X|Y],N,Z):-N2 is N-1,

"mas não tenho o valor de N.

queria inicializar N2 e ir subindo até que Z=X" Ric

Refazendo:

elemento([X|Y],N,Z):-N2 is N-1(*), elemento(Y,N2,Z).

"de novo estou fazendo isso! não pode fazer isso aqui (*)

A consciência do novo meio: *"não pode fazer isso aqui"*. O sujeito já é capaz de pensar no processo de execução sem a ferramenta, que havia sido utilizada, no mesmo problema, quando era pedido o "enésimo elemento de uma lista".

Após simular "mentalmente" o processo da máquina de inferência, refaz:

elemento([X|Y],N,Z):-elemento(Y,N2,Z), N2 is N+1.

"incrementa o N quando voltar da recursão!" Ric

Para esse sujeito, as ferramentas do Módulo Operacional parecem fornecer elementos para que ele construa uma ponte de sua perspectiva procedural a partir de uma introjeção nos mecanismos da máquina, para o código Prolog. Isso não significa, ainda, uma mudançaa em sua perspectiva inicial de abordagem ao problema. As ferramentas estão apenas instrumentalizando o sujeito para resolver o problema, segundo sua perspectiva inicial. Entretanto esse pode ser um caminho necessário para o sujeito construir estruturas que o levem a uma mudança de perspectiva. Por exemplo, no caso do problema da potência, foi necessário ao sujeito esse tipo de

"exercício" antes que ele pudesse olhar de forma analítica para a definição matemática de potência de um número.

8.4.4 Síntese da Análise do Grupo II

Os resultados apresentados mostram que o processo de aquisição da nova linguagem (Prolog) pelo novato-Prolog é fortemente influenciado pelos elementos do paradigma procedural de programação. Num primeiro momento, esses elementos constituem a base de seu modelo conceitual inicial de linguagem e é esse modelo que o novato usa para captar o novo conhecimento. Esse aspecto aparece refletido na atividade de criação de programas, quando o sujeito tenta usar clichês já conhecidos no meio procedural.

A partir do uso das ferramentas, especialmente as do Módulo Operacional, o sujeito começa a reconhecer os elementos da máquina virtual da nova linguagem. Num segundo momento, a linguagem passa a ser vista como um novo "meio" para a representação da solução de problemas. Isso requer dele uma nova maneira de "pensar" sobre a representação da solução de problemas. Conceitos de programação conhecidos anteriormente, como por exemplo o conceito de recursão, são reconstruídos no novo "meio". Isso envolve um exercício não mais de tradução de uma linguagem para outra, mas de adaptação à nova linguagem.

8.5 Síntese de Resultados do Estudo

Ao contrário da idéia da *tabula rasa* o novato em determinada linguagem de programação traz para o ambiente de aprendizado elementos de sua experiência ante-

rior (não necessariamente com computadores), que interferem no seu entendimento da máquina virtual da linguagem em questão.

Enquanto esse modelo inicial da linguagem é fortemente influenciado pelo discurso da linguagem natural no caso do novato-novato (novato em programação), no caso do novato-Prolog esse modelo inicial é fortemente influenciado pelo formalismo de outras linguagens de programação.

A construção do conhecimento da nova linguagem é consequência de uma evolução nos seus modelos conceituais e se dá a medida em que o novato se engaja no ciclo hipótese-criação-execução-feedback. As ferramentas propostas atuam de maneira a enriquecer o *feedback* do ambiente e possibilitar ao sujeito uma mudança de perspectiva em sua maneira de "enxergar" o conhecimento sendo representado (programa) e seu interlocutor (máquina virtual).

A análise dos resultados mostra, no caso novato-novato, a maneira como ele se apropria do novo formalismo (Prolog) e o papel das ferramentas como uma janela para que ele veja a perspectiva da máquina para o problema sendo representado. A mudança de paradigma de programação em que o novato-Prolog é submetido envolve uma mudança de perspectiva em relação a sua maneira de "enxergar" a máquina virtual da linguagem. Em ambos os casos, a construção do conhecimento da nova linguagem se dá através desse exercício de mudança do "meio" onde a solução para o problema é representada (programa): do contexto da linguagem natural para Prolog; do contexto do paradigma procedural para o paradigma de Prolog.

As ferramentas serviram tanto ao grupo novato-novato quanto ao grupo novato-Prolog. Ao contrário do que era esperado, não houve uma diferença significativa no ritmo com que o sujeito do grupo novato-novato (relatado no estudo de caso) e os sujeitos do grupo novato-Prolog evoluiram nas tarefas propostas. O grupo novato-Prolog mostrou maior agilidade no uso das ferramentas e do ambiente

computacional como um todo, como era de se esperar. Entretanto, considerando seu conhecimento anterior de linguagens de programação, o mesmo não se pode dizer a respeito do processo de aquisição da nova linguagem. O modelo conceitual de linguagem de programação, no caso do novato-Prolog, é fortemente influenciado por elementos do paradigma procedural e passa por transformações ao longo do processo de interação com o ambiente proposto.

Capítulo 9

Discussão, Conclusões e Perspectivas

Capítulo 9

Discussão, Conclusões e Perspectivas

"O peixe é o único que não sabe que nada"

(proverbio chinês, em Ackermann, 1992)

Meu*1 objetivo neste capítulo é tomar distância do trabalho de pesquisa em que estive envolvida para tentar entender mais profundamente e expressar com maior clareza minhas reflexões, como investigadora, sobre a pesquisa realizada.

A minha preocupação durante este trabalho tem sido entender a atividade de programação enquanto atividade intelectual em que o sujeito (programador) se engaja, sob a perspectiva epistemológica. Segundo minha visão, a natureza do conhecimento envolvido na atividade de programar deve nortear o *design* de ferramentas e ambientes computacionais para novatos. Por outro lado, o uso dessas ferramentas e ambientes fornece respostas que representam janelas para a visualização de alguns aspectos do processo que estou interessada em conhecer. Uma discussão específica sobre ferramentas para ambientes de programação, portanto, deve endereçar uma discussão mais ampla de modo a entender, sob várias perspectivas (computacional, educacional, psicológica) o que o novato faz enquanto está programando, qual o papel do paradigma da linguagem que intermedia sua comunicação com o computador e como projetar ferramentas computacionais que possibilitem a ele engajar-se no ambiente de aprendizado.

*1 Em detrimento do estilo, tomo a liberdade de usar a primeira pessoa do singular neste capítulo, para expressar a ênfase na minha própria reflexão (no papel de aprendiz) sobre o problema que está sendo discutido.

Um estudo deste tipo poderia ter sido conduzido de muitas maneiras diferentes. Minha abordagem teve a influência da teoria construcionista de Papert do "objeto para se pensar sobre", conforme descrito a seguir:

> "Papert vê o aprendizado como particularmente efetivo quando ele tem lugar no contexto de uma atividade rica e concreta, que o aprendiz (criança assim como adulto) experimenta enquanto constrói um produto significativo tal como um pedaço de trabalho de arte, uma estória ou um relatório de pesquisa. Por várias razões - relacionadas ao seu *background*, estilo de pensar e suas crenças sobre o futuro da sociedade - Papert coloca grande ênfase em "pensar concretamente", em aprender enquanto constrói um programa de computador tangível ou alguma espécie de máquina funcional" (Harel, 1991, p. 26).

As ferramentas computacionais para o ambiente Prolog, desenvolvidas como parte desta pesquisa representam sob minha perspectiva como aprendiz, o "objeto para pensar sobre". Do ponto de vista de IA, o "objeto para se pensar sobre" refere-se à construção de artefatos que abrem janelas para a visualização de como se dá a apropriação do conhecimento na atividade de programar.

É através do processo de *design* das ferramentas e da observação de seu uso por novatos que busquei entender as questões mencionadas de início e que passarei a discutir.

A criação de ferramentas computacionais para constituir um ambiente de programação para novatos está centrada em dois aspectos básicos: primeiro o sujeito para quem a ferramenta é projetada (o novato), sugere que o ambiente de programação deva ser um "ambiente de aprendizado", no sentido de que programar significa mais do que representar uma solução já conhecida para um problema, numa dada linguagem. Segundo, um ambiente para programação pressupõe a atividade de expressar a solução de problemas segundo a visão que o aprendiz tem do

paradigma implícito na linguagem de programação em questão (Baranauskas, 1991a; Baranauskas 1991b). Como tal, as ferramentas desse ambiente devem refletir o meio gerado pelo paradigma da linguagem.

O Ambiente de Aprendizado Construcionista

As ferramentas propostas neste trabalho constituem parte de um cenário de como elas são usadas pelo novato e pelo facilitador e, como tal, devem ser analisadas dentro do quadro teórico do que entendo por "ambiente de aprendizado".

O *design* das ferramentas pressupôs uma imagem do aprendiz e do processo de aprender que integram a influência da linguagem e de aspectos sociais no aprendizado (Vygotsky 1962; Vygotsky 1978), com aspectos de desenvolvimento cognitivo baseado na construção e interação do sujeito com objetos da mídia computacional (Papert, 1980; Papert 1986).

Na abordagem construcionista, de Papert, o aprendizado é visto como uma construção e reconstrução de conhecimento pelo próprio sujeito em vez de transmissão de conhecimento de um sujeito mais informado para um sujeito receptor. A abordagem de Vygotsky focaliza o papel da linguagem no aprendizado e no desenvolvimento cognitivo. Enquanto Vygotsky enfatiza a interação do aprendiz com um "adulto informado" através da linguagem, Papert desloca esse diálogo para o computador. Ambos enfatizam o papel do diálogo na resolução de problemas (Harel, 1991).

Minha proposta de "ambiente de aprendizado construcionista" integra essas duas perspectivas: a aquisição de conhecimento se dá a partir da ação do sujeito no ambiente e esse diálogo é mediado pelo paradigma de programação implícito na linguagem. O sujeito de minha pesquisa interage num ambiente computacional constituído do interpretador Prolog acrescido das ferramentas, para construir

o conhecimento de seu interlocutor com a máquina (linguagem de programação). Essa interação é facilitada/influenciada por um experto (que pode ser o professor, um colega mais informado, um monitor, um observador participante, por exemplo): o mediador no sentido usado por Vygotsky.

O paradigma da aquisição de conhecimento pela sua transmissão direta, é substituído no construcionismo pela construção de artefatos que possibilitem que o sujeito aprenda. O ambiente constituído pelas ferramentas representa, portanto, uma proposta construcionista para o trabalho do novato em programação Prolog. Isso envolve aprender *sobre* e aprender *com* uma linguagem de programação baseada no paradigma da lógica, a partir da interação do próprio sujeito com os elementos do ambiente computacional.

A abordagem construcionista, pela sua própria origem tem sido adotada em trabalhos baseados no uso da linguagem Logo, dentro do que se convencionou chamar "cultura Logo". Este trabalho representa uma tentativa de transpor a teoria do construcionismo para outras linguagens e outros ambientes, mantendo a "estética Logo". A concepção de um ambiente baseado no uso das ferramentas aqui propostas representa um modelo construcionista de ambiente computacional para acesso (ensino/aprendizagem) de novatos ao paradigma da programação em lógica.

As ferramentas tradicionais disponíveis nos chamados "ambientes de programação" abrangem desde editores estruturados, checagem de sintaxe e propriedades semânticas a ferramentas para depuração. São ferramentas que dão suporte ao sujeito enquanto este já é um programador. Dessa maneira, nos estudos que se concentram nos estágios iniciais do processo de programação mencionados na literatura, o ambiente de programação não tem tido papel significativo (Van Someren, 1990). Nesses estudos as ferramentas são entendidas como apoio aos sujeitos, em situações de erro que são identificadas em um contexto convencional de ensino/aprendizado de Prolog (através de aulas, livros-texto). O estudo realizado

neste trabalho pretendeu captar a evolução dos aprendizes na aquisição da linguagem de programação (Prolog), a partir exclusivamente de sua interação no ambiente proposto. O *design* das ferramentas aqui apresentadas, portanto, teve como alvo o novato e representou uma nova concepção de "ambiente de programação", onde as ferramentas tem um papel relevante no processo de aprender do sujeito. A idéia subjacente é a do aprendiz como agente no processo, onde a assimilação dos conceitos acontece de acordo com seus esquemas mentais ao invés de instruções ou intervenções "inteligentes", que são elaboradas *a priori*. Nesse sentido, qualquer tentativa de instrução é inócua.

O ciclo hipótese-criação-execução-feedback

No cenário proposto neste trabalho, o sujeito interage num ambiente composto do sistema da linguagem (interpretador Prolog) acrescido das ferramentas (MOP, MDS, EDS) e tem o acompanhamento de um experto (facilitador) na tarefa de programar. A atividade de programar em Prolog envolve o ciclo criar e depurar uma base de dados. Para o novato, a criação e a depuração da base de dados não são duas atividades dissociadas mas fazem parte de um ciclo em que o novato se engaja que compreende: a partir de um conjunto inicial de hipóteses a respeito do problema em questão e a respeito do interlocutor (linguagem de programação), criação de uma base de dados (fatos e regras sobre o domínio do problema) e execução pela máquina virtual que fornece respostas a perguntas formuladas pelo sujeito. A resposta dada pela máquina nem sempre gera *feedback* suficiente para realimentar o ciclo. A proposta das ferramentas foi enriquecer o *feedback* de modo a levar o sujeito a refletir e reformular seu conjunto de hipóteses, reformular sua base de dados e continuar o ciclo.

A reflexão do sujeito se dá não apenas ao nível do problema que ele está resolvendo mas, o que é mais importante, ao nível do processo de aprender do próprio sujeito, como mostrado no capítulo anterior (ver depoimento de Ric, na seção 8.2.5).

Podemos explicar, do ponto de vista psicológico, a efetividade do ciclo em que o novato se engaja, citando o trabalho de Ackermann:

> "Enquanto tudo funciona suavemente e conseguimos o que queremos fazendo o que fazemos, não há necessidade de grande esforço de reflexão ou justificação. É quando insatisfação ou dúvida aparece, que um novo nível de reflexão é requerido. Dúvida aparece em uma variedade de situações que são diferentes para diferentes pessoas. Por exemplo, dúvida pode aparecer quando não estamos mais certos sobre o valor de um pensamento, quando somos repetidamente contraditos em nossas crenças ou quando não conseguimos o que queremos fazendo o que fazemos. Genericamente dúvida aparece quando percebemos discrepância entre um estado desejado e um estado corrente. Perceber uma discrepância na maioria das vezes chama por mudança e coloca em movimento uma busca por novas e mais acuradas estratégias, para diminuir a diferença. ... Poderíamos dizer que crescimento requer *feedback* e *feedback* requer engajamento e ação. Assim, ambos ação e avaliação dos efeitos de uma ação são essenciais para o aprendizado." (Ackermann, 1991, p.1, p.2).

No capítulo anterior foram mostrados vários exemplos de como o novato se engaja nesse ciclo e os tipos de *feedback* gerados pelas ferramentas.

A literatura tem mostrado uma polarização nas abordagens ao ensino de Prolog que enfatizam ou o aspecto declarativo ou o aspecto mecanístico. A abordagem declarativa ao Prolog, em sua falta de ênfase nos mecanismos da linguagem, pode levar o novato, inicialmente, a ver Prolog como uma simplificação da linguagem natural. Por outro lado, os estudantes introduzidos à semântica procedural

tendem a interpretar as estratégias de execução de Prolog como mais "inteligentes" do que realmente são (Taylor, 1990). Os resultados de nosso estudo apontaram na mesma direção, quando observamos novatos que utilizaram exclusivamente ou o Módulo Declarativo ou o Módulo Operacional.

No cenário proposto neste trabalho, o *feedback* gerado pelas ferramentas aborda o aspecto declarativo e/ou o aspecto operacional, dependendo da escolha que o sujeito faz da ferramenta ou o facilitador sugere para trabalhar um aspecto específico do problema em questão.

No capítulo anterior mostramos exemplos onde o *feedback* operacional era necessário para realimentar o ciclo (ver exemplo de Rog em [idade], na seção 8.2.3). Também vimos casos onde o *feedback* declarativo foi necessário para o sujeito "reorganizar as variáveis" de forma que a cláusula fizesse sentido a ele (ver exemplo 2 na seção 8.4.2).

Dessa maneira, o *feedback* de que falamos no ciclo hipótese-criação-execução-feedback não premia uma ou outra abordagem, mas coloca a disposição ambas para que a escolha seja feita pelo sujeito (de acordo com sua "preferência pessoal") ou pelo facilitador (de acordo com o que ele avalia da situação, que seria mais conveniente do ponto de vista metodológico).

Nossos resultados sugerem uma tendência de o novato-Prolog (acostumado a linguagens procedurais) usar mais as ferramentas do Módulo Operacional. Isso pode ser explicado em função do Modelo Conceitual Inicial que eles fazem da máquina virtual, que é fortemente influenciado por sua experiência anterior, conforme será discutido na próxima seção.

O Modelo Inicial da Máquina Virtual Criado pelo Sujeito

Estudos citados na literatura têm mostrado que o novato aplica à tarefa de programar o ciclo de entender e interpretar o discurso humano (Pea, 1986; Taylor, 1990). A principal pressuposição feita é a de que os aprendizes não vêm para a situação de aprendizado como "recipientes vazios", mesmo que o domínio seja novo para eles. Taylor explica o uso por novatos de uma analogia entre o discurso humano e o discurso com computadores, da seguinte maneira:

> "...porque os novatos não têm um conhecimento do domínio específico, eles são *obrigados* a interpretar uma nova situação a partir do nível de discurso geral. Nessa atividade os aprendizes necessitam usar entendimento de discurso de propósito geral e estratégias de raciocínio. Conforme eles aplicam essas estratégias, eles trazem qualquer informação existente que eles têm, que parece a eles relevante. Essa informação será montada em uma estória ou um conjunto de explicações que constitui seu modelo mental em crescimento, do discurso de programação." (Taylor, 1990, p.309)

No estudo experimental, o uso de conhecimento e estratégias do mundo real na construção pelo sujeito de sua própria visão de Prolog foram constatadas principalmente no grupo novato-novato. O uso do discurso humano na interpretação da linguagem e do comportamento da máquina apareceram, por exemplo, na seção 8.2.2 do capítulo anterior, quando analisamos a interferência da Linguagem Natural no formalismo clausal para representar sentenças. Na mesma seção mostramos o uso do "contexto estendido" como uma forma de o sujeito atribuir à base de dados mais conhecimento do que ela possui, originário do senso comum.

Enquanto Pea descreve o uso de estratégias do discurso humano na interpretação que o novato faz da linguagem e do comportamento da máquina, como

um "superbug", Taylor o descreve como uma forma de "raciocínio abdutivo" descrito a seguir:

> "se uma dada observação pode, juntamente com crenças já estabelecidas, ser explicada pela adoção de uma hipótese particular, então aquela hipótese é adotada (e a observação é explicada concordantemente). Abdução é vista como a primeira fase do raciocínio no desenvolvimento de novo conhecimento porque ela gera a hipótese a partir da qual outro raciocínio pode, em princípio, se seguir." (Taylor, 1990, p.285).

Abdução explica, a nosso ver os modelos conceituais iniciais que o novato (dos dois grupos: novato-novato e novato-Prolog) constrói sobre a máquina virtual. Mais do que o uso do discurso geral em sua interpretação da máquina, entretanto, os resultados nos levam a dizer que o novato faz uso de seu conhecimento anterior na construção desse modelo inicial, que pode ser o do discurso humano (mais frequente entre o novato-novato) ou o de um discurso formal segundo um paradigma conhecido anteriormente (um sistema formal). Esse conhecimento inicial pode ser explicado como o esquema mental que permite ao sujeito interpretar o mundo, segundo Piaget.

O modelo conceitual inicial que o novato-novato cria da máquina virtual Prolog é fortemente associado à imagem que ele tem do computador que, por sua vez, se apresenta segundo componentes do padrão humano (possuidor de memória, uso de lógica, respondedor de perguntas, etc.).

O modelo conceitual inicial da máquina virtual que o novato-Prolog constrói é fortemente influenciado pela sua experiência anterior com sistemas formais (linguagens natural e artificial). Nas seções 8.4.1, 8.4.2 e 8.4.3 mostramos como os sujeitos representam a solução de problemas em função de seu modelo inicial e a influência do paradigma procedural de programação nesse processo.

Em vez de falarmos do "discurso humano", então, achamos mais apropriado falar do discurso segundo o modelo conceitual inicial que o sujeito cria e modifica ao longo de seu processo de aprendizado.

Entendimento de discurso é um processo cíclico que envolve interpretação, raciocínio, produção e avaliação de produção (Taylor, 1990). Suposições são feitas pelo novato em diferentes fases no discurso que nem sempre correspondem a informações corretas (como as apresentadas em livros-texto, por exemplo). Tendo abduzido uma hipótese, o raciocínio abdutivo futuro é baseado nessa "teoria" que o sujeito fez e que pode não ser o modelo correto. O *feedback* gerado pelas ferramentas pode proporcionar ao sujeito um "redirecionamento" no sentido da depuração de sua teoria anterior.

Outro problema associado ao uso do raciocínio abdutivo, mencionado na literatura é que ele pode prevenir o aprendiz de considerar todas as informações relevantes necessárias e contribuir para um emaranhado de más-interpretações. Esse é um aspecto, no cenário para o ambiente de aprendizado que propusemos neste trabalho, que poderia ser investigado e instigado pela figura do facilitador (experto).

A Atividade de Programar e o "Meio" Gerado pelo Paradigma da Linguagem

O computador toma formas diferentes conforme o paradigma embutido na linguagem que intermedia o processo de interação do novato com a máquina.

Programação tem sido tratada na literatura como uma tarefa de criação de "planos"*2 (Soloway e Woolf 1980; Rist, 1986). O programador deve construir planos executáveis pelo computador para chegar a uma saída dada uma entrada (Van Someren, 1990). Linguagens procedurais possuem primitivas que repre-

*2 *planning* no original

sentam abstrações complexas e poderosas como parte inerente da linguagem. Blocos de construção desses planos, como por exemplo "repita uma operação até que uma condição seja satisfeita", são abstrações que já embutem essas primitivas.

Por outro lado, como o novato tem um entendimento parcial da semântica da linguagem de programação, ele tende a usar seu conhecimento de linguagem natural para escrever os planos. Esse processo de dois estágios onde, primeiro o novato constrói um plano com instruções em linguagem natural e depois tenta transformar esse plano em código Pascal foi analisado por Bonar (citado em Van Someren, 1990). Os resultados de Bonar mostraram que mesmo quando o plano está correto (o que geralmente é o caso), muitos problemas acontecem em função da tradução para Pascal.

Esses resultados reforçam a idéia da construção do programa como um processo único, função do meio gerado pelo paradigma da linguagem e não como dois processos dissociados: planejamento e codificação ou resolução do problema e programação. Argumentamos que programar significa "moldar" a solução do problema de acordo com o "meio" gerado pelo paradigma da linguagem. Podemos entender "meio" como as "primitivas" (num sentido mais amplo do que "comandos") da linguagem de programação. No caso de Prolog, por exemplo, podem ser consideradas "primitivas", para sua semântica operacional, os processos de unificação e busca em profundidade, enquanto que em linguagens procedurais as primitivas seriam atribuição de valor a variável e sequenciação.

As ferramentas que compõem o ambiente tiveram como meta principal refletir o meio gerado pelo paradigma da linguagem, ao mesmo tempo em que deveriam possibilitar ao sujeito construir e se expressar de acordo com os modelos de como ele enxerga esse meio.

Refletir o meio gerado por Prolog envolve possibilitar ao sujeito o trabalho em aspectos operacionais da representação da solução do problema, bem

como em aspectos declarativos onde não existe um componente de "controle" na mão do sujeito; o problema é visto em termos das relações entre os elementos de sua base de dados e das relações que podem ser deduzidas das previamente definidas. Ackermann enfatiza a importância de múltiplas abordagens para (e descrições de) um problema como a chave para o aprendizado, que transcrevemos a seguir:

> "Um ambiente de aprendizado rico é um ambiente que provê oportunidade de "passear" dentro e fora de um problema, da mesma maneira que passeamos dentro e ao redor de uma cidade para descobrir suas várias facetas. Passear dentro e fora do problema envolve olhar para o problema de diferentes perspectivas, imergindo no problema, saindo do problema e olhando para ele à distância. Envolve expressar e moldar o entendimento pela construção de modelos ou descrições da situação conforme se vai ao longo do processo investigando esses modelos ou descrições e compartilhando-os com outros." (Ackermann, 1991, p.1)

Pude observar que a maneira de o sujeito engajar-se no processo é diferente dependendo de vários fatores, possivelmente de seu estilo cognitivo, momento em que ele se encontra no processo de resolver o problema, etc.. Encontramos, por exemplo, o sujeito que usa as ferramentas do Módulo Operacional para "mergulhar" no problema, colocando-se no papel do interpretador. Só após esse exercício de mudança de perspectiva é que ele consegue distanciar-se do problema e ter uma visão analítica dele (ver, por exemplo, Ric no exemplo 4 da seção 8.4.2).

No capítulo anterior mostrei o modelo inicial que os novatos usam para a comunicação com a máquina virtual, como esses modelos são influenciados pela experiência anterior do sujeito (com linguagem natural principalmente no caso do novato-novato e com linguagens procedurais no caso do novato-Prolog) e como esses modelos são modificados ao longo do processo, a partir do *feedback* gerado pelas ferramentas.

A construção do conhecimento da nova linguagem envolveu captar o "meio" gerado pelo paradigma da linguagem e aconteceu a partir de transformações nesse modelo inicial. O entendimento desse processo tem implicações tanto a nível do facilitador, enquanto atua no ambiente, quanto a nível do *designer* de ferramentas. Esse processo de criação e transformação de modelos fornece um contexto para que o facilitador situe as dificuldades conceituais do novato no aprendizado da linguagem, colocando uma mudança de perspectiva em relação ao entendimento dos "erros de programação" do sujeito. A nível do *designer* de ferramentas, o entendimento de como se dá o processo de apropriação da linguagem, especialmente no caso de linguagens não procedurais, representa um importante *feedback* para a criação de ferramentas para ambientes baseados em novos paradigmas de programação, onde muda a relação do programador com o aspecto de "controle" da máquina virtual.

Ferramentas Cognitivas

Conceitos de linguagens de programação são, em geral, apresentados como teorias abstratas, por representações estáticas enquanto que muitos dos conceitos e técnicas envolvidos são dinâmicos em sua natureza. Uma das razões pelas quais o novato, em geral, tem dificuldade em aprender conceitos de programação é relacionada ao problema de associar a natureza abstrata da descrição de um conceito com a estrutura e conteúdo dos programas que ele está tentando escrever (Rajan, 1990).

Os trabalhos apresentados na literatura envolvendo Prolog como linguagem de programação têm lidado com essa questão apresentando ferramentas que propõem representações gráficas ou de animação para mostrar a execução de programas (Rajan, 1990; Eisenstadt, 1990; Plummer 1988; Dewar 1986). Reconheço a necessidade de ferramentas para facilitar ao novato a visualização de como seu

programa está sendo executado pela máquina e os resultados do estudo experimental (capítulo 8) mostram a efetividade do uso das ferramentas do Módulo Operacional (MOP-árvore e MOP-trace estendido) para realimentação do ciclo hipótese-criação-execução-feedback.

Por outro lado, os aspectos mecanísticos da linguagem representam apenas parte do meio gerado pelo seu paradigma. Programas Prolog podem ser interpretados como especificações lógicas ou como programas em sua forma convencional (instruções para a máquina) e, como tal, dois tipos de discurso formal são requeridos no diálogo em que o sujeito é intermediado por Prolog: um lógico e um mecanístico, como descrito a seguir:

> "... o domínio mecanístico é um domínio no qual expressões em Prolog são interpretadas por referência a mecanismos computacionais ou virtuais, enquanto que o domínio lógico é um domínio no qual expressões em Prolog são entendidas ou por referência a um modelo hipotético ou são dadas uma interpretação puramente sintática." (Taylor, 1990, p. 287)

A ênfase nos aspectos operacionais coloca o sujeito num discurso mecanístico no processo de resolver o problema. De igual importância a se considerar, então, são ferramentas que possibilitem ao sujeito engajar-se no discurso lógico, dentro do discurso formal de resolução de problemas.

A abordagem da interpretação declarativa ao ensino de Prolog tem sido defendida na literatura (Kowalski, 1982), embora não esteja claro qual é o papel que o significado declarativo de um programa pode ter na tarefa de programação (Van Someren, 1990). Até o presente momento, não tenho conhecimento, a partir da literatura, de ferramentas dirigidas a novatos no ambiente Prolog, que abordem o aspecto declarativo da programação, além de minha iniciativa anterior (Baranauskas, 1991b). A dificuldade em lidar com o aspecto declarativo envolvido

na tarefa de programar parece ser uma herança da cultura computacional em que estamos imersos, cuja arquitetura subjacente (sequencial) induz o aspecto mecanístico. É possível que, em novas arquiteturas, a linguagem esteja mais isenta das especificidades da arquitetura da máquina e o foco do processo de programar seja tirado de seus aspectos mecanísticos.

O *design* de ferramentas para o ambiente Prolog apresentado neste trabalho teve como princípio norteador possibilitar ao novato uma visualização de ambos os aspectos envolvidos no programa Prolog: o mecanístico (com ênfase no processo de execução do programa) e o declarativo (com ênfase no inter-relacionamento entre os objetos/conceitos presentes no texto do programa).

As ferramentas do Módulo Declarativo (EDS e MDS) representam uma iniciativa de possibilitar ao novato engajar-se no discurso lógico na atividade de programar em Prolog. Ao mesmo tempo, tentei entender o papel que o significado declarativo de um programa pode ter na tarefa de programação do novato.

Os resultados obtidos do estudo realizado (capítulo 8) mostraram um envolvimento relativo muito maior do grupo novato-novato (sujeitos sem prévio conhecimento de programação) com as ferramentas do Módulo Declarativo. Isso pode ser explicado pela ausência, nesse grupo, de conhecimento especializado de programação. O novato-Prolog já adquiriu um nível de discurso formal especializado (programação procedural) que os deixa mais próximos do mecanístico.

As ferramentas do Módulo Declarativo mostraram a importância de seu *feedback*, principalmente nas situações em que o novato começa a escrever sentenças na forma clausal, mapeando seu discurso geral dentro do mundo real (sentença comum), para o discurso formal (formalismo clausal). A sentença que ele tem em mente (discurso geral) é escrita segundo um formalismo que nem sempre reflete o significado pretendido (seu conhecimento do formalismo clausal não está completo). Isso envolve, para o novato, fazer uma distinção entre o que *ele* acha que

a sentença significa e o que *ele* supõe sobre a capacidade do computador entender e interpretar o código que escreveu. A representação gráfica da sentença codificada (gerada por EDS, MDS) pode dar ao sujeito pistas do significado da sentença codificada no formalismo clausal, segundo a perspectiva da máquina (como mostrado na seção 8.2.5, por exemplo, uma regra mal escrita, onde um dos objetos fica "solto" no diagrama).

Van Someren observa que novatos tendem a avaliar partes do código *localmente*, como módulos separados e assim não levam em consideração como, por exemplo, as cláusulas colaboram (Van Someren, 1990). Neste estudo mostrei como esse tipo de novato fez uso do Módulo Declarativo (MDS), não mais ao nível de cláusula, mas para analisar a informação contida em sua base de dados, considerada como todo. Esse processo de análise do programa, baseado em fatores outros que não construção exaustiva da execução do programa é chamado na literatura, "meta-análise" (Hook, 1990). A ferramenta MDS facilita ao novato a meta-análise de sua base de dados, enquanto tira de foco o conhecimento expresso em uma cláusula específica para mostrar o inter-relacionamento das cláusulas que têm objetos em comum.

A literatura tem mostrado a ênfase que tem sido dada em prover o programador novato com modelos da máquina virtual Prolog (também chamados máquinas notacionais). Pain e Bundy (1987) fazem uma análise de alguns modelos possíveis, que chamam de "estórias" Prolog. Algumas ferramentas foram desenvolvidas implementando alguns dos modelos mencionados (por exemplo, Eisenstadt, 1988).

Uma questão que se coloca refere-se à "granulação", isto é, ao nível de descrição do processo de execução que deve ser mostrado nessas máquinas e os consequentes modelos de representação adotados. Eisenstadt, por exemplo, observa que usuários de diagramas AORTA (em sua ferramenta TPM), na primeira vez que

os usam acham "*os diagramas mais difíceis de entender do que o próprio Prolog*" (Eisenstadt, 1990). Ele acrescenta, entretanto, que essa dificuldade dura pouco e que existe um ponto no processo em que os estudantes revertem a situação, escrevendo diagramas AORTA no papel, para entender o que seus programas estão fazendo.

Se, por um lado, essa dificuldade inicial do novato é diretamente proporcional ao nível de sofisticação da ferramenta (TPM foi projetada para atender às necessidades de ambos novato e *expert*), por outro lado, simplificações no modelo adotado para as máquinas notacionais podem levar o novato a interpretações enganosas, como ilustra Burstein (citado em Van Someren, 1990). Ele mostra, por exemplo, que representar uma variável por uma "caixa", um valor por um "objeto" e o símbolo de atribuição de valor a variável, ":=" , como sendo a ação de colocar um objeto em uma caixa pode levar o novato a entender que, após execução dos comandos

A:=1

B:=A

a "caixa" A estaria vazia porque a segunda linha poderia sugerir que o objeto 1 seria tirado da caixa A e colocado na caixa B.

A questão da representação utilizada nas ferramentas é, a meu ver, uma solução de compromisso entre a simplicidade inicial (que deve considerar o sujeito para quem a ferramenta é projetada) e o grau de fidelidade do modelo aos processos/conceitos que ele representa, de modo a possibilitar uma transição gradual ao invés de brusca, pelos conceitos abordados.

No caso do ambiente proposto neste trabalho, esse impacto inicial das representações (árvore de busca, diagrama semântico) foi menos evidente, talvez porque o novato não tinha um conhecimento inicial de Prolog (adquirido de livros-texto ou aulas expositivas); o conhecimento da linguagem foi sendo construído junto

e através do uso das ferramentas. Alguns novatos passaram a desenhar no papel diagramas semânticos para entender ou explicar seu entendimento de uma base de dados (ver por exemplo, protocolo de Rog em [idade], na seção 8.2.3).

O diagrama da árvore de busca parece ter funcionado como suporte para a "execução mental" que os sujeitos do grupo novato-Prolog costumavam fazer (ver seção 8.4.3), antecipando uma resposta do interpretador ou analisando como sua base de dados estaria se comportando. Os diagramas da árvore de busca ofereceram ao sujeito um modelo concreto para ele pensar sobre conceitos de natureza dinâmica como por exemplo, a "volta da recursão" (ver exemplo na seção 8.4.2).

Os resultados do estudo experimental mostram, também, que os sujeitos estabelecem tipos diferentes de relação com as ferramentas, o que pode ser explicado por uma combinação de fatores como por exemplo, seu conhecimento anterior, o momento que se encontra no processo de aprendizado da linguagem, seu estilo cognitivo. Há o sujeito que usa as ferramentas no seu sentido restrito de ferramenta, como um "utilitário". As ferramentas funcionam, para esse tipo de sujeito, como "lentes" para visualização de "o quê a máquina faz com seu código". Seu engajamento no ambiente acontece ao estilo desafio: o sujeito tenta realizar a tarefa (escrever/depurar o programa) e usa a ferramenta como forma de "checar" seu raciocínio, confirmando-o ou não.

Opostamente ao analítico, há o sujeito que literalmente se envolve no ambiente e usa as ferramentas como um meio de exercitar o processo de imergir e emergir, tomando distância (*diving in* e *stepping back*) do problema em questão. Nesse tipo de relação, quando usa o Módulo Operacional, por exemplo, o sujeito se coloca antes no papel da máquina virtual, para só então conseguir uma visão analítica do problema.

Em vez de projetar na ferramenta a capacidade humana, ferramentas cognitivas representam um esforço no sentido de possibilitar ao sujeito que ele faça o

máximo uso de seu próprio entendimento e percepção, na tarefa de resolver um problema (criar um texto, um programa, etc.). Meu trabalho de propor, desenvolver e estudar o uso das ferramentas esteve afinado com essa idéia. Para o sujeito, as ferramentas representaram uma janela por onde ele pôde visualizar ou "mergulhar" no processo. Para o pesquisador, a janela possibilitou entender o que está envolvido enquanto o sujeito programa. Esse entendimento contribui para o ciclo que o desenvolvimento de qualquer sistema baseado no computador deve possuir, do *design* para a experiência e de volta ao *design*.

Perspectiva de Trabalho Futuro

Várias questões surgem dos resultados apresentados neste trabalho, que, como comentarei a seguir, poderão dar continuidade à pesquisa em novas direções:

- Pesquisa de natureza investigativa para responder a novas perguntas levantadas, considerando o ambiente como inicialmente idealizado.

- Pesquisa e desenvolvimento de sistemas computacionais que incorporam as idéias de representações gráficas para apresentação visual da informação.

A seguir apresentamos novos questionamentos que surgem a partir deste trabalho, limitações do ambiente proposto e novas direções para a pesquisa.

O Tipo do Novato

O estudo experimental realizado baseou-se em dois tipos de sujeitos: o novato em sua primeira experiência com computadores (novato-novato) e o novato no paradigma da programação em lógica, mas com boa experiência em linguagens procedurais. O novato-Prolog reflete a situação mais comum de aprendizes da linguagem Prolog até o momento atual, pois a grande maioria de usuários de linguagens de programação, foram "alfabetizados" em linguagens procedurais. O novato-

novato, por outro lado, representa o usuário que não tem a influência anterior de um paradigma de programação. Essa escolha foi direcionada, portanto, para a observação da questão da dualidade de semânticas envolvidas na atividade de programar em Prolog, refletidas pelas ferramentas que constituem o ambiente aqui representado.

Em face dos resultados observados e já discutidos, novas perguntas podem ser feitas envolvendo outros tipos de sujeitos, como por exemplo:

De que maneira, novatos com outro tipo de *background* interagem nesse ambiente? Estudantes da área de Humanidades, por exemplo, se apropriariam do ambiente de forma diferente? Sujeitos com conhecimento da Teoria da Lógica tenderiam a escolher as ferramentas do Módulo Declarativo com maior frequência? O *design* das ferramentas é flexível o suficiente para não se contrapor aos modelos conceituais iniciais desses outros tipos de sujeitos? Foi feito um estudo piloto com dois sujeitos do curso de Filosofia, usando MDS. Uma análise inicial dos resultados observados mostrou que a problemática dos sujeitos esteve muito próxima da problemática do novato-novato no ambiente Prolog, no sentido de seu modelo conceitual inicial da linguagem ser baseado no discurso da linguagem natural. Entretanto, a formulação de hipóteses, pelos sujeitos, sobre "como as inferências são realizadas pela máquina" foi muito mais elaborada, provavelmente em função de terem um conhecimento anterior de Teoria da Lógica (os sujeitos haviam feito uma disciplina introdutória em Lógica). As perguntas levantadas e outras, exigem uma investigação mais prolongada e poderiam ser objeto de novos trabalhos de pesquisa.

O Papel do Facilitador no Ambiente

O ambiente de aprendizado baseado nas ferramentas propostas conta com a participação de um facilitador que é um experto (no sentido de experiente). A atuação do facilitador deve estar afinada com a proposta metodológica de um am-

biente construcionista. O *Manifesto de Genebra para Ambientes de Aprendizado Inteligentes* reflete a dificuldade do papel do facilitador:

> "A quantidade de ajuda necessitada pelo aprendiz não é uma constante educacional universal, mas um parâmetro a ser modificado através de ações educacionais." (TECFA, 1990, frase n. 6)

Em relação ao seu papel no ambiente de forma a intervir no processo de aprendizado do sujeito e, em particular, na interação do novato com as ferramentas do ambiente, pode-se perguntar:

Até que ponto o facilitador (experto) pode (ou deve) interferir na escolha que o sujeito faz das ferramentas? Seria desejável contrabalançar o uso das ferramentas em relação aos aspectos declarativo e operacional envolvidos na programação Prolog? O papel do facilitador poderia ser feito por uma parte do próprio sistema computacional? Até que ponto?

A Influência da Natureza do Problema

Prolog é uma linguagem de programação de propósito geral e, como tal, pode expressar tanto problemas "abertos", de natureza declarativa como por exemplo a criação de uma base de dados com informações (fatos e regras) sobre um certo domínio, quanto problemas cuja solução envolve um processo bem definido como por exemplo, o problema de ordenar uma série de itens. Reconhecidas essas diferenças pode-se perguntar:

Qual é o papel da natureza do problema que o novato está resolvendo, na abordagem que o sujeito usa (operacional *versus* declarativa) para apropriar-se da linguagem? Trabalhar em processamento de listas, no problema de "elevar um

número a uma certa potência" ou no problema da "busca ao tesouro" interfere na construção do meio gerado pelo paradigma da linguagem?

Programação em Lógica e o Raciocínio Lógico

Estudos extensivos de psicólogos a respeito do raciocínio humano refutam a visão de que as pessoas tipicamente raciocinam de acordo com a lógica formal e apontam para a conclusão geral de que as pessoas têm uma baixa performance em tarefas que envolvem raciocínio proposicional (Colbourn, 1988). Taylor mostra que quando se apresenta a pessoas comuns, problemas envolvendo expressões lógicas, elas se transportam para o "mundo real" para raciocinar, mas, então, não mapeiam seu entendimento de volta para o domínio da Lógica em nenhum ponto subsequente do processo de resolução do problema. A solução do sujeito faz sentido no "mundo real", mas não seria aceitável no contexto da Lógica (Taylor, 1990).

O ambiente baseado em Prolog, embora contenha características restritivas que não são inerentes à Lógica, por outro lado, tem os benefícios de trazer o sujeito para um sistema formal onde ele possa "praticar" o raciocínio dedutivo, sem a necessidade de entendimento da teoria da Lógica.

Simetricamente, para os estudantes de Lógica, o ambiente baseado em Prolog pode funcionar "*como um meio de consolidar o entendimento do estudante sobre Lógica, bem como mostrar como a análise lógica pode ser aplicada na prática*" (Spencer-Smith, 1990).

Através das ferramentas propostas, o sujeito pode testar o conteúdo lógico da definição de sua base de dados. Além disso, ele pode ver a análise lógica sendo posta para funcionar, de maneira concreta.

O uso do ambiente proposto força o aprendiz a fazer distinções claras entre tipos diferentes de informação, organizá-las consistentemente e tornar explíci-

tas as conexões lógicas entre tipos diferentes de informação, ao mesmo tempo possibilitando e impondo coerência lógica.

O potencial dos ambientes baseados em Prolog como um meio de externalizar o conhecimento revelando sua não completeza e inconsistência, representam um contexto rico para investigações sobre o desenvolvimento do raciocínio lógico e da metacognição.

Uma das possibilidades para continuar a investigação dessas questões é colocar as notações propostas pelas ferramentas, em uso através de diferentes mídias (diagramas em livro-texto, animações em vídeo, implementação em estações de trabalho gráficas), para diferentes tipos de usuários, em um projeto interdisciplinar. Nesse tipo de investigação seria fundamental a participação de pesquisadores de outras áreas para uma análise mais profunda dos aspectos psicológicos e educacionais envolvidos.

Limitações do Ambiente Proposto e Novas Possibilidades

Nossa escolha do ambiente de desenvolvimento das ferramentas representou uma solução de compromisso entre tornar o ambiente acessível a usuários não sofisticados e qualidade e desempenho do sistema.

As ferramentas propostas foram desenvolvidas para computadores da linha PC usando Arity Prolog (Arity, 1990) como linguagem de implementação.

As restrições em termos de capacidade de armazenamento e recursos gráficos, impostos pelo ambiente de desenvolvimento refletem no problema de escala. A atual implementação das ferramentas está restrita a representação de problemas no nível (tamanho) dos apresentados em livros-texto, com objetivos didáticos. Deve-se observar que essa restrição é uma consequência do ambiente de implementação e não se estende às representações propostas (diagrama semântico e

árvore de busca). O problema de escala pode ser minimizado com a possibilidade de implementação em ambientes com maiores recursos (estações de trabalho, por exemplo).

Do ponto de vista de ambiente de aprendizado, a principal dificuldade refere-se à medida de controle que o aprendiz tem sobre suas próprias atividades. Esse é um aspecto que divide os proponentes de micro-mundos e os proponentes de ITSs. Enquanto que muitos pesquisadores acreditam que o sistema deva dosar o controle dado ao aprendiz de acordo com seu nível de competência, o Manifesto de Genebra coloca a questão num contexto educacional mais amplo, explicando:

> "A ineficiência relativa de controle no aprendiz é mais um artefato de vários anos de passividade na classe que uma limitação intrínseca do crescimento cognitivo. O design de sistemas com alto grau de diretividade simplesmente reproduzirá estes efeitos escolares e matará qualquer chance de inovação educacional." (TECFA, 1990)

Em minha proposta, a escolha das ferramentas do ambiente (Módulo Operacional, Módulo Declarativo) é deixada para o aprendiz ou é parcialmente transferida para o facilitador que deve estar afinado com a proposta metodológica do ambiente construcionista. A principal dificuldade para o facilitador está na determinação do momento a intervir e na forma de intervir, de modo a permitir que o aprendiz use as ferramentas que sejam adequadas à situação-problema e relevantes às suas estruturas de conhecimento.

O principal desafio dos ambientes de aprendizado baseados na idéia de micro-mundos, como é o caso deste trabalho, é promover um estilo de interação que quebre o modelo de aprendizagem passiva.

A seguir apresento alguns tópicos em pesquisa e desenvolvimento de sistemas, que podem ser explorados a partir dos resultados apresentados neste trabalho.

Uso das Representações propostas em Sistemas Existentes

Sistemas computacionais já existentes podem incorporar as idéias de representação gráfica propostas, de forma a apresentar visualmente informação de contexto. Existe pesquisa sendo feita na direção de sistemas com habilidade para localizar erros em programas. Tais sistemas, uma vez que encontram e analisam um erro, tendem a fornecer pouca explicação, apresentada, em geral, na forma textual, sobre o erro detectado. O sistema PDS (Shapiro, 1983) é um exemplo de sistema de depuração guiada que poderia incorporar as idéias de transparência propostas neste trabalho, como uma maneira de possibilitar ao novato a visualização do "contexto" onde decisões são tomadas pelo sistema. Os novatos principalmente, mas não exclusivamente, têm dificuldades em criar hipóteses sobre qual é o erro e, a partir daí, corrigir ambos: o erro e seu modelo conceitual falho da execução do programa. Ilustrações através de representações gráficas de características da linguagem de programação ou técnicas de programação, nos moldes das que apresentamos neste trabalho, poderiam ser usadas para demonstrar ao usuário de tais sistemas, *como* e *porquê* determinado erro foi detectado.

Outros Domínios

Visualizações de programas animados por representações gráficas, por outro lado, não precisam estar confinadas a linguagens de programação. Os princípios de *design* adotados neste trabalho, poderiam igualmente bem ser aplicados a outros sistemas dinâmicos que são complexos e que não têm seu uso restrito a espe-

cialistas da área de Ciência da Computação, como por exemplo, ambientes interativos baseados em computador, sistemas programáveis de controle, etc.

Além das possibilidades para trabalho futuro mencionadas, de natureza investigativa e de desenvolvimento de sistemas, a idéia que tem me atraído, após esta experiência com representações visuais e programação é a de reverter o ambiente gerado pelas ferramentas. Isto é, em vez de o sujeito trabalhar na notação convencional e usar o *feedback* visual das ferramentas, criar um ambiente que possibilite a ele programar através da própria representação gráfica.

Programação Visual

O uso de representações gráficas em ambientes de computador e, em particular, o interesse em programação visual, têm aumentado à medida em que recursos de interfaces gráficas de alta qualidade têm sido colocadas a disposição para aplicações. Um argumento usado na literatura para o esforço nesse sentido é que as pessoas podem absorver muito mais rapidamente informação visual que textual (O'Shea, 1983). Logo e Smalltalk são exemplos antigos e bem sucedidos de ambientes onde a interação com objetos computacionais é mediada por representações visuais. Os resultados observados do uso das ferramentas aqui apresentadas, por novatos, encorajam um trabalho de pesquisa e desenvolvimento de um ambiente visual para programação em lógica.

Este trabalho requisitou, desde o início, incursões em teorias de *design,* aprendizagem e desenvolvimento cognitivo que foram enriquecedoras e intelectualmente instigantes sob minha perspectiva pessoal como aprendiz. Respostas para as questões colocadas aqui e outras poderão ser buscadas em novos trabalhos de pesquisa, envolvendo também pesquisadores de outras áreas (de Psicologia, Linguística e Educação), o que certamente possibilitará colocarmos em prática, no nível da pesquisa, a questão da multiplicidade de visões como promotora da articulação do

conhecimento. A IA, a meu ver, depende desses elos para crescer, tanto na direção do desenvolvimento de tecnologia quanto na direção do entendimento dos processos de comunicação que acontecem entre o próprio Homem e a máquina.

Referências

Referências

Ackermann, E. (1990). Pathways into a child's mind: helping children become epistemologists. Em P. Heltne & L. Marquardt, ed - Symposium Proceedings Science Learning in the Informal Setting, Chicago pp. 7-19.

Ackermann, E. (1991). Perspective-Taking and reality construction. Paper presented at the 22nd Annual Symposium of the Jean Piaget Society, Montréal.

Anderson, J. R.,Pirolli, P, Farrell, R. (1988). Learning to Program Recursive Functions. Em M. T. H. Chi, R. Glaser, M. J. Farr (Eds.), *The Nature of Expertise*. Hillsdale,NJ: Lawrence Erlbaum Associates Inc..

APGRAPH (1990). Biblioteca de rotinas gráficas para o Arity/Prolog v5.1. CTI, Campinas.

Arity (1988). The Arity/Prolog Language Reference Manual Arity Corporation, Concord, MA.

Baranauskas, M. C. C. (1991a). Procedure,Function, Object or Logic? Proceedings of The Eighth International Conference on Technology and Education, Toronto, Ontario, pp. 730-731.

Baranauskas, M. C. C. (1991b). Creating a computer-based learning environment to help novices to develop Prolog programs. Proceedings of the Sixth International PEG Conference- Knowledge Based Environments for Teaching and Learning. Rapallo, Genova, pp. 92-98.

Bobrow, D. (1985). If Prolog is the answer, what is the question? or what it takes to support AI programming Paradigms. *IEEE Transactions on Software Engineering*, 11(11) pp 1401-1408.

Briggs, J. H. (1988). Why teach Prolog? The uses of Prolog in Education. Em J. Nichol, J. Briggs e J. Dean (Eds), *Prolog, Children and Students*. London: Kogan Page.

Brna, P. (1988 a). Improving Prolog Environments: a Review of Available Tools. DAI Research Paper no. 386. Department of Artificial Intelligence, University of Edinburgh.

Brna, P. (1991). Teaching Prolog techniques. Proceedings of The Sixth International PEG Conference. Rapallo, Genova, pp. 647-654.

Brna, P., Brayshaw, M., Bundy, A., Elsom-Cook, M., Fung, P. Dodd, T. (1988 b). An Overview of Prolog Debugging Tools. DAI Research Paper no. 398. Department of Artificial Intelligence, University of Edinburgh.

Brna, P., Pain, H., Du Boulay, B. (1990). Teaching, learning and using Prolog: understanding Prolog. *Instructional Science*, 19:247-256.

Brna, P., Pain, H., e Du Boulay, B. (1990b). Teaching, learning and using Prolog: supporting the programmer. *Instructional Science*, 20, 81-87.

Byrd, L. (1980). Understanding the control flow of Prolog programs. Em S. Tarnlund (Ed.), *Proceedings of the Logic Programming Workshop*, pp. 127-138.

Casanova, M. A., Giorno, F. A. C. e Furtado, A. L. (1987). *Programação em lógica e a linguagem Prolog*. SP: Editora Edgard Blucher Ltda.

Clocksin, W. F., Mellish, C. S. (1984). *Programming in Prolog* (2nd edition). New York: Springer-Verlag.

Colbourn, C. e Light, P. (1988). Peers, problem-solving and programming: projects and prospects. Em J. Nichol, J. Briggs e J. Dean (Eds), *Prolog, Children and Students*. London: Kogan Page.

Coombs, M. J. e Stell, J. G. (1985). A model for debugging Prolog by symbolic execution: the separation of specification and procedure. Research report MMIGR137, Dept. of Computer Science, University of Strathclyde.

Deliyanni, A. e Kowalski, R. A. (1979). Logic and semantic networks. *Communications of the ACM*, 22(3), pp. 184-192.

Dewar, A. D. e Cleary, J. G. (1986). Graphical display of complex information within a Prolog debugger. *International Journal of Man Machine Studies*, 25, 503-511.

du Boulay, J. B. H., O'Shea, T. e Monk, J. (1981). The black box inside the glass box: presenting computing concepts to novices. *International Journal of Man-Machine Studies*, 14, 237-249.

Eisenstadt, M. e Brayshaw, M. (1988). The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming. *The Journal of Logic Programming*, 5, 277-342.

Eisenstadt, M. e Brayshaw, M. (1990). A fine-grained account of Prolog execution for teaching and debugging. *Instructional Science*, 19(4/5), 407-436.

Eisenstadt, M., Brayshaw, M. (1988). The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming. *Journal of Logic Programming*, 5(4), 277-342.

Ennals, R. (1983). *Beginning Micro-Prolog*. New York: Harper and Row.

Fung, P., Brayshaw, M., Du Boulay, B. e Elsom-Cook, M. (1990). *Instructional Science*, 19:311-336.

Harel, I. (1991). *Children Designers*. Norwood, NJ: Ablex Publishing Corporation.

Hawley, R. (1987). *Artificial Intelligence Programming Environments*. Chichester: Ellis Horwood.

Holland, S. (1991). Visual programming in Prolog. Proceedinds of The Sixth International PEG Conference. Rapallo, Genova, pp 676-684.

Hook, K., Taylor, J. e Du Boulay, B. (1990). Redo "Try once and pass": the influence of complexity and graphical notation on novices' understanding of Prolog. *Instructional Science*, 19(4/5), 337-360.

Johanson, R. P. (1988). Computers, Cognition and Curriculum: Retrospect and Prospect. *Journal of Educational Computing Research*, 4(1), 1-30.

Kowalski, R. (1982). Logic as a programming language for children. Proceedings of the European Conference on AI, 2-10.

Loyd, J. W. (1984). *Foundations of Logic Programming*. New York: Springer-Verlag.

Ludke, M. e André, M. E. D. A. (1986). *Pesquisa em Educação: Abordagens Qualitativas*. SP: Editora Pedagógica e Universitária Ltda.

Mendelsohn, P., Green, T. R. G., Brna, P. (1990). Programming Languages in Education. Document TECFA 90-8, Université de Geneve.

Monteiro, R. C. (1991). A pesquisa qualitativa como opção metodológica. *Pro-Posições*, 5, pp. 27-35.

Muir, R., J. (1989). Object-Oriented Programming: Initial Experiences. *The Logo Exchange*, abril, 28-30.

Newell, A. e Simon, H. A. (1972). *Human Problem Solving*. New York: Prentice Hall.

Newman, D. Griffin, P. e Cole, M. (1989). *The Construction Zone*. Cambridge University Press.

O'Shea, T. e Self,J. (1983). Computer as a tool. Em T. O' Shea e J. Self (Eds.) *Learnig and teaching with computers*. Harvester Press Ltd.

Pain, H. e Bundy, A. (1987). What stories should we tell novice Prolog programmers? Em R. Hawley (Ed.), *Artificial Intelligence Programming Environments*. Chichester: Ellis Horwood.

Papert, S. (1980). *Mindstorms: Children, computers and powerful ideas*. New York: Basic Books.

Papert, S. (1986). *Constructionism: A new opportunity for elementary science education*. (A proposal to the National Science Foundation). Cambridge, MA: MIT Technology Laboratory.

Papert, S. (1987). A critique of technocentrism in thinking about the school of the future. Conference presented at Children in an Information Age: Opportunities for Creativity, Innovation & New Activities, Sofia, Bulgaria.

Papert, S. (1991). *New images of programming: in search of an educationally powerful concept of technological fluency*. (A proposal to the National Science Foundation). Cambridge, MA: MIT Technology Laboratory.

Park, W. B. (1990). User-Centered design. Em D. Norman (Ed.), *The Psychology of Everyday Things*, New York: Basic Books Inc., pp 187-217.

Patterson, J. H., Smith, M. S. (1986). The Role of Computers in Higher-order Thinking. Em J. A. Culbertson e L . L. Cunningham (Eds.), *Microcomputers and Education*, Chicago: University of Chicago Press.

Pea, R. D. (1985). Beyond Amplification: Using Computer to Reorganize Mental Functioning. *Educational Psychologist*, 20(4), 167-182.

Pea, R. D. (1986). Language-independent conceptual "bugs" in novice programming. *Journal of Educational Computing Research*, 2(1), 25-36.

Piaget, J. (1977). A teoria de Piaget. Em P. Mussen e S. Pfromm Netto (Eds.). *Manual de Psicologia da Criança*, (tradução brasileira de Carmichael's Manual of Child Psychology), S.P.: EDUSP , pp. 72-117.

Plummer, D. (1988). Coda: an extended debugger for Prolog. Em D. Plummer (Ed.) Logic Programming: Proceeding of the Fifth International Conference and Symposium, pp 496-511, Cambridge, MA: MIT Press.

Rajan, T. (1986). APT: A Principled Design for an Animated View of Program Execution for Novice Programmers. Technical Report 19, Human Cognition Research Laboratory, The Open University.

Rajan, T. (1990). Principles for the design of dynamic tracing environments for novice programmers. *Instructional Science*, 19(4/5), 377-406.

Rist, R. S. (1986). Plans in programming: definition, demonstration and development. Em M. Soloway e S. Iyengar (Eds.), *Empirical Studies of programmers*. Norwood: Ablex Publishing Corporation.

Ritchie, G. D. e Hanna, F. K. (1983). Semantic networks - a general definition and a survey. *Information Technology: Research and Development*, 2, 187-231.

Rocha, H. V. (1991). Representações Computacionais Auxiliares ao entendimento de conceitos de programação. Tese de Doutorado, outubro, FEE UNICAMP.

Scherz, Z., Maler, O., Shapiro, E. (1988). Learning with Prolog - A New Approach. Em J. Nichol, J. Briggs e J. Dean (Eds), *Prolog, Children and Student*, London: Kogan Page.

Schon, D. A. (1990). The design process. Em *Varieties of Thinking*. New York: Routledge, pp. 11-141.

Schubert, L. K. (1976). Extending the expressive power of semantic networks. *Artificial Intelligence* 7, 163-198.

Schubert, L.K., Goebel, R. G. e Cercone, N. J. (1979). The structure and organization of a semantic net for comprehension and inference. Em *Associative Networks*. Academic Press, Inc..

Sebesta, R. W. (1989). *Concepts of Programming Languages*. California: The Benjamin/Cummings Publishing Company,Inc..

Seviora, R. E. (1987). Knowledge Based Program Debugging Systems. *IEEE Software*, maio, 20-32.

Shapiro, E. Y. (1983). *Algorithm Program Debugging*. Cambridge, MA: MIT Press.

Soloway, E. M. e Woolf, B. (1980). Problems, plans and programs. *SIGSCE Bulletin*, 12, 16-24.

Spencer-Smith, R. (1988). Teaching logical analysis with Prolog. Em J. Nichol, J. Briggs e J. Dean (Eds), *Prolog, Children and Students*. London: Kogan-Page.

Steele, G. L., Jr. e Sussman, G. J. (1978). The art of the interpreter or the modularity. Technical memorandum AIM-453, MIT AI-Lab.

Sterling, L., Shapiro, E. (1986). *The Art of Prolog - Advanced Programming Techniques*. Cambridge, MA: MIT Press.

Taylor, J. (1990). Analysing novices analysing Prolog: what stories do novices tell themselves about Prolog?. *Instructional Science*, 19(4/5), 283-309.

Taylor, J. du Boulay, B. (1987). Studying novice programmers: why they may find learning Prolog hard. Em J. Rutkowska e C. Crook (Eds.), *The computer and human development*. Chichester: Wiley.

TECFA (1990). The Geneva Manifesto of Intelligent Learning Environments. Université de Geneve.

Valente, A. B. (1988). Como o computador é dominado pelo adulto. *Cadernos de Pesquisa*, 65, pp 30-37.

Valente, J. A. (1991). O computador no processo de construção do conhecimento e da inteligência. Conferência apresentada no III Congresso Brasileiro Logo. Universidade Católica de Petrópolis, Petrópoli.

Van Someren, M. W. (1985). Beginners problems in learning Prolog. Memorandum 54, Department of Experimental Psychology, University of Amsterdam.

Van Someren, M. W. (1990). What's wrong? Understanding beginners' problems with Prolog. *Instructional Science*, 19(4/5), 257-282.

Vygotsky, L. S. (1962). *Thought and language*. Cambridge, MA: MIT Press.

Vygotsky, L. S. (1978). *Mind in society: The development of higher psychological processes*. Cambridge, MA: Harvard University Press.

Weizenbaum, J. (1981). *Puissance de l'ordinateur et raison de l'home: du jugement au calcul*. Paris: Editions Informatics.

Winograd, T. e Flores, F. (1988). *Understanding Computers and Cognition A New Foundation for Design*, NY: Addison-Wesley Publishing Company, Inc..

Apêndice 1

Apêndice 1

Definições Básicas

Programas Prolog têm um significado operacional preciso: eles são instruções para execução em um computador - a máquina virtual Prolog. Algumas definições básicas necessárias para o entendimento do modelo de execução de Prolog (descrito no capítulo 3) são apresentadas*2, a seguir (definições 1 a 8).

Definição 1:

Um *programa Prolog* (ou Base de Dados*3) é um conjunto de cláusulas Prolog.

Definição 2:

O conjunto das *cláusulas Prolog* é o menor conjunto satisfazendo às seguintes condições:

(i) se A é uma fórmula atômica Prolog, então *A.* é uma cláusula Prolog chamada *cláusula unitária* (ou fato) Prolog.

(ii) se A e B1,...Bn são fórmulas atômicas Prolog, então a expressão *A:-B1,B2,...Bn* é uma cláusula Prolog chamada *cláusula não unitária* Prolog (ou regra), onde A é a *cabeça* e B1,B2,...Bn é o *corpo* da cláusula.

(iii) se C1,...Cn são fórmulas atômicas Prolog, então C1,...Cn é uma cláusula Prolog chamada *cláusula objetivo* Prolog (meta ou pergunta).

---

*2 Uma apresentação mais formal dos conceitos abordados nesta seção pode ser encontrada em (Casanova, 1987; Sterling, 1986, p. 68; Loyd, 1984).

*3 Na terminologia da area de Bancos de Dados, o termo tem outro significado e o mais apropriado para designar o programa Prolog seria "Base de Conhecimento". Estamos, entretanto, usando a terminologia de uso comum na literatura sobre Prolog.

Definição 3:

Uma *fórmula atômica* Prolog é uma cadeia da forma p(t1,...tn), para n>=0, onde t1,...tn são termos Prolog e p é um átomo no papel de um símbolo predicativo n-ário.

Definição 4:

O conjunto dos *termos* Prolog é o menor conjunto satisfazendo às seguintes condições:

(i) toda variável Prolog é um termo Prolog.

(ii) toda constante Prolog é um termo Prolog.

(iii) se t1,...tn são termos Prolog e f é um átomo, então f(t1,...tn) é um termo Prolog, onde o átomo f tem o papel de um símbolo funcional n-ário. Diz-se ainda que a expressão f(t1,...tn) é um *termo funcional* Prolog.

Para ilustrar as definições, o conjunto de sentenças a seguir representa um programa Prolog que define a concatenação de duas listas:

```
concatena([H|T],L,[H|T1]):- concatena(T,L,T1).
concatena([],L,L).
```

A primeira sentença é uma cláusula não unitária Prolog (regra); a segunda sentença é uma cláusula unitária Prolog (fato); *concatena([],L,L)* é uma fórmula atômica Prolog; *[] e L* são termos Prolog.

Definição 5:

Um termo t é uma *instância comum* de dois termos t1 e t2 se existirem substituições s1 e s2 tal que t iguala t1s1 e t2s2.

Definição 6:

Um *unificador* de dois termos é uma substituição que faz os termos idênticos. Se dois termos têm um unificador, dizemos que eles *unificam*.

Exemplo:

concatena([1,2,3],[3,4],Lista) e concatena([X|Xs],Ys,[X|Zs] unificam. Uma substituição *unificadora* é :

{X=1, Xs=[2,3], Ys=[3,4], Lista=[1|Zs]}.

Sua *instância comum*, determinada por esta substituição unificadora é dada por:

concatena([1,2,3],[3,4],[1|Zs]).

Definição 7:

O *unificador mais geral* ou *mgu* de dois termos é um unificador tal que a instância comum associada é *mais geral.*

Definição 8:

Um termo w é *mais geral* que um termo t se t é uma instância de w, mas w não é uma instância de t.

Um programa Prolog tem um significado declarativo, que é herdado de programação em lógica. O significado declarativo de um programa em lógica, P é dado pelo seu "modelo mínimo", apresentado a partir da sequência de definições a seguir.

Definição 9

O *Universo de Herbrand de P (U(P))* é o conjunto de todos os termos básicos que podem ser formados de constantes e símbolos funcionais que aparecem em P.

Definição 10.

A *Base de Herbrand (B(P))* é o conjunto de todas as metas que podem ser formadas dos predicados em P e dos termos em U(P).

Definição 11.

Uma *interpretação* para um programa em lógica é um subconjunto da base de Herbrand. Uma interpretação associa verdade e falsidade aos elementos da base de Herbrand. Uma meta em B(P) é verdadeira com respeito a uma interpretação se é membro dela e é falsa em caso contrário.

Definição 12.

Uma interpretação *I* é um *modelo* para um programa em lógica se, para cada instância básica de uma cláusula no programa do tipo A:-B1,...Bn, A pertence a I se B1,...Bn pertencem a I.

Definição 13.

O modelo obtido como intersecção de todos os modelos para um programa em lógica é chamado *modelo mínimo (M(P))*.

Para ilustrar as definições, consideremos a definição do programa que "concatena" duas listas, usado no capítulo anterior e repetida aqui:

concatena([X|Xs],Ys,[X|Zs]):- concatena(Xs,Ys,Zs).
concatena([],Ys,Ys).

O Universo de Herbrand é formado por todas as listas que podem ser construídas usando a constante [] (lista vazia), ou seja: [],[[]],[[],[]], etc.

A Base de Herbrand é constituída de todas as combinações do predicado "concatena" com listas. Exemplo de elemento de B(P): concatena([],[],[[]]).

O significado declarativo do programa em lógica é dado por todas as instâncias básicas de "concatena([],Xs,Xs)", isto é: concatena([],[[]],[[]]) etc. e por metas que são implicadas logicamente por aplicação da regra, como por exemplo: concatena([[]],[],[[]]). Tais instâncias básicas são um subconjunto de B(P).

Apêndice 2

Folha-Tarefa I

Considere os exemplos de Bases de Dados, a seguir. Examine-as formulando "perguntas" e analisando as respostas obtidas.

Base de Dados 1:

```
ladrao(joao).
gosta_de(maria,doce).
gosta_de(maria,vinho).
gosta_de(joao,X):-gosta_de(X,vinho).
pode_roubar(X,Y):-ladrao(X),gosta_de(X,Y).
```

a. Perguntas: (metas)

```
? gosta_de(maria,vinho).

? gosta_de(maria,chocolate).

? gosta_de(joao,X).

? pode_roubar(X,Y).

? ...

? ...

? ...

? ...
```

b. Expanda a Base de Dados do exemplo, incluindo novos fatos e regras.

c. "Salve" a Base de Dadós expandida em seu disquete.
nome do arquivo:...

d. Questione a nova Base de Dados.

```
? ...

? ...

? ...

? ...
```

Folha-Tarefa I

Base de Dados 2:

```
% relacoes de parentesco.
pai_de(maria,joao).
mae_de(pedro,maria).
pai_de(pedro,paulo).
pais_de(X,Y):-mae_de(X,Y).
pais_de(X,Y):-pai_de(X,Y).
ancestral_de(X,Y):-pais_de(X,Y).
```

a. Perguntas (metas):

? ...

? ...

? ...

b. Inclua na Base de Dados outras relações de parentesco, como por exemplo: irmão, tio, avó, etc.

c. "Salve" sua nova Base de Dados em disquete
nome do arquivo:...

d. Questione a nova Base de Dados.

Folha-Tarefa I

Base de Dados 3.

```
lados(quadrado,4).
lados(retangulo,4).
lados(triang_eq,3).
lados(losango,4).
lados(hexagono,6).
quadrilatero(X):-lados(X,4).
lados_iguais(quadrado).
lados_iguais(losango).
lados_iguais(triang_eq).
ang_iguais(quadrado).
ang_iguais(retangulo).
ang_iguais(triang_eq).
lados_paralelos(X):-lados_iguais(X),quadrilatero(X).
lados_paralelos(X):-ang_iguais(X),quadrilatero(X).
```

Folha-Tarefa I

Base de Dados 4.

```
% operacoes em listas.
%e_lista(X) :- e verdadeiro se X e uma lista.
e_lista([]).
e_lista([X|Y]).

% ultimo(L,X) :- o ultimo elemento da lista L e X.
ultimo([X],X).
ultimo([X|Y],Z) :- ultimo(Y,Z).

% comprimento(L,X) :- X e o comprimento da lista L.
comprimento([],0).
comprimento([X|Y],Z) :- comprimento(Y,Z1), Z is Z1 + 1.

% soma(L,X) :- X e a soma dos elementos da lista L.
soma([],0).
soma([X|Y],Z) :- soma(Y,Z1), Z is X + Z1.
```

Folha-Tarefa I

Base de Dados 5.

```
refeicao(X,Y,Z):-entrada(X),pprincipal(Y),sobremesa(Z).
pprincipal(X):-carne(X).
pprincipal(X):-peixe(X).
carne(X):-boi(X).
carne(X):-ave(X).
peixe(dourado).
peixe(pintado_na_brasa).
ave(galinha).
ave(pato).
boi(picanha).
boi(file_mignon).
entrada(salada).
entrada(frios).
entrada(sopa).
sobremesa(sorvete).
sobremesa(fruta).
sobremesa(pudim).
```

Folha-Tarefa B

Para cada uma das Bases de Dados e respectivas perguntas (metas), responda o item a) antes de iniciar atividade no micro, o item b) enquanto estiver no micro e o item c) após o uso da(s) ferramenta(s).

a) Qual seria a resposta para a pergunta?
Explique, se achar necessário.

b) Que resposta você obteve?
Como você acha que o sistema chegou a essa resposta?

c) Descreva o processo usado pelo sistema para chegar a uma resposta (última página).

B.D. 1:

```
local(P,L):-em(P,L).
local(P,L):-visita(P,O), local(O,L).

em(alan,sala19).
em(jane,sala54).
em(bete,escritorio).

visita(davi,alan).
visita(alberto,davi).
visita(janete,bete).
```

Perguntas:

```
? local(alberto,X).
```
a1)

```
?. local(X,escritório).
```
a2)

```
?. local(X,Y).
```
a3)

b1)

b2)

b3)

Folha-Tarefa B

B.D. 2:

```
p:-a,b.
a.
b.
```

pergunta:

```
? p.
```

a)

b)

B.D. 3:

```
p:-a,b.
a:-d,e.
b.
a.
```

pergunta:

```
? p.
```

a)

b)

Folha-Tarefa B

B.D. 4:

```
p:-a,b,c.
p:-a,b.
a.
a.
b.
```

Pergunta:

? p.

a)

b)

B.D. 5:

```
p:-a,b.
a:-d,e,f.
a.
b.
```

Pergunta:

? p.

a)

b)

Folha—Tarefa B

B.D. 6:

```
a:-d,e,f.
a.
b.
p:-a,b.
```

Pergunta:

```
? p.
```

a)

b)

B.D. 7:

```
p:-a,b,c.
b.
b.
a.
```

Pergunta:

```
? p.
```

a)

b)

Folha-Tarefa B

B.D. 8:

```
gosto(X):-vermelho(X), carro(X).
gosto(X):-azul(X),bicicleta(X).

vermelho(cereja).
vermelho(tijolo).
vermelho(fusca).

carro(mercedes).
carro(santana).

azul(jeans).
azul(honda).
azul(vidro).

bicicleta(caloi).
bicicleta(minha).
```

Pergunta:

```
? gosto(X).
```

a)

b)

Folha_Tarefa B

B.D. 9:

```
p(X):-a(X),b(X).
a(jane).
b(sue).
b(jane).
```

Pergunta:

```
? p(X).
```

a)

b)

B.D. 10:

```
p(X):-a(X),b(X),c(X,Y).
a(jane).
b(jane).
b(sue).
c(sue,helen).
c(helen,fred).
```

Pergunta:

```
? p(X).
```

a)

b)

Folha-Tarefa C

Em cada uma das Bases de Dados a seguir, verifique se existe algum tipo de "erro" e responda o item a) antes de iniciar atividade no micro, o item b) enquanto estiver no micro e o item c após o uso da(s) ferramenta(s):

a) O quê você acha que está errado?

b) Que erro foi detectado? Proponha uma "correção", modificando a B.D. e salvando-a em disquete. (Escreva o nome do arquivo).

c) Que erro foi detectado? Qual é a causa do erro?

B.D. 1:

```
mais_velho(X,Y):- IX > IY,
                  idade(X,IX),
                  idade(Y,IY).

idade(joao,17).
idade(tom,18).
idade(sueli,15).
```

a)

b)

nome do arquivo que contem a nova B.D.:

c)

Folha-Tarefa C

B.D. 2:

```
mdc(I,J,K):- J > 0,
             R is I mod J,
             mdc(J,R,K).
```

% X mod Y fornece o resto da divisao inteira de X por Y.

a)

b)

nome do arquivo que contem a nova B.D.:

c)

B.D. 3:

```
pai_de(maria,joao).
mae_de(pedro,maria).
pai_de(pedro,paulo).
pais_de(X,Y):-mae_de(X,Y).
pais_de(X,Y):-pai_de(X,Y).
ancestral_de(X,Y):-pais_de(X,Z),ancestral_de(X,Y).
ancestral_de(X,Y):-pais_de(X,Y).
```

a)

b)

nome do arquivo que contem a nova B.D.:

c)

Folha-Tarefa C

B.D. 4:

```
caminho(No,No).
caminho(No1,No2):- caminho(No1,X),
                   adjacente(X,No2).

adjacente(a,b).
adjacente(a,c).
adjacente(b,d).
adjacente(c,d).
adjacente(d,e).
adjacente(f,g).
```

a)

b)

nome do arquivo que contem a nova B.D.:

c)

B.D. 5:

```
intervalo(I,J,I).
intervalo(I,J,K) :- I =< J,
                    I1 is I + 1,
                    intervalo(I1,J,K).
```

a)

b)

nome do arquivo que contem a nova B.D.:

c)

Folha- Tarefa C

B.D. 6:

```
ordenada([X]).
ordenada([X,Y|Z]) :- X < Y , ordenada(Z).
```

a)

b)

nome do arquivo que contem a nova B.D.:

c)

B.D. 7:

```
insere(X,[Y|Z]),[X,Y|Z]) :- X  < Y.
insere(X,[Y|Z],[X|Z1]) :- X >  Y, insere(Y,Z,Z1).
insere(X,[],[X]).
```

a)

b)

nome do arquivo que contem a nova B.D.:

c)

Folha-Tarefa A

Para cada "problema" formulado, crie uma Base de Dados que "represente" o problema de forma a responder perguntas a respeito do domínio em questão.

Problema 1.

Crie uma Base de Dados que descreva a classe dos animais pelas suas características (física, comportamental, etc.).

Salve sua B.D. em disquete.
Nome do arquivo:....

Perguntas à B.D.:
?
?
?
?

Folha-Tarefa A

Problema 2.

Crie uma B.D. que represente o fatorial de um número.

Salve sua B.D. em disquete.
Nome do arquivo:...

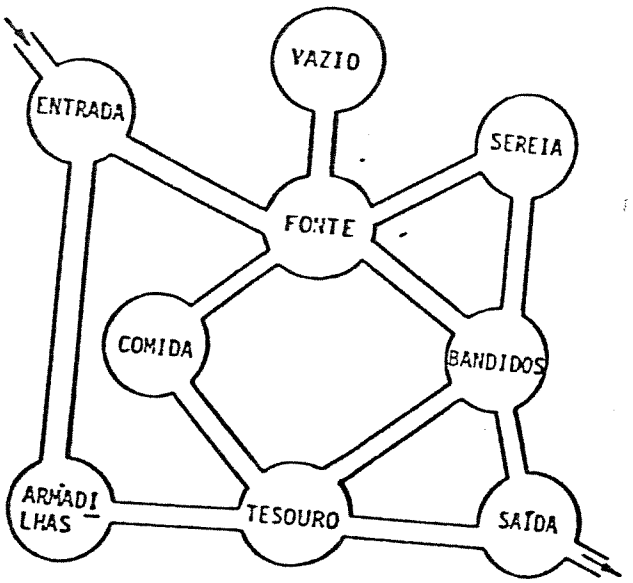
Perguntas à B.D.:
?
?
?

Folha-Tarefa A

Problema 3.

Busca ao Tesouro.
Considere o mapa a seguir. A partir da entrada, nosso objetivo é atravessar o labirinto, sem passar pelos perigos (armadilhas e bandidos), encontrar o tesouro e sair.

Construa uma B.D. que represente o problema e através dela consigamos todos os percursos válidos. da entrada até a saida.



Salve sua B.D. em disquete.
Nome do arquivo: ...

Perguntas à B.D.:

?
?
?
?

Folha-Tarefa A

Problema 4.

Crie uma Base de Dados para representar a potência de um número, de forma a gerar respostas para questões do tipo "X elevado a Y é igual a ...".

Salve sua B. D. em disquete.
Nome do arquivo: ...

Perguntas à B.D.:

?
?
?

Folha-Tarefa A

Problema 5.

Crie uma B.D. através da qual possamos obter o produto dos elementos de uma lista de números inteiros.

Salve sua B.D em disquete.
Nome do arquivo: ...

Perguntas à B.D.:

?
?
?